Algorithmique et complexité

Olivier Ridoux

Algorithmique et complexité

Olivier Ridoux - 2020

Introduction

Les **algorithmes** sont l'abstraction des procédures que mettent en œuvre les systèmes informatiques. Un **algorithme** est le moyen de résoudre des **instances** d'un **problème** par une méthode de calcul **automatique**, c-à-d. sans jugement par un opérateur. Ex. calculer le **produit de deux entiers en notation décimale** est un problème, la méthode de **multiplication apprise à l'école** est un algorithme, et **123 × 45** est une instance. **Problème** n'est donc pas synonyme de **difficulté**. Noter que la représentation des données fait partie de la spécification du problème.

Plusieurs algorithmes peuvent résoudre le même problème. Se pose alors la question de les comparer. Un critère possible est leur **complexité**, c-à-d. une mesure du coût de les exécuter pour résoudre une instance (**complexité** n'est pas non plus synonyme de **difficulté !**). On en tire immédiatement une question plus abstraite qui est **Quelle complexité attendre des meilleurs algorithmes qui résolvent un problème donné ?** On s'intéresse alors à une propriété **intrinsèque** des problèmes, indépendante des technologies mises en œuvre, langage de programmation ou machine d'exécution, y compris des technologies qui restent à découvrir.

L'approche de l'algorithmique choisie ici est donc celle d'une **science des relations entre problèmes et algorithmes**. On abordera donc aussi la théorie de la **calculabilité**, pour montrer que certains problèmes, pourtant bien posés, **n'ont pas de solution algorithmique**. En ce sens, cette science joue le rôle des théories physiques qui fixent des limites à l'ingénierie : pas de mouvement perpétuel, pas de température en dessous du zéro absolu, pas de vitesse plus grande que celle de la lumière, etc., mais aussi **pas de démonstration automatique pour tout le calcul des prédicats, ni de solution optimale et efficace dans tous les cas pour la planification automatique**, etc.

Conclusions et perspectives

La **pensée algorithmique** est au **cœur** de l'informatique. Tout traitement automatisé met en œuvre des algorithmes. Cependant, il n'existe pas de langage d'usage généralisé pour noter des algorithmes. En fait, un algorithme n'est jamais lié à une notation, c'est un **concept**. En cela, les algorithmes diffèrent des programmes car ces derniers sont liés aux langages de programmation dans lesquels ils sont écrits. Les algorithmes sont bien des **abstractions des programmes**.

La pensée algorithmique contient entre autres des éléments de stratégie (ex. **diviser pour régner** qui donne d'excellents algorithmes), et même des éléments de posture quasi politique (ex. **répéter une action locale pour un effet global**, ou **la bonne représentation fait le bon algorithme**).

Enfin, la pratique algorithmique doit être empreinte de pragmatisme. Un algorithme de complexité médiocre peut être préféré à un algorithme qui serait meilleur sur le papier.
Ex. 1, l'algorithme du **SIMPLEX** pour résoudre les **systèmes d'(in)équations linéaires** : il est de complexité exponentielle dans le pire des cas, mais c'est un pire cas rarement réalisé ; il est souvent préféré à des algorithmes polynomiaux plus modernes car il est plus facile à mettre en œuvre.
Ex. 2, l'algorithme **quicksort** pour le **tri** : il n'est pas optimal en théorie , mais il est excellent en pratique.

Bibliographie

• « ***On Computable Numbers with an Application to the*** **Entscheidungsproblem** », Turing (1936), à lire dans « ***The Annotated Turing*** », Petzold (Wiley, 2008).
• « ***The Feeling of Power*** », Asimov (1958).
• « ***Computer Algorithms*** », Baase (Addison-Wesley, 1983).
• « ***Computers LTD, what they really can't do*** » et « ***Algorithmics, The Spirit of Computing*** », Harel (OUP, 2003, et Addison-Wesley, 1987).
• « ***The new Turing Omnibus*** », Dewdney (Owl Books, 2001).
• « ***Introduction to the Design and Analysis of Algorithms*** », Levitin (Addison Wesley, 2003).
• « **Petite introduction à l'algorithmique** », Damphousse (Ellipses, 2005).
• « **Calculateurs, calculs, calculabilité** », Ridoux et Lesventes (Dunod, 2008).
• « ***Algorithms Unplugged*** », Vöcking et al. (Springer, 2011), voir aussi « ***Algorithmus des Woche*** » (www-i1.informatik.rwth-aachen.de/~algorithmus).
• « ***Algorithms Unlocked*** », Cormen (MIT Press, 2013).

Même pas peur de $\mathcal{NP}$!

