MINESnancy
ARTEM

Algorithmes et Complexité

Responsable du cours : Xavier Goaoc

Mode d'emploi

Calculer, c'est faire traiter de l'information par un système physique. Cette définition soulève plusieurs questions, parmi lesquelles :

Qu'est-il possible de calculer ? Comment ? Avec quelle efficacité ?

L'objectif de ce cours d'informatique théorique est de vous permettre d'appréhender ces questions. C'est essentiel pour comprendre les possibilités et les limites de l'informatique, pour saisir la démarche de ce champ disciplinaire, ses problématiques et développements actuels (sécurité des systèmes, consommation énergétique, apprentissage automatique, informatique quantique, ...), et les changements profonds qu'il connaîtra dans le futur.

Ce cours comporte deux parties : une partie d'**algorithmique**, qui étudie l'organisation efficace du calcul, et une partie de **théorie de la complexité**, qui étudie les calculs qu'il est impossible d'organiser efficacement. Ce cours touche aussi, plus ponctuellement, à l'*architecture des ordinateurs*, aux *mathématiques discrètes* et à la *théorie de la calculabilité*.

Ce cours est composé de six **séances méthodologiques** (2, 3, 4, 6, 7 et 8), de deux **cas d'étude** (1 et 5), et d'une **séance d'ouverture** (9). À chaque séance correspond un chapitre de ce polycopié. Le cours et le polycopié sont complémentaires : le cours insiste sur l'essentiel et donne une image que l'on espère vivante du sujet, et le polycopié fournit les détails. Lire le polycopié avant le cours est recommandé (et ne présente aucun risque de divulgâchage). Les TD visent à vous aider à assimiler les méthodes et concepts introduits en cours et à fournir de nouveaux exemples. Les TD se font sur papier car réfléchir aux algorithmes et réfléchir à leur programmation sont deux activités distinctes, et il est important de décomposer les difficultés. L'usage et l'annotation du polycopié en TD sont encouragés.

Le matériel pédagogique (version électronique du polycopié, sujets de TD, corrigés, ...) sont disponibles sur **la page Arche du cours**. J'y indiquerai aussi les horaires des **permanences hebdomadaires** à mon bureau à l'École des Mines (R329) où je serai disponible pour répondre à vos questions.

Ce module est **évalué** par un examen final, sur table, de 3h. Les copies ne vous seront pas rendues mais je vous proposerai des créneaux pour les consulter. Un **devoir de maison optionnel** vous sera proposé à mi-parcours. Il sera corrigé et noté, mais la note sera uniquement indicative et **n'entrera pas en compte dans la note finale** du module.

Le seul document autorisé à l'examen est la **copie papier** du polycopié, qui peut être annotée. Aucun système électronique ne sera autorisé, pas même pour vous donner l'heure.

Remerciements. Ce polycopié a bénéficié des conseils, relectures, suggestions et explications de Xavier Bonnetain, Mathilde Bouvel, Fabienne Buffet, Véronique Cortier, Alexandre Guernut, Samuel Hornus, Emmanuel Jeandel, Florent Koechlin et Benjamin Testart. Les erreurs, coquilles et autres approximations sont bien évidemment de ma responsabilité.

Table des matières

Chapitre 0

Passerelle

Ce chapitre passe en revue quelques notions utilisées dans les séances ultérieures, que vous avez vraisemblablement déjà abordé au fil de votre scolarité, mais dont il est utile de préciser le formalisme ou l'usage.

Objectifs. Cette séance vise à vérifier que vous...

- savez modéliser une problématique générale en un problème algorithmique précis,
- maitrisez les notations asymptotiques $O()$, $\Omega()$ et $\Theta()$ pour une variable entière tendant vers l'infini,
- êtes à l'aise avec les fonctions exponentielles et logarithmiques,
- savez coder de l'information par des mots binaires et mesurer la taille du codage d'une information (nombre de bits, à $O()$ près),
- savez mener une analyse de complexité asymptotique pire cas simple.

0.1 Problématique de calcul et problème algorithmique

Dans ce cours...

- une « problématique de calcul » désigne un calcul à mener qui n'est pas formalisé, par exemple « *organiser l'appariement entre élèves 1A de Mines Nancy et les offres de stage opérateur en fonction des préférences exprimées* ». Ce calcul n'est pas formalisé en ce qu'on ne sait ni sous quelle forme sont données les préférences, ni de quelle manière on souhaite les prendre en compte.
- un « problème algorithmique » désigne la formalisation de ce que l'on souhaite calculer, c'est à dire la définition des entrées, des sorties, et des propriétés que la sortie doit satisfaire relativement à l'entrée.

Le premier pas pour résoudre une problématique de calcul est de la formaliser en un problème algorithmique. Il peut exister plusieurs problèmes algorithmiques formalisant une même problématique de calcul, et le choix entre eux est une activité de **modélisation** : il s'agit de les comparer selon divers critères (simplicité, précision, disponibilité d'une solution, ...) et de choisir le compromis qui nous semble le plus adapté.

Ce cours se concentre sur des méthodologies d'étude des problèmes algorithmiques et n'évoque que peu les problématiques de calcul.

Vous aurez l'occasion de pratiquer la modélisation dans le cours de recherche opérationnelle.

Définition de problème algorithmique. Pour tout ensemble X, on note 2^X l'ensemble des sous-ensembles de X, $\emptyset$ et X inclus.

Dans ce cours, un **problème algorithmique** est une fonction $P : E \to 2^S$ où E et S sont des ensembles dénombrables.

On appelle E **l'ensemble des entrées** du problème. Un élément $e \in E$ est une **entrée** (ou une **instance**) du problème. On appelle S **l'ensemble des sorties** du problème. Pour chaque entrée $e \in E$, $P(e)$ désigne l'ensemble des sorties acceptées comme réponse au problème sur cette entrée e.

Présentation. Dans ce cours nous présentons les problèmes algorithmiques comme suit :

MINIMUM D'UNE LISTE D'ENTIERS
Entrée : Une liste finie d'entiers non vide.
Sortie : La valeur du plus petit entier de la liste.

Dans cet exemple, E est l'ensemble des listes finies d'entiers, S est l'ensemble des entiers, et P envoit toute liste finie d'entiers $[i_1, i_2, \ldots, i_k]$ sur le singleton $\{\min\{i_1, i_2, \ldots, i_k\}\}$.

Dans MINIMUM D'UNE LISTE D'ENTIERS, pour toute entrée il existe une unique sortie acceptée. Ce n'est pas nécessairement le cas, comme l'illustre le problème suivant :

POSITION DU MINIMUM D'UNE LISTE D'ENTIERS
Entrée : Une liste finie d'entiers non vide.
Sortie : Un indice du plus petit entier de la liste.

Dans cet exemple, E est à nouveau l'ensemble des listes finies d'entiers, S est à nouveau l'ensemble des entiers, et P envoit toute liste finie d'entiers $[i_1, i_2, \ldots, i_k]$ sur l'ensemble $\{j : i_j = \min\{i_1, i_2, \ldots, i_k\}\}$ des indices où la liste atteint son minimum.

Par convention, tout paramètre apparaissant sans quantificateur dans l'entrée est libre de prendre toutes les valeurs pour lequel l'énoncé a un sens. Par exemple :

RÉSOLUTION D'ÉQUATION DIOPHANTIENNE
Entrée : Un polynôme P à coefficients entiers et $n < \infty$ variables.
Sortie : Vrai ou faux, $P(x_1, x_2, \ldots, x_n) = 0$ admet une solution entière.

Dans RÉSOLUTION D'ÉQUATION DIOPHANTIENNE, n peut prendre toute valeur entière supérieure ou égale à 1.

0.2 Algorithme et présentation d'un algorithme

Une solution à un problème algorithmique P prend la forme d'un *algorithme* qui, partant de n'importe quelle entrée e du problème, produit une sortie valide pour cette entrée au sens où elle appartient à $P(e)$. Formellement, un *algorithme* est une séquence finie d'*instructions élémentaires*, une notion dont la définition s'avère délicate ; nous la traiterons au Chapitre 2, lors de la définition précise du *modèle de calcul*.

Un algorithme **n'est pas** un code. On donne en Annexe A des conventions de présentation d'un algorithme dans ce cours.

0.3 Fini, borné, constant, ...

Lors de la description ou l'analyse d'un algorithme, les grandeurs qui interviennent dépendent généralement de l'entrée : compteur de boucle `i`, taille `t` d'un tableau intermédiaire, valeur `m` du maximum d'un tableau, ... Il convient de distinguer les termes suivants :

- On dit qu'une grandeur est **finie** si elle est finie quelque soit l'entrée.

- On dit qu'une grandeur est **bornée** si elle est majorée par une constante *indépendante de l'entrée.*
- On dit qu'une grandeur est **constante** si elle est *indépendante de l'entrée.*

Ainsi, si on résout MINIMUM D'UNE LISTE D'ENTIERS en parcourant le tableau par une boucle `for i=1..n`, la variable `i` est finie mais n'est ni bornée, ni constante.

0.4 Notation asymptotique

L'analyse de la complexité asymptotique d'un algorithme amène à manipuler des notations asymptotiques qui sont usuelles, mais qu'il est peut-être utile de rappeler. On s'intéresse uniquement au comportement asymptotique de fonctions (ici f et g) de $\mathbb{N}^*$ dans $\mathbb{N}^*$, lorsque la variable tend vers $+\infty$.

$f = O(g)$ s'il existe n_0 et M tels que pour tous entiers $n \geq n_0$ on a $f(n) \leq Mg(n)$.

Une fonction f est **bornée** si $f = O(1)$, au plus **linéaire** si $f = O(n)$, au plus **quadratique** si $f = O(n^2)$, au plus **cubique** si $f = O(n^3)$ ... Une fonction f est **polynomiale** si $f = O(n^t)$ avec t borné. On dit qu'une fonction f est au plus **logarithmique** si $f = O(\log n)$. Une fonction f est au plus **polylogarithmique** si il existe une constante c telle que $f = O(\log^c(n))$. Un abus de langage courant consiste à oublier le terme « au plus ». [1]

$f = o(g)$ si $\lim_{n\to\infty} f(n)/g(n) = 0$.

Une fonction f est **sous-linéaire** si $f = o(n)$, **sous-quadratique** si $f = o(n^2)$, **sous-cubique** si $f = o(n^3)$ ...Si $f = o(g)$ alors $f = O(g)$ mais la réciproque n'est pas vraie en général (prendre $f = g$).

$f = \Omega(g)$ si et seulement s'il existe un réel $M > 0$ tel que pour tout $n \geq 0$ il existe $n' \geq n$ tel que $f(n') \geq Mg(n')$.

C'est précisément la négation de « $\lim_{n\to\infty} f(n)/g(n) = 0$ ». Autrement dit, $f = \Omega(g)$ si et seulement si f n'est pas $o(g)$. Dans la littérature, cette notation est aussi parfois utilisée pour signifier que $g = O(f)$, c'est-à-dire qu'il existe des réels n_0 et M tels que $\forall n \geq n_0, f(n) \geq Mg(n)$. La définition que l'on rappelle ici, elle aussi largement utilisée dans la littérature, est moins exigeante.

$f = \Theta(g)$ si et seulement si $f = O(g)$ *et* $f = \Omega(g)$.

Notons qu'en particulier, s'il existe des constantes n_0, $m > 0$ et $M > 0$ telles que pour tout $n \geq n_0$ on a $mg(n) \leq f(n) \leq Mg(n)$, alors $f = \Theta(g)$. Soulignons que les constantes m et M peuvent être distinctes. Quand elles sont égales, on dit que f et g sont **équivalentes** (en $n \to \infty$).

0.5 Exponentiation et logarithme

Soit a un réel strictement positif. La **fonction exponentielle de base** a est la fonction

$$\exp_a : \left\{ \begin{array}{ccc} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & a^x \end{array} \right.$$

Elle peut se définir comme l'unique fonction continue $f : \mathbb{R} \to \mathbb{R}$ qui satisfait $f(1) = a$ et $f(x + y) = f(x)f(y)$. Cette propriété, importante, caractérise ce que l'on appelle une « variation exponentielle » (croissance exponentielle si $|a| > 1$, décroissance exponentielle si $|a| < 1$). Voici quelques exemples d'apparition de fonctions exponentielles dans les sujets de ce cours :

1. En particulier, il est ainsi raisonnable de dire qu'une fonction linéaire est quadratique...

- Un entier dont la représentation binaire est de longueur n est compris entre 2^{n-1} et $2^n - 1$. (Cf ci-après.)
- Un arbre de hauteur h dont chaque nœud est de degré au plus k a au plus k^h feuilles. (Cf Chapitre 4)
- Si on souhaite résoudre une instance du problème SOUS-SOMMES par force brute, cela nous amène à examiner tous les sous-ensemble de $\{1, 2, \ldots, n\}$, c'est à dire 2^n possibilités. (Cf TD 3 et Chapitre 8)

Ces exemples illustrent le fait que les exponentielles que l'on va croiser seront souvent de base $a = 2$, mais pas toujours,

Le **logarithme de base** a $\log_a$ peut se définir comme la fonction inverse de $\exp_a$:

$$y = \log_a(x) \quad \Leftrightarrow \quad x = \exp_a(y) = a^y.$$

Voici quelques exemples d'apparition de fonctions logarithmiques dans les sujets de ce cours :

- La taille de la représentation binaire d'un entier x est $\lceil \log_2(1 + x) \rceil$ (Cf ci-après.)
- Un arbre binaire ayant n feuilles est de hauteur au moins $\log_2 n$. (Cf Chapitre 6)
- Supposons que l'on parte d'un tableau de taille n et qu'on le coupe, récursivement, en deux parties comme le fait le tri fusion. Le nombre de coupes qu'il faut faire pour aboutir à un tableau de taille bornée est $\Theta(\log n)$.

Fixons $a > 0$ et $b > 0$. Puisque $\exp_a$ et $\log_a$ sont inverse l'une de l'autre, on a que $b = a^{\log_a b}$. Ainsi,

$$\forall x \in \mathbb{R}, \quad a^x = \left(b^{\log_b a}\right)^x = b^{(\log_b a)x} = b^{x \log_b a}.$$

Cette identité permet d'opérer des « conversions » entre fonctions exponentielles. De même, en utilisant que $\log_a(x^y) = y \log_a x$ et en appliquant le logarithme en base a à l'identité $x = b^{\log_b x}$, on obtient :

$$\forall x \in \mathbb{R}, \quad \log_a x = \frac{\log_b x}{\log_b a}.$$

En particulier, pour $n \to \infty$ et a, b constantes, on a $\log_a n = \Theta(\log_b n)$.

> Lorsqu'un logarithme de base **constante** apparaît dans une notation asymptotique $O(), \Omega()$, ou $\Theta()$, on peut omettre la base.

0.6 Numérisation de l'information

Une caractéristique fondamentale d'un système de calcul est la manière dont il représente l'information. La majorité des systèmes actuels réalisent cela au moyen de composants physiques prenant deux états [2] ; selon que le composant est dans l'un ou l'autre état, on considère qu'il « représente » le chiffre 0 ou le chiffre 1. En associant plusieurs composants, on peut stocker un **mot binaire**, c'est-à-dire une suite *finie* d'éléments de $\{0, 1\}$. Cette restriction est fondamentale :

> Toute information manipulée par un ordinateur (classique) est représentée par un mot binaire.

Chaque élément d'un mot binaire est appelé un **bit** (contraction de *binary digit*). [3] La **taille** d'un mot binaire w est son nombre de bits, et est généralement notée $|w|$. Ainsi, $\{0, 1\}^n$ est l'ensemble des mots binaires de longueur n et on note $\{\mathbf{0, 1}\}^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \{0, 1\}^n$ l'ensemble des mots binaires. Lorsque l'on a besoin d'expliciter les bits d'un mot binaire w, on les note généralement [4] $w = w_1 w_2 \ldots w_{|w|}$.

2. Les systèmes de calcul quantiques constituent une exception notable. Nous y reviendrons au Chapitre 9.
3. Ainsi, dans le mot binaire 01101 le premier bit est 0 et deuxième est 1.
4. Lorsque w est la représentation binaire d'un entier on utilise souvent une convention différente, cf ci-après.

0.6.1 Convention d'encodage et représentation binaire d'un entier

La correspondance entre un ensemble d'objets (nombres, textes, images, ...) et les mots binaires est appelée **convention d'encodage** de ces objets. Formellement, une convention d'encodage pour un ensemble X est une *injection* de X dans $\{0,1\}^*$.

Un exemple important d'encodage est la **représentation binaire d'un entier naturel**. Tout entier naturel $x \in \mathbb{N}$ peut s'écrire $x = \sum_{i=0}^{k-1} x_i 2^i$ avec $x_i \in \{0,1\}$ et cette écriture est unique si l'on impose $x_{k-1} \neq 0$ ou, pour $x = 0$, que $k = 1$. Cette écriture définit une convention d'encodage "naturelle" consistant à encoder tout entier naturel a par le mot binaire $x_{k-1}x_{k-2}\dots x_0$. Ce mot binaire est appelé la **représentation binaire** de x. Par exemple, la représentation binaire de 6 est 110 et celle de 17 est 10001.

Soulignons que la **taille** de la représentation binaire d'un entier positif x est $\lceil \log_2(1+x) \rceil$. En effet, les entiers positifs ayant une représentation binaire de taille n sont les entiers compris entre 2^{n-1} (représenté par $100\dots0$) et $2^n - 1$ (représenté par $11\dots1$). Ainsi, la représentation binaire de x est de taille n si et seulement si

$$2^{n-1} \leq x < 2^n \quad \Leftrightarrow \quad 2^{n-1} < 1 + x \leq 2^n \quad \Leftrightarrow \quad n - 1 < \log_2(1+x) \leq n \quad \Leftrightarrow \quad n = \lceil \log_2(1+x) \rceil.$$

0.6.2 Taille d'encodage d'une entrée

Il est courant que l'entrée d'un algorithme soit une séquence de nombres, comme par exemple les tableaux d'entiers ou de préférences considérés au chapitre précédent. On peut naturellement encoder chaque nombre par son écriture binaire et considérer la *séquence de mots binaires* obtenue comme un encodage de cette entrée. La **taille d'encodage d'une entrée** est alors définie comme la somme des tailles des mots binaire de cette séquence.

Définir la taille d'une entrée suppose de fixer une convention d'encodage, mais le principe suivant rend souvent cette convention d'encodage assez secondaire :

> Dans ce cours, on ne mesure la taille d'encodage d'une donnée qu'à un $O()$ près.

Ainsi, la taille d'un entier positif $x \in \mathbb{N}$ est $O(\log_2 x) = O(\log x)$. La taille d'un graphe à n sommets représenté par sa matrice d'adjacence est $O(n^2)$. La taille d'un tableau de préférence $n \times n$ fourni comme entrée à l'algorithme de Gale-Shapley est $O(n^2 \log n)$.

0.7 Complexité asymptotique pire-cas

Dans ce cours, on mesure l'efficacité d'un algorithme au moyen de sa *complexité asymptotique pire cas*. Rappelons que pour un algorithme $\mathcal{A}$ d'ensemble d'entrées E, elle est définie comme l'ordre de grandeur asymptotique, pour $n \to \infty$, du nombre maximum d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter une entrée de taille n. Cette définition appelle plusieurs remarques :

a. Notons $C(e)$ le nombre d'instructions élémentaires utilisées par $\mathcal{A}$ pour traiter l'entrée e. Notons $C(n) \stackrel{\text{def}}{=} \max\{C(e) \mid e \in E \text{ et } e \text{ est de taille } n\}$. La complexité asymptotique pire-cas de $\mathcal{A}$ est l'ordre de grandeur de $C(n)$.

b. On parle de « pire-cas » car on s'intéresse, à n donné, au coût de traitement *maximum* d'une entrée de taille n. C'est, au choix, pessimiste ou prudent.

c. Cette définition suppose que l'on ait choisi une manière de mesurer la taille de chaque entrée $e \in E$. Cette mesure peut sembler aller de soi et être souvent laissée implicite. Par exemple si l'entrée est un tableau d'entiers il est courant de définir sa taille comme le nombre d'entrées du tableau. On a néanmoins souvent le choix. [5]

d. Cette définition repose sur la notion d'instruction élémentaire. On introduira, au Chapitre 2, deux modèles de calcul ayant chacun sa notion d'instruction élémentaire ; chacun a donc sa notion d'algorithme, et de complexité asymptotique pire-cas.

5. Par exemple, pour des considérations de théorie de la complexité, on définira la taille d'un tableau d'entier comme la somme des tailles d'encodages de chacune de ses entrées.

e. Dire que « la complexité asymptotique pire-cas de l'algorithme $\mathcal{A}$ est $O(f(n))$ » n'en donne qu'une *majoration*. En particulier cet énoncé est correct même si $f(n)$ n'a pas le même ordre de grandeur asymptotique de la complexité asymptotique pire-cas de $\mathcal{A}$.[6]

f. De même, dire que « la complexité asymptotique pire-cas de l'algorithme $\mathcal{A}$ est $\Omega(f(n))$ » n'en donne qu'une *minoration*.

g. On ne connait la complexité asymptotique pire-cas d'un algorithme $\mathcal{A}$ que si on peut la majorer et la minorer par une même fonction.[7]

6. Par exemple, il est correct de dire que « le tri d'un tableau de taille n par tri-fusion est de complexité asymptotique pire-cas $O(n^2)$ ».

7. Autrement dit, établir que c'est un $O(f(n))$ et un $\Theta(f(n))$. Attention, cela ne prouve pas que la complexité est égale à $f()$, mais simplement que pour n assez grand elle est encadrée par des multiples de $f()$.

Chapitre 1

Cas d'étude : problèmes d'affectation et algorithme de Gale-Shapley

Cette première séance situe l'algorithmique et ses enjeux au moyen d'un cas d'étude. Plusieurs des concepts manipulés dans ce chapître, à commencer par la notion d'*algorithme*, seront formalisés au chapitre suivant.

Dans cette séance, nous partons d'une problématique (apparier des élèves et des places en colocation) et la modélisons par un **problème algorithmique** (le calcul d'un mariage stable). Nous présentons ensuite un algorithme qui résout ce problème (l'*acceptation retardée*, de Gale et Shapley) puis établissons certaines de ses propriétés (terminaison, efficacité et optimalité pour les demandeurs) avant d'examiner son adaptabilité à des variations du problème algorithmique. Nous concluons par quelques considérations sur le déploiement pratique de cet algorithme (au travers notamment du système Parcoursup).

Cette séance a pour objectif que vous sachiez

- modéliser une problèmatique par un problème algorithmique,
- présenter un algorithme, textuellement ou par pseudocode,
- adapter une analyse d'un algorithme.

1.1 Modélisation d'une problématique en problème algorithmique

1.1.1 Problématique d'affectation de ressources

Les problématiques d'affectation de ressources relèvent du domaine de la recherche opérationnelle. La version étudiée ici met en jeu deux ensembles d'agents qu'il s'agit d'apparier. Cela recouvre des situations aussi diverses que l'affectation de nouveaux internes en médecine à des postes hospitaliers vacants, de bacheliers à des formations d'enseignement supérieur, de clients affamés à des tables de restaurant, de dons d'organes à des receveurs (patients en attente de greffe), *etc.* Les agents peuvent être de deux types distincts (nouveaux internes en médecine et postes hospitaliers vacants) ou d'un seul type (par exemple des paires donneur/receveur lors de dons d'organes croisés [Wika]).

Contraintes de compatibilité et de préférence. L'appariement recherché peut être soumis à des contraintes de « compatibilité ». Par exemple, dans le cas de dons d'organes, la compatibilité des systèmes immunitaires du donneur et du receveur est nécessaire pour éviter les rejets. En pratique, il peut ne pas être évident qu'il *existe* un appariement satisfaisant toutes les contraintes de compatibilité ; le *calcul* d'un tel appariement peut être encore moins évident.

L'appariement recherché peut aussi être soumis à des contraintes de « préférence ». Par exemple, si on apparie des groupes de clients affamés à des tables de restaurant, chaque groupe peut avoir des préférences en matière de table (loin de la porte, en terrasse, à l'ombre, ...). Ces préférences peuvent émaner d'un ou des deux types d'agents. Elles peuvent être spécifiques à chaque agent, ou

être les mêmes pour l'ensemble des agents d'un type. Ces différents types de contraintes peuvent se combiner. [1]

Acceptabilité et instabilité. On peut envisager la résolution d'un problème d'affectation comme un effort pour optimiser *globalement* un ensemble d'arbitrages entre les préférences des agents concernés. Pour qu'une solution proposée soit pertinente, il est nécessaire qu'elle soit acceptable pour les agents. Nous allons ici nous concentrer sur une des sources d'inacceptabilité, appelée *instabilité*.

Supposons que l'on ait à affecter trois élèves (Alice, Bob et Charlie) à trois services (scolarité, communication et relations entreprises). Supposons que l'on affecte Alice à la scolarité et Bob à la communication. S'il s'avère qu'Alice préfère le service communication au service scolarité *et* que le service communication préfère Alice à Bob, deux des agents (Alice et le service communication) ont intérêt à refuser l'affectation globale proposée et à s'entendre directement. Un tel comportement déstabilise la solution que l'on a élaborée (puisque maintenant, Bob et la scolarité se trouvent sans affectation) et mine l'optimisation globale des arbitrages que l'on souhaitait mettre en place.

On va modéliser cette problématique par le calcul d'une affectation entre agents de deux types, *sans* contrainte de compatibilité, mais *avec* des préférences individuelles pour *chaque* agent. Cette affectation devra, en outre, ne présenter aucune instabilité telle qu'illustrée ci-dessus.

1.1.2 Le problème algorithmique des mariages stables

La première étape pour résoudre une problématique comme l'affectation sans instabilité décrite en Section 1.1 consiste à la formaliser en un *problème algorithmique*, c'est-à-dire à expliciter le type de données en entrée du calcul, le type de résultat en sortie du calcul, et les propriétés devant lier entrées et sorties.