Appartenir à $\mathcal{NP}$ n'est pas une mauvaise nouvelle en soit. Ce qui est inquiétant c'est d'appartenir à $\mathcal{NP}$ avec peu d'espoir d'appartenir à $\mathcal{P}$. C'est le cas des problèmes **$\mathcal{NP}$-complets**. **SAT** est le premier problème à avoir été démontré **$\mathcal{NP}$-complet**. À sa suite des centaines de problèmes l'ont aussi été, et parmi eux de très nombreux problèmes de grande importance pratique. En voici une sélection :

• **TSP** (***Traveling Salesman Problem*** – Problème du voyageur de commerce) : soit **n** points (ex. **n villes**), la donnée des **distances** qui les séparent (ex. **distance kilométrique** ou **coût du voyage**), trouver **le plus court chemin** qui part d'un point choisi à l'avance, passe par tous les points et revient au point de départ.

• **COL** (**coloriage de graphe**) : soit un **graphe** à **n** sommets, trouver **le plus petit nombre de couleurs** qui permette de colorier les sommets du graphe de telle façon que deux sommets voisins aient toujours des **couleurs différentes**. C'est le prototype de problèmes de planification ou d'allocation de ressources.

• **KP** (***Knapsack Problem*** – Problème du sac-à-dos) : soit un **contenant** (ex. un **sac-à-dos**), sa **capacité** (ex. en litre), et des objets caractérisés par un **volume** et une **valeur**, trouver un chargement du sac qui respecte sa capacité et soit de **valeur maximale**. C'est le prototype de problèmes de sélection.

Ces problèmes formalisent des questions auxquelles il serait extrêmement utile d'avoir des réponses et ils demandent à être résolus en réalité pour de très grosses instances. Comment doit-on considérer le fait qu'ils sont **$\mathcal{NP}$-complets** ?

• **Tenter sa chance** : essayer en espérant que ça passe. C'est ce qu'on fait, avec subtilité, pour **SAT**. Les **SAT *solvers*** font l'objet d'une compétition effrénée.

• **Accepter des quasi-solutions** : ne plus rechercher les réponses **optimales**, mais des réponses **acceptables**. Ex. il existe de très bonnes **heuristiques** pour les problèmes listés. L'une d'elles consiste à construire progressivement les éléments d'une réponse en faisant toujours le choix apparemment le plus favorable sans jamais revenir dessus (on appelle cette heuristique la **gloutonnerie**). Ex. pour **TSP**, toujours choisir le point le plus proche du point courant, et s'y rendre. C'est sub-optimal car parfois il vaut mieux faire maintenant un choix moins favorable (un sacrifice) qui permet un choix plus favorable plus tard (le bénéfice du sacrifice).

• **Utiliser un quasi-algorithme** : utiliser des méthodes d'optimisation comme **l'algorithmique génétique**, ou le **recuit simulé**, ou des méthodes de **l'intelligence artificielle** comme l'apprentissage profond (***deep learning***).

Dans tous les cas, il est souvent nécessaire de résoudre des instances de problèmes **$\mathcal{NP}$-complets** et il ne faut pas hésiter à le faire, en en prenant les moyens.

Pourquoi $n^{\log_2 7}$ et $n \times \log n$?

On invente rarement des algorithmes *ex-nihilo* ; on applique plutôt des stratégies pour combiner des traitements élémentaires. **Diviser pour régner** en est un bon exemple quand 3 conditions sont réunies :

1. On peut **diviser** toute instance $I^{(0)}$ du problème en plusieurs instances $I^{(1)}$ telles que la solution $S^{(0)}$ de $I^{(0)}$ peut être **reconstruite** à partir de celles des $I^{(1)}$.
2. La somme des coûts de construction et de traitement des $I^{(1)}$ et des coûts de reconstruction de $S^{(0)}$ est **moindre** que le coût de traitement direct de $I^{(0)}$.
3. On peut **itérer** le procédé en construisant des instances $I^{(i+1)}$ dérivées des instances $I^{(j)}$ jusqu'à obtenir des instances que l'on ne divise plus.

Ex. **Strassen** a montré en 1969 que toute instance $I^{(0)}$ du problème de produit matriciel $n \times n$ (avec $n=2^k$ pour simplifier) pouvait être divisée en **7** instances $I^{(1)}$ de taille $2^{k-1} \times 2^{k-1}$. En itérant, on obtient 7^k instances $I^{(k)}$, et $7^k = 7^{\log_2 n} = n^{\log_2 7} = n^{2,81\ldots}$.

De même, toute instance $I^{(0)}$ du problème du tri d'une liste de **n** éléments (avec $n=2^k$ à nouveau) peut être décomposée de façon à trier la **première moitié** de la liste, puis la **seconde**, puis **fusionner** les deux. Le coût de la fusion est proportionnel à la longueur des listes à fusionner. À chaque itération on a 2^i instances $I^{(j)}$ de tri de listes de longueurs 2^{k-i}, soit à chaque fois 2^k éléments à fusionner. Il y a **k** itérations, et le coût total est donc de $2^k \times k = n \times \log_2 n$.