On modélise les ensembles d'agents à apparier par deux ensembles finis disjoints A et B. On suppose pour l'instant que les ensembles A et B sont de même taille $|A| = |B| = n$. Un **mariage de A et B** est un sous-ensemble $M \subset A \times B$ tel que tout élément de $A \cup B$ soit dans exactement un couple de M. (On parle parfois de *couplage parfait*, ou *perfect matching* en anglais.)

On modélise les préférences des agents par un **système de préférences** $\prec$, défini comme la donnée :

- pour chaque élément $a \in A$ d'un ordre total $\prec_a$ sur B, où $b_1 \prec_a b_2$ si a préfère b_1 à b_2,
- pour chaque élément $b \in B$ d'un ordre total $\prec_b$ sur A, où $a_1 \prec_b a_2$ si b préfère a_1 à a_2.

Une **instabilité** pour un mariage M de A et B relativement à un système de préférences $\prec$ est un couple d'éléments (a, b) qui satisfait trois conditions :

(i) a et b ne sont pas appariés ; autrement dit, $(a, b) \notin M$.

(ii) a préfère b à son appariement [2] ; autrement dit, M contient un couple (a, b') avec $b \prec_a b'$.

(iii) b préfère a à son appariement ; autrement dit, M contient un couple (a', b) avec $a \prec_b a'$.

Un mariage $M \subset A \times B$ est **stable** pour un système de préférences $\prec$ s'il n'existe pas d'instabilité pour ce mariage relativement à ce système.

Voici enfin le problème algorithmique qui nous intéresse :

MARIAGE STABLE

Entrée : Un système de préférences entre deux ensembles A et B de même taille.

Sortie : Un mariage stable entre A et B.

A priori, il n'est pas évident que ce problème admet une solution pour n'importe quelle entrée. Il s'avère que c'est le cas :

1. Par exemple, dans le déroulement des études de médecine il était d'usage que les élèves reçu·es à l'issu de la 1ère année réalisent un stage infirmier ; chaque élève formulait alors son ordre de préférence sur les stages proposés, et tous les stages ordonnaient les élèves par ordre de classement de fin de 1ère année.

2. Soulignons que les préférences des appariements de a et b (c'est-à-dire b' et a') n'entrent pas en ligne de compte pour déterminer si (a, b) est une instabilité.

Théorème 1.1. *Si A et B sont disjoints et de même taille, alors pour tout système de préférences entre A et B il existe un mariage stable.*

Nous allons prouver ce théorème en donnant un algorithme de construction d'un tel mariage stable, et en prouvant que cet algorithme termine et est correct. (C'est une preuve « constructive » en un sens assez fort du terme.)

Présentation : tableaux de préférence

Il peut être pratique de présenter un système de préférences $\prec$ sous la forme d'un **tableau de préférences** sur le modèle suivant :

	b_1	b_2	b_3
a_1	$1, 3$	$2, 2$	$3, 1$
a_2	$3, 1$	$1, 3$	$2, 2$
a_3	$2, 2$	$3, 1$	$1, 3$

Dans un tel tableau, les lignes sont indexées par les éléments d'un ensemble (ici A) et les colonnes par les éléments de l'autre ensemble (ici B). L'entrée à la ligne $a \in A$ et colonne $b \in B$ est un couple i, j où i est le rang de b dans $\prec_a$, et j le rang de a dans $\prec_b$. Ainsi, dans les préférences représentées par le tableau ci-dessus, b_1 est le préféré de a_1 (il a rang 1) mais la réciproque n'est pas vraie (a_1 est le 3ème, et donc dernier, choix de b_1).

1.2 L'algorithme d'acceptation retardée (dit « de Gale-Shapley »)

L'algorithme que nous présentons maintenant est attribué[3] au système d'affectation des internes dans le système hospitalier de Boston, et a été analysé par Gale et Shapley. C'est un algorithme dit *glouton* : il détermine une solution globale au problème posé au travers d'une séquence d'optimisations locales. Comme de nombreux algorithmes gloutons, l'algorithme de Gale-Shapley a une description très simple, et l'essentiel de la difficulté réside dans sa preuve de correction et l'analyse de son efficacité. Comprendre ces preuves est indispensable pour pouvoir adapter l'algorithme à des variations du problème.

Comme discuté en Annexe A, on présente les algorithmes dans ce cours par une combinaison de pseudocode et de présentation textuelle. Dans leur article original [GS62], Gale et Shapley présentent leur algorithme textuellement, en mettant en scène son application à des demandes en mariage. Nous gardons ce principe de présentation et l'essentiel du texte original, dont on va voir qu'il est très efficace pour faire comprendre le principe de l'algorithme. Nous adaptons cependant le scénario au public et à l'époque en mettant ici en scène des élèves postulant à des colocations. Nous avons donc deux ensembles de même taille : les élèves (« students ») et les colocations (« houses »). Chaque élève cherche une place en colocation et chaque colocation offrant *exactement une* place. Chaque élève a connaissance de toutes les colocations et les a classées par ordre de préférence. Chaque colocation a connaissance de tou·te·s les élèves et les a classé·es par ordre de préférence.

Voici l'algorithme d'acceptation retardée (presque) tel que présenté par Gale et Shapley :

a. *To start, let each student propose to his or her favorite house. Each house that receives more than one proposal rejects all but its favorite from among those who have proposed to it. However, it does not accept him or her yet, but keeps him or her on a string to allow for the possibility that someone better may come along later.*

b. *We are now ready for the second stage. Those students who were rejected now propose to their second choices. Each house receiving proposals chooses its favorite from the group consisting of the new proposers and the student on its string, if any. It rejects all the rest and again keeps the favorite in suspense.*

3. Ceci est un exemple du principe de Stigler : une notion est rarement attribuée à ses premiers inventeurs (y compris le principe de Stigler).

c. *We proceed in the same manner. Those who are rejected at the second stage propose to their next choices, and the houses again reject all but the best proposal they have had so far.*

d. *As soon as the last house gets a proposal, each house accepts the student on its string and the algorithm is declared over.*

1.3 Quelques propriétés de l'algorithme d'acceptation retardée

La présentation de Gale et Shapley peut sembler laisser beaucoup d'implicite. On peut par exemple se demander ce que l'on doit faire si un·e élève a *toutes* ses demandes refusées. L'algorithme ne le précise pas car... cela est impossible. Pour bien comprendre l'algorithme, il convient d'étudier son fonctionnement sur une entrée générique à n élèves et n colocations.

Terminaison

Un algorithme **termine** si pour toute donnée d'entrée *finie*, l'algorithme s'arrête après un nombre *fini* de pas de calcul. [4] Concernant la terminaison de leur algorithme, Gale et Shapley écrivent :

> *Eventually (in fact, in at most n^2-2n+2 stages) every house will have received a proposal, for as long as any house has not been proposed to there will be rejections and new proposals, but since no student can propose to the same house more than once, every house is sure to get a proposal in due time.*

Détaillons cela...

Appellons *phase* de l'algorithme une séquence consistant en l'émission d'une (nouvelle) demande par les élèves rejeté·es et leur examen par les colocations et les rejets éventuels. Cela correspond aux *stages* de Gale et Shapley. Ainsi, le paragraphe (a) décrit la 1ère phase et le (b) la 2ème. On dit qu'un·e élève est *libre* s'il ou elle n'est pas en attente dans une colocation. De même, on dit qu'une colocation est libre si elle n'a mis aucun·e élève en attente. Notons quatre propriétés élémentaires :

(P1) Au début d'une phase, le nombre d'élèves libres égale le nombre de colocations libres.

(P2) Dès qu'une colocation a reçu au moins une demande, elle n'est plus jamais libre.

(P3) Dès lors qu'un·e élève a adressé sa dernière demande, aucune colocation n'est libre.

(P4) L'algorithme s'arrête lorsqu'aucune colocation n'est libre.

En effet, (P1) découle du fait qu'il y a autant d'élèves que de colocations, que chaque élève est en attente dans au plus une colocation, et chaque colocation met en attente au plus un·e élève. La propriété (P2) découle du fait qu'une colocation ne rejette un·e élève en attente que pour la remplacer par un·e autre. Quant à (P3), elle découle de (P2) et du fait que si un·e élève a adressé toutes ses demandes, alors chaque colocation a reçu au moins une demande (celle de cet·te élève). La propriété (P4) exprime la condition d'arrêt de l'étape (d) de l'algorithme.

Proposition 1.2. *L'algorithme d'acceptation retardée termine.*

Démonstration. Notons d_i le nombre cumulé de demandes émises au cours des phases 1, 2, ..., i. Supposons que l'algorithme n'ait pas terminé au cours des T premières phases. D'une part, d_i égale la somme des nombres de demandes adressées par chaque élève. Les propriétés (P3) et (P4) impliquent qu'aucun·e élève n'a adressé toutes ses demandes au cours des T premières phases ; on a donc $d_T < n(n-1)$. D'autre part, $d_i - d_{i-1}$ égale le nombre de demandes adressées à la $i$ème phase, ce qui égale le nombre d'élèves libres et donc de colocations libres par (P1) ; ainsi, pour $i \leq T$ on a $d_i - d_{i-1} > 0$ et la suite $d_1, d_2, \ldots, d_T$ est strictement croissante. Toute suite d'entiers strictement croissante et majorée étant finie, pour toute entrée il existe T tel que l'algorithme termine en au plus T phases. □

On peut affiner l'analyse ci-dessus pour majorer le nombre de phases, comme le font Gale et Shapley. Cette question fait l'objet de l'exercice 6 du TD.

4. On formalisera ce qu'est un algorithme et un pas de calcul au chapître suivant.

Correction

Un algorithme **résout** un problème algorithmique $P : E \to 2^S$ si, partant d'une entrée quelconque[5] $e \in E$, l'algorithme retourne en sortie un élément (quelconque) de $P(e)$, c'est-à-dire une sortie valide pour cette entrée. Concernant la correction de leur algorithme, Gale et Shapley écrivent :

> *We assert that this set of marriages is stable. Namely, suppose Stéfanie and Blandan house are not paired-up but Stéfanie prefers Blandan to her assigned house. Then Stéfanie must have proposed to Blandan at some stage and subsequently been rejected in favor of someone that Blandan liked better. It is now clear that Blandan must prefer its student to Stéfanie and there is no instability.*

À nouveau, leur argument mérite d'être développé.

Lemme 1.3. *L'algorithme de Gale-Shapley retourne un mariage.*

Démonstration. Le résultat de l'algorithme est l'ensemble des couples (a, b) où a est une colocation et b est l'élève mis en attente par a au moment où l'algorithme termine. Chaque colocation appartient à au moins un couple puisque l'algorithme ne termine que quand aucune colocation n'est libre. Chaque colocation appartient à au plus un couple puisqu'une colocation ne garde jamais plus d'un·e élève en attente. L'algorithme retourne donc exactement n couples et chaque colocation appartient à exactement un de ces couples. De plus, chaque élève appartient à au plus un couple car un·e élève n'a jamais plus d'une demande en attente. Comme il y a autant d'élèves que de colocations, chaque élève appartient à exactement un couple. L'algorithme retourne donc bien un mariage. □

Afin de montrer que le mariage retourné par l'algorithme ne contient pas d'instabilité, commençons par établir une propriété de monotonie du côté des colocations :

Lemme 1.4. *À la fin de chaque phase, chaque colocation préfère l'élève mis·e en attente à tout·e autre élève qui lui a adressé une demande depuis le début de l'exécution de l'algorithme.*

Démonstration. Considérons une colocation et notons i_0 le numéro de la phase à laquelle elle reçoit pour la première fois une demande (de quelque élève que ce soit). L'énoncé est vrai pour la phase i_0 (la demande mise en attente est la préférée parmi celles reçues à cette phase) et aussi, trivialement, pour toute phase d'index $i < i_0$. L'énoncé se propage alors par récurrence pour les phases $i \geq i_0$. En effet, pour $i \geq i_0$ notons e_i l'élève mis·e en attente à la phase i. D'une part, e_{i+1} est préféré·e ou égal·e à chacune des demandes reçues à la phase $i + 1$. D'autre part, e_{i+1} est préféré·e ou égal·e à e_i, demande qui est elle-même (par hypothèse de récurrence) la préférée parmi tou·tes les candidat·es qui lui ont adressé une demande dans les phases $\leq i$. □

On peut maintenant conclure la preuve de correction :

Proposition 1.5. *L'algorithme de Gale-Shapley retourne un mariage qui est stable pour le système de préférence donné en entrée.*

Démonstration. Le Lemme 1.3 assure que la sortie de l'algorithme est un mariage M. Supposons qu'un couple (Stéfanie, Blandan) ne soit pas dans M. Pour que ce soit une instabilité, on doit avoir que :

(ii) Stéfanie préfère Blandan à son appariement (disons Commanderie) dans M.

(iii) Blandan préfère Stéfanie à son appariement (disons Charles) dans M.

Comme Stéfanie fait ses demandes par ordre décroissant de préférence et a adressé une demande à Commanderie (puisqu'elle y est en attente à la fin de l'algorithme), le (ii) assure que Stéfanie a fait une demande à Blandan. Le Lemme 1.4 assure alors que Blandan préfère Charles à Stéfanie, ce qui contredit (iii). □

5. Dans certaines applications, il est toléré qu'un algorithme fasse occasionnellement des erreurs. Pas dans ce cours : on *prouvera* la correction de nos algorithmes.

Optimalité

Dans la formulation de Mariage stable, les deux ensembles A et B jouent des rôles symétriques. L'algorithme de Gale-Shapley casse cette symétrie : l'un des ensembles adresse des demandes, l'autre les arbitre. On peut donc résoudre une instance de Mariage stable par l'algorithme de Gale-Shapley de deux manières, selon que l'on fait jouer le rôle des étudiants à A ou à B. Il s'avère qu'une instance du problème Mariage stable peut admettre plusieurs solutions possibles (cf les exercices pour un exemple). Ces deux utilisations de Gale-Shapley peuvent donc ne pas donner le même résultat.

Il s'avère que le mariage calculé par l'algorithme de Gale-Shapley est optimal pour les demandeurs au sens suivant. Considérons une entrée P de Mariage stable, c'est-à-dire un système de préférences pour deux ensembles A et B de même taille. Étant donnés $a \in A$ et $b \in B$, on dit que b est **accessible** à a s'il existe un mariage stable pour le système P qui contient (a, b).

Théorème 1.6. *L'algorithme de Gale-Shapley apparie chaque demandeur à l'arbitreur qu'il préfère parmi ceux qui lui sont accessibles.*

On prouve cela à l'exercice 4 du TD.

1.4 Adapter l'algorithme à des variantes du problème

Maintenant que l'on comprend l'algorithme de Gale-Shapley, adaptons le à des variantes du problème.

1.4.1 Ensembles de tailles différentes

Pour commencer, considérons une variante du problème dans laquelle on autorise les ensembles A et B à être de tailles différentes.

Définitions et problème algorithmique. Commençons par remarquer que les notions de **système de préférences** et de **tableau de préférences** restent valides dans ce cadre.

La notion de *mariage* est, elle, à revoir. Un **mariage** de A et B est un sous-ensemble M de $A \times B$ tel que chaque élément de $A \cup B$ apparaît dans au plus un élément de M. La notion d'**instabilité pour un mariage relativement à un système de préférence** reste inchangée.

Remarquons que l'ensemble vide est, trivialement, un mariage stable pour tout système de préférence. Pour quel le problème algorithmique reste intéressant [6], il est naturel de le modifier pour demander que le résultat soit un mariage stable « aussi grand que possible ». Cela se formalise généralement de deux manières :

- un mariage (stable) de A et B est dit **maximal** (ou **maximal pour l'inclusion**) s'il n'est pas contenu strictement dans un autre mariage (stable) de A et B,
- un mariage (stable) de A et B est dit **maximum** (ou **de cardinalité maximum**) s'il n'existe pas de mariage (stable) de A et B de taille strictement plus grande.

Tout mariage (stable) maximum est maximal, mais la réciproque peut être fausse. [7] Dans le cas des mariages stables, maximalité pour la taille et pour l'inclusion s'avèrent équivalentes [8]. Voici donc notre problème :

Mariage stable (différentes tailles)

Entrée : Un tableau décrivant un système de préférences entre deux ensembles A et B.

Sortie : Un mariage stable maximal entre A et B.

6. Au sens où il permet de modéliser des problématiques intéressantes.
7. Intuitivement, la « maximalité pour l'inclusion » est une condition d'optimalité locale, tandis que le « cardinal maximal »est une condition d'optimalité globale.
8. Cela affleure dans l'exercice 5 du TD.

Algorithme. Il faut remettre en question deux des observations faites dans l'analyse de l'algorithme de Gale-Shapley :

a. Un demandeur peut désormais épuiser sa liste. Cela est en fait inévitable si le nombre de demandeurs excède le nombre d'arbitreurs.

b. Certains arbitreurs peuvent ne jamais recevoir de demande. Cela est en fait inévitable si le nombre d'arbitreurs excède le nombre de demandeurs.

Cela nous amène à modifier la condition d'arrêt de l'algorithme. Voici une solution, les changements étant marqués en gras :

a. *To start, let each student propose to his favorite house. Each house who receives more than one proposal rejects all but its favorite from among those who have proposed to it. However, it does not accept him/her yet, but keeps him/her on a string to allow for the possibility that someone better may come along later.*

b. *We are now ready for the second stage. Those students who were rejected now propose to their second choices. Each house receiving proposals chooses its favorite from the group consisting of the new proposers and the student on its string, if any. It rejects all the rest and again keeps the favorite in suspense.*

c. *We proceed in the same manner. Those who are rejected at the second stage propose to their next choices, and the houses again reject all but the best proposal they have had so far.* ***Students who reach the end of their list stop sending proposals.***

d. *As soon as* ***we reach a stage where no new proposal is sent****, each house accepts the student on its string***, if any,** *and the algorithm is declared over.*

Il convient alors de réexaminer l'analyse :

- Chaque demandeur adresse au plus une demande à chaque arbitreur. Il y a donc au plus $|A| * |B|$ phases auxquelles une nouvelle proposition est adressée. L'algorithme termine donc, et ce en au plus $|\mathbf{A}| * |\mathbf{B}|$ phases.
- Le résultat de l'algorithme est un ensemble de couples dans lequel chaque élément apparaît bien au plus une fois. En effet, chaque arbitreur garde au plus une proposition en réserve et chaque demandeur a au plus une proposition en attente. Le résultat est bien un mariage.
- Les Observations 1.4 et 1.5 restent valides, avec les mêmes preuves, aussi le mariage calculé est bien stable.

Il ne reste donc qu'à prouver que le mariage donné en résultat est maximal pour l'inclusion et de généraliser le Théorème 1.6 d'optimalité pour les demandeurs. On traite le premier point à l'exercice 5 du TD.

Cet algorithme produit *le même résultat* que l'algorithme de la Section 1.2 dans le cas où A et B sont de même taille ; il le *généralise*.

1.4.2 Capacités

Une généralisation naturelle de la problématique consiste à autoriser chaque agent à être appariés avec plusieurs agents de l'autre type. L'autorisation peut ne concerner qu'un seul type d'agents [9] ou les deux types [10] Le nombre d'appariements possibles pour un agent est généralement limité, et est appelé sa **capacité**.

Pour modéliser cette nouvelle problématique, on considère deux ensembles A et B de tailles finies (mais pas nécessairement égales) et une fonction $c : A \cup B \to \mathbb{N}$. L'entier $c(x)$ représente la capacité de

9. Par exemple l'affectation de bachelier·e·s dans des établissements d'enseignement supérieur permet à chaque établissement d'accueillir plusieurs bachelier·e·s mais n'affecte chaque bachelier·e que dans au plus une affectation.

10. On peut imaginer l'affectation de spécialistes sur des tâches, chaque spécialiste pouvant prendre en charge plusieurs tâches et chaque tâche pouvant être partagée entre plusieurs spécialistes.

l'agent x. Les définitions de système de préférences et de tableau de préférences se généralisent telles quelles à ce cadre. Un **mariage de capacité** c est un sous-ensemble $M \subseteq A \times B$ tel que chaque élément $x \in A \cup B$ apparaît dans au plus $c(x)$ couples.[11] Soulignons qu'un mariage de capacité c est un *ensemble*, et contient donc 0 ou 1 copie de chaque couple.[12] La définition de stabilité ne change pas et on cherche à nouveau à maximiser l'utilité d'un mariage en le demandant maximal pour l'inclusion. Voici donc notre problème :

MARIAGE STABLE (AVEC CAPACITÉS)

Entrée : Un tableau décrivant un système de préférences entre deux ensembles A et B, ainsi qu'une fonction de capacité c.

Sortie : Un mariage de capacité c entre A et B qui est stable et maximal pour l'inclusion.

On peut adapter l'algorithme comme suit :

a. *To start, let each student* $\mathbf{x}$ *propose to his favorite* $\mathbf{c(x)}$ *house**s**. Each house* $\mathbf{y}$ *who receives more than* $\mathbf{c(y)}$ *proposal rejects all but its* $\mathbf{c(y)}$ *favorite**s** from among those who have proposed to it. However, it does not accept **them** yet, but keeps **them** on a string to allow for the possibility that someone better may come along later.*

b. *We are now ready for the second stage. **Each** student* $\mathbf{x}$ *who **was** rejected now proposes to **its next** choices **so as to have** $\mathbf{c(x)}$ **proposals out**. Each house* $\mathbf{y}$ *receiving proposals chooses its favorite* $\mathbf{c(y)}$ *from the group consisting of the new proposers and the student**s** on its string, if any. It rejects all the rest and again keeps the favorite* $\mathbf{c(y)}$ *in suspense.*

c. *We proceed in the same manner. Those who are rejected at the second stage propose to their next choices, and the houses again reject all but the best proposal**s** they have had so far. **Students who reach the end of their list stop sending proposals.***

d. *As soon as **we reach a stage where no new proposal is sent**, each house accepts the student**s** on its string and the algorithm is declared over.*

Les clefs d'analyse restent les mêmes :

- chaque demandeur adresse au plus une demande à chaque arbitreur, soit au plus $|A|*|B|$ demandes émises,
- à tout moment, chaque élément $x \in A \cup B$ participe à au plus $c(x)$ demandes en suspens,
- L'Observation 1.4 se généralise en remplaçant « par une colocation est son préféré » par « par une colocation y est un de ses $c(y)$ préférés » ; la preuve s'adapte facilement.

On en déduit que l'algorithme termine en au plus $|A| * |B|$ phases et produit un mariage de capacité c qui est stable. La preuve que ce mariage est maximal pour l'inclusion procède du même argument que pour l'algorithme de la Section 1.4.1 (traité en exercice).

1.4.3 Listes de préférences tronquées

On peut envisager qu'un agent n'ordonne qu'une partie des agents de l'autre type, les agents non ordonnés étant considérés comme indésirable (l'agent préfère être non-apparié qu'apparié avec l'un d'eux).

Cette variante amène à modifier la notion de système de préférences (pour chaque agent, on se donne un ordre total sur un sous-ensemble des agents de l'autre type) et la notion de tableau de

11. En particulier, un mariage au sens de la Section 1.4.1 est un mariage de capacité c pour la fonction c constante égale à 1.

12. Si on souhaite autoriser plusieurs appariements entre deux agents, à supposer que leurs capacités le permette, il faudrait travailler avec des multiensembles. Rien ne l'empêche, c'est simplement une autre variante, qui demande une autre adaptation de l'algorithme.

préférences (une entrée peut être − pour indiquer que l'agent n'est pas classé). La notion de mariage (respectant les capacités) reste inchangée, mais la notion d'instabilité doit être ajustée (pour inclure le cas où un agent est apparié avec un agent qu'il n'a pas classé).

Il est remarquable que les algorithmes précédents (sans et avec capacité), sans autre modification, calculent dans le cadre de listes de préférences tronquées un mariage respectant les capacités, stable et maximal pour l'inclusion. Bien entendu, il ne faut pas me croire sur parole mais vérifier les preuves...

1.5 Après l'algorithmique, le déploiement : Parcoursup

En France, l'affectation des néo-bachelier·es (entre autres) en études supérieures est en grande partie centralisée par une plateforme de gestion des préférences. Depuis 2018, cette gestion est organisée par la plateforme Parcoursup. De 2009 à 2017, elle était organisée par la plateforme Apb. Ces systèmes gèrent des centaines de milliers d'élèves et des milliers de formations [13] et rendent des décisions lourdes d'enjeux. C'est donc sans surprise que la première campagne de fonctionnement de Parcoursup au printemps 2018 a suscité de nombreux débats publics.

Ces deux plateformes traitent en fait une variante du problème d'affectation intégrant des capacités (pour les établissements), des listes tronquées (pour les néo-bacheliers) et d'autres critères (taux minimum de boursiers, taux minimum d'élèves résidant dans l'académie, internat, dispositif « meilleurs bacheliers », ...). Le cœur de l'algorithme n'en demeure pas moins une variante de l'algorithme de Gale-Shapley, c'est-à-dire un dialogue entre *demandeurs* et *arbitreurs*. La documentation officielle de l'algorithme de parcoursup est disponible librement à

https://framagit.org/parcoursup/algorithmes-de-parcoursup

(dans le répertoire `doc`). Nous allons souligner deux aspects de cet algorithme à la lumière de notre compréhension de Gale-Shapley. Il ne s'agit nullement de trancher, ni même d'aborder franchement, un quelconque débat autour de parcoursup. L'objectif est d'illustrer comment ce débat peut être enrichi par une compréhension fine de l'algorithme sous-jacent.