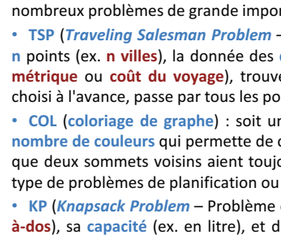
Comment peut-on être dans $\mathcal{NP}$?

Rappelons que $\mathcal{P} \subseteq \mathcal{NP}$. Il n'est donc pas extraordinaire d'être dans $\mathcal{NP}$; le produit matriciel et le tri sont dans $\mathcal{P}$ donc dans $\mathcal{NP}$!

Le plus fameux problème de $\mathcal{NP}$ est le problème **SAT** du **calcul des propositions**. Il s'agit de déterminer si une formule **F** à **n** variables est **satisfaisable ou non**. Une solution naïve consiste à construire la **table de décision** de **F** (2^n lignes), puis tester si elle contient une ligne où le résultat est **Vrai**. SAT est donc dans $O(2^n)$. Une fois données des valeurs pour les variables de **F** il est facile (**polynomial**) de **vérifier** si elles satisfont **F**. SAT est donc dans $\mathcal{NP}$. Et cette fois-ci on ne sait pas si SAT est dans $\mathcal{P}$; on ne connaît pas d'algorithme dont l'ordre de grandeur soit meilleur que 2^n, mais on n'a pas la preuve qu'il n'en existe pas.

SAT a une autre propriété ; il est **$\mathcal{NP}$-complet** (**Cook**, 1971). Cela signifie que tout problème de $\mathcal{NP}$ peut être traduit en **SAT** de façon à le résoudre en utilisant un solveur de **SAT**. **SAT** représente donc la **$\mathcal{NP}$-ité** par excellence.

La théorie de la complexité à pour objet de déterminer le **coût d'exécution des algorithmes** et le **coût de résolution des problèmes**. Le coût s'évalue le plus souvent en **temps de calcul**, parfois en **mémoire utilisée**, mais pourrait décrire une autre dimension, comme l'**énergie**. Le coût dépend évidemment de l'**instance** du problème considéré. Ex. il est plus facile de calculer **la somme de 12 et 13** que celle de **16578 et 87659**. Donc ce n'est pas tant le coût qui est évalué que la façon dont il dépend des instances. Cela se représente par une fonction d'une caractéristique **n** des instances (ex. **n** est le nombre de chiffres) vers le coût : **caractéristique → coût**.



On dit que $g = O(f)$ (à lire **impérativement** $g \in O(f)$, ou **f domine g**), si $\exists N . \exists C . \forall n > N . [\, g(n) \leq C \times f(n) \,]$. On y lit que **f** est **au delà** d'un seuil **N**, on peut trouver un facteur **C**, tel que **g** est toujours sous **C×f**. Remarquer que **en aucun cas g = O(f) ne signifie que g est inférieure à f ou que g tend vers f** ! Ex. **cos(n) = O(n)**. D'autres expressions d'ordre de grandeur sont **Ω(f)** pour les **fonctions qui dominent f**, et **θ(f)** pour les **fonctions qui dominent et sont dominées par f**, et d'autres encore.

On peut montrer que $O(f) = O(c \times f)$ si **c** est une constante ou que $O(f) = O(f+g)$ si **g=O(f)**. Un ordre de grandeur ne dépend donc pas des facteurs constants ni des ajouts d'ordres de grandeur plus faibles. En particulier, $O(\log_a x) = O(\log_b x) = O(\log x^a) = O(\log x^b)$, pour tout **a** et **b** ; on peut donc oublier la base du logarithme. Très souvent, les ordres de grandeur s'expriment à l'aide de fonctions simples : élévation à une puissance (n^c), logarithme ($\log n$), exponentielle (b^n), ou des produits de fonctions simples (ex. $n \times \log n$), où **n** est la taille de l'instance.

L'effort de formalisation de la logique de la fin du XIX^e^ siècle et du début du XX^e^ a fait se demander ce qu'est le calcul afin de concevoir des méthodes de démonstration automatique ou de vérification de preuve qui soient incontestables. En 1936, Alan Turing (1912-1954) propose un mécanisme de calcul appelé aujourd'hui **Machine de Turing**. Ce mécanisme comporte un ruban pour stocker des données, et qu'un jeu de règles **R** réalise **une fonction f** (noté $\mathcal{M}[R] = f$), si à chaque fois qu'on représente une entrée **x** sur le ruban, **R** réalise un calcul qui s'arrête avec la représentation de **f(x)** sur le ruban. Pour certaines entrées, le calcul **ne s'arrête pas et rend donc aucun résultat** ; ce sont des entrées où **f n'est pas définie**.