Examinons tout d'abord la distribution des rôles au sein de Parcoursup. On l'a vu, la symétrie du problème d'affectation permet de distribuer les rôles de *demandeurs* et d'*arbitreurs* de deux manières dans l'algorithme de Gale-Shapley. On a aussi vu (Théorème 1.6) que cette distribution des rôles n'est pas neutre : l'algorithme de Gale-Shapley calcule une affectation qui satisfait au mieux les préférences des demandeurs. Dans Parcoursup, ce sont les formations qui demandent et les étudiants qui arbitrent. [14] Cela est apparent dans le fait qu'au fil de l'exécution de l'algorithme, lorsqu'un étudiant reçoit de nouvelles propositions, il doit n'en garder qu'une (qu'il pourra accepter) et doit refuser toutes les autres. L'affectation calculée va donc satisfaire au mieux les préférences des formations.

Examinons ensuite le choix initial (mis en œuvre uniquement en 2018) fait par Parcoursup de demander aux étudiants une liste de vœux non-ordonnée et de les consulter à chaque fois que l'algorithme nécessite un arbitrage. Ce choix a plusieurs conséquences :

- L'algorithme doit rendre visibles les états internes, c'est-à-dire les affectations temporaires (« on a string ») à la fin de chacune des phases.
- L'algorithme ne confirme définitivement *aucune affectation* avant la phase finale. Cependant, il est possible que de nombreuses affectations se stabilisent rapidement. Il est difficile à un·e candidat·e d'apprécier, à un instant donné, dans quelle mesure son affectation temporaire a encore des chances d'évoluer.
- La durée d'une phase doit laisser le temps aux élèves consulté·es de décider et communiquer leurs arbitrages. Cela ralentit donc l'algorithme et impose aux étudiants de consulter régulièrement l'état interne courant.

13. D'après wikipedia, ∼660 000 élèves et ∼15 500 formations en 2020.

14. Soulignons que la question n'est pas de savoir si les élèves sont en situation de demandeurs en ce qu'ils demandent leur admission dans l'établissement. La question est de savoir lequel des deux rôles (demandeur ou arbitreur) la modélisation choisie par Parcoursup leur fait jouer.

- Le calendrier de la campagne d'affectation étant contraint, le nombre de phases doit être limité. Un paramètre ayant un impact notable sur le nombre de phases requis est le nombre maximum de vœux autorisés. Le choix de ne pas classer les vœux a priori encourage donc à limiter leur nombre.

Tout cela a des conséquences sociales et sociétales importantes. Par exemple, l'allongement de la durée d'exécution ajoutée à l'incertitude du caractère final de l'affectation courante peut poser des difficultés matérielles : si l'affectation courante est géographiquement éloignée des meilleures propositions espérées, où et quand commencer à chercher un logement ? Un autre exemple est que la qualité de la solution en attente augmentant à mesure que progresse l'algorithme (cf Observation 1.4), les premiers états (que l'étudiant *doit* consulter) peuvent être pessimistes en regard de l'affectation finale et s'avérer inutilement démoralisants. On peut noter qu'un système de répondeur automatique a été ajouté à l'algorithme dès 2019.

1.6 Prolongements

Cette séance s'est concentré sur les appariements bipartis, c'est-à-dire entre agents de deux types différents, sujet classique au croisement de la **théorie algorithmique des jeux**, de l'**optimisation combinatoire**, de la **recherche opérationnelle**, . . .Les couplages (parfaits, maximaux, . . .) ont aussi été largements étudiés, notamment en **théorie (algorithmique) des graphes**.

L'analyse de l'algorithme de Gale-Shapley peut être considérablement élargie. On peut par exemple étudier sa résistance à la **manipulation** : un agent peut-il obtenir un meilleur résultat en mentant sur ses préférences ? On peut aussi étudier sa résistance à la **triche** : un agent ayant réussi à obtenir une copie des préférences de tous les autres agents avant de transmettre ses préférences à lui peut-il calculer des vœux lui assurant le meilleur résultat ? On trouvera des éléments de réponses par exemple dans l'article de Chung-Piaw Teo, Jay Sethuraman et Wee-Peng Tan [TST01].

L'algorithme de Gale-Shapley a de nombreuses applications y compris théoriques. Sur son blog [Kal], Gil Kalai présente la preuve d'une conjecture (de Dinitz) sur une propriété de généralisations des carrés latins dont l'algorithme de Gale-Shapley est un ingrédient essentiel. L'ensemble des mariages qui sont stables pour un ensemble de préférences donné a par ailleurs une structure mathématique riche et largement étudiée. Voir [Wikb].

Références bibliographiques

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1) :9–15, 1962.

[Kal] Gil Kalai. Galvin's proof of the dinitz conjecture. https://gilkalai.wordpress.com/2012/04/26/galvins-proof-of-dinitzs-conjecture/.

[TST01] Chung-Piaw Teo, Jay Sethuraman, and Wee-Peng Tan. Gale-shapley stable marriage problem revisited : Strategic issues and applications. *Management Science*, 47(9) :1252–1267, 2001.

[Wika] Wikipedia. Don croisé d'organes. https://fr.wikipedia.org/wiki/Don_d'organes#Don_croisé.

[Wikb] Wikipedia. Lattice of stable matchings. https://en.wikipedia.org/wiki/Lattice_of_stable_matchings.

À retenir.

- Un *problème algorithmique* explicite les entrées et les sorties d'un calcul, ainsi que le sous-ensemble des sorties admis comme réponse pour une entrée donnée.
- De nombreuses problématiques d'affectation entre agents ayant des préférences se modélisent par le *calcul d'un mariage stable*.
- L'*algorithme d'acceptation retardée*, aussi appelé algorithme de Gale-Shapley, résout de nombreuses variantes du calcul d'un mariage stable.
- Dans cet algorithme, un types d'agents (les *demandeurs*) adresse ses propositions par ordre de préférence décroissante, tandis que l'autre type d'agent (les *arbitreurs*) garde en réserve, à tout moment, la meilleure proposition reçue et refuse les autres.
- L'algorithme de Gale-Shapley est correct et termine. La solution calculée est optimale pour les demandeurs.

Chapitre 2

Modèles de calcul classiques

Après le cas d'étude du Chapitre 1, nous posons ici les bases de l'étude du calcul en définissant formellement les notions d'*algorithme* et de *complexité d'un algorithme*. Cela nous demande de préciser le modèle de calcul dans lequel on travaille. Le modèle usuel d'étude de l'algorithmique, souvent implicite, est le modèle *RAM taille arbitraire*. Sa simplicité cache une grande naïveté : si l'on n'y prend garde, on peut facilement « établir » que tout problème algorithmique solvable peut y être résolu en temps $O(n)$, ce qui est peu réaliste. On explicite donc des règles d'usage de ce modèle par comparaison à un modèle rigoureux, le *RAM 8 bits*, via la notion d'*usage abusif* d'instructions.

Objectifs. À l'issue de cette séance, il est attendu que vous...

- compreniez les notions d'instruction élémentaire et d'algorithme dans les deux modèles RAM considérés et sachiez présenter des algorithmes pour ces modèles.
- sachiez mener une analyse de complexité asymptotique pire cas dans ces deux modèles et pour des fonctions de taille diverses,
- sachiez éviter les usages abusifs d'instruction.

Préambule

Un *modèle de calcul* décrit la manière dont un système *calcule*, c'est-à-dire traite de l'information. La grande variété de modèles est organisée en familles : séquentiels (machine de Turing, automate fini, RAM, machine de Turing quantique, ...), fonctionnels (systèmes de réécriture, lambda calcul, ...), concurrents (systèmes asynchrones à messages, automates cellulaires, ...), etc. Dans ce cours on ne va utiliser que des **modèles séquentiels**.

Les premiers modèles de calcul ont été formalisés pour décrire précisément *ce qui est calculable* et donc *ce qui ne l'est pas*. Ils ont permis d'établir un fait fondamental :

Il existe des problèmes algorithmiques qui n'admettent aucune solution.

Il ne s'agit pas là du simple constat qu'aucun algorithme connu ne résout ces problèmes, mais de la propriété, très forte, qu'il *est impossible* de concevoir un algorithme les résolvant. La *théorie de la calculabilité* se consacre à l'étude de tels problèmes, dits *indécidables*.

Les modèles de calcul sont par ailleurs indispensables à **l'algorithmique**. En effet, informellement, un algorithme décrit une manière de décomposer un calcul donné en pas élémentaires, or c'est le modèle de calcul qui définit ce que sont ces pas élémentaires. En complétant le modèle de calcul par un **modèle de coût**, qui associe à chaque opération élémentaire une quantité (par exemple le temps ou l'énergie qu'elle requiert), on peut étudier l'efficacité des algorithmes voire la *complexité d'un problème*, définie comme la complexité minimale d'un algorithme résolvant ce problème. Ce dernier point est l'objet principal de la **théorie de la complexité**.

L'algorithmique et la théorie de la complexité imposent des exigences contraires sur le modèle de calcul : plus l'ensemble des pas élémentaires est riche, plus la présentation d'*algorithmes individuels* est

aisée mais plus le raisonnement sur des *ensembles d'algorithmes* est compliqué. Il est donc raisonnable d'adapter le modèle selon les circonstances. Ce cours utilise principalement deux modèles, le *RAM 8 bits* et le *RAM taille arbitraire*, mais d'autres modèles interviendront occasionnellement (arbres de décision au Chapitre 6, vérificateurs au Chapitre 8, et circuits quantiques au Chapitre 9).

2.1 Préambule : principes d'architecture des ordinateurs

Avant de définir nos modèles de calcul formellement, examinons quelques principes généraux de conception d'un système de calcul. Cette section est à prendre comme une digression culturelle ayant pour seul but de motiver les modélisations qui suivent. Elle est volontairement succinte, mais les élèves intéressé·e·s trouveront plus de détails et d'exemples en Complément **??**.

Stockage de l'information

Le dispositif de mémorisation d'information d'un ordinateur comporte plusieurs niveaux. Rappelons qu'on s'intéresse ici à des ordinateurs classiques, qui représentent donc l'information au moyen de mots binaires finis. La construction d'une *cellule mémoire élémentaire* permet de mémoriser un nombre 1-bit. Ce premier niveau a peu d'influence sur la définition du modèle de calcul. Nous n'en disons pas plus et renvoyons les élèves intéressé·e·s à l'ouvrage de Nisan et Schocken [NS21, §3.1].

L'assemblage de b cellules mémoire élémentaires en une *case mémoire* permet de mémoriser un nombre b-bits. Généralement $b = 8$, c'est à dire que chaque nombre mémorisé est un **octet**, mais d'autres choix sont possibles. L'organisation de M cases mémoire b-bits en une *unité mémoire*, qui permet ainsi de mémoriser M octets.

La construction d'une unité mémoire introduit un mécanisme important : l'**adressage**. L'unité mémoire numérote les cases de 0 à $M-1$, et le processeur peut demander à la mémoire de lui transmettre le mot binaire contenu dans une case de numéro donné, ou d'écrire un mot binaire donné dans une case de numéro donné. En première approximation,[1] on peut donc considérer la mémoire comme un (gigantesque) tableau de mots binaires, typiquement des octets.

Fonctions booléennes et instructions élémentaires

Puisque toute information est mémorisée sous la forme de mots binaires, toute transformation d'information est essentiellement une fonction de $\{0,1\}^*$ dans $\{0,1\}^*$. Le principe fondateur de la *construction* d'ordinateurs est le principe suivant :

> Pour toutes constantes ℓ, m, pour toute fonction $f : \{0,1\}^\ell \to \{0,1\}^m$, il existe un système physique qui calcule f.

Soulignons-le : on sait construire un système calculant une fonction $f : \{0,1\}^\ell \to \{0,1\}^m$ quand f, ℓ et m sont fixés *à la construction du système*. On peut donc construire un système réalisant n'importe quelle opération modélisable par une fonction booléenne de taille bornée. Autrement dit, on sait construire toute *opération élémentaire* au sens suivant :

> Une **opération élémentaire** est définie comme une transformation de données d'entrée en données de sortie telle que
>
> (i) les entrées déterminent les sorties, et
>
> (ii) les tailles d'encodage des entrées et des sorties sont bornées.

Ainsi, additionner deux entiers n'est pas une opération élémentaire car la taille de leur encodage est non-bornée, mais additionner deux entiers modulo 2^{100} en est une. De même, comparer deux mots binaires arbitraires pour l'ordre lexicographique n'est pas une opération élémentaire mais trier un tableau de 100 entiers, chacun à valeur dans $\{0, 1, \ldots, 2000\}$, en est une.

1. En réalité, des mécanismes plus complexes entrent en jeu mais ceci n'est pas un cours d'architecture des ordinateurs...

Le principe fondateur ci-dessus assure que l'on peut munir un processeur de *n'importe quelle* opération élémentaire ainsi définie. En effet, la propriété (ii) assure que l'on peut coder les entrées (resp. sorties) par un mot binaire de taille constante ℓ (resp. m). Avec (i), l'exécution de l'opération élémentaire se réduit au calcul de la fonction qui envoie chaque mot de $\{0,1\}^\ell$ codant des entrées sur le mot de $\{0,1\}^m$ codant les sorties associées.

Processeurs, assembleur et langages

Un **processeur** est un système physique qui calcule. À la conception d'un processeur, on choisit les opérations élémentaires dont on le munit. Ces opérations élémentaires constituent son **langage assembleur** et peuvent, en principe, varier d'un processeur à l'autre.[2] Pour qu'un calcul puisse être réalisé par un processeur, il doit être décomposé en une succession d'instructions assembleur.

Un **langage de programmation** est un ensemble d'opérations, pas nécessairement élémentaires, qui permettent de décrire un programme plus confortablement qu'en assembleur. La conception d'un langage s'accompagne de l'écriture, pour *chaque* processeur supporté, d'un traducteur[3] entre ce langage et l'assembleur du processeur. Le confort apporté peut être par exemple de la *portabilité* (un programme, une fois écrit, pourra être facilement traduit en de nombreux assembleurs) ou une plus grande *abstraction* (le langage offre des concepts plus proches de l'intuition humaine que ceux disponibles en assembleur).

Le **langage machine** d'un processeur est une convention d'encodage de ses instructions assembleur par des mots binaires. Un programme est ainsi une suite de mots binaires, chacun encodant une instruction élémentaire. En première approximation, l'exécution d'un programme par un processeur revient donc à répéter trois opérations : chargement du mot binaire codant l'instruction assembleur suivante, décodage de ce mot binaire pour déterminer l'opération élémentaire à réaliser, réalisation de cette opération.

Et les algorithmes dans tout cela ?

Formellement, un **algorithme** est la décomposition d'un calcul en une séquence d'opérations élémentaires. Le choix des opérations élémentaires utilisées pour décrire un algorithme est laissé libre, aussi un algorithme peut ne pas être directement interprétable dans l'assembleur d'un processeur donné, voire dans un langage plus haut niveau comme le `C` ou le `python`. Le travail de traduction d'un algorithme dans un langage donné s'appelle son **implantation**.

On peut envisager le traitement d'un problème algorithmique comme l'utilisation conjointe d'un algorithme, d'un langage de programmation et d'une architecture matérielle. L'implantation de l'algorithme dans le langage donne un programme dont la compilation ou l'interprétation produit un code en langage machine exécutable par l'architecture matérielle pour résoudre le problème. Algorithmes, langages et architectures s'adaptent continuellement les uns aux autres[4], raison de plus pour distinguer la conception d'algorithmes de leur programmation.

2.2 Modèle RAM 8 bits

Notre premier modèle de calcul s'inspire directement des principes d'architecture exposés ci-dessus mais les épure pour faciliter la description des algorithmes et permettre l'analyse de leur complexité.

2.2.1 Définitions : modèle et algorithme

Les **modèles RAM** comportent deux parties, l'unité mémoire et l'unité de calcul.

2. Pour des raisons pratiques, il existe des familles de processeurs de langages rétro-compatibles. Ainsi, chaque processeur intel de la famille `x86` peut exécuter tout code écrit pour ses prédécesseurs (mais pas l'inverse, car de nouvelles instructions ont pu apparaître).

3. Ce traducteur peut être un *interpréteur*, comme pour `python`, ou un *compilateur*, comme pour `C`.

4. Par exemple, les *tensor cores* de la microarchitecture Volta des cartes graphiques Nvidia ont une opération élémentaire qui prend en entrée trois matrices 4×4 A, B et C, et retourne $A \times B + C$. Cette opération a été ajoutée à l'assembleur *parce qu*'elle est utilisée intensivement dans les *algorithmes* d'entrainement de réseaux de neurones.

L'**unité de mémoire** mémorise de l'information. Elle dispose d'emplacements où ranger l'information, appelés *cases mémoire*. Chaque case mémoire peut contenir un mot binaire. Les cases mémoire sont numérotées $0, 1, 2, \ldots$ et le numéro d'une case mémoire est appelée son **adresse**. Deux cases mémoire distinctes ont des adresses distinctes, mais peuvent contenir des mots binaires identiques [5].

L'**unité de calcul** peut exécuter des *instructions élémentaires* sur des données contenues dans des cases mémoire dont on connait les adresses. [6] L'unité de calcul est en charge de l'exécution du programme et a accès au code du programme : elle dispose d'un pointeur sur l'instruction courante, fait avancer ce pointeur une fois l'instruction courante exécutée, et peut exécuter certaines instructions telles que `if/elif/else`, `for`, `while`, ou encore `break/continue`, qui modifient (conditionnellement) ce pointeur.

Le **modèle RAM 8 bits** est une version du modèle RAM dans laquelle :

- le nombre de cases mémoire est **infini** et l'ensemble des adresses est $\mathbb{N}$,
- chaque case mémoire contient un mot binaire de taille exactement 8,
- une **instruction élémentaire** est la transformation d'entrées décrites par un nombre borné de cases mémoire en des sorties décrites par un nombre borné de cases mémoire et déterminées uniquement par les entrées.

La valeur 8 est, ici, choisie arbitrairement, et toute autre constante donnerait un modèle équivalent pour notre usage. Soulignons deux points :

- La définition d'instruction élémentaire dans ce modèle RAM 8 bits correspond exactement à la notion d'opération élémentaire décrite dans le préambule sur l'architecture des ordinateurs : elle modélise des transformations de données que l'on sait réaliser physiquement.
- Si une séquence d'instructions élémentaires est de longueur **bornée**, alors elle constitue, par définition, elle même une instruction élémentaire.

On peut maintenant définir formellement la notion d'algorithme :

Un **algorithme en RAM 8 bits** est une séquence d'instructions élémentaires pour ce modèle.

2.2.2 Mot binaire de taille arbitraire en RAM 8 bits

Dans le modèle RAM 8 bits, pour stocker un mot binaire arbitraire en mémoire il est nécessaire de le découper en mots 8 bits, chacun pouvant être stocké dans une case mémoire. Par exemple :

$$\underbrace{01011011101111011111011111101111}_{\text{mot 32 bits}} \quad \to \quad \underbrace{01011011}_{\text{8 bits}}\ \underbrace{10111101}_{\text{8 bits}}\ \underbrace{11110111}_{\text{8 bits}}\ \underbrace{11101111}_{\text{8 bits}}.$$

Pour stocker en mémoire un mot binaire dont la taille n'est pas multiple de 8, on choisi une convention de « complétion » (*padding*) ; un choix courant est d'ajouter des 0 en préfixe jusqu'à ce que la taille soit le multiple de 8 immédiatement supérieur. Par exemple :

$$\underbrace{10110111011101111}_{\text{mot 17 bits}} \quad \to \quad \underbrace{000000101101110111101111}_{\text{mot 24 bits}} \quad \to \quad \underbrace{00000010}_{\text{8 bits}}\ \underbrace{11011101}_{\text{8 bits}}\ \underbrace{11101111}_{\text{8 bits}}.$$

Ainsi, en RAM 8 bits, on représente un mot binaire par un *tableau* d'octets :

$$w = 10110111011101111 \quad \rightsquigarrow \quad w[1..3] = [00000010, 11011101, 11101111].$$

5. Par exemple, les cases mémoire d'adresses 5 et 12 peuvent toutes deux contenir le mot « 101010 ».
6. L'unité de calcul peut communiquer à l'unité de mémoire une demande de lecture (« quel est le contenu de la case n° A ? ») ou d'écriture (« écrire B dans la case n° A ») quand elle dispose des informations A et B en interne.

Stockage d'entiers longs

Le stockage d'entiers de taille arbitraire donne lieu à une convention particulière qu'il est utile de détailler. L'idée est de représenter un entier a par un tableau $a[1..n]$ ayant la propriété que

$$a = \sum_{i=1}^{n} a[i] * 256^{i-1}.$$

Pour cela, (i) on part de la représentation binaire de a, (ii) on ajoute des zéros à gauche jusqu'à obtenir un mot de taille multiple de 8, (iii) on découpe le mot obtenu en mots 8 bits, et (iv) on stocke les mots dans l'ordre en commençant par la fin. (Noter que seule l'étape (iv) a changé.) Par exemple, l'entier 187887 de représentation binaire 101101110111101111 est codé par le tableau $w[1..3] = [11101111, 11011101, 00000010]$, de sorte que $w = w[3] * 256^2 + w[2] * 256 + w[1]$. Dans ce cours, on appelle cette méthode de stockage la **présentation d'un entier par un tableau d'octets**.

2.2.3 Présentation d'un algorithme RAM 8 bits

Un algorithme est défini comme une suite d'instructions élémentaires. Cependant, il n'est pas toujours raisonnable de *présenter* un algorithme par une telle suite. Illustrons cela sur un exemple, avant de formaliser les usages à suivre pour la présentation d'algorithmes.

Tentative de décomposition en instructions élémentaires...

À titre d'exemple, décomposons en instructions élémentaires la première phrase de l'algorithme de Gale-Shapley vu au Chapitre 1 :

> To start, let each student propose to its favorite house

On peut commencer par décomposer cela en...

```
pour i = 1..n
   l'élève i demande à sa colocation préféré
```

mais pour le le modèle RAM 8 bits, ces instructions **ne sont pas** élémentaires. En effet, si on décompose[7] le `pour` en

```
1  i = 1
2  l'élève i demande à sa colocation préférée
3  i = i+1
4  si i != n+1 aller à la ligne 2
```

on voit qu'il cache un `i=i+1` qui **n'est pas** une instruction élémentaire puisque la donnée d'entrée (un entier arbitraire) n'est pas de taille bornée ! Pour descendre au niveau d'instructions élémentaires, il faut en fait présenter i par un tableau d'octets et expliciter son incrémentation (qui, par propagation des retenues, est susceptible de modifier toutes les cases du tableau). Une fois cela fait, il resterait à s'occuper des instructions des lignes 2 et 4, qui ne sont pas non plus élémentaires...

Instructions décomposables

Pour comprendre un algorithme, il n'est pas nécessaire (ni même utile) qu'il soit donné par une liste d'instructions élémentaires : il suffit de donner suffisamment d'information pour que la lectrice ou le lecteur puisse produire cette liste si besoin. Autrement dit, il suffit, lorsqu'on présente un algorithme, que chaque étape soit facilement *décomposable* en instructions élémentaires.

7. D'un point de vue pratique, il s'agit réellement d'une décomposition : peu de langages assembleurs proposent une instruction `for` tandis que la plupart proposent des instructions sauts conditionnels.

Formellement, on définit une **instruction** comme la transformation d'entrées décrites par un nombre *fini* de cases mémoire en des sorties décrites par un nombre *fini* de cases mémoire et déterminées uniquement par les entrées. Une instruction est dite **décomposable** si elle peut être réalisée par une séquence finie d'instructions élémentaires. Autrement dit, une instruction est décomposable s'il existe un algorithme en RAM 8 bits qui la calcule.

La présentation d'un algorithme peut utiliser des instructions décomposables.

Pratique dans ce cours. On autorise la présentation d'algorithmes en RAM 8 bits à utiliser des instructions décomposables à condition qu'il soit *facile* de les décomposer en instructions élémentaires et que le choix de cette décomposition soit sans importance pour l'algorithme dès lors qu'il est raisonnable. On renforcera ces conditions en Section 2.3 en une *convention de présentation des algorithmes*.

Exemple : multiplication d'entiers

Illustrons la présentation d'un algorithme RAM 8 bits pour le problème algorithmique suivant :

MULTIPLICATION D'ENTIERS : Étant données les représentations binaires de deux entiers, calculer la représentation binaire de leur produit.

Ce problème peut sembler élémentaire mais s'avère à la fois profond [8] et fondamental ; on y reviendra plusieurs fois dans ce cours. En voici une formulation adaptée au modèle RAM 8 bits :

MULTIPLICATION D'ENTIERS PRÉSENTÉS PAR TABLEAU D'OCTETS

Entrée : Des tableaux $A[1..n]$ et $B[1..n]$ présentant des entiers a et b.

Sortie : Le tableau $C[1..2n]$ présentant l'entier $a * b$.

Une manière simple de résoudre ce problème consiste à développer naïvement la formule

$$\left(\sum_{i=1}^{n} A[i] * 256^{i-1}\right) * \left(\sum_{j=1}^{n} B[j] * 256^{j-1}\right) = \sum_{k=1}^{2n-1} \underbrace{\left(\sum_{1\le i,j\le n,\ i+j=k+1} A[i] * B[j]\right)}_{\stackrel{\text{def}}{=} D[k]} 256^{k-1},$$

à évaluer chaque terme $D[k]$ naïvement pour $k = 1..2n-1$, puis à "propager les retenues" entre les $D[k]$ pour en déduire $C[1..2n]$. Cela devrait ressembler à la manière dont vous avez appris à multiplier en petite classe [9]. On peut le faire par exemple comme suit :

```
multiplication_naive(A[1..n],B[1..n])
1  créer un tableau d'octets C[1..2n] = [0,0, ..., 0]
1  créer un tableau d'entiers D[1..2n] = [0,0, ..., 0]
2  pour i allant de 1 à n
3      pour j allant de 1 à n
4        ajouter A[i]*B[j] à D[i+j-1]
5  pour k=1..2n-1
6      ajouter l'entier D[k]/256 à D[k+1]
7      C[k] = D[k] modulo 256
8  retourner C[1..2n]
```

(La division de la ligne 6 est entière.) Soulignons que la multiplication ('$*$') de la ligne 4 prend en entrée deux nombres 8 bits et est donc élémentaire. Aucune des lignes de cet algorithme n'est une instruction élémentaire, mais chacune est facilement décomposable.