Turing démontre les trois résultats suivants (en notation moderne et simplifiée) :

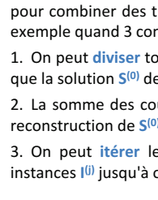
1. Il existe un jeu de règles **MTU** tel que pour tout jeu de règles **R**, il est vrai que $\mathcal{M}[MTU](R, x) = \mathcal{M}[R](x)$. Autrement dit, l'machine MTU peut simuler tout jeu de règles à condition qu'il soit représenté comme une donnée sur le ruban. On appelle cela une **Machine de Turing Universelle**, et c'est le modèle théorique de tous les ordinateurs. Pour ceux-ci la mémoire est le ruban, le processeur est les règles MTU, et leurs programmes stockés en mémoire sont autant de jeux de règles de calcul **R**.
2. Il n'existe pas de jeu de règles **H** tel que pour tout jeu de règles **R** et valeur **v** représentés initialement sur le ruban, $\mathcal{M}[H](R, v) = \mathit{Vrai}$ si $\mathcal{M}[R](v)$ **s'arrête**, **Faux sinon**. Le problème qui serait résolu par **H** s'appelle le **Problème de l'arrêt** (***Halting Problem***). C'est le premier problème dont on ait montré formellement qu'il n'avait pas de solution par une machine de Turing. On dit qu'il est **indécidable**.
3. Il n'existe pas de jeu de règles **P** tel que pour toute formule ϕ du **calcul des prédicats** représentée initialement sur le ruban, $\mathcal{M}[P](\phi) = \mathit{Vrai}$ si ϕ **est démontrable**, **Faux sinon**. Turing démontre cela en montrant que si une telle machine existait on pourrait s'en servir pour résoudre le Problème de l'arrêt en le modélisant dans la logique du calcul des prédicats. **Le calcul des prédicats est donc lui aussi indécidable**. Cette méthode de preuve, dite par **réduction**, est typique de l'algorithmique.

On pourrait objecter à ces résultats négatifs qu'ils viennent peut-être de ce que les machines de Turing ne constituent pas un mécanisme de calcul assez puissant. Il n'y a jamais eu de preuve ni de contraire, mais un faisceau de présomptions extrêmement fort. En effet, à partir de 1936 un très grand nombre de mécanismes alternatifs ont été proposés, d'une variété extraordinaire, mais ils se sont tous montrés équivalents aux machines de Turing. On en conjecture ce qu'on appelle la **Thèse de Church-Turing** qui dit essentiellement que tout ce qui peut être calculé peut l'être par une machine de Turing, et donc par tout autre mécanisme équivalent.

Considérons l'algorithme de **produit matriciel** qui effectue les produits scalaires des lignes d'une matrice par les colonnes de l'autre. Pour des matrices de taille $n \times n$, il calcule n^2 coefficients qui chacun demande de faire **n** produits élémentaires. Il fait donc n^3 produits élémentaires. Le problème **produit matriciel** est donc dans $O(n^3)$, mais on peut aussi affirmer qu'il est dans $\Omega(n^2)$ car il y a n^2 coefficients à calculer. Le **Θ** du coût intrinsèque du problème est donc entre n^2 et n^3. On connaît des algorithmes qui font moins de n^3 produits, et plus que n^2, mais on ne connaît pas le **Θ** de ce problème, pourtant assez banal.

Voici des ordres de grandeur et des problèmes et algorithmes qui les illustrent :

- **log n** : la recherche **dichotomique** (ex. comme dans un dictionnaire).
- **n** (= **linéaire**) : la recherche séquentielle (ex. comme dans un livre sans index).
- **n×log n** : un O et un **Ω** du coût du problème du tri par comparaison-échange, donc le coût intrinsèque de ce problème, son **Θ**.
- **n^2** (= **quadratique**) : le pire des cas de l'algorithme du **tri rapide** (***quicksort***), donc son **Θ**. Ce n'est donc pas un algorithme optimal pour le problème du tri.
- $n^{\log_2 7} = n^{2,81\ldots}$: **l'algorithme de Strassen** pour le produit matriciel.
- **n^3** : l'algorithme usuel pour le produit matriciel.
- **n^c** (= **polynomial**) : ordre de grandeur des coûts qualifiés de **supportables**, même si un coût de n^{100} est certainement **exorbitant**.
- **b^n** (= **exponentiel**) : ordre de grandeur de coûts qualifiés d'**insupportables**. Noter que des coûts bien plus élevés sont possibles et fréquents en pratique.



On appelle **polynomiaux** (on note $\mathcal{P}$) les problèmes dont l'ordre de grandeur du coût est borné par un polynôme.

Pour certains problèmes on ne connaît pas de solution polynomiale, mais on sait **vérifier** en temps polynomial qu'une proposition de réponse est correcte. On dit qu'ils sont dans $\mathcal{NP}$. Ex. on ne connaît pas d'algorithme polynomial pour calculer les facteurs premiers d'un nombre, mais il est facile de vérifier que des nombres sont les facteurs d'un autre ; il suffit de les multiplier, ce qui a un coût polynomial.

$\mathcal{P} \subseteq \mathcal{NP}$, car **calculer la réponse** permet de **vérifier une réponse proposée**, mais on ne sait pas si $\mathcal{P} \neq \mathcal{NP}$. C'est une des grandes questions ouvertes des mathématiques du XXI^e^ siècle, même si le consensus est plutôt que $\mathcal{P} \neq \mathcal{NP}$.

Le nom **d'algorithme** vient d'un hommage très ancien au mathématicien persan **Al Khwarizmi** (~780 – ~850). Celui-ci a, entre autres, étudié et développé la **représentation positionnelle des nombres** qui avait été découverte en Inde. Ses écrits traduits en latin ont fait connaître cette méthode en Europe à partir du XII^e^ siècle. L'usage de la notation positionnelle et des opérations associées était tellement révolutionnaire par rapport à l'utilisation d'**abaques** que cette méthode a reçu globalement le nom **d'algorithme**, en hommage à celui qui l'a fait connaître, mais cela n'a pas beaucoup à voir avec **l'algorithmique moderne**. Le mot a ensuite été utilisé pour toute méthode de calcul systématique jusqu'à ce que la notion moderne **d'algorithme** apparaisse au premier tiers du XX^e^ siècle avec les travaux de **Hilbert**, **Ackermann**, **Gödel**, **Turing**, **Church**, **Post**, etc.

Cet état de fait archaïque a duré jusque vers le XVI^e^ siècle à partir duquel sont apparus progressivement les éléments des notations mathématiques modernes : **variables**, **opérateurs**, **expressions**, etc. Mais ce n'est qu'au XX^e^ siècle que les concepts modernes **d'itération**, de **conditionnelle**, de **récursion**, etc., sont apparus en tant que tels.

Mode d'emploi :

- Lire cette page de prérequis, calcul et logique, comprendre, apprendre le cas échéant.
- Préparer les plis (voir traits gris rentrants et traits rouges saillants au verso de cette page).
- Découper selon le trait rouge entre les deux ● du verso de cette page, puis achever le pliage.

① Puissances, racines et logarithmes

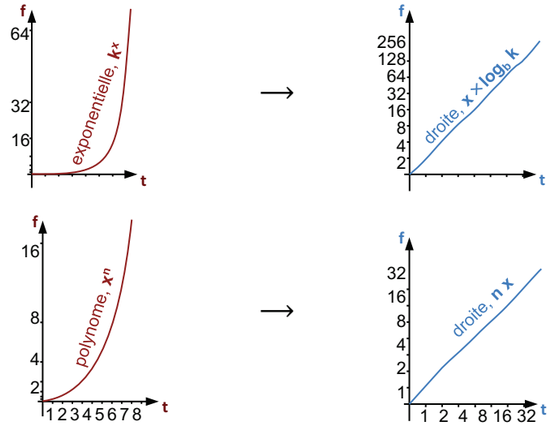
La $n^{ème}$ puissance d'un nombre x, notée x^n, est le produit de n copies de ce nombre. La racine $n^{ème}$ d'un nombre x, notée $\sqrt[n]{x}$, est un nombre y tel que $y^n = x$, soit de quoi x est le produit de n copies ; y est en général irrationnel. Par exemple, $\sqrt[12]{2} = 1.05946...$ est le rapport des fréquences de deux notes de musique séparées par 1 demi-ton ; c'est de quoi 2 (le rapport de fréquence d'un octave) est le produit de 12 copies (les 12 demi-tons de l'octave). Initialement conçues pour n à valeur entière, ces opérations ont été progressivement généralisées par continuité à toutes sortes de nombres, si bien que puissances et racines sont maintenant deux instances du même concept. On montre que $x^{a+b} = x^a \times x^b$ et $x^{a\times b} = (x^a)^b = (x^b)^a$, que $x^{-n} = 1/x^n$ et $x^{1/n} = \sqrt[n]{x}$, et que $x^{a/b}$ est un nombre y tel que $x^a = y^b$.

Le logarithme en base b d'un nombre x, noté $\log_b x$, est un nombre y tel que $b^y = x$, soit de quel nombre de copies de b x est le produit ; y est en général non-algébrique, c-à-d. pas racine d'un polynôme à coefficients rationnels. On montre que $\log_b x$ est strictement croissante, que $\log_b 1 = 0$ et $\log_b b = 1$, que $\log_b (x\times y) = (\log_b x) + (\log_b y)$, que $\log_b x = (\log_{b'} x) \times (\log_b b')$, et que $x^{\log_b y} = y^{\log_b x}$. Noter que en général $\log_b x \ll x$. Par exemple, le logarithme en base 10 d'un nombre est à peu près le nombre des chiffres de sa représentation décimale : 1,95... pour 90, 2,99... pour 990 et 5,999... pour 999000. L'erreur faite en arrondissant un logarithme a un entier voisin est bornée de la façon suivante : $n \le b^{\lfloor \log_b n \rfloor} < b\times n$ et $n/b < b^{\lfloor \log_b n \rfloor} \le n$. Par exemple, $4 \le 2^{\lfloor \log_2 4 \rfloor} < 8$, $7 \le 2^{\lfloor \log_2 7 \rfloor} = 8 < 2\times 7 = 14$, $8/2 = 4 < 2^{\lfloor \log_2 8 \rfloor} \le 8$, et $5/2 = 2{,}5 < 2^{\lfloor \log_2 5 \rfloor} = 4 \le 5$. Noter enfin que pour tout $\delta \ge 0$, $\lim_{x\to\infty} x^\delta/\log_b x = \infty$; une puissance finit toujours par dépasser un logarithme.