8. On multiplie depuis des millénaires, et pourtant les derniers développements de la recherche sur ce problème ne datent que de 2019.

9. Vous avez normalement appris, étant données les écritures décimales de deux entiers, à déterminer l'écriture décimale de leur produit. On est simplement passé d'une écriture en base 10 à une écriture en base 256.

2.3 Complexité asymptotique pire-cas

L'efficacité d'un algorithme peut s'apprécier selon plusieurs critères, tels que le temps pris, l'espace mémoire utilisé, ou encore l'énergie consommée. On va ici s'intéresser au temps, mais les méthodes que l'on présente sont applicables aux autres critères. On commence par esquisser la définition de *complexité pire-cas* d'un algorithme. On souligne ensuite les nombreuses lacunes de cette esquisse, puis on observe que ces lacunes disparaissent lorsque l'on adopte un point de vue *asymptotique*.

L'esquisse

Considérons un algorithme[10] $\mathcal{A}$ qui résout un problème algorithmique d'espace d'entrée E. Pour toute entrée $e \in E$, on définit le **coût de $\mathcal{A}$ sur e**, noté $c_{\mathcal{A}}(e)$, comme le nombre d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter e.

En supposant que l'algorithme $\mathcal{A}$ ait été présenté au moyen d'instructions toutes élémentaires, la fonction $c_{\mathcal{A}}$ reste difficile à utiliser, ne serait-ce que parce que l'ensemble E peut être compliqué à décrire. On résout cela en *choisissant* une **fonction de taille** sur E, c'est à dire une fonction $t : E \to \mathbb{N}$, et en agrégeant ensemble les valeurs $c_{\mathcal{A}}(e)$ pour toutes les entrées e de même taille $t(e)$. Il y a souvent (au moins) deux choix naturels pour cette mesure de taille :

- Un « paramètre naturel » utilisé dans la description de l'entrée, comme par exemple le nombre n de demandeurs et d'arbitreurs pour Mariage stable, le nombre n d'octets dans un tableau pour Multiplication d'entiers présentés par tableau d'octets.
- La taille d'encodage de l'entrée, c'est à dire le nombre N de bits de son encodage en mot binaire (cf. Section 0.6.2).

Ces deux choix sont légitimes et nous les utiliserons selon le contexte.

Une fois fixée la fonction de taille, il convient de choisir la méthode d'agrégation. Pour **l'analyse pire-cas**, l'agrégation des valeurs de $c_{\mathcal{A}}$ d'une taille donnée se fait par la fonction :

$$C_{\mathcal{A}} : \left\{ \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ t & \mapsto & \max\{c_{\mathcal{A}}(e) \colon e \in E, t(e) = t\}. \end{array} \right. \tag{2.1}$$

Autrement dit, $C_{\mathcal{A}}(t)$ est le nombre maximum d'instructions élémentaires exécutées lors du traitement d'une entrée de taille t.

Des lacunes.. que résout l'asymptotique

La définition de $C_{\mathcal{A}}$ souffre de plusieurs imprécisions problématiques. Soulignons-en trois :

- Une présentation de l'algorithme $\mathcal{A}$ respectant la convention de présentation de la Section 2.2.3 ne suffit pas à déterminer la fonction $c_{\mathcal{A}} : E \to \mathbb{N}$: il faut savoir combien d'instructions élémentaires se cachent derrière chaque instruction décomposable.
- Si l'on choisit comme fonction de taille la taille d'encodage, la définition de $C_{\mathcal{A}}(t)$ demande de préciser la convention d'encodage utilisée.
- La notion d'instruction élémentaire autorise les regroupements : l'enchainement de deux instructions élémentaires est elle-même une instruction élémentaire. On peut donc diviser par 2 (ou toute autre constante) la complexité d'un algorithme à peu de frais[11].

Il s'avère que ces trois problèmes, et d'autres[12], disparaissent essentiellement[13] lorsque l'on s'intéresse non pas à la fonction $C_{\mathcal{A}}$, mais *à son comportement asymptotique*.

> La **complexité asymptotique pire-cas** d'un algorithme $\mathcal{A}$ est l'ordre de grandeur asymptotique, pour $t \to \infty$, du nombre maximum d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter une entrée de taille t.

10. À ce stade, on n'a formellement défini que les *algorithmes RAM 8 bits*, mais cette section s'applique aussi telle quelle aux algorithmes du modèle RAM taille arbitraire que l'on définit ultérieurement.
11. Ce phénomène est décrit plus généralement par le *Théorème d'accélération linéaire*.
12. Par exemple, le caractère discutable du choix d'assigner un coût unité à chaque instruction élémentaire.
13. Tous les problèmes ne disparaissent pas. Ainsi, par exemple, certaines données n'ont pas d'encodage optimal (on dit *entropique*) connu.

Lorsque le contexte est clair, on abrège « complexité asymptotique pire-cas » en **complexité**.

Renforcement de la convention de présentation des algorithmes

Il est courant d'avoir à déterminer la complexité asymptotique pire-cas d'un algorithme présenté au moyen d'instructions décomposables. Pour permettre cela, on étend la convention amorcée en Section 2.2.3 en la **convention de présentation des algorithmes** suivante :

> La présentation d'un algorithme peut utiliser des instructions décomposables si
>
> (i) la manière dont cette décomposition est faite est sans importance pour l'algorithme dès lors qu'elle est « raisonnable », et
>
> (ii) l'asymptotique pire-cas du nombre d'instructions élémentaires utilisées dans cette décomposition est facile à déterminer.

Cette convention vaut pour le modèle RAM 8 bits mais vaudra aussi pour le modèle RAM taille arbitraire que l'on définit ci-après.

Exemple : complexité de l'algorithme d'acceptation retardée

Pour prendre un exemple concret, évaluons la complexité de l'algorithme d'acceptation retardée du Chapitre 1 dans le modèle RAM 8 bits. On considère la version symétrique de cet algorithme, avec autant de demandeurs que d'arbitreurs.

(a) Relativement au nombre d'agents

Commençons par faire l'analyse en prenant comme fonction de taille le nombre n de demandeurs (et donc d'arbitreurs). On est ici dans le cas d'un « paramètre naturel ». L'algorithme procède par phase, la première étant :

> a. *To start, let each student propose to his favorite house. Each house who receives more than one proposal rejects all but its favorite from among those who have proposed to it. However, it does not accept him/her yet, but keeps him/her on a string to allow for the possibility that someone better may come along later.*

Cette phase réalise $O(n)$ opérations de copie et comparaison sur des entiers à valeur dans $\{1, 2, \ldots, n\}$. Chacun de ces entiers occupe $O(\log n)$ cases mémoire, aussi ces instructions ne sont pas élémentaires. On peut vérifier que la copie et la comparaison d'entiers occupant k cases mémoire peut se faire en $O(k)$ instructions élémentaires. Ces instructions décomposables sont donc chacune de complexité $O(\log n)$ et la complexité de la 1ère phase est $O(n \log n)$.

L'algorithme continue par des phases d'un type similaire impliquant uniquement les proposeurs ayant essuyé un rejet à la phase précédente :

> b. *We are now ready for the second stage. Those students who were rejected now propose to their second choices. Each house receiving proposals chooses its favorite from the group consisting of the new proposers and the student on its string, if any. It rejects all the rest and again keeps the favorite in suspense.*
>
> c. *We proceed in the same manner. Those who are rejected at the second stage propose to their next choices, and the houses again reject all but the best proposal they have had so far.*

On a vu au Chapitre 1 qu'il y a $O(n^2)$ phases. En majorant le nombre de proposeurs participants par n, les mêmes arguments que ceux utilisés pour la 1ère phase montrent que la complexité de chacune des phases ultérieures est $O(n \log n)$. De même, la condition de terminaison peut être vérifiée en $O(n \log n)$ instructions élémentaires.

Dans l'ensemble, la complexité de l'algorithme de Gale-Shapley dans le modèle RAM 8 bits est $O(n^3 \log n)$ où n est le nombre de demandeurs. Il y a $O(n^3)$ instructions décomposable manipulant chacune des entrées de taille $O(\log n)$.

(b) Relativement à la taille d'encodage de l'entrée

Notons maintenant N la taille d'encodage de l'entrée. L'entrée consiste en $2n(n-1)$ entiers, chacun compris entre 1 et n, et chaque entier entre 1 et n apparaît $2n-2$ fois. On a donc $N = \Theta(n^2 \log n)$. On peut alors reprendre l'analyse ci-dessus en estimant le coût de chaque phase en fonction de N. Puisque l'analyse a déjà été faite en fonction de n, on peut aussi substituer $n = O\left(\sqrt{\frac{N}{\log N}}\right)$ dans le résultat précédent, et obtenir que l'algorithme de Gale-Shapley dans le modèle RAM 8 bits est $O(N\sqrt{N}\log N)$ où N est la taille d'encodage de l'entrée. (Cette majoration peut être affinée par un examen plus attentif...)

Discussion : pertinence et limites de la complexité asymptotique pire-cas

Il peut sembler naturel d'évaluer l'efficacité d'un algorithme par un banc d'essais, mesurant ses performances sur un ensemble d'entrées. Outre que ce type d'étude demande un effort substantiel, son résultat dépend de la qualité de l'algorithme, mais aussi de la qualité de son implantation, voire de son adéquation au langage et à l'architecture matérielle choisie. On souhaite ici se donner les moyens de comparer l'efficacité d'algorithmes a priori de ces autres facteurs. Pour cela, on se contente d'apprécier la manière dont la quantité de ressources qu'un algorithme consomme croît avec la taille de l'entrée à traiter, lorsque celle-ci devient significative, c'est-à-dire, en première approximation, quand elle tend vers l'infini.

Cette méthode d'analyse enchaîne les simplifications : coût identique pour toutes les instructions élémentaires, passage au pire-cas pour une taille donnée, focalisation sur l'asymptotique. Au final, la mesure que l'on obtient est un bien piètre prédicteur des ressources consommées par un algorithme sur une donnée spécifique. Son intérêt est autre : cette mesure donne une *garantie* (grâce au « pire-cas ») sur le *rythme* auquel les ressources consommées augmentent lorsqu'on augmente la taille d'une instance.

2.4 Modèle RAM taille arbitraire

L'étude de l'algorithmique se fait usuellement dans un modèle implicite qui est plus permissif que le RAM 8 bits. Par exemple, les cases mémoire peuvent généralement contenir des mots binaires de *taille quelconque*. Nous formalisons maintenant ce modèle afin de pouvoir faire de l'algorithmique « comme d'habitude », mais installons toutefois des garde-fous pour prévenir les usages déraisonnables au sens où ils sont incompatibles avec le modèle RAM 8 bits.

Définitions : modèle, algorithme et complexité

Le modèle RAM à taille de mots arbitraire est une variation du modèle RAM (et comporte donc lui aussi deux parties, l'unité mémoire et l'unité de calcul, etc.). L'unique différence avec le RAM 8 bits est la relaxation de la contrainte sur la taille du mot binaire que peut contenir une case mémoire.

Le **modèle RAM taille arbitraire** est une version du modèle RAM dans laquelle :

- le nombre de cases mémoire est **infini** et l'ensemble des adresses est $\mathbb{N}$,
- chaque case mémoire contient un mot binaire fini mais de **taille arbitraire**,
- une **instruction élémentaire** est la transformation d'entrées décrites par un nombre borné de cases mémoire en des sorties décrites par un nombre borné de cases mémoire et déterminées uniquement par les entrées.

Un **algorithme en RAM taille arbitraire** est une séquence d'instructions élémentaires pour ce modèle.

Une fois choisi une fonction de taille sur les entrées, on peut définir la **complexité (asymptotique pire-cas)** d'un algorithme $\mathcal{A}$ dans le modèle RAM taille arbitraire comme l'ordre de grandeur asymptotique, pour $t \to \infty$, du nombre d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter une entrée de taille t.

2.4.1 L'algorithmique change peu entre RAM 8 bits et RAM taille arbitraire

Comparons maintenant nos deux modèles du point de vue de leur utilisation en algorithmique.

Les instructions élémentaires sont différentes.

Toute instruction élémentaire en RAM 8 bits est aussi une instruction élémentaire en RAM taille arbitraire. Pour voir que la réciproque est fausse, considérons par exemple l'instruction `i=i+1` de la ligne 3 de l'extrait d'algorithme suivant :

```
1  i = 1
2  l'élève i demande à sa colocation préférée
3  i = i+1
4  si i != n+1 aller à la ligne 2
```

On a vu que cette instruction n'est pas élémentaire en RAM 8 bits. Dès lors que la variable i est encodée sur un mot binaire écrit dans une seule case mémoire, elle est élémentaire en RAM taille arbitraire.

Les instructions décomposables sont les mêmes.

En revanche, on peut facilement se convaincre que toute instruction élémentaire dans le modèle RAM taille arbitraire peut se décomposer en un nombre fini d'instructions élémentaires en RAM 8 bits. Cela découle du fait que tout mot binaire fini peut se décomposer en un nombre fini de mots binaires 8 bits, comme on l'a vu en Section 2.2.2. En particulier, toute instruction décomposable en RAM taille arbitraire est aussi décomposable en RAM 8 bits.

Les *présentations* d'algorithmes sont les mêmes.

Les notions d'*algorithme* sont, formellement, différentes en RAM 8 bits et en RAM taille arbitraire. En effet, un algorithme est une séquence d'instructions élémentaires, et ces modèles ont des notions d'instruction élémentaire différentes. Cependant, comme les notions d'instructions décomposables sont les mêmes dans nos deux modèles, toute présentation d'un algorithme $\mathcal{A}$ dans un modèle vaut présentation d'un algorithme $\mathcal{A}'$ dans l'autre modèle. Les algorithmes $\mathcal{A}$ et $\mathcal{A}'$ sont formellement différents, mais ils résolvent généralement le même problème algorithmique (à quelques nuances d'encodage près).

Dans ce cours, on identifie un algorithme avec sa présentation et on abrège « l'algorithme présenté par $\mathcal{A}$ dans le modèle $\mathcal{M}$ » en « l'algorithme $\mathcal{A}$ dans le modèle $\mathcal{M}$ ».

On parle ainsi de « l'algorithme d'acceptation retardée en RAM 8 bits » et de « l'algorithme d'acceptation retardée en RAM taille arbitraire ».

Intuitivement, ces modèle reflètent les différentes idées que l'on peut se faire du fonctionnement d'un ordinateur suite à une première expérience de programmation : RAM taille arbitraire pour le `python` et RAM 8 bits pour le `C`.

2.4.2 La complexité change entre RAM 8 bits et RAM taille arbitraire

Reprenons notre exemple de l'algorithme d'acceptation retardée, et en particulier l'instruction `i=i+1` discutée ci-dessus. En RAM taille arbitraire, cette instruction est élémentaire et a donc un coût de 1. En RAM 8 bits, comme i prend des valeurs allant de 1 à n, cette instruction est décomposable et peut être réalisée par un algorithme de complexité $O(\log n)$ en RAM 8 bits. Plus systématiquement, si on reprend l'analyse de la Section 2.3, on obtient que l'algorithme d'acceptation retardée est de complexité $O(n^3)$ en RAM taille arbitraire, pour n le nombre de demandeurs.

La complexité asymptotique pire cas d'un algorithme en RAM 8 bits est parfois appelée sa **complexité numérique** (*bit complexity*).

La complexité asymptotique pire cas d'un algorithme en RAM taille arbitraire est parfois appelée sa **complexité arithmétique** (*arithmetic complexity*).

On peut, à l'inverse, obtenir la complexité numérique de l'algorithme d'acceptation retardée comme valant $O(n^3 \log n)$ car (i) cet algorithme est de complexité arithmétique $O(n^3)$, et (ii) chaque instruction élémentaire de cet algorithme est de complexité numérique $O(\log n)$. Dans cet exemple la complexité change assez peu, mais on va voir qu'elle peut changer bien plus drastiquement.

Exemple d'abus : multiplication (bis)

Revenons sur le problème de multiplication d'entiers dans sa formulation compatible avec le modèle RAM 8 bits :

MULTIPLICATION D'ENTIERS PRÉSENTÉS PAR TABLEAU D'OCTETS

Entrée : Des tableaux $A[1..n]$ et $B[1..n]$ présentant des entiers a et b.

Sortie : Le tableau $C[1..2n-1]$ présentant l'entier $a * b$.

Voici un algorithme élémentaire :

```
1  multiplication_magique(A[1..n],B[1..n])
2     va,vb,p = 0,0,1
3     pour i allant de 1 à n
4        ajouter A[i]*p à va
5        ajouter B[i]*p à vb
6        multiplier p par 256
7     vc = va * vb
8     créer C[1..2n-1] = [0,0, ..., 0]
9     pour i allant de 1 à 2n-1
10       C[i] = vc modulo 256
11       vc = vc/256
12    retourner C[1..2n-1]
```

Précisons que la division de la ligne 11 est entière. On peut vérifier que chaque instruction est élémentaire dans le modèle RAM taille arbitraire à l'exception de la création du tableau `C[1..2n-1]` ligne 8, mais elle se décompose facilement en $O(n)$ instructions élémentaires en RAM taille arbitraire. Cet algorithme est donc de complexité $O(n)$ en RAM taille arbitraire, pour n la taille du tableau.

Comme annoncé, cet algorithme est tout aussi valide en RAM 8 bit car chaque instruction est décomposable. On peut cependant remarquer que l'instruction `vc = va*vb` de la ligne 7 ne demande rien de moins que de résoudre le problème MULTIPLICATION D'ENTIERS. Cette présentation ne respecte donc pas la convention de la page 34 d'être facilement décomposable.

Réduction à l'absurde

Poussons le raisonnement précédent un peu plus loin. Commençons par établir un principe d'encodage simple.

> Toute entrée d'un problème algorithmique peut être encodée en **un seul mot binaire**.

La **concaténation** des mots binaires w et w' est le mot binaire $w \cdot w' \stackrel{\text{def}}{=} w_1 w_2 \dots w_{|w|} w'_1 w'_2 \dots w'_{|w'|}$ obtenu en « accolant » w' à la fin de w. Cette opération est associative (mais pas commutative). On peut ainsi concaténer n'importe quelle suite finie de mots binaires en un unique mot binaire. Soulignons que cette opération n'est pas injective, puisque par exemple

$$001 = 0 \cdot 01 = 00 \cdot 1.$$

Cela ne permet donc pas d'encoder deux mots binaires en un... Pour cela, définissons la fonction $D : \{0,1\}^* \to \{0,1\}^*$ comme l'unique fonction satisfaisant :

$$D(0) = 00, \quad D(1) = 11, \quad \text{et pour tous } w, w' \in \{0,1\}^*, \quad D(w \cdot w') = D(w) \cdot D(w').$$

(L'unicité de D se prouve par récurrence sur la taille des mots considérés.) Cela permet de définir une convention d'encodage pour l'ensemble des *suites finies de mots binaires* par

$$(w(1), w(2), \dots, w(k)) \mapsto D(w(1)) \cdot 01 \cdot D(w(2)) \cdot 01 \cdot \dots \cdot 01 \cdot D(w(k)).$$

Par exemple, on peut encoder le couple d'entiers $(8, 5)$ en encodant chaque entier par son écriture binaire, puis en appliquant D :

$$(8,5) \mapsto (1000, 101) \mapsto 11000000 \cdot 01 \cdot 110011 \mapsto 1100000001110011.$$

On laisse en exercice la preuve que cette fonction est bien injective.

De même, toute sortie d'un problème algorithmique peut s'encoder sur un seul mot binaire. Dès lors, la résolution de ce problème revient à transformer une entrée décrite par une unique case mémoire en une sortie décrite par une unique case mémoire et déterminée uniquement par l'entrée. Cette résolution satisfait donc à la définition d'instruction élémentaire dans le modèle RAM taille arbitraire !

Lorsque le problème n'encode pas l'entrée par un unique mot binaire, comme c'est le cas pour Multiplication d'entiers présentés par tableau d'octets, il est facile de s'y ramener en temps $O(n)$ où n est le nombre de cases mémoires occupées par l'entrée. C'est ce que fait l'algorithme abusif ci-dessus pour la multiplication ; une réduction plus générale est facile à réaliser à partir de la fonction D définie ci-dessus. On peut ainsi facilement établir que...

> Tout problème algorithmique qui peut être résolu en RAM taille arbitraire peut l'être par un algorithme de complexité $O(n)$, où n est le nombre de cases mémoire occupées par l'entrée.

Autrement dit, le modèle RAM taille arbitraire est tellement simpliste que la notion de *complexité d'un problème*, définie comme complexité du meilleur algorithme résolvant ce problème, y est triviale.

2.4.3 Usage abusif d'une instruction décomposable

On peut définir un bon usage du modèle RAM taille arbitraire en se référant au modèle RAM 8 bit, via la notion d'*usage abusif* d'une instruction. Informellement, cela revient à se poser la question suivante :

> Le coût de cette instruction en RAM 8 bits *telle que l'algorithme l'utilise* est-il très différent de 1 ?

Prenons un exemple. L'instruction `i=i+1` est de complexité linéaire en la taille de `i`. Si elle est utilisée par un algorithme traitant d'une entrée de taille n, avec `i` prenant des valeurs entre 0 et n, alors son coût est $O(\log n)$, ce qu'on peut considérer comme proche de 1.

Formalisons cela. Considérons une instruction décomposable `instr` dans la présentation d'un algorithme $\mathcal{A}$. Notons $f(t)$ l'asymptotique du nombre maximum d'opérations élémentaires en RAM 8 bits exécutées pour réaliser `instr` *au cours du traitement par* $\mathcal{A}$ *d'une entrée de taille* t. L'usage d'`instr` par $\mathcal{A}$ est **raisonnable** si $f(t)$ est polylogarithmique en t, et **abusif** sinon. Soulignons deux subtilités :

- Le caractère raisonnable ou abusif de l'usage d'une instruction dépend de deux facteurs : la complexité de cette instruction et la taille de l'entrée sur laquelle elle opère *relativement à la taille de l'entrée de l'algorithme*. Ainsi, pour une entrée de taille t, utiliser `i=i+1` sur un entier `i` de taille d'encodage polylogarithmique en t est raisonnable ; en effet, l'addition est de complexité linéaire en la taille des nombres additionnés, ici un polylogarithme de t. En revanche, utiliser `i=i+1` sur un entier `i` de taille d'encodage linéaire en t est abusif.
- Il peut être difficile de *prouver* qu'un usage est abusif. En effet, supposons qu'un algorithme $\mathcal{A}$ pour traiter une entrée de taille t utilise une instruction `instr` sur une entrée de taille $g(t)$. Prouver que cet usage est abusif revient à prouver qu'il n'existe aucun algorithme pour `instr` de complexité f assez faible pour que $f \circ g(t)$ soit polylogarithmique. C'est typiquement le type de problèmes que l'on étudiera à partir du Chapitre 6.

Un usage abusif d'une instruction n'est pas nécessairement problématique ; il souligne simplement que le coût de cette instruction *telle qu'elle est utilisée dans cet algorithme* doit être soigneusement examiné car l'estimer à $O(1)$ induirait une erreur assez grossière dans l'appréciation de la complexité *de l'algorithme*. Dans ce cours, on appliquera la précaution suivante :

> Tout usage abusif d'une instruction doit être commenté.

Quand l'instruction est triviale à décomposer il suffit d'en énoncer la complexité ; par exemple, si un algorithme compare deux entiers dont les représentations binaires sont de taille $\sqrt{t}$, il suffit d'énoncer que cette comparaison est de complexité $O(\sqrt{t})$. Lorsque l'instruction correspond à un "sous-algorithme", il peut être utile de l'expliciter avant d'en indiquer la complexité ; par exemple, si l'usage de l'instruction « `trier T[1..`$\sqrt{n}$`]` » semble abusif, il faut reformuler cette instruction pour expliciter l'algorithme de tri utilisé (par exemple, en « `faire un tri_fusion de T[1..`$\sqrt{n}$`]` » et expliciter que sa complexité est $O(\sqrt{n}\log n)$).

2.5 Prolongements

Le modèle de calcul de référence en théories de la calculabilité et de la complexité est la **machine de Turing**. Nous la présentons en Complément B mais ne l'utilisons pas dans ce cours, lui préférant le modèle RAM 8 bits qui lui est *polynomialement équivalent* : tout calcul qui peut être réalisé par un algorithme de complexité polynomiale sur un machine de Turing peut être réalisé par un algorithme de complexité polynomiale en RAM 8 bits, et vice-versa. Autrement dit, ces modèles peuvent se **simuler** l'un-l'autre en temps polynomial.

Une excellente référence pour approfondir le sujet de l'*architecture des ordinateurs* est l'ouvrage « From NAND to Tetris » de Nisan et Schocken [NS21]. Une illustration de la manière dont les algorithmes peuvent influencer l'architecture des ordinateurs est donnée dans l'article [DTH20] : il détaille comment la conception d'un matériel dédié à l'implémentation de l'algorithme de Smith-Waterman pour l'alignement de séquences en analyse génomique a permis de diviser par 26 000 le coût énergétique de son exécution par rapport à une implantation soigneuse sur un processeur de type `Intel E5`.

L'analyse d'algorithmes se prolonge naturellement par l'analyse des *structures de données*. C'est un sujet à part entière, que l'on ne traitera pas ici faute de temps. Un très bon ouvrage de référence sur ce sujet est le livre de Brass [Bra08], et le Complément C propose une rapide introduction au sujet.

2.6 Références bibliographiques

[Bra08] Peter Brass. *Advanced data structures*, volume 193. Cambridge University Press Cambridge, 2008.

[DTH20] William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7) :48–57, 2020.