Le logarithme permet des changements de variable intéressants. Quand un phénomène a une dimension qui croit par multiplication d'un facteur constant à chaque pas de temps, par exemple, loi de Moore, multiplication par 2 de la capacité des disques durs tous les 2 ans, doublement du nombre de grains de riz à chaque case de l'échiquier, on dit qu'il a une croissance exponentielle. Ces phénomènes sont difficiles à représenter graphiquement car le rapport entre les plus petites valeurs et les plus grandes est énorme : par exemple, plusieurs millions sur l'histoire des capacités des disques durs, c'est-à-dire beaucoup plus que le nombre de points d'impression sur la hauteur d'une page A4.

Le changement de variable $(t, f(t)) \longrightarrow (t, \log_b f(t))$, c-à-d. afficher $\log_b f(x)$ plutôt que $f(x)$ à l'abscisse x, rend le graphe de f plus lisible et l'hypothèse de croissance exponentielle facile à vérifier si les points s'alignent sur une droite ; on appelle cela la projection lin × log.

On peut aussi convenir du changement de variable $(t, f(t)) \longrightarrow (\log_b t, \log_b f(t))$, c-à-d. afficher $\log_b f(x)$ à l'abscisse $\log_b x$ plutôt que $f(x)$ à l'abscisse x pour former la projection log × log. Son avantage est que le graphe des phénomènes de croissance polynomiale y tend vers une droite dont la pente est le degré du polynôme. C'est un moyen très simple de vérifier si un phénomène a une croissance polynomiale ou non.



N'oubliez jamais !

Un modèle est toujours imparfait, parfois utile, et c'est tout ce qu'on peut lui demander (d'après George Box 1978, voir aussi les cartes à l'échelle 1/1 de Jorge Luis Borges 1946 et Lewis Carroll 1893 qui sont parfaites et inutiles, parfaitement inutiles).

L'informatique ne travaille que sur la représentation des choses ; il lui faut des capteurs et des actionneurs pour toucher les choses. Par contre, travailler sur des représentations de représentations de … est courant. Par exemple, 440 est une représentation d'un entier qui peut être la représentation de la mesure d'une fréquence, tout comme 1B8, et La_3 pourrait être une autre représentation de la même fréquence. Il faut toujours se demander ce qui a du sens par rapport à ce qui est modélisé.

① Calcul des propositions

Propositions atomiques (ou atomes) **:** Ce sont des formules élémentaires à qui on peut attribuer une valeur de vérité, Vrai ou Faux. On les note souvent par des lettres minuscules : ex. a, b, c.

Formules propositionnelles : Ce sont des formules, élémentaires ou non, qui sont reliées par des connecteurs logiques, généralement ∧, ∨, ¬, ou ⇒.

- conjonction, $\phi_1 \wedge \phi_2$ **:** relie deux formules pour en former une troisième qui est vraie ssi les deux premières le sont. Le connecteur ∧ est commutatif, associatif, et a pour élément neutre la valeur de vérité Vrai. Cela permet la notation de conjonction étendue comme $\bigwedge_{i\in[1,n]} \phi_i$. Si on convenait que Faux < Vrai, $\phi_1 \wedge \phi_2$ serait le minimum de ϕ_1 et ϕ_2. Si on convenait que Faux = 0 et Vrai = 1, $\phi_1 \wedge \phi_2$ serait $\phi_1 \times \phi_2$.
- disjonction, $\phi_1 \vee \phi_2$ **:** relie deux formules pour en former une troisième qui est vraie ssi au moins une des deux premières l'est. Le connecteur ∨ est commutatif, associatif, et a pour élément neutre la valeur de vérité Faux. Cela permet la notation de disjonction étendue comme $\bigvee_{i\in[1,n]} \phi_i$. Si on convenait que Faux < Vrai, $\phi_1 \vee \phi_2$ serait le maximum de ϕ_1 et ϕ_2. Si on convenait que Faux = 0 et Vrai = 1, $\phi_1 \vee \phi_2$ serait $1 - (1-\phi_1) \times (1-\phi_2)$.
- implication, $\phi_1 \Rightarrow \phi_2$ **:** relie deux formules pour en former une troisième qui est fausse ssi ϕ_1 est vraie alors que ϕ_2 est fausse. Le connecteur ⇒ n'est ni commutatif ni associatif. Si on convenait que Faux < Vrai, $\phi_1 \Rightarrow \phi_2$ serait Vrai ssi $\phi_1 \le \phi_2$.
- négation, $\neg\,\phi$ **:** constitue une formule qui est fausse ssi ϕ est vraie. Si on convenait que Faux = 0 et Vrai = 1, $\neg\,\phi$ serait $1 - \phi$.