[NS21] Noam Nisan and Shimon Schocken. *The elements of computing systems : building a modern computer from first principles*. MIT press, 2021.

À retenir.

- Le **modèle RAM** comporte une *unité mémoire* et une *unité de calcul*. Les définitions des deux variantes qu'on utilise diffèrent uniquement en la taille du mot binaire que peut stocker une case mémoire (8 bits vs fini mais non borné).
- Une **instruction** est n'importe quelle opération qui transforme des entrées décrites par un nombre fini de cases mémoire en des sorties décrites par un nombre fini de cases mémoire et déterminées uniquement par les entrées.
- Une instruction est **élémentaire** si le nombre de cases mémoire décrivant ses entrées et ses sorties est *borné*. Une instruction est **décomposable** si elle peut être réalisée par une séquence finie d'instructions élémentaires.
- Un **algorithme** est une séquence d'instructions élémentaires. La **présentation** d'un algorithme peut utiliser des instructions décomposables sous conditions (page 34) et peut s'interpréter différement en RAM 8 bits et en RAM taille arbitraire.

- La **complexité** d'un algorithme $\mathcal{A}$ est l'asymptotique, pour $t \to \infty$, du nombre d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter une entrée de taille t.
- L'usage d'une instruction dans la présentation d'un algorithme $\mathcal{A}$ est **raisonnable** si lors du traitement *par* $\mathcal{A}$ d'une entrée de taille t, cette instruction est décomposable en $O(\text{polylog } t)$ instructions élémentaires en RAM 8 bits. Un usage non-raisonnable est dit **abusif** et doit être commenté.

Chapitre 3

Méthode récursive

Cette séance présente les principes de conception d'*algorithmes récursifs* et ses trois déclinaisons les plus classiques. L'idée n'est pas de développer une théorie de la récursion, mais de donner une vision unifiée de principes que vous avez pu croiser par le passé, et de développer la pratique de la conception d'algorithmes récursifs élémentaires.

Objectifs. À l'issue de cette séance, il est attendu que vous sachiez mettre en œuvre les trois principaux « patrons » d'algorithmes récursifs : recherche dichotomique, diviser pour régner, et exploration par retour arrière.

L'analyse de complexité des algorithmes récursifs fera l'objet du chapitre suivant.

3.1 Méthodologie récursive : simplifier et déléguer

Informellement, le principe de la résolution d'un problème algorithmique par récursion se résume à *simplifier et déléguer* :

> On réduit le problème à résoudre à la résolution du *même problème* sur une ou plusieurs entrées *plus simples*, et on *délègue* ces résolutions.

La **simplicité** d'une entrée peut être mesurée par une fonction de taille, au sens de la Section 2.3, ou de manière plus *ad hoc*, l'essentiel étant de garantir la propriété suivante :

> Il n'existe pas de suite infinie $e_0, e_1, \ldots$ d'entrées distinctes où e_{i+1} est strictement plus simple que e_i pour tout $i \in \mathbb{N}$.

Une récursion ayant cette propriété est dite **bien fondée**. Le reste de cette section explicite quelques points techniques qui se cachent derrière ce principe. Les sections suivantes présentent trois mises en œuvre classiques de cette méthodologie récursive.

Les cas de base

Le principe ci-dessus assure que toute séquence de simplifications conduit à une ou plusieurs entrées insimplifiables. Pour de telles entrées, le problème doit être résolu *directement*, c'est-à-dire sans récursion.

On appelle **cas de base** d'un algorithme récursif une entrée que cet algorithme traite directement, c'est-à-dire sans déléguer la résolution d'entrées plus simples. Toute entrée insimplifiable doit être un cas de base, mais l'inverse n'est pas nécessairement le cas : s'il s'avère facile de résoudre directement toute entrée de taille ≤ 17, il n'y a pas de raison de se l'interdire. Le fait qu'une récursion est bien fondée énonce simplement que le choix des cas de base et du mécanisme de simplification assurent conjointement que toute séquence de simplifications mène à un cas de base.

Lorsque l'on *implante* un algorithme récursif il est important de traiter efficacement les cas de base ; cette étape est souvent non-triviale et spécifique à chaque problème. Du point de vue de l'*algorithmique*, détailler le traitement d'un cas de base est inutile *dès lors que cette entrée est définie par un nombre borné de cases mémoire*. En effet, sous cette hypothèse, ce traitement est une instruction élémentaire pour notre modèle de calcul (cf Chapitre 2).

> Dans ce cours, on peut fixer une constante (par exemple 42) et d'utiliser l'instruction élémentaire `traiter directement les entrées de taille au plus 42`.

Soulignons qu'une telle instruction n'est élémentaire que si la fonction de taille à laquelle elle fait référence est telle que les entrées de taille au plus 42 occupent un nombre borné de cases mémoire.

Apprendre à déléguer

Lorsque l'on conçoit un algorithme récursif, il convient de *s'interdire* de réfléchir à la manière dont les entrées simplifiées seront elle-même résolues : cette résolution doit être déléguée. Si le fait que cette délégation se fasse par une auto-référence à l'algorithme en cours de conception vous trouble, n'hésitez pas à considérer dans un premier temps que les sous-problèmes sont délégués à *des boîtes noires*.

Premier exemple

L'algorithme d'Euclide pour le calcul de *plus grand commun diviseur* (pgcd) est un exemple classique d'algorithme récursif. L'idée de départ est élémentaire :

> Pour tous $a \geq b \in \mathbb{N}$ on a $\text{pgcd}(a, b) = \text{pgcd}(a - b, b)$.

En effet, un entier c divise a et b si et seulement si c divise $a - b$ et b. L'algorithme d'Euclide consiste à simplifier l'entrée (a, b) en $(a - b, b)$ si $a \geq b$ et en $(b - a, a)$ sinon, puis à déléguer le calcul du pgcd de l'entrée simplifiée.[1] Pour voir ceci comme une simplification, on peut mesurer la « simplicité » d'une entrée (a, b) par le nombre $\min(a, b)$: plus ce nombre est petit, plus l'entrée est simple. Cette notion satisfait bien au principe qu'il n'existe pas de suite infinie d'entrées de plus en plus simples. Pour compléter l'algorithme, il convient de préciser les cas de base : si $\min(a, b) = 0$ on renvoit $\max(a, b)$.

3.2 Recherche dichotomique (binary search)

Voyons maintenant le premier patron[2] de mise en œuvre de cette méthodologie récursive :

> Un algorithme de **recherche dichotomique** est un algorithme récursif qui
>
> (i) divise l'entrée e de taille n en deux parties de tailles respectives au plus αn et βn, où $0 < \alpha, \beta < 1$ sont indépendants de n,
>
> (ii) identifie une des deux parties comme inutile à la résolution du problème, et
>
> (iii) délègue la résolution du problème sur l'autre partie.

On fixe par ailleurs, à la conception de l'algorithme, une constante t_0 arbitraire et on considère les entrées de taille au plus t_0 comme des cas de base : si la taille de l'entrée est au plus t_0, on la `traite directement`.

> Les paramètres α et β doivent être majorés par un $C < 1$ **indépendant** de n.

1. Une variante de cet algorithme simplifie (a, b) en $(a \mod b, b)$ si $a \geq b$ et en $(b \mod a, a)$ sinon.
2. Ce terme est utilisé ici dans le même sens qu'en couture, d'un modèle qu'on peut reproduire.

Exemple

Voici l'exemple canonique de problème que la recherche dichotomique traite facilement :

RECHERCHE DANS UN TABLEAU TRIÉ

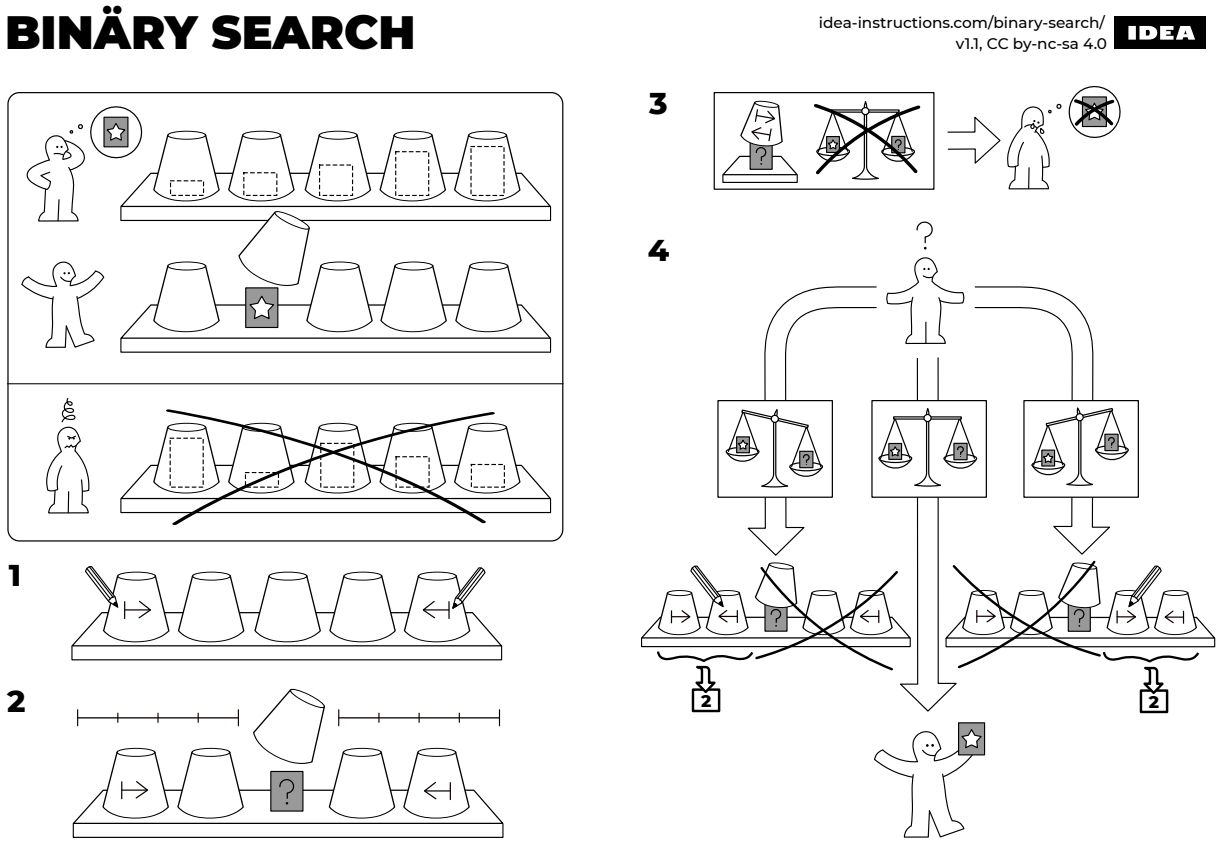
Entrée : Un tableau $T[1..n]$ d'entiers tel que $T[1] < T[2] < \ldots < T[n]$ et un entier x.

Sortie : L'indice i tel que $T[i] = x$ ou -1 si cet indice n'existe pas.

Fixons un indice p tel que $1 \leq p \leq n$. Comparer x à $T[p]$ permet de déterminer que x *n'est pas* dans une partie du tableau, soit $T[1..p]$ soit $T[p..n]$. On a donc :

```
rechercher(T[1..n],x)
   si n < 14 résoudre directement
   si x < T[n/2] retourner rechercher(T[1..n/2],x)
   sinon retourner rechercher(T[n/2..n],x)+n/2-1
```

Ici, les deux morceaux sont essentiellement disjoints ($\alpha + \beta = 1$) et de tailles égales ($\alpha = \beta$), mais aucune de ces conditions n'est nécessaire. Nous verrons d'autres exemples en exercices.



3.3 Diviser pour régner

Voici notre deuxième patron de mise en œuvre de la méthodologie récursive.

Un algorithme de type **diviser pour régner** est un algorithme récursif qui

(i) divise l'entrée donnée en entrées *plus petites* et *indépendantes* du même problème,

(ii) délègue la résolution du problème sur chacune de ces petites entrées, et

(iii) combine les solutions à chacune de ces petites entrées en une solution à l'entrée initiale.

À nouveau, les entrées dont la taille tombe en dessous d'un seuil constant, fixé arbitrairement, sont traitées directement.

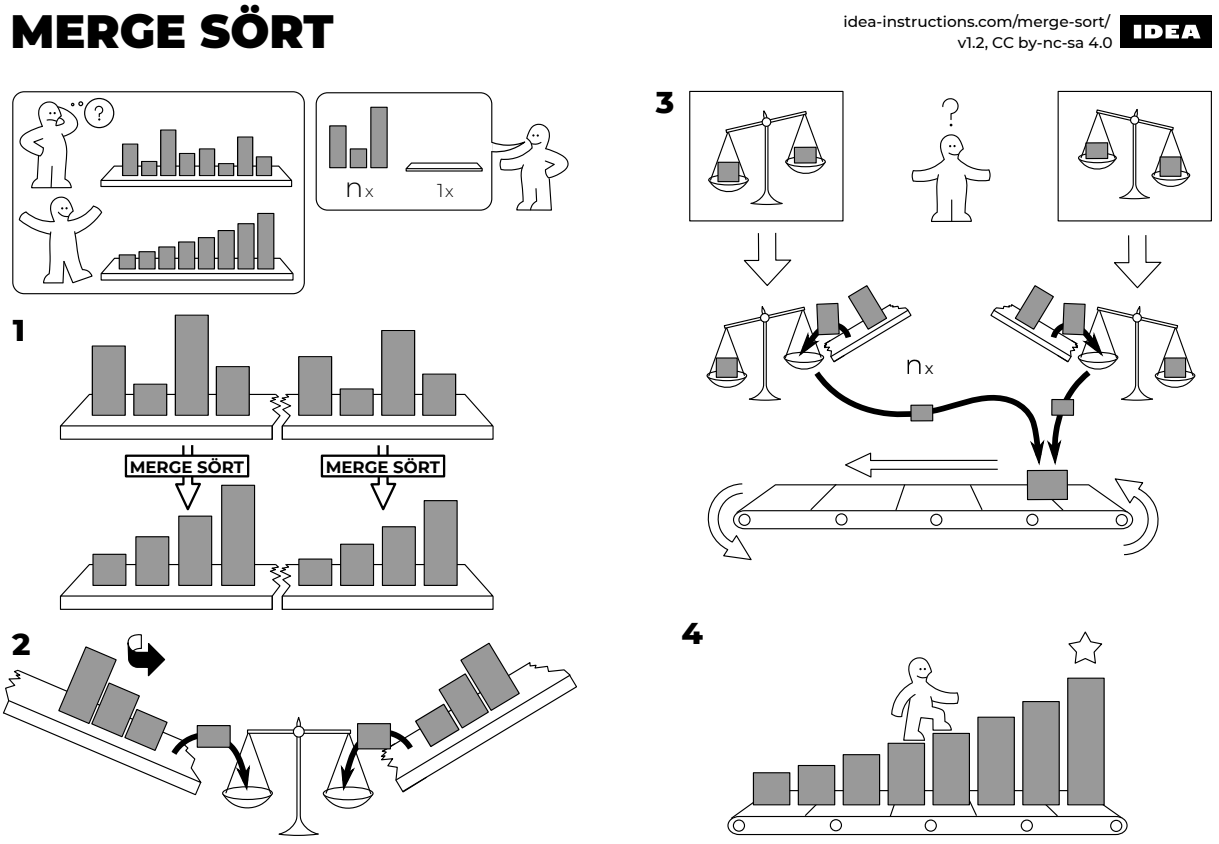
1er exemple : tri fusion

Un exemple classique de problème facile à aborder par « diviser pour régner » est le problème du tri. En voici une version :

> TRI
>
> **Entrée :** Un tableau $T[1..n]$ d'entiers distincts.
>
> **Sortie :** Un tableau $S[1..n]$ contenant les entiers de $T[1..n]$ dans l'ordre croissant.

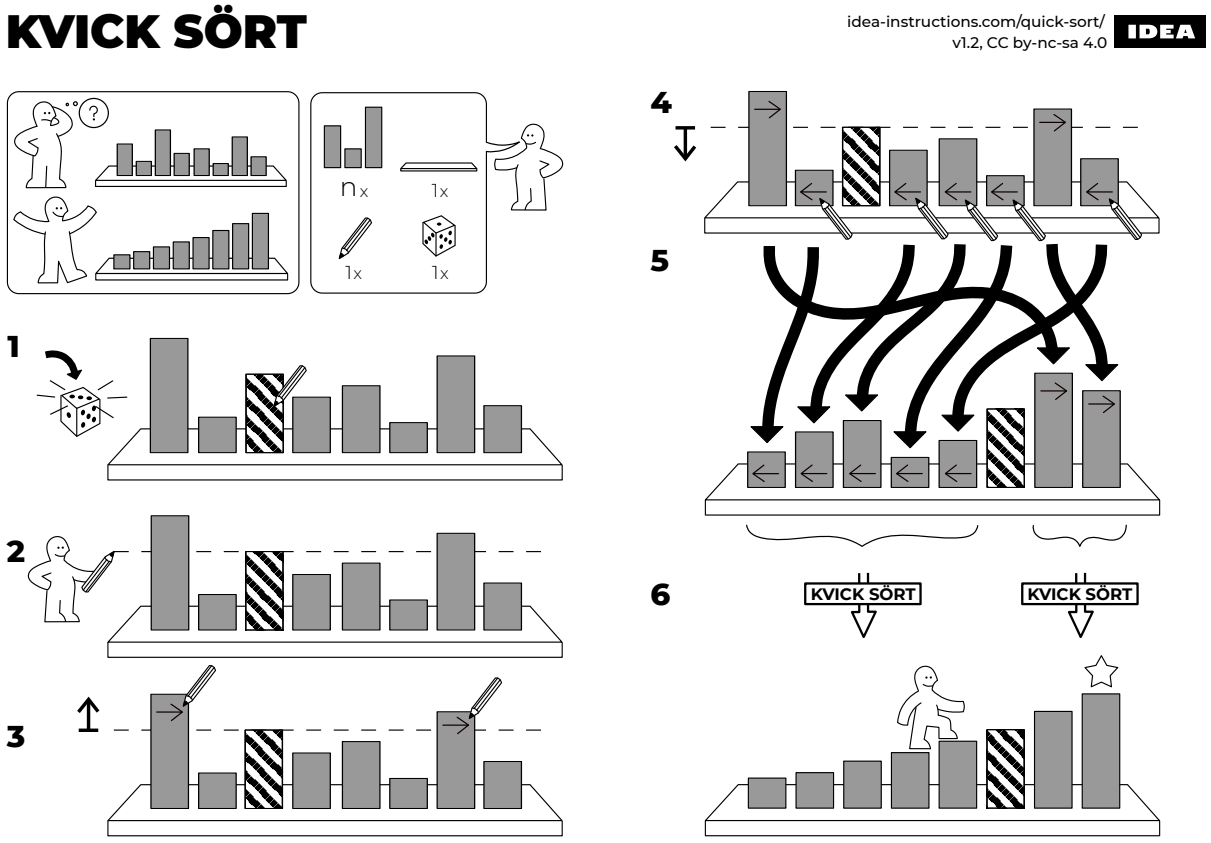
L'algorithme de *tri fusion* pose $m = n/2$ et divise l'entrée en $T[1..m]$ et $T[m+1..n]$, délègue le tri de chacune de ces parties, puis fusionne les parties triées comme suit :

```
i, j, k = 0, n/2, 0
tant que i < n/2 et j < n
   si T[i]<T[j]
     R[k] = T[i] et i = i+1
   sinon R[k] = T[j] et j = j+1
   k = k+1
si i == n/2 compléter R avec T[j..n-1]
sinon compléter R avec T[i..n/2-1]
copier R dans T
```



2ème exemple : tri rapide

Le *tri rapide* est un autre exemple classique d'algorithme « diviser pour régner ». Pour résoudre Tri, il choisit[3] un pivot p, divise $T[1..n]$ en deux parties, l'une contenant les éléments $\leq p$ et l'autre contenant les éléments $> p$, délègue le tri de ces parties, puis les concaténe.



3.4 Exploration par retour arrière (backtracking)

Notre troisième et dernier patron vise à *construire* une solution à un problème algorithmique en explorant des *séquences* de décisions élémentaires. Sa mise en œuvre suppose que l'on ait au préalable reformulé le problème algorithmique comme la recherche d'un objet défini par une séquence de décision.

Traduction d'un problème en séquence de décisions

Considérons un premier exemple venant de la théorie algorithmique des graphes. Une *clique* dans un graphe G est un ensemble de sommets C tel que toute paire d'éléments de C forme une arête dans G.

Clique

Entrée : Un graphe G à n sommets donné par sa matrice d'adjacence $A \in \{0,1\}^{n \times n}$ et un entier k.

Sortie : Vrai ou faux, G contient une clique de taille k.

On peut construire une clique C au travers d'une séquence de décisions élémentaires :

- Met on le 1er sommet dans C ?
- Met on le 2ème sommet dans C ?

3. Formellement, il s'agit plutôt d'une famille d'algorithmes, un par méthode de choix du pivot (comme pour la famille d'algorithmes du simplexe que vous verrez en recherche opérationnelle).

- ...
- Met on le $n$ième sommet dans C ?

Pour résoudre CLIQUE il suffit de tester *toutes* les combinaisons possibles de décisions élémentaires et répondre vrai si l'une de ces combinaisons donne lieu à une clique de taille k ou plus. L'exploration par retour arrière organise cette exploration systématique en évitant des combinaisons facilement détectables comme inintéressantes : il ne sert à rien d'ajouter à C un sommet qui ne forme pas une arête avec *tous* les sommets déjà dans C.

Prenons un deuxième exemple avec le problème classique suivant :

n REINES

Entrée : Un entier n.

Sortie : Vrai ou faux, est-il possible de placer n reines sur un échiquier $n \times n$ sans qu'aucune reine ne puisse en attaquer une autre ?

Autrement dit, est-il possible de choisir n cases dans une grille $n \times n$ sans que deux cases ne soient alignées horizontalement, verticalement ou diagonalement ? On peut vérifier à la main que le problème a une solution pour $n = 1$ mais pas pour $n = 2$. Une analyse de cas rapide permet de s'assurer aussi qu'il n'a pas de solution pour $n = 3$ mais en a pour $n = 4$. Dans toute solution valide, chaque ligne contient *exactement une* reine. On peut donc construire une solution valide en décidant :

- où placer une reine sur la 1ère ligne, puis
- où placer une reine sur la 2ème ligne, sans conflit avec la reine de la ligne précédent, puis
- où placer une reine sur la 3ème ligne, sans conflit avec les reines des lignes précédentes, puis
- ...
- où placer une reine sur la $n$ième ligne, sans conflit avec les reines des lignes précédentes.

L'exploration par retour arrière permet d'organiser l'exploration de ces combinaisons de décisions.

Le patron

Supposons que l'on ait formulé un problème algorithmique comme l'exploration des combinaisons possibles d'une séquence de décision (comme pour CLIQUE ou n-REINES ci-dessus). Voici le patron d'exploration récursive de ces combinaisons :

Un algorithme d'**exploration par retour arrière** prend en entrée une séquence de décisions élémentaires *déjà prises*, et :

(i) Identifie une décision élémentaire restant à trancher ; s'il n'en existe pas, il retourne que la recherche a réussi.

(ii) Liste les choix possibles pour la décision élémentaire identifiée ; s'il n'en existe pas, il retourne que la recherche a échoué.

(iii) Délégue récursivement l'exploration de *chacun* de ces choix ; si au moins une des recherches déléguées réussi, il retourne que la recherche a réussi, sinon il retourne qu'elle a échoué.

Soulignons qu'à l'étape (ii), on peut ne lister que les choix dont l'exploration a un intérêt (par exemple, pour CLIQUE, on peut n'envisager d'ajouter un sommet à C que s'il forme une arête avec tous les sommets déjà dans C).

Exemples

Pour mettre en œuvre ce patron, il reste à préciser la manière dont on identifie les choix intéressants au (ii) et on transmet la liste des décisions élémentaires déjà prises au (iii).

Pour CLIQUE, on peut par exemple transmettre un tableau $C[1..n]$ dont l'entrée $C[i]$ vaut 1 si le $i$ème sommet a déjà été ajouté à C et 0 sinon. Cette démarche conduit à l'algorithme suivant (r est le numéro du sommet à examiner pour la prochaine décision) :

```
clique(G,k,C[1..n],r)
   si r > n et C[1..n] contient au moins k '1', retourner Vrai
   si {r,i} est une arête dans G pour tout i<r tel que C[i]=1
     mettre C[r] à 1
     si clique(G,k,C[1..n],r+1) est vrai retourner vrai
   mettre C[r] à 0
   si clique(G,k,C[1..n],r+1) est vrai retourner vrai
   retourner Faux
```

Pour n REINES, on peut par exemple transmettre un tableau $Q[1..n]$ dont l'entrée $Q[i]$ indique le numéro de la colonne contenant la reine de la i-ème ligne (et, disons, -1 si cette décision n'a pas encore été prise). Ainsi, après un appel récursif il y a une reine, placée en $(1, Q[1])$, après deux appels récursifs il y en a deux en $(1, Q[1])$ et $(2, Q[2])$, etc. Cette démarche conduit à l'algorithme suivant (r est le numéro de la ligne de la prochaine décision) :

```
reines(Q[1..n],r)
   si r > n retourner Vrai
   pour j=1..n
     si (r,j) n'est en conflit avec aucun (i,Q[i]) pour i=1..r-1
       Q[r] = j
       si reines(Q[1..n],r+1):
         retourner Vrai
   retourner Faux
```

Commentaires

Plusieurs décompositions en décision élémentaire sont généralement envisageables pour un problème donné. Différentes décompositions peuvent conduire à des algorithmes plus ou moins compliqués, et plus ou moins efficaces. Ainsi, pour n REINES, on pourrait prendre comme r-ième décision la position de la r-ième reine sans contrainte de ligne. Adopter cette notion ouvrirait plus de choix possibles et augmenterait donc le nombre d'appels récursifs à faire à chaque étape.

Les décisions élémentaires sont, de fait, traitées dans un certain *ordre* ; par exemple, l'algorithme proposé pour n REINES place les reines par ordre croissant de numéro de ligne sur l'échiquier. Cet ordre peut être induit par le problème ou être fixé arbitrairement.

Les sous-problèmes délégués récursivement *dépendent généralement* des choix effectués au cours des appels précédents. Il convient donc de transmettre à chaque appel récursif un **résumé des décisions prises**. Ainsi, à mesure que le nombre de décisions restant à prendre diminue, la quantité d'information passée récursivement augmente.

De nombreux problèmes admettent des variations, notamment des versions *décision* (« existe-t-il un mouton à 5 pattes ? »), *comptage* (« combien existe-t-il de moutons à 5 pattes ? ») et *énumération* (« lister tous les moutons à 5 pattes qui existent. »). L'exploration par retour arrière est souvent robuste au sens où un algorithme résolvant une variante peut facilement être adapté pour résoudre les autres variantes. Une bonne pratique consiste donc à résoudre d'abord la variante *la plus simple possible* d'un problème, qui est souvent la version *décision*.

3.5 Prolongements

La **programmation dynamique** est une autre méthodologie récursive, importante de par son efficacité et ses très nombreuses applications. Elle met l'accent sur l'évitement de la répétition de calculs redondants lors des appels récursifs, au travers notamment de la mémoisation de calculs intermédiaires. Nous en verrons un exemple avec l'algorithme de Floyd-Warshall au Chapitre 5.

La méthode récursive s'applique naturellement aux structures de données récursives, dont la définition guide la manière de décomposer l'entrée en parties. Par exemple, pour déterminer la hauteur

d'un arbre, il suffit de déterminer la hauteur de chacun de ses sous-arbres (dont on délègue le calcul), puis d'ajouter 1 au maximum.

Chercher à résoudre un problème par la méthode récursive (*simplifier et déléguer*) est une bonne manière de mettre au point un algorithme de bonne complexité. Cela ne tient pas au fait que l'algorithme est formulé récursivement, mais au fait que la méthode récursive incite à trouver une manière de *simplifier* efficacement l'entrée. Une question naturelle est : dans quelle mesure une amélioration de l'efficacité de la simplification se traduit en un algorithme de meilleure complexité globale ? Ce sera l'enjeu du Chapitre 4.

Certains algorithmes récursifs sont facilement « dérécursifiables », au sens où l'idée principale de l'algorithme (la simplification) est facilement répétable sans appel récursif (délégation). Par exemple, une recherche dichotomique dans un tableau peut se faire comme suit :

```
rechercher(T[1..n],x)
   deb,fin = 1,n
   tant que fin-deb > 14
      si x < T[(deb+fin)/2]
         fin = (deb+fin)/2
      sinon deb = (deb+fin)/2
   résoudre directement sur T[deb,fin]
```

Opérer une telle dérécursification lors de l'implantation d'un algorithme peut améliorer les performances pratiques du code produit, notamment parce que tout appel de fonction induit généralement des coûts cachés (par exemple la sauvegarde du contexte). Cela reste cependant sans conséquence sur la complexité de l'algorithme, aussi on ne s'y intéresse pas dans ce cours.

À retenir.

- Un algorithme récursif est avant tout une **idée de simplification**. Cette simplification produit des problèmes plus simples dont on **délégue** la résolution (récursivement).
- Il est inutile (voire nuisible) de réfléchir à comment les problèmes délégués sont résolus.

- Tout algorithme récursif identifie un ensemble d'entrées à traiter directement. Ce sont ses **cas de base**. Les cas de base sont définis par un nombre borné de cases mémoire peuvent être traités en une seule instruction élémentaire : `traiter directement`.
- L'idée de simplification doit assurer qu'il n'existe pas de suite infinie d'entrées chacune plus simple que la précédente. Autrement dit, partant de n'importe quelle entrée, on doit aboutir à un cas de base après un nombre fini de simplifications.
- La **dichotomie** est une méthode récursive qui coupe l'entrée en deux parties, chacune de taille linéaire, et identifie une partie inutile à la résolution du problème.
- Le **diviser-pour-régner** est une méthode récursive qui coupe l'entrée en parties et construit la solution au problème sur l'entrée à partir des solutions au problème sur les parties (que l'on délègue).
- Le **backtracking** est une méthode récursive qui explore les combinaisons possibles pour une séquence de décisions élémentaires. L'algorithme choisit une décision élémentaire et, pour chaque choix possible, délègue l'exploration des combinaisons compatible avec ce choix ; au fil des délégations, le nombre de décisions élémentaires restant à prendre diminue.

Chapitre 4

Complexité asymptotique pire-cas d'algorithmes récursifs

La fonction de complexité asymptotique pire-cas d'un algorithme récursif peut généralement s'étudier au travers d'une récurrence. Si l'algorithme $\mathcal{A}$ traite une entrée de taille n en déléguant la résolution de k sous-problèmes, chacun de taille αn, puis en les recombinant en temps $O(n^d)$, sa complexité asymptotique pire cas $T(n)$ satisfait

$$T(n) = kT(\alpha n) + O(n^d).$$

Nous allons voir que l'analyse de ce type de récurrence peut s'automatiser. Soulignons que les outils présentés dans ce chapitre s'appliquent indifféremment dans nos deux modèles (RAM 8 bits et RAM taille arbitraire).

Objectifs. À l'issue de cette séance, il est attendu que vous sachiez

- utiliser des arbres enracinés pour modéliser l'exécution d'algorithmes,
- construire des bornes inférieures pour des algorithmes simples,
- déterminer la complexité asymptotique d'un algorithme récursif par application du théorème maître lorsque c'est possible, et par analyse d'arbre d'exécution sinon,
- utiliser le théorème maître pour guider la conception d'algorithmes récursifs de type *diviser-pour-régner*.

4.1 Préliminaires : rappels sur les arbres

Commençons par quelques rappels sur arbres. Dans ce cours, nous envisageons les arbres comme des structures mathématiques discrètes et nous nous en servons comme des outils de raisonnement abstrait.[1] Nous en définissons ici deux variantes mais utilisons principalement la seconde. Chacun des encadrés suivants énonce un principe, que l'on développe dans le paragraphe qui suit.

Un **arbre** est un **graphe** dans lequel toute paire de sommets est reliée par un unique chemin simple.

Détails : étant donné un ensemble V notons $\binom{V}{2}$ l'ensemble des paires (non-ordonnées, donc) d'éléments de V. Un **graphe**[2] est un couple (V, E) où V est un ensemble arbitraire et $E \subseteq \binom{V}{2}$. Un **chemin** (entre a et b) dans un graphe (V, E) est une suite $v_0, v_1, \ldots, v_\ell$ de sommets tels que $\{v_{i-1}, v_i\} \in E$ pour $i = 1, 2, \ldots \ell$ (et $\{v_0, v_\ell\} = \{a, b\}$). Un chemin est **simple** si les sommets qui le composent sont deux à deux distincts.

1. Les arbres sont traités et utilisés sous divers points de vue. En particulier, soulignons qu'ici un arbre *n'est pas* une structure de donnée.
2. Un graphe est parfois appelé **graphe non orienté**.

Un **arbre enraciné** est un **graphe orienté** ayant une unique source r et dans lequel pour tout sommet $v \neq r$ il existe un unique chemin de r à v.

Détails : étant donné un ensemble V notons $\Delta_V \stackrel{\text{def}}{=} (V \times V) \setminus \{(v,v) \colon v \in V\}$ l'ensemble des paires d'éléments de V distincts. Un **graphe orienté** est un couple $(V, \vec{E})$ où V est un ensemble arbitraire et $\vec{E} \subseteq \Delta_V$. Une **source** dans un graphe orienté $(V, \vec{E})$ est un élément $s \in V$ tel que $\vec{E} \cap \{(v,s) \colon v \in V\} = \emptyset$.

On peut **enraciner** un arbre en n'importe lequel de ses sommets pour définir un arbre enraciné.

Détails : Fixons un arbre $A = (V, E)$ et choisissons un sommet $r \in V$. Notons $u \prec v$ si et seulement si u appartient au chemin simple de v à r dans A. Remarquons que si $\{u, v\} \in E$ alors[3] soit $u \prec v$, soit $v \prec u$ et on ne peut pas[4] avoir $u \prec v$ et $v \prec u$. Ainsi, en posant

$$\vec{E} \stackrel{\text{def}}{=} \{(u,v) \colon \{u,v\} \in E \text{ et } u \prec v\}$$

on obtient un graphe orienté $(V, \vec{E})$ qui est un arbre enraciné de source r ; on l'appelle **enracinement de (V, E) en r**. Réciproquement, à tout arbre enraciné $(V, \vec{E})$ on peut associer l'arbre (V, E), appelé **arbre sous-jacent** où

$$E \stackrel{\text{def}}{=} \{\{u,v\} \colon (u,v) \in \vec{E}\}.$$

Soulignons que E est un ensemble et ne tient pas compte de multiplicités. Si $A = (V, E)$ est un arbre et $r \in V$, enraciner A en r produit un arbre enraciné de source r et d'arbre sous-jacent A. Inversement, si $\vec{A} = (V, \vec{E})$ est un arbre enraciné de source $r \in V$, prendre l'arbre A sous-jacent et l'enraciner en r produit l'arbre enraciné initial $\vec{A}$.

Les arbres enracinés ont une terminologie spécifique : ascendant, descendant, racine, feuille, degré, hauteur, ...

Précisons d'abord quelques termes de théorie des graphes.[5] Si $G = (V, E)$ est un graphe, alors un élément $\{a, b\} \in E$ est appelé une **arête** de G. Si $\vec{G} = (V, \vec{E})$ est un graphe orienté, alors un élément $(a, b) \in \vec{E}$ est appelé une **arête orientée** de G. On écrit $a \to b$ pour exprimer que (a, b) est une arête orientée. La **longueur** d'un chemin $s_0, s_1, \ldots, s_\ell$ dans un graphe (orienté) égale ℓ, c'est-à-dire le nombre d'arêtes (orientées) du chemin.

Venons-en aux arbres enracinés. Un élément de V est appelé un **nœud**. La **racine** d'un arbre enraciné est son nœud source. Si $u \to v$ on dit que u est un **ascendant** de v et v un **descendant** de u. Ainsi, dans un arbre enraciné la racine n'a aucun ascendant et tout autre nœud en a exactement un.[6] Une **feuille** est un nœud sans descendant et un nœud est **interne** s'il a au moins un descendant. Le **degré d'un nœud** est son nombre de descendants[7] et l'**arité** d'un arbre enraciné est le degré maximum d'un de ses nœuds. La **hauteur d'un nœud** est la longueur du chemin de la racine à ce nœud ; ainsi la racine à hauteur 0, ses descendants ont hauteur 1, leurs descendants ont hauteur 2, etc. La **hauteur d'un arbre enraciné** est la hauteur maximale d'un de ses nœuds.

Un arbre d'arité k et de hauteur $h \geq 0$ a moins de k^{h+1} nœuds et au plus k^h feuilles.

Cette propriété se prouve facilement en fixant k et en procédant par récurrence sur h. De nombreuses autres propriétés se prouvent similairement. Par exemple, si dans un arbre enraciné tous les nœuds interne ont degré 2, le nombre de feuilles égale le nombre de nœuds internes plus 1.

3. En effet, supposons $u \not\prec v$ et notons $v = v_0, v_1, \ldots, v_\ell = r$ l'unique chemin simple de v à r. Alors $u, v, v_1, v_2, \ldots, v_\ell = r$ est un chemin dans A de u à r et il est simple. Par conséquent, $v \prec u$.

4. Pour le prouver, supposer que c'est le cas et montrer qu'il existe deux chemins simples distincts de u à r dans A.

5. En cas de besoin, la page https://en.wikipedia.org/wiki/Glossary_of_graph_theory est un bon point de départ.

6. Pour prouver cela, montrer que u est un ascendant de v si et seulement si $u \to v$ est la dernière arête de tout chemin simple de r à v.

7. En particulier, le degré d'un nœud dans un arbre enraciné est différent de son degré dans l'arbre sous-jacent, puisque ce dernier désigne son nombre de voisins, ce qui inclut l'ascendant.

La classe des arbres enracinés peut être définie récursivement.

Détails : Soient $A_1 = (V_1, \vec{E}_1)$, $A_2 = (V_2, \vec{E}_2)$, …, $A_k = (V_k, \vec{E}_k)$ des arbres enracinés d'ensembles de nœuds disjoints et de racines respectives v_1, v_2, …, v_k. Pour $v_0 \notin V_1 \cup V_2 \cup \ldots \cup V_k$ on note $A' \stackrel{\text{def}}{=} \star_{v_0}(A_1, A_2, \ldots, A_k)$ l'arbre enraciné $(V', \vec{E}')$ avec

$$V' \stackrel{\text{def}}{=} \{v_0\} \cup V_1 \cup V_2 \cup \ldots \cup V_k,$$
$$\vec{E}' \stackrel{\text{def}}{=} \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_k)\} \cup \vec{E}_1 \cup \vec{E}_2 \cup \ldots \cup \vec{E}_k.$$

Le nœud v_0 est la racine de A' et a chacun des v_i comme descendant.

Pour un ensemble V donné, notons $A(V)$ l'ensemble des arbres enracinés à nœuds dans V. On peut définir $A(V)$ de manière récursive, comme le plus petit ensemble qui contient :

- $(\{v\}, \emptyset)$ pour tout $v \in V$, et
- $\star_{v_0}(A_1, A_2, \ldots, A_k)$ pour tous $k \geq 1$, $v_0 \in V$ et $A_1, A_2, \ldots, A_k \in A(V)$ d'ensembles de nœuds disjoints et ne contenant pas v_0.

Avec cette définition, tout arbre enraciné $A \in A(V)$ est soit un seul sommet, soit $\star_{v_0}(A_1, A_2, \ldots, A_k)$. Dans le second cas, $A_1, A_2, \ldots, A_k$ son les **sous-arbres de la racine** de A (cette racine étant v_0).

Les arbres (enracinés) peuvent être **représentés graphiquement**, leurs nœuds et leurs arêtes peuvent être **étiquetés**, etc.

4.2 Borne inférieure sur la complexité d'un algorithme

Comment peut-on s'assurer qu'une majoration établie sur la complexité d'un algorithme ne peut pas être améliorée ?

Un premier exemple

Commençons par examiner cela sur l'exemple de l'algorithme de `tri_rapide` (introduit au Chapitre 3), que l'on considère ici dans le modèle RAM taille arbitraire (mais notre raisonnement s'applique aussi bien au modèle RAM 8 bits).

L'algorithme de `tri_rapide` dépend d'un choix de pivot. On suppose ici que le pivot est le *premier* élément du tableau :

```
tri_rapide(T[1..n]):
   p = T[1]
   A[1..a] = éléments de T[1..n] inférieurs à p
   B[1..b] = éléments de T[1..n] égaux à p
   C[1..c] = éléments de T[1..n] supérieurs à p
   quicksort(A[1..a])
   quicksort(C[1..c])
   recoller les tableaux A, B et C
   retourner le tableau ainsi obtenu
```

Notons $f(n)$ le nombre d'instructions élémentaires exécutées par `tri_rapide`, avec ce choix de pivot, sur l'entrée $T[1..n] = [1, 2, \ldots, n]$, c'est-à-dire une entrée de taille n déjà triée.[8] Soulignons que $f(n)$ *minore* la complexité (asymptotique pire-cas) de l'algorithme. La subdivision de $T[1..n]$ en $A[]$, $B[]$ et $C[]$ examine les n cases, ce qui requiert au moins Kn instructions élémentaires, pour une certaine

8. Comme `tri_rapide` ne manipule l'entrée qu'au travers de copies et de comparaisons entre éléments, les séquences d'instructions exécutées sur deux entrées *déjà triées* sont rigoureusement identiques. Plus généralement, la séquence d'instructions exécutées par `tri_rapide` ne dépend que de la permutation qu'il convient d'appliquer à l'entrée pour la trier.

constante K.[9] Les tableaux $A[]$, $B[]$ et $C[]$ qu'elle produit sont de tailles $a = 0$, $b = 1$ et $c = n - 1$ et, de plus, $C[]$ est trié. On a ainsi

$$f(n) \geq f(n-1) + K.n \qquad \text{et donc} \qquad f(n) \geq K\frac{n(n+1)}{2}.$$

La complexité du `tri_rapide` est donc $\Omega(n^2)$, c'est-à-dire « au moins quadratique ». Il est assez facile de prouver que l'algorithme de `tri_rapide` avec le premier élément comme pivot est de complexité $O(n^2)$. Cette borne quadratique ne peut donc pas être améliorée.

Formalisation

Plus généralement, on peut minorer la complexité pire-cas d'un algorithme $\mathcal{A}$ en examinant son comportement sur une suite d'entrées de tailles strictement croissantes. On suppose fixés un modèle de calcul (par exemple RAM 8 bits) et une fonction de taille $e \mapsto t(e)$.

> La **complexité d'un algorithme** $\mathcal{A}$ est $\Omega(f(t))$ s'il existe une suite infinie d'entrées $e_1, e_2, \ldots$ telle que (i) $\lim_{i\to\infty} t(e_i) = +\infty$, et (ii) pour traiter e_j, l'algorithme $\mathcal{A}$ exécute au moins $K \cdot f(t(e_j))$ instructions élémentaires, où K est indépendant de j.

Une telle minoration de la croissance asymptotique de la complexité pire-cas de $\mathcal{A}$ est appellée une **borne inférieure** sur la complexité de l'algorithme $\mathcal{A}$; on parle de *borne inférieure pour* $\mathcal{A}$.

Lorsque l'on a établi des bornes supérieure et inférieure égales sur la complexité d'un algorithme, on dit que cette borne de complexité est **fine**. Une borne de complexité fine s'exprime par un $\Theta()$ et signifie que le travail d'analyse de complexité pour *cet* algorithme est terminé.

4.3 Arbres d'exécution

On commence par montrer comment l'exécution d'un algorithme récursif dans un modèle de calcul peut être modélisée par un arbre enraciné. Cela nous permet notamment d'en calculer la complexité.

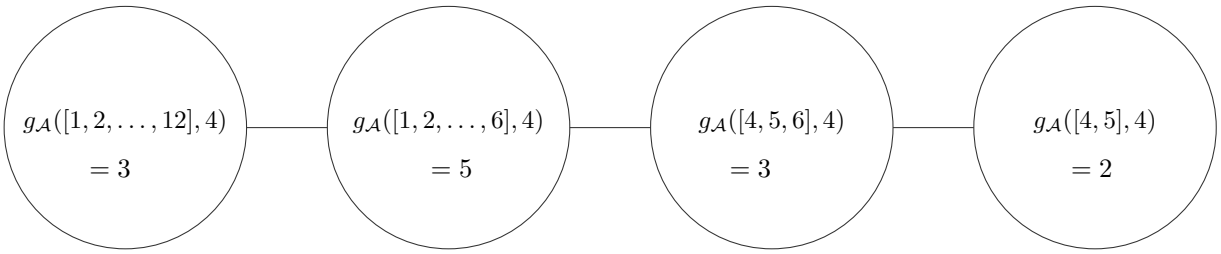
4.3.1 Arbre d'exécution d'une entrée

Considérons un algorithme récursif $\mathcal{A}$ et un modèle (RAM 8 bits ou RAM taille arbitraire). Pour toute entrée e de $\mathcal{A}$, notons $g_{\mathcal{A}}(e)$ le nombre d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter e **hors appels récursifs**. On modélise l'exécution de $\mathcal{A}$ sur e par un arbre enraciné $\mathcal{T}_{\mathcal{A}}(e)$ défini récursivement comme suit :

- si e est un cas de base pour $\mathcal{A}$, alors $\mathcal{T}_{\mathcal{A}}(e)$ a un seul nœud étiqueté $g_{\mathcal{A}}(e)$,
- sinon la racine de $\mathcal{T}_{\mathcal{A}}(e)$ est étiquetée par $g_{\mathcal{A}}(e)$ et ses sous-arbres sont $\mathcal{T}_{\mathcal{A}}(e_1)$, $\mathcal{T}_{\mathcal{A}}(e_2)$, …, $\mathcal{T}_{\mathcal{A}}(e_k)$ où $e_1, e_2, \ldots, e_k$ sont les entrées dont le traitement est délégué récursivement lors du traitement de e.

Illustrons cela sur deux exemples du Chapitre 3 : l'algorithme `rechercher` de recherche dichotomique d'un élément dans un tableau trié (avec les entrées de taille ≤ 2 comme cas de base) et l'algorithme `tri fusion` de tri d'un tableau d'entiers par fusion. On examine ces exemples en RAM taille arbitraire.

Dans l'algorithme `rechercher`, la récursion donne lieu à un seul appel récursif, donc chaque nœud interne de $\mathcal{T}_{\texttt{rechercher}}(e)$ a un descendant : c'est un *chemin*. Le coût $g_{\texttt{rechercher}}(e)$ égale 2 lorsque e est un cas de base (le `si...` et le `résoudre...`) et 3 ou 5 sinon (selon la branche prise). La longueur du ce chemin dépend de l'entrée. Pour l'entrée $T[1..12] = [1, 2, \ldots, 12]$ et $x = 4$ on obtient le chemin :



9. On utilise ici qu'une instruction élémentaire examine $O(1)$ cases.

Dans l'algorithme `tri_fusion`, hors cas de base, le traitement d'une entrée donne systématiquement lieu à deux appels récursifs. Chaque nœud interne de $\mathcal{T}_{\texttt{tri\_fusion}}(e)$ est donc de degré 2 : c'est un *arbre (enraciné) binaire* . Chaque appel récursif divise la taille de l'entrée par ≈ 2, aussi l'arbre est de hauteur $\Theta(\log_2 t(e))$. La fonction $g_{\texttt{tri\_fusion}}(e)$ est, elle, plus laborieuse à expliciter.

4.3.2 Arbre d'exécution d'un algorithme

Les arbres que l'on vient de définir constituent une application de l'ensemble des entrées de l'algorithme dans l'ensemble des arbres (à nœuds étiquetés par $\mathbb{N}$). Comme pour la fonction de complexité, il est souhaitable de simplifier cet objet en (i) fixant une fonction de taille, puis (ii) agrégeant les entrées en paquets en considérant le pire cas parmi les entrées d'une taille t donnée, et enfin (iii) en passant d'un comptage précis des instructions élémentaires à une estimation asymptotique pour $t \to \infty$.

Commençons par la simplification (ii), et pour cela, fixons une convention : dans la description de $\mathcal{T}_{\mathcal{A}}(e)$, les sous-arbres $\mathcal{T}_{\mathcal{A}}(e_1)$, $\mathcal{T}_{\mathcal{A}}(e_2)$, …, $\mathcal{T}_{\mathcal{A}}(e_k)$ sont ordonnés par *tailles* $t(e_i)$ *décroissantes*. On suppose de plus qu'il existe une constante n_0 telle que $\mathcal{A}$ traite toute entrée de taille $n \leq n_0$ *sans* appel récursif (ce sont des cas de base). On peut dès lors définir, pour tout entier n, un arbre $\mathcal{T}_{\mathcal{A}}(n)$ comme suit :

- si $n \leq n_0$ alors $\mathcal{T}_{\mathcal{A}}(n)$ a un seul nœud étiqueté par $\max\{g_{\mathcal{A}}(e) \colon e \text{ entrée de taille } n\}$,
- sinon la racine de $\mathcal{T}_{\mathcal{A}}(n)$ est étiquetée par $\max\{g_{\mathcal{A}}(e) \colon e \text{ entrée telle que } t(e) = n\}$ et ses sous-arbres sont $\mathcal{T}_{\mathcal{A}}(n_1)$, $\mathcal{T}_{\mathcal{A}}(n_2)$, …, $\mathcal{T}_{\mathcal{A}}(n_k)$ où [10] $n_i = \max\{t(e_i) \colon e \text{ entrée de taille } n\}$.

Pour la simplification (iii), on remplace, dans l'étiquetage, les fonctions précises $\max\{g_{\mathcal{A}}(\cdot) \colon \ldots\}$ par leur comportement asymptotique, que l'on note $g_{\mathcal{A}}(n)$.

$$\mathcal{T}_{\mathcal{A}}(n) = \begin{cases} O(1) & \text{si } n \leq n_0 \\ g_{\mathcal{A}}(n) \text{ avec les sous-arbres } \mathcal{T}_{\mathcal{A}}(n_1),\ \mathcal{T}_{\mathcal{A}}(n_2),\ \ldots\ \mathcal{T}_{\mathcal{A}}(n_k) & \text{sinon} \end{cases}$$

L'arbre [11] $\mathcal{T}_{\mathcal{A}}(n)$ ainsi obtenu est appelé **arbre d'exécution** de l'algorithme $\mathcal{A}$.

Premiers exemples

Reprenons nos deux exemples, l'algorithme `rechercher` de recherche dichotomique d'un élément dans un tableau trié de taille et l'algorithme `tri fusion` de tri d'un tableau d'entiers. On se place à nouveau ici en RAM taille arbitraire (mais le raisonnement serait similaire en RAM 8 bits).

Pour l'algorithme `rechercher`, quelque soit l'entrée, la récursion se fait par un seul appel, aussi $\mathcal{T}_{\texttt{rechercher}}(n)$ est un chemin. Chaque appel récursif divise la taille de l'entrée considérée par ≈ 2 ; ainsi, après p appels cette taille est $\approx n/2^p$. Le nombre d'appels nécessaires à atteindre la taille n_0 d'un cas de base est donné par $n/2^p \approx n_0$, soit $p \approx \log_2(n/n_0) = O(\log n)$. Ainsi, $\mathcal{T}_{\texttt{rechercher}}(n)$ est un chemin de longueur $O(\log n)$. À chaque étape de la récursion, le coût de simplification est $O(1)$. Chaque nœud de ce chemin est donc étiqueté $O(1)$.