a	b	c	(a∨b)∧c
Vrai	Vrai	Vrai	Vrai
Vrai	Vrai	Faux	Faux
Vrai	Faux	Vrai	Vrai
Vrai	Faux	Faux	Faux
Faux	Vrai	Vrai	Vrai
Faux	Vrai	Faux	Faux
Faux	Faux	Vrai	Faux
Faux	Faux	Faux	Faux

a	a∧¬a
Vrai	Faux
Faux	Faux

a	a∨¬a
Vrai	Vrai
Faux	Vrai

Satisfaisabilité : les formules du calcul des propositions peuvent être vues comme des fonctions des variables qu'elles contiennent : $\{Vrai, Faux\}^n \longrightarrow \{Vrai, Faux\}$ pour une formule qui compte n variables. On peut représenter chacune de ces fonctions par une table de vérité. Il s'agit d'un tableau comportant 2^n lignes, et n+1 colonnes. Dans leurs n premières colonnes, les lignes représentent toutes les entrées possibles (appelées valuations des variables), et dans leur n+1-ème colonne, elles représentent ce que vaut la fonction pour cette entrée. Ex. les formules (a∨b)∧c, a∨¬a, et a∧¬a peuvent être représentées par les tables de vérité ci-contre.

On peut remarquer que toutes les lignes de la table de a∨¬a ont la valeur Vrai en position de résultat ; on dit que cette formule est une tautologie, elle vaut Vrai pour toutes ses entrées. Toutes les lignes de la table de a∧¬a ont la valeur Faux ; on dit que cette formule est contradictoire ou absurde. Finalement, les lignes de la table de (a∨b)∧c ont soit Vrai soit Faux ; on dit que cette formule est satisfaisable (bien sûr, les tautologies sont aussi satisfaisables). Il est souvent intéressant de savoir pour quelles entrées une formule vaut Vrai.

② Calcul des prédicats

Dans sa définition la plus naïve le calcul des prédicats est la formalisation de la logique employée tous les jours dans les activités un peu scientifiques. Elle permet d'exprimer des jugements sous forme de formules, et de décider formellement si ils sont vrais ou faux. Les formules de la logique des prédicats sont formées des connecteurs du calcul des propositions (ex. ∧, ∨, ¬, et ⇒) et des quantificateurs ∀ et ∃.

Formules quantifiées : Ce sont des formules, élémentaires ou non, dont la valeur de vérité s'évalue par rapport à un ensemble d'objets, plutôt que par rapport à un objet. Syntaxiquement, une quantification lie une variable, comme le font $\int \ldots dx$ ou $\partial \ldots / \partial x$. Sémantiquement, les quantifications sont définies comme suit :

- quantification universelle, $\forall x\,.\,\phi\,(x)$ **:** constitue une formule qui est vraie ssi ϕ est vraie de tous les éléments du domaine. Si le domaine est fini, la quantification universelle est juste une conjonction des applications de ϕ à tous les éléments du domaine.
- quantification existentielle, $\exists x\,.\,\phi\,(x)$ **:** constitue une formule qui est vraie ssi ϕ est vraie d'au moins un élément du domaine. Si le domaine est fini, la quantification existentielle est juste une disjonction des applications de ϕ à tous les éléments du domaine.

Parenthèses et priorité des opérateurs : Il est courant d'assigner des priorités d'opérateurs aux différents connecteurs et quantificateurs afin de spécifier comment se lisent les formules complexes en l'absence de parenthèses. C'est utile mais absolument pas fondamental car ces conventions dépendent trop des outils, et même quand ce n'est pas le cas, il n'est jamais très prudent de se reposer trop lourdement sur les priorités des opérateurs quand le coût de quelques parenthèses est si faible devant le coût d'une grosse erreur. Ce genre d'expertise est à réserver aux experts. Il vaut mieux ne pas être avare de parenthèses, ex. utiliser les parenthèses rondes ((…)) pour structurer les connecteurs, et les parenthèses carrées (ou crochets, […]) pour les quantificateurs.

③ Idiomes logiques

En pratique, on n'utilise pas n'importe quelle formule de la logique des prédicats. On a tendance à utiliser des imbrications de quantificateurs et de connecteurs qui sont idiomatiques et qu'il convient de reconnaître et interpréter correctement au premier coup d'œil.