Pour l'algorithme `tri fusion`, l'entrée (un tableau de taille n) est coupée en deux parties (des tableaux de taille $n/2$), dont le tri est délégué récursivement. Ainsi, $\mathcal{T}_{\texttt{tri\_fusion}}(n)$ est un arbre binaire. Sa profondeur est $O(\log n)$, pour la même raison que pour l'algorithme `rechercher`. Quand au coût de simplification $g_{\texttt{tri\_fusion}}(n)$, il vaut $O(n)$: on peut réaliser la division en $O(1)$ mais la fusion des sous-tableaux triés coûte $O(n)$.

10. C'est là qu'on utilise la convention que les entrées $e_1, e_2, \ldots, e_k$ dont le traitement est délégué par $\mathcal{A}$ sont ordonnées par taille d'entrée e_i décroissantes.

11. Formellement, c'est une famille d'arbres paramétrée par n, *i.e.* une fonction de $\mathbb{N}$ dans l'ensemble des arbres.

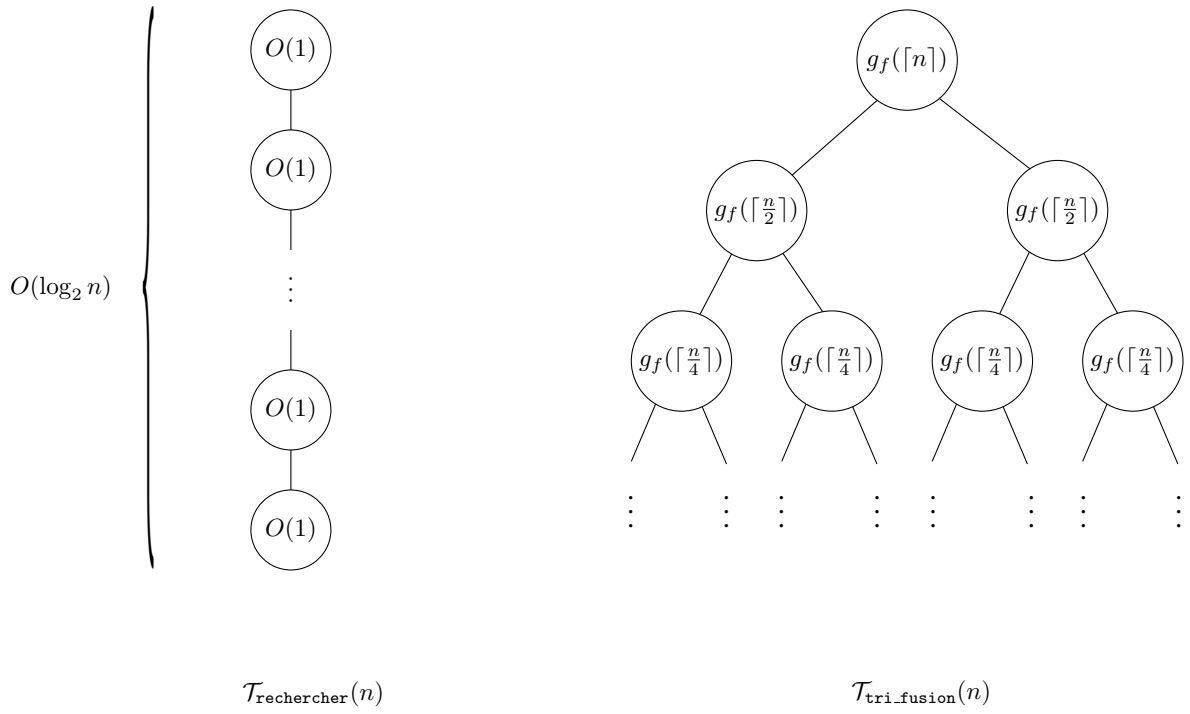


FIGURE 4.1 – Arbres d'exécution pour la recherche dichotomique (à droite) et le tri fusion (à gauche).

Ces deux arbres sont présentés Figure 4.1. Dans ces deux exemples, on peut facilement se convaincre que pour tout n, il existe une entrée e_n de taille n dont l'arbre d'exécution $\mathcal{T}(e_n)$ est proche de l'arbre d'exécution agrégé $\mathcal{T}(n)$. Ce n'est pas le cas pour l'algorithme `tri_rapide`, lui aussi vu au Chapitre 3. Quelque soit l'entrée (un tableau de taille n), l'algorithme la coupe en deux parties dont on délègue le tri ; l'arbre $\mathcal{T}_{\texttt{tri\_rapide}}(n)$ est donc binaire. Cependant, les tailles des parties dépendent du choix du pivot. La taille de la plus grande partie est, au pire, $n-1$ et la taille de la petite partie est au pire $n/2$.[12] Ainsi, $\mathcal{T}_{\texttt{tri\_rapide}}(n)$ est de hauteur $O(n)$. Quand au coût de simplification $g_{\texttt{tri\_rapide}}(n)$, il vaut $O(n)$ puisque chaque élément du tableau d'entrée doit être comparé au pivot.

4.4 De l'arbre d'exécution à la complexité

Dans de nombreux cas, on peut facilement déterminer la complexité d'un algorithme à partir de son arbre d'exécution.

L'intuition

Notons $\text{val}(v)$ l'étiquette du nœud v, ce que l'on appelle aussi sa **valeur**. La définition des arbres d'exécution assure que la complexité pire cas de $\mathcal{A}$ sur une entrée de taille n est majorée par la somme des valeurs des nœuds de $\mathcal{T}_{\mathcal{A}}(n)$:

$$C_{\mathcal{A}}(n) = O\left(\sum_{v \text{ nœud de } \mathcal{T}_{\mathcal{A}}(n)} \text{val}(v)\right). \tag{4.1}$$

On peut donc déduire facilement la complexité des algorithmes dont on a décrit les arbres d'exécution :

- $\mathcal{T}_{\texttt{rechercher}}(n)$ a $O(\log n)$ nœuds, chacun de valeur $O(1)$, d'où $C_{\texttt{rechercher}}(n) = O(\log n)$.

- $\mathcal{T}_{\texttt{tri\_fusion}}(n)$ a des nœuds de profondeur 0 à $h = O(\log n)$, où h est la hauteur de l'arbre. Il y a 2^p nœuds de profondeur p (cela se prouve par récurrence) et chaque nœud de profondeur p a

12. Soulignons que ces deux pire-cas ne peuvent se produire pour la même entrée. C'est précisément ce qui fait que l'arbre d'exécution de l'algorithme rendra mal compte des arbres d'exécution des entrées.

valeur $g_{\texttt{tri\_fusion}}\left(\lceil \frac{n}{2^p} \rceil\right) = O\left(\frac{n}{2^p}\right)$. On en déduit :

$$C_{\texttt{tri\_fusion}}(n) = O\left(\sum_{p=0}^{O(\log n)} 2^p O\left(\frac{n}{2^p}\right)\right) = O\left(n \log n\right). \tag{4.2}$$

Le calcul de l'équation (4.2) opère de manière abusive sur les $O()$, mais peut se formaliser proprement en remplaçant les $O(x)$ par des majorations de la forme $\leq C \cdot x$ où C est une constante **indépendante de n et de p**. Dans ce cours, vous pouvez utiliser ce type de présentation *dans la mesure où vous savez la formaliser.* [13]

Le cas des récursions uniformes

La récursion faite par un algorithme $\mathcal{A}$ est dite **uniforme** si pour tout $n > n_0$, $\mathcal{A}$ traite une entrée de taille n par k appels récursifs, chacun portant sur une entrée de taille au plus αn, avec $k \geq 1$ et $\alpha \in (0,1)$ **indépendants** de n. Les récursions des algorithmes `rechercher` et `tri_fusion` sont toutes les deux uniformes. En revanche, la récursion de `tri_rapide` n'est pas uniforme.

L'arbre d'exécution $\mathcal{T}_{\mathcal{A}}(n)$ d'un algorithme $\mathcal{A}$ de récursion uniforme est d'arité k et est **complet**, c'est-à-dire que chaque nœud interne a exactement k descendants. De plus, cet arbre est **équilibré**, c'est-à-dire que les feuilles ont toutes la même hauteur, qui vaut la hauteur de l'arbre ; notons la h. Remarquons que h est le plus petit entier tel que $\alpha^h n \leq n_0$, d'où $h = \log_{\frac{1}{\alpha}} n + O(1)$. Pour tout $0 \leq p \leq h$, $\mathcal{T}_{\mathcal{A}}(n)$ contient k^p nœuds de profondeur p, chacun de valeur $g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right)$.

Reprenons alors l'équation (4.1) et sommons ensemble les contributions des nœuds de même profondeur ; on obtient

$$C_{\mathcal{A}}(n) = O\left(\sum_{p=0}^{h} k^p g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right)\right) \tag{4.3}$$

Il y a au moins trois cas dans lesquels la somme (4.3) est simple à exprimer.

Cas 1. Si $p \mapsto k^p g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right)$ suit une **décroissance exponentielle**, alors on a, par comparaison à une série géométrique, que $C_{\mathcal{A}}(n)$ est au plus une constante fois le premier terme : $C_{\mathcal{A}}(n) = O(g_{\mathcal{A}}(n))$. Dans ce cas, la complexité de l'algorithme récursif est dominée par le coût de traitement, hors appel récursif, du seul *premier pas*.

Cas 2. Si $p \mapsto k^p g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right)$ reste **quasi-constante**, au sens où elle reste dans un intervalle $[u, v]$ avec $0 < u \leq v < \infty$ indépendants de p et n, alors $C_{\mathcal{A}}(n) = O\left(h \cdot g_{\mathcal{A}}(n)\right) = O(g_{\mathcal{A}}(n) \log n)$. Dans ce cas, la complexité de l'algorithme récursif est dominée par le coût de traitement, hors appel récursif, du premier pas de récursion **multiplié par** $O(\log n)$.

Cas 3. Si $p \mapsto k^p g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right)$ suit une **croissance exponentielle**, alors on a, par comparaison à une série géométrique, que $C_{\mathcal{A}}(n)$ est au plus une constante fois le dernier terme : $C_{\mathcal{A}}(n) = O\left(k^h\right) = O\left(k^{\log_{\frac{1}{\alpha}} n}\right) = O\left(n^{\log_{\frac{1}{\alpha}} k}\right)$.

4.5 Le « théorème maître »

L'analyse des récursions uniformes résumée ci-dessus, et les trois types de comportements qu'elle fait apparaître, sont généralement synthétisées dans ce que l'on appelle le « théorème maître ». Dans l'expression des complexités, il est parfois utile de « cacher les facteurs logarithmiques ». On fait cela au moyen de la notation $\tilde{O}()$:

$f = \tilde{O}(g)$ signifie qu'il existe une constante c telle que $f(t) = O(g(t) \log^c t)$.

13. Corollaire immédiat : je serai impitoyable avec les erreurs découlant de ce genre de manipulations.

Cette notation étant posée, voici le **théorème maître** :

Théorème 4.1. *Soit k un entier positif, $\alpha \in (0,1)$ et d un réel positif. Si un algorithme $\mathcal{A}$ traite toute entrée de taille n suffisamment grande par k appels récursifs, chacun sur une entrée de taille $\lceil \alpha n \rceil$ et avec un coût hors appels récursif $g_{\mathcal{A}}(n) = \tilde{O}\left(n^d\right)$, alors*

$$C_{\mathcal{A}}(n) \leq \begin{cases} O\left(n^{\frac{\log k}{\log \frac{1}{\alpha}}}\right) & \text{si } k\alpha^d > 1 \quad \textit{(les appels récursifs dominent)}, \\ O\left(g_{\mathcal{A}}(n) \log n\right) & \text{si } k\alpha^d = 1 \quad \textit{(point d'équilibre)}, \\ O\left(g_{\mathcal{A}}(n) + n^d\right) & \text{si } k\alpha^d < 1 \quad \textit{(le traitement initial domine)}. \end{cases}$$

Soulignons les points suivants :

- L'idée de la preuve est assez simple : de $g_{\mathcal{A}}(n) = \tilde{\Theta}\left(n^d\right)$ on tire $k^p g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right) = \left(k\alpha^d\right)^p \cdot \tilde{\Theta}\left(n^d\right)$, et une comparaison entre $k\alpha^d$ et 1 permet alors d'identifier l'un des trois régimes identifiés précédemment (décroissance exponentielle, quasi-constance et croissance exponentielle).
- Si $g_{\mathcal{A}}(n) = \tilde{O}\left(n^d\right)$ alors $g_{\mathcal{A}}(n) = \tilde{O}\left(n^\delta\right)$ pour tout $\delta > d$. C'est un bon exercice que de vérifier que le théorème maître énoncé ci-dessus donne le meilleur résultat avec le paramètre d minimal pour lequel $g_{\mathcal{A}}(n) = \tilde{O}\left(n^d\right)$.
- Tel qu'énoncé, le théorème maître s'applique aux récursions uniformes assez « rigides » ; par exemple, les tailles des sous-entrées des appels récursifs doivent être rigoureusement égales entre elles.

On conclue ce chapitre par l'énoncé d'une généralisation du théorème maître, dûe à Akra et Bazzi :

Théorème 4.2. *Soit $k \geq 1$ et n_0 des entiers positifs, $\alpha_1, \alpha_2, \ldots, \alpha_k$ des réels tels que $1 > \alpha_i > 0$, et $h_1, h_2, \ldots, h_k$ des fonctions. On suppose qu'il existe $\epsilon > 0$ tel que $h_i(n) \leq \frac{n}{\log^{1+\epsilon} n}$ pour $n \geq n_0$. Notons p le réel tel que $\sum_{i=1}^k {\alpha_i}^p = 1$.*

Si un algorithme $\mathcal{A}$ traite une entrée de taille $n \geq n_0$ par des appels récursifs à k sous-entrées, de taille respectives $\alpha_i n + h_i(n)$ pour $1 \leq i \leq k$, alors

$$C_{\mathcal{A}}(n) \leq \begin{cases} O\left(n^p\right) & \textit{si } g_{\mathcal{A}}(n) = O\left(n^{p-\epsilon}\right) \textit{ pour une constante } \epsilon > 0, \\ O\left(g_{\mathcal{A}}(n) \log n\right) & \textit{si } g_{\mathcal{A}}(n) = \tilde{O}\left(n^p\right), \\ O\left(g_{\mathcal{A}}(n)\right) & \textit{sinon.} \end{cases}$$

Notons que l'indice p annoncé existe bien et est unique. Cela découle de la continuité et de la décroissance de la fonction $t \mapsto \sum_{i=1}^k {\alpha_i}^t$.

Cet énoncé inclut le théorème maître comme cas particulier, lorsque tous les α_i sont égaux et toutes les fonctions h_i sont nulles. Autrement dit, cet énoncé généralise le théorème maître d'une part en autorisant des appels récursifs de tailles distinctes, et d'autre part en autorisant des fluctuations dans la taille des appels récursifs, dès lors que ces fluctuations sont suffisamment sous-linéaires. Ces deux améliorations couvrent (très largement) les approximations que l'on peut faire en ignorant, par exemple, les parties entières et autres constantes additives.

La preuve du Théorème d'Akra-Bazzi sort du cadre de ce cours et s'avère un solide exercice d'analyse réelle. On la présente en Complément D afin que ce cours comporte une preuve complète et rigoureuse (d'une généralisation) du théorème maître.

À retenir.

- Un arbre enraciné d'arité k et de hauteur h a $< k^{h+1}$ nœuds et $\leq k^h$ feuilles.
- Une **borne inférieure** ($\Omega()$) sur la complexité d'un algorithme s'obtient en construisant une famille $\{e_n\}_n$ d'entrées dont les tailles divergent et en analysant la complexité de traitement de e_n par cet algorithme.
- Une borne de complexité est **fine** s'il existe une borne inférieure de même ordre de grandeur asymptotique. Une borne de complexité fine s'exprime par un $\Theta()$.

- On peut modéliser le déroulement d'un algorithme récursif par un **arbre d'exécution**, puis en déduire une majoration de sa complexité.
- Une récursion est **uniforme** si le nombre d'appels récursifs et le coefficient de réduction sur la taille des entrées sont indépendants de n, la taille de l'entrée.
- Le **théorème maitre** automatise l'analyse de complexité d'un algorithme récursif de récursion uniforme à partir du nombre k d'appels récursifs, du coefficient α et de la complexité asymptotique du coût de traitement **hors appels récursifs**.

Chapitre 5

Cas d'étude : Systèmes de vote

Cette séance constitue un intermède entre les séances méthodologiques d'algorithmique (2 à 4) et celles de théorie de la complexité (6 à 8). Nous y examinons la problématique suivante : comment déterminer le ou la gagnant·e d'une élection ? Cette question, fondamentale en *théorie du choix social*, apparaît par exemple aussi dans certaines méthodes d'*apprentissage automatique*. Après une introduction rapide à cette problématique, soulignant notamment un résultat d'impossibilité fondamental (le théorème de Gibbard–Satterthwaite), nous décrivons quelques exemples de *méthodes de Condorcet* et des questions d'algorithmique sous-jacentes.

Objectifs. À l'issue de cette séance, il est attendu que vous sachiez analyser une propriété d'un système de vote donné par un algorithme.

5.1 Problématique

L'enjeu d'un vote est de déterminer une préférence collective à partir de l'expression de préférences individuelles. On examine ici des questions d'algorithmique soulevées par des méthodes de vote.

Supposons à titre d'exemple que l'on organise une consultation des habitant·e·s de Nancy sur la manière dont il convient d'améliorer les conditions de circulation en vélo dans l'agglomération. Supposons que cette consultation intervienne après que le processus de décision politique ait acté de mettre en œuvre l'une de ces trois options suivantes :

(a) Supprimer des voies de circulation motorisée pour créer des double-voies vélo séparées.

(b) Supprimer des espaces de stationnement pour créer des double-voies vélo séparées.

(c) Limiter à 30 km/h la vitesse de circulation de tous les véhicules (hors véhicules d'urgence) dans toute l'agglomération.

La consultation ait pour objectif de déterminer l'option à mettre en œuvre.

Il n'y a pas que le scrutin majoritaire...

Une solution naturelle consiste à demander à chaque votant·e de choisir une option, puis de retenir l'option qui obtient le plus grand nombre de suffrages. Un tel vote, appelé **scrutin majoritaire**, peut conduire à une décision mal acceptée. [1] Pour améliorer la prise de décision, il est courant de demander aux votant·e·s de fournir plus d'information, comme dans les bulletins de la Figure 5.1.

Dans cette séance, on s'intéresse à des votes qui demandent à chaque votant·e d'*ordonner* les choix qui lui sont proposés par ordre de **préférences décroissantes**, sans ex æquo.

1. Par exemple, il se peut que (a), (b) et (c) récoltent respectivement 34%, 33% et 33% des suffrages, mais que les 66% ayant voté (b) ou (c) soient très opposés à (a), ce qui ne se voit pas dans les votes.

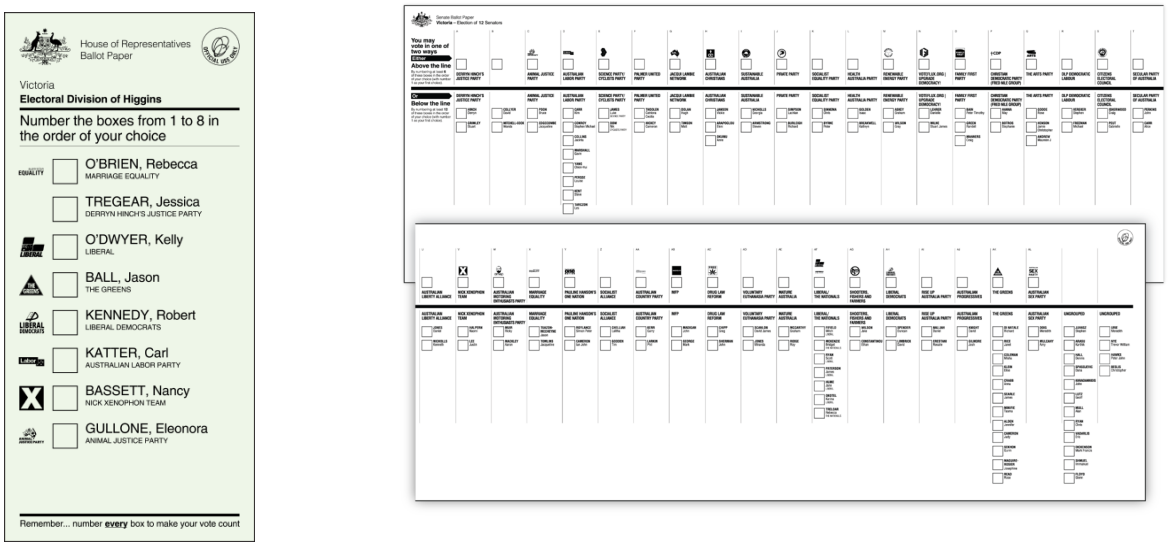


FIGURE 5.1 – Exemples de bulletins de vote tirés d'élections Australiennes de 2016 (source : wikipedia). Gauche : House of Representatives. Droite : Australian Capital Territory.

Duel, vainqueur de Condorcet et paradoxe de Condorcet

Supposons donc que pour notre exemple cycliste l'on ait demandé à chaque votant·e d'ordonner nos trois choix par ordre de préférence, sans autoriser d'ex æquo ; on écrit ainsi $x < y < z$ pour indiquer que l'on préfère x à y, et y à z (et donc, par transitivité, x à z). [2] Il y a donc 6 votes possibles. Supposons que 100 000 votant·e·s se soient exprimé·e·s, avec les résultats suivants (en milliers de votes) :

Préférences	a<b<c	a<c<b	b<a<c	b<c<a	c<a<b	c<b<a
Nombre de votes	22	12	6	27	17	16

Une approche naturelle pour déterminer le résultat d'une élection consiste à comparer les choix deux à deux. On dit qu'un choix x **gagne son duel** avec un choix y si une majorité des votant·e·s préfère x à y ; ainsi, dans l'exemple ci-dessus, (a) gagne son duel contre (b) puisque $22 + 12 + 17 = 51\%$ des votes placent (a) devant (b).

Un choix x qui gagne tous ses duels est appellé un **vainqueur de Condorcet**. Il s'avère que la relation « gagner son duel » n'est pas transitive : dans l'exemple ci-dessus, une majorité (51%) préfère (a) à (b), tandis qu'une autre majorité (55%) préfère (b) à (c), et qu'une troisième majorité (60%) préfère (c) à (a). Cette non-transitivité est connue sous le nom de **paradoxe de Condorcet**.

Suffrage et méthodes de comptage

Un **suffrage** est un ensemble de votes. Une **méthode de comptage** est une fonction qui associe à tout suffrage un choix ou un ordre vainqueur, ou alors rien en cas d'ex æquo. Ainsi, le vote majoritaire est une méthode de comptage. Il en existe des dizaines d'autres, dont voici quelques exemples historiques.

Dans la **méthode de Borda**, chaque vote apporte un nombre de points à chaque choix en fonction de sa position : ici, 2 points pour le premier, 1 point pour le deuxième, et aucun point pour le troisième. [3] On additionne les points obtenus par chaque choix sur l'ensemble des votes, puis on classe les choix par nombre décroissant de points. C'est le résultat du comptage. Pour le suffrage ci-dessus, après division par 1 000, on obtiendrait 91 points pour (a), 104 points pour (b) et 105 points pour (c). Le résultat serait donc $c < b < a$.

2. On utilise ici la même notation pour les ordres qu'au chapitre 1. Autrement dit, "$x < y$" dans un vote exprime le fait que lorsque cet·te votant·e ordonne les choix par préférences décroissantes, le rang de x est inférieur à celui de y.

3. Pour c choix, un vote apporte 0 point au dernier choix, 1 point à l'avant-dernier, ..., $c-1$ points au premier choix.

La **méthode de Dodgson**[4] compte, pour chaque choix, le nombre minimum de transpositions[5] à opérer sur les votes exprimés pour faire de ce choix un vainqueur de Condorcet, et déclare vainqueur le choix pour lequel ce nombre est minimal. Pour le suffrage ci-dessus, ces nombres de transposition sont 10 000 pour (a), 1 000 pour (b) et 5 000 pour (c). Le résultat est donc (b), avec une avance plus confortable.

La **règle de Copeland** désigne vainqueur le ou les choix gagnant le plus de duels. Pour le suffrage ci-dessus, les trois choix arrivent ex æquo et il n'y a donc pas de vainqueur.

Les **méthodes minimax** associent à chaque choix un score basé sur son moins bon duel, par exemple le nombre minimum de votant·e·s ayant voté pour ce choix dans un duel, et désignent vainqueur le ou les choix de score maximal. Pour ce suffrage, ce score est de 40 pour (a), 49 pour (b) et 45 pour (c) ; c'est donc (b) qui est désigné vainqueur.

On le voit, des méthodes différentes peuvent donner des résultats différents... Sur quels critères comparer ces méthodes (et d'autres) ?