Quantifications dans un domaine : Très souvent, on écrit des formules comme $\forall x \in E\,.\,\phi(x)$ ou $\forall x \text{ tq } \psi(x)\,.\,\phi(x)$. Les $x \in E$ et tq $\psi(x)$ sont une façon de dire dans quel domaine doit être évaluée la quantification. Ces formules doivent être lues $\forall x\,.\,[x \in E \Longrightarrow \phi(x)]$ et $\forall x\,.\,[\psi(x) \Longrightarrow \phi(x)]$. Une conséquence directe est qu'une quantification universelle sur un domaine vide est trivialement vraie. Cela paraît une situation étrange, mais c'est banal en algorithmique, spécialement quand on considère les cas d'initialisation : ex. Tous les utilisateurs sont … lorsqu'il n'y a pas d'utilisateurs. De la même façon, on écrit des formules comme $\exists x \in E\,.\,\phi(x)$ ou $\exists x \text{ tq } \psi(x)\,.\,\phi(x)$. Ces formules doivent être lues $\exists x\,.\,[x \in E \wedge \phi(x)]$ et $\exists x\,.\,[\psi(x) \wedge \phi(x)]$. On voit alors qu'une quantification existentielle sur un domaine vide est trivialement fausse.

Cascades de quantifications : On imbrique souvent les quantifications. Certaines imbrications doivent faire réfléchir, et d'autres moins.

- $\exists x\,.\,\exists y\,.\,\phi\,(x, y)$ est équivalent à $\exists y\,.\,\exists x\,.\,\phi\,(x, y)$ qu'on note souvent $\exists x, y\,.\,\phi\,(x, y)$; l'ordre des quantificateurs ne compte pas.
- $\forall x\,.\,\forall y\,.\,\phi\,(x, y)$ est équivalent à $\forall y\,.\,\forall x\,.\,\phi\,(x, y)$ qu'on note souvent $\forall x, y\,.\,\phi\,(x, y)$; l'ordre des quantificateurs ne compte pas non plus.
- $\forall x\,.\,\exists y\,.\,\phi\,(x, y)$ exprime que pour chaque x il y a un y, qui peut dépendre de x, qui a la propriété désirée. Ex. dans $\forall x\,.\,\exists y\,.\,[x \times y = 0]$, le même y convient pour tous les x, mais dans $\forall x\,.\,\exists y\,.\,[x \times y = 1]$, à tout x correspond un y différent. Un y existentiel est donc implicitement une fonction de tous les x universels qui le précèdent.
- $\exists x\,.\,\forall y\,.\,\phi\,(x, y)$ exprime qu'un x unique a la propriété désirée pour tous les y. Ex. $\exists x\,.\,\forall y\,.\,[x \times y = 0]$ est vrai, mais pas $\exists x\,.\,\forall y\,.\,[x \times y = 1]$. On voit donc que lorsqu'il y a une alternance de quantificateurs l'ordre compte ! Ces alternances sont l'essence de nombreuses définitions importantes en mathématiques (ex. continuité, convergence) et en informatique (ex. ordres de grandeurs au verso de cette page).

② Sommations

La somme et le produit étant des opérations associatives, commutatives et ayant chacune un élément neutre, on peut se permettre des notations de sommes et produits itérés : $\sum_{i\in[bi\,bs]} t_i$ et $\prod_{i\in[bi\,bs]} f_i$ où les t_i et f_i sont des termes et facteurs dépendants de i. Si l'intervalle des indices est vide (ex. [0 0[) la valeur conventionnelle de l'expression est l'élément neutre de l'opération, ex. 0 pour la somme et 1 pour le produit, mais aussi Faux pour la disjonction (⋁), qui est une sorte de somme, et Vrai pour la conjonction (⋀), qui est une sorte de produit.

Certaines sommations doivent être connues ; il est même bon de savoir les retrouver :

- $\sum_{i\in[1\,n[} i$: somme des n premiers entiers (c-à-d. de 1 à n). C'est égal à $n \times (n+1) / 2$. Ex. $\sum_{i\in[1\,17[} i = 153$.
- $\sum_{i\in[0\,n[} x^i$: somme des n premières puissances de x (c-à-d. de x^0 à x^{n-1}). C'est égal à $(x^n - 1) / (x - 1)$. Ex. $\sum_{i\in[0\,n[} 2^i = 2^n - 1$.

Il est bon aussi de se rappeler quelques transformations élémentaires :

- $\sum_{i\in[0\,n[} (k \times t_i) = k \times \sum_{i\in[0\,n[} t_i$ et $\sum_{i\in[0\,n[} k = k \times n$, si k ne dépend pas de i.
- $\sum_{i\in[0\,n[} (s_i + t_i) = \sum_{i\in[0\,n[} s_i + \sum_{i\in[0\,n[} t_i$. Attention, rien de tel pour $\sum_{i\in[0\,n[} (s_i \times t_i)$!