Formalisation

Pour définir ce que devrait être une bonne méthode de comptage, commençons par modéliser ces méthodes en termes mathématiques :

- On modélise les **choix** entre lesquels les votants doivent se prononcer par un ensemble fini, que l'on note C. Ainsi $C = \{a, b, c\}$ dans notre exemple introductif cycliste. Pour le premier tour de l'élection présidentielle 2022 en France l'ensemble des choix était

$$\begin{aligned} C = \{&\text{Nathalie ARTHAUD}, \text{Nicolas DUPONT-AIGNAN}, \text{Anne HIDALGO}, \text{Yannick JADOT},\\ &\text{Jean LASSALLE}, \text{Marine LE PEN}, \text{Emmanuel MACRON}, \text{Jean-Luc MELENCHON},\\ &\text{Valerie PECRESSE}, \text{Philippe POUTOU}, \text{Fabien ROUSSEL}, \text{Eric ZEMMOUR}\} \end{aligned}$$

- On modélise un **vote** par un ordre total sur C et on note V l'ensemble des votes possibles. Dans notre exemple introductif cycliste, $V = \{a < b < c, a < c < b, b < a < c, b < c < a, c < a < b, c < b < a\}$. Dans le cas de l'élection présidentielle 2022 en France, puisqu'il y a 12 candidat·es, l'ensemble V est de taille 12!.
- On modélise le **suffrage** correspondant à l'expression de n votant·e·s par un vecteur $S \in V^n$. Le **suffrage** associé à notre exemple introductif cycliste peut donc être (selon l'ordre dans lequel on range les votes) :

$$\begin{aligned} S = (&\underbrace{a<b<c,\ \dots,\ a<b<c}_{22\,000\text{ copies}}, \underbrace{a<c<b,\ \dots,\ a<c<b}_{12\,000\text{ copies}}, \underbrace{b<a<c,\ \dots,\ b<a<c}_{6\,000\text{ copies}},\\ &\underbrace{b<c<a,\ \dots,\ b<c<a}_{27\,000\text{ copies}}, \underbrace{c<a<b,\ \dots,\ c<a<b}_{17\,000\text{ copies}}, \underbrace{c<b<a,\ \dots,\ c<b<a}_{16\,000\text{ copies}}) \in V^{100\,000}. \end{aligned}$$

On peut alors modéliser une **fonction de comptage** de deux manières différentes :

- Si le résultat est un choix, c'est une fonction $f : V^n \to C$; on appelle cela une **fonction de choix social**. C'est le cas de la méthode de Dodgson.
- Si le résultat est un ordre sur les choix, c'est une fonction $f : V^n \to V$; on appelle cela une **fonction d'ordre social**. C'est le cas pour la méthode de Borda.

Une fonction de choix/d'ordre social est généralement décrite par un algorithme qui la calcule.

Propriétés et théorème d'impossibilité

On peut maintenant exprimer quelques propriétés que l'on souhaite que notre fonction de comptage idéale satisfasse. Pour cela il nous faut une dernière notation : pour $s \in V$ on note $\text{pref}(s)$ le choix préféré du vote s. Ainsi, dans notre exemple cycliste on a $\text{pref}(a < b < c) = \text{pref}(a < c < b) = a$.

4. Mieux connu sous son nom de plume, Lewis Caroll.
5. Changer un vote « a<b<c » en « c<b<a » requiert deux transpositions.

Une fonction de choix social f est **unanime** si pour tout $S \in V^n$, si $\text{pref}(S_1) = \text{pref}(S_2) = \ldots = \text{pref}(S_n) = c$ alors $\text{pref}(f(S)) = c$. Autrement dit, lorsque tous les votes préfèrent le même choix, alors ce choix est vainqueur. Il est facile de vérifier que la méthode de Dodgson est unanime.

Une fonction de choix social f est **dictatoriale** s'il existe un entier $1 \leq i \leq n$ tel que pour tout suffrage $S \in V^n$ on ait $f(S) = \text{pref}(S_i)$. Autrement dit, le résultat de l'élection par cette fonction coïncide [6] systématiquement avec la préférence d'un·e votant·e prédéfini·e. Il est là-aussi facile de vérifier que la méthode de Dodgson est *non*-dictatoriale.

Une fonction de choix social f est **manipulable** s'il existe un entier $1 \leq k \leq n$ et deux suffrages $S, S' \in V^n$ qui diffèrent en le $k$ème vote et seulement lui [7] tels que le vote S_i préfère $f(S'_i)$ à $f(S_i)$. Autrement dit, il est possible à un votant d'améliorer le résultat du point de vue de ses préférences en votant *différemment* de ses préférences. [8] On peut montrer que la méthode de Dodgson est manipulable.

Théorème 5.1 (Gibbard–Satterthwaite)**.** *Pour $|C| \geq 3$ choix, il n'existe pas de fonction de choix social qui soit à la fois unanime, non-manipulable et non-dictatoriale.*

Il est facile de traduire certaines des propriétés définies pour les fonctions de choix social aux fonctions d'ordre social. Par exemple :

- Une fonction d'ordre social f est **unanime** si pour tout $S \in V^n$, si $S_1 = S_2 = \ldots = S_n = v$ alors $f(S) = v$. Il est facile de vérifier que la méthode de Borda est unanime.
- Une fonction d'ordre social f est **dictatoriale** s'il existe un entier $1 \leq i \leq n$ tel que pour tout suffrage $S \in V^n$ on ait $f(S) = S_i$. Il est là-aussi facile de vérifier que la méthode de Borda est *non*-dictatoriale.
- Une fonction d'ordre social f est **manipulable** s'il existe un entier $1 \leq k \leq n$ et deux suffrages $S, S' \in V^n$ qui diffèrent en le $k$ème vote et seulement lui tels que le vote S_i préfère $f(S'_i)$ à $f(S_i)$. La méthode de Borda s'avère manipulable (cf TD 4)...

Un énoncé similaire au Théorème 5.1 existe dans ce cadre, c'est le Théorème d'Arrow. Les preuves de ces résultats dépassent le cadre de ce cours (on renvoit les élèves intéressé·e·s à l'article de Sen [Sen01]) et on en retient principalement un enseignement :

Certaines propriétés désirables de fonctions de choix/ordre social sont incompatibles.

5.2 Algorithmique et comptage

Dans le prolongement du Théorème 5.1, une part importante de l'étude des fonctions de comptage consiste à définir des algorithmes, à formuler des propriétés sur ce que calculent ces algorithmes, et à prouver que ces algorithmes ont ou n'ont pas ces propriétés.

Préliminaire algorithmique : la matrice des duels

On considère dans ce qui suit une élection d'ensemble de choix $C = \{c_1, c_2, \ldots, c_k\}$ et un suffrage $S \in V^n$. Les fonctions de choix/d'ordre social qui nous intéressent commencent par extraire du profil de votes considéré les résultats des confrontations entre paires de choix. Pour $1 \leq i, j \leq k$, $i \neq j$, on définit $d(i, j)$ comme le nombre de votant·e·s préférant c_i à c_j. On note $D = (d(i, j))_{1 \leq i,j \leq k}$ la **matrice des duels** ; c'est une matrice $k \times k$ dont chaque entrée est un entier compris entre 0 et n et tel que $d_{i,j} + d_{j,i} = n$.

La matrice des duels suffit à déterminer le résultat de certaines méthodes de comptage (par exemple Borda, Copeland et Minimax) mais s'avère insuffisante pour d'autres méthodes (par exemple Dodgson).

6. Soulignons que c'est un constat purement descriptif. On ne cherche pas à modéliser de lien de causalité.
7. Autrement dit, $S_i = S'_i$ si et seulement si $i \neq k$.
8. L'évocation de ce phénomène est souvent appelé un « appel au vote utile ».

Exemple de propriété : l'insensibilité aux clones

Introduisons une propriété que l'on va étudier à titre d'exemple. **Cloner** un choix $c \in C$ dans un suffrage S, c'est produire un suffrage, noté $\text{clone}(S, c)$, en ajoutant un nouveau choix c' à l'ensemble des choix, et en insérant c' *immédiatement après* c dans chaque vote S_i. Le suffrage $\text{clone}(S, c)$ a le même nombre de votes que S et porte sur exactement un choix de plus.

> Une fonction de choix social f est **insensible aux clones** si pour tout suffrage S et tout choix $c \in C$ on a $f(S) = f(\text{clone}(S, c))$.

Cette propriété modélise le fait qu'une élection ne puisse pas être manipulée par simple ajout d'un choix très proche d'un choix existant. Établir qu'une méthode de comptage est insensible aux clones requiert une preuve valide pour tout nombre k de choix, tout suffrage S et tout choix c cloné. Inversement, prouver qu'une méthode de comptage est sensible aux clones ne demande qu'un exemple (un nombre de choix, un suffrage et un choix cloné).

Le préféré de Borda est sensible aux clones

Il est facile de constater que si l'on élit le préféré du classement produit par la méthode de Borda, cela donne un choix *sensible* aux clones. Prenons $k = 2$, $n = 5$ et clonons c_2 dans le suffrage suivant :

Vote	$c_1 < c_2$	$c_2 < c_1$
# votes	3	2

$\rightsquigarrow$

Vote	$c_1 < c_2 < c_2'$	$c_2 < c_2' < c_1$
# votes	3	2

Avant clonage, c_1 gagne l'élection par 3 points contre 2 pour c_2. Après clonage, les scores de Borda sont de 6 pour c_1 et 7 pour c_2 et c'est donc c_2 qui gagne. (Le score du clone c_2', ici 2, importe peu puisqu'il est par définition inférieur au score du choix cloné.)

Copeland est sensible aux clones

La méthode de Copeland est elle aussi *sensible* aux clones. Prenons $k = 4$, $n = 10$ et clonons c_2 dans le suffrage

Vote	$c_1 < c_2 < c_3 < c_4$	$c_2 < c_3 < c_4 < c_1$	$c_3 < c_4 < c_1 < c_2$	$c_4 < c_1 < c_2 < c_3$
# votes	1	4	2	3

$\downarrow$

Vote	$c_1 < c_2 < c_2' < c_3 < c_4$	$c_2 < c_2' < c_3 < c_4 < c_1$	$c_3 < c_4 < c_1 < c_2 < c_2'$	$c_4 < c_1 < c_2 < c_2' < c_3$
# votes	1	4	2	3

Avant clonage, la méthode de Copeland désigne c_3 vainqueur car c_3 gagne deux duels tandis que chacun des trois autres choix gagne un duel (le duel entre c_2 et c_4 n'ayant pas de vainqueur). Le clonage ne change pas ces duels. Le clone c_2', quant à lui, gagne son duel contre c_3 et perd son duel contre c_1 et c_2. La méthode de Copeland ne déclare pas de vainqueur, les choix c_1, c_2 et c_3 étant premiers ex æquo.

5.2.1 Méthode de Kemeny-Young

La **règle de Kemeny-Young** est une fonction d'ordre social. Son résultat est le vote, c'est-à-dire l'élément de V, qui minimise la somme des distances aux votes exprimés ; la distance entre deux ordres est ici mesurée par le nombre d'inversions entre ces ordres.[9] Pour tout vote $r \in V$ et tout suffrage $S \in V^n$ on définit le désaccord

$$\text{des}(r, S) \stackrel{\text{def}}{=} \sum_{t=1}^{n} \sum_{1 \le i < j \le k} \mathbb{1}_{\{c_i, c_j\} \text{ est une inversion entre } r \text{ et } S_t}$$

Le résultat est le ou les ordres de désaccord minimum.

9. Rappelons qu'une *inversion* entre deux ordres $r, s \in V$ est une paire $\{i, j\} \in C$ telle que que i et j n'apparaissent pas dans le même ordre dans r et dans s. La distance entre permutation définie par le nombre d'inversions est parfois appelée *distance de Kendal-τ*.

Algorithmique

Il est possible de calculer le nombre d'inversion entre deux ordres de taille k en temps $O(k \log k)$. [10] Calculer ainsi les nombres d'inversions pour chaque ordre r et chaque vote S_i permet de déterminer le vainqueur selon Kemeny-Young en temps $O(nk!k \log k)$. Cette méthode est efficace si le nombre de choix est petit, et ce même si le nombre de votes est grand.

Lorsque le nombre de choix est grand, la complexité de cet algorithme naïf est prohibitive. Aucun algorithme efficace n'est connu pour un grand nombre de choix, et ce même si le nombre de vote est très faible. La notion de NP-difficulté que l'on introduira au Chapitre 8 permet de formaliser le fait qu'il est peu vraisemblable qu'un algorithme efficace existe : il est NP-difficile (relativement à k) de décider, étant donné un profil S de 4 votes sur k choix et un seuil σ, si le résultat r de la règle de Kemeny-Young satisfait $\text{des}(r, S) \leq \sigma$.

Propriétés

Il est facile d'établir que la méthode de Kemeny-Young est unanime et *non*-dictatoriale. On peut adapter la propriété d'insensibilité aux clones aux fonctions d'ordre social : une fonction d'ordre social f est **insensible aux clones** si pour tout suffrage S et tout choix $c \in C$, on obtient l'ordre $f(S)$ lorsque l'on supprime le clone c' de $f(\text{clone}(S, c))$. La méthode de Kemeny-Young s'avère sensible aux clones. En effet, prenons $k = 2$ et $n = 13$, et considérons le suffrage S suivant :

Vote	$c_1 < c_2 < c_3$	$c_2 < c_3 < c_1$	$c_3 < c_1 < c_2$
# votes	4	5	4

On peut vérifier que la méthode de Kemeny-Young désigne $c_2 < c_1 < c_3$ vainqueur. Clonons c_2 dans S pour obtenir un nouveau suffrage S', puis clonons c_2 dans S' pour obtenir un suffrage S''. Autrement dit, $S'' = \text{clone}(\text{clone}(S, c_2), c_2)$. Le suffrage S'' :

Vote	$c_1 < c_2 < c_2'' < c_2' < c_3$	$c_2 < c_2'' < c_2' < c_3 < c_1$	$c_3 < c_1 < c_2 < c_2'' < c_2'$
# votes	4	5	4

La méthode de Kemeny-Young désigne alors $c_1 < c_2 < c_2'' < c_2' < c_3$ vainqueur. On peut remarquer que la suppression des deux clones dans cet ordre produit un ordre différent de celui obtenu en appliquant la méthode de Kemeny-Young à S. C'est donc que la méthode de Kemeny-Young a été sensible à l'un des deux clonages.

5.2.2 Méthode des paires ordonnées

La **méthode des paires ordonnées** est une fonction de choix social proposée en 1987 par Nicolaus Tideman. Notons G le graphe orienté ayant un sommet pour chacun des k choix de C, et une arête orientée de chaque sommet i à chaque sommet j, cette arête étant étiquetée par l'entrée $d(i, j)$ de la matrice des duels. (En particulier, le graphe non-orienté sous-jacent à G_S est complet.) Considérons le sous-graphe orienté G' de G obtenu par le procédé suivant :

```
G'= graphe orienté de sommets 1,2, ..., k et sans arête
L = liste des arêtes de G triées par poids décroissants
Tant que L n'est pas vide
   Enlever de L l'arête A de poids maximal
   Ajouter A à G' si cela ne crée pas de cycle
```

Si plusieurs arêtes ont même poids, on teste chacune des manières de les départager ; chaque test conduit à un vainqueur, et ces vainqueurs sont déclarés ex æquo.

10. Cela s'entend en RAM taille arbitraire. Cf exercice 6 du TD4.

Cette construction assure que G' est acyclique. En particulier, G' contient *au moins une* source.[11] Cette construction assure aussi que pour qu'une arête (x, y) n'appartienne pas à G', il faut que G' contienne un chemin de y à x. En particulier, G' doit contenir un chemin de la source u vers tout autre sommet, et par conséquent G' contient *au plus une* source. La méthode des paires ordonnées retourne cette source comme résultat.

Reformulons cela. L'algorithme ci-dessus sélectionne un sous-ensemble des duels de sorte qu'il existe un unique choix qui n'en perde aucun. En particulier, si l'on s'en tient aux seuls duels sélectionnés, ce choix bat « par transitivité » tous les autre choix. La sélection examine les duels par ordre décroissant de « force de départage », et retient un duel si et seulement s'il ne crée aucun « cycle de Condorcet » avec des duels déjà retenus.

Algorithmique

En RAM taille arbitraire, il est possible de déterminer en temps $O(a + b)$ si un graphe orienté à a sommets et b arêtes est acyclique. Ainsi, l'algorithme des paires ordonnées est de complexité $O(k^4)$, ce à quoi il faut ajouter la complexité du calcul de la matrice D à partir du profil de vote e. Dans l'ensemble, le calcul de cette fonction de choix social peut donc se faire en temps $O(k^4 + k^2 n)$ en RAM taille arbitraire.

Propriétés

Il est facile de vérifier que la méthode des paires ordonnée est *unanime*[12] et *non-dictatoriale*. Cette méthode a de fait été conçue pour avoir la propriété suivante :

Proposition 5.2. *La méthode des paires ordonnée est insensible aux clones.*

Idée de preuve. Notons $S' = \text{clone}(S, c)$ et c' le clone de c, et considérons le calcul du graphe G' relatif au suffrage S'. Supposons qu'il n'y a pas d'égalité entre poids autre que celles causées par le clonage. L'arête $c \to c'$ est de poids n, ce qui est maximal, et est donc traitée la première. Ensuite, pour tout choix $c_i \in C \setminus \{c, c'\}$ les arêtes $c \to c_i$ et $c' \to c_i$ sont de même poids et sont traitées simultanément. La manière dont on les départage n'a aucune importance : soit elles créent toutes les deux un cycle et aucune n'est ajoutée à G', soit aucune ne crée de cycle et toutes les deux sont ajoutées à G'. Le raisonnement est similaire pour les arêtes $c_i \to c$ et $c_i \to c'$. Cela permet de montrer, par récurrence sur le nombre d'arêtes enlevées de L, que le graphe G' créé lors du traitement de S' coincide, après suppression de c', avec le graphe G' créé lors du traitement de S. □

5.2.3 Méthode de Schulze

La **méthode de Schulze** est une fonction de choix social proposée en 2010 par Tobias Schulze. Partons (comme pour les paires ordonnées) du graphe orienté G ayant un sommet pour chacun des k choix de C, et une arête orientée de chaque sommet i à chaque sommet j, étiquetée par l'entrée $d(i, j)$ de la matrice des duels. On construit un premier sous-graphe orienté G' de G en ne gardant que les arêtes $i \to j$ telles que $d(i, j) > d(j, i)$. On définit alors la *force* $\delta(i, j)$ de l'arête $i \to j$ de l'une des manières suivantes (au choix) :

$$\delta(i,j) \stackrel{\text{def}}{=} d(i,j), \qquad \text{ou } \delta(i,j) \stackrel{\text{def}}{=} d(i,j) - d(j,i), \qquad \text{ou encore } \delta(i,j) \stackrel{\text{def}}{=} d(i,j)/d(j,i).$$

(Ce qui suit est valable pour ces trois définitions.) La *force* d'un chemin $\gamma = s_1 \to s_2 \to \ldots \to s_\ell$ de G' est définie comme le minimum de $\{\delta(s_i, s_{i+1}) \colon 1 \leq i \leq \ell - 1\}$. On définit alors $P(i, j)$ comme la force du chemin le plus fort de G' de i à j ; s'il n'existe pas de chemin de i à j dans G' alors $P(i, j) \stackrel{\text{def}}{=} 0$. On définit alors un second graphe orienté G'' de sommets C et avec une arête $i \to j$ si et seulement si $P(i, j) > P(j, i)$. Le graphe G'' est acyclique, et satisfait même une propriété plus forte :

11. En effet, voici un algorithme qui trouve une source. Choisissons un premier sommet (quelconque) et vérifions si c'est une source. Si oui, on le retourne et on termine. Sinon, on choisit une de ses arêtes entrantes, on saute au sommet voisin via cette arête, et on recommence. Cette exploration ne peut pas visiter le même sommet plus d'une fois (cela révèlerait un cycle), aussi elle doit terminer et donc trouver une source.

12. En cas de vote unanime, chaque $d(i, j)$ vaut soit 0, soit n. Cela produit beaucoup de cas d'égalité entre paires, mais le résultat est le même quelque soit la manière dont on les départage.

Lemme 5.3. *Si G'' contient $a \to b$ et $b \to c$ alors G'' contient aussi $a \to c$.*

Démonstration. Notons α, β et γ les chemins de forces maximales de, respectivement, a à b, b à c et c à a. Notons $\odot$ l'opération de concaténation de deux chemins, uniquement définie lorsque l'arrivée du premier coïncide avec le départ du second. Notons $f_\bullet$ la force du chemin $\bullet$.

Comme $\alpha \odot \beta$ est un chemin de a à c, on a $P(a,c) \geq f_{\alpha\odot\beta} = \min(f_\alpha, f_\beta)$. Dès lors, si G'' ne contient pas $a \to c$, on a $f_\gamma \geq \min(f_\alpha, f_\beta)$ et de deux choses l'une :

- soit $f_\alpha \leq f_\beta$, et alors $P(b,a) \geq f_{\beta\odot\gamma} \geq f_\alpha = P(a,b)$, contredisant $a \to b \in G''$,
- soit $f_\alpha > f_\beta$, et alors $b(c,b) \geq f_{\gamma\odot\alpha} \geq f_\beta = P(b,c)$, contredisant $b \to c \in G''$.

Le graphe G'' doit par conséquent contenir $a \to c$. □

Le graphe G'' étant acyclique, il contient nécessairement au moins une source. Si cette source est unique, c'est le résultat de la méthode de Schulze. Si G'' contient plus d'une source, il convient de départager ces ex æquo ; en pratique, cela se produit rarement.

Algorithmique

Examinons l'algorithmique en RAM taille arbitraire. Le calcul du graphe G' et des poids $\delta(\cdot,\cdot)$ peut facilement se faire en temps $O(k^2)$. Le calcul de G'' demande, en revanche, de calculer la force maximale des chemins entre *toute* paire de sommets. On peut calculer les $\binom{k}{2}$ forces en un temps total $O(k^3)$ par une adaptation simple de l'**algorithme de Floyd-Warshall** de calcul de distances dans un graphe pondéré. C'est un exemple d'application d'une méthodologie récursive appelée **programmation dynamique**.

On a en entrée un graphe orienté G'' de sommets $[k] \stackrel{\text{def}}{=} \{1, 2, \ldots, k\}$ dont chaque arête $i \to j$ a un poids $\delta(i,j)$, l'ensemble étant donné par la matrice δ. On souhaite calculer la fonction, définie sur $[k]^2$,

$$f(i,j) = \max_{\gamma \text{ chemin de } i \text{ à } j} \min_{a\to b \text{ arête de } \gamma} \delta(a,b)$$

On introduit une fonction auxiliaire, définie sur $[k]^3$, qui décompose le problème :

$$g(i,j,m) = \max_{\gamma \text{ chemin de } i \text{ à } j \text{ sans sommet intermédiaire} > m} \min_{a\to b \text{ arête de } \gamma} \delta(a,b).$$

Comme $f(i,j) = g(i,j,k)$, la fonction g détermine la fonction f. Le calcul de g s'avère plus commode que celui de f car on peut procéder par valeurs croissantes de k *pour tous* i *et* j grâce à la récurrence :

$$\forall m \geq 2, \quad g(i,j,m) = \max\{g(i,j,m-1), \min(g(i,m,m-1), g(m,j,m-1))\}. \tag{5.1}$$

En effet, le meilleur chemin de i à j sans sommet intermédiaire $> m$ soit évite m (premier terme), soit passe par m (second terme). Cette récurrence donne l'algorithme simple :

```
Calculer g(i,j,1) pour tous i,j
Pour m=2..k
   Pour i=1..k
     Pour j=1..k
       g(i,j,m) = max (g(i,j,m-1), min(g(i,m,m-1), g(m,j,m-1)))
```

Le calcul des $g(i,j,1)$ se fait naïvement en temps $O(k^2)$ et l'ensemble de l'algorithme est de complexité $O(k^3)$.

Propriétés

Il est facile de vérifier que la méthode de Schulze est *unanime* et *non-dictatoriale*. Cette méthode s'avère aussi insensible aux clones.

5.3 Prolongements

Certaines des méthodes de votes dont l'usage se répand actuellement sont étonnamment récentes. La méthode de Schulze, que l'on retrouve par exemple dans divers *partis pirates* ou encore dans des communautés telles que *Debian*, date de 1997. Autre exemple, *le jugement majoritaire*, utilisé par exemple par la ville de Paris pour son budget participatif, date de 2007. (Précisons que le jugement majoritaire n'est pas une fonction de choix social au sens de la Section **??** puisqu'il demande à chaque votant de *noter* chaque choix, et déclare vainqueur le choix de meilleure note médiane.)

Les *méthodes à second tour instantané* simulent une élection uninominale à plusieurs tours à partir de profils de votes ordonnant tout ou partie des candidats. Esquissons par exemple la méthode *single transferable vote* (STV), utilisée dans plusieurs pays du Commonwealth pour une élection à plusieurs sièges. Elle procède par phase. À chaque phase, on ne tient compte que du premier choix de chaque bulletin. Si un candidat recueille un nombre de voix suffisant pour être élu, il est rayé de tous les bulletins ; les bulletins dont c'était le premier choix se reportent donc sur leur second choix. Cette méthode considère cependant qu'un bulletin votant pour un candidat élu a été partiellement « consommé » : son poids (initialement 1) est réduit d'autant plus que le candidat a été élu de justesse. Gare à la précision arithmétique nécessaire pour dépouiller de grandes élections...

Au-delà des méthodes de comptage, se pose la question des *protocoles* de vote. On entend par là tout processus par lequel l'expression des votes est collectée, dépouillée, comptée et proclamée. Tout comme les fonctions de comptage, les protocoles de votes peuvent être formalisés et analysés. Il est par exemple souhaitable qu'un protocole garantisse (i) que tout votant puisse s'assurer que son vote est bien pris en compte, et que (ii) qu'aucun votant ne puisse prouver à autrui la teneur de son vote. Remarquons que dans le vote papier en France, la propriété (ii) n'est garantie que depuis l'obligation, à partir de 1913, de mettre sous enveloppe des bulletins imprimés de manière uniforme lors d'un passage dans l'isoloir. Cette obligation est par ailleurs le résultat d'une longue lutte sous la troisième république [Tan04]. Ces questions sont un enjeu primordial dans les discussions sur les méthodes de vote à distance ou de vote électronique. Elles sont aussi assez subtiles, voir

`https://interstices.info/vote-par-internet/`

pour un exemple de discussion. Ce sujet est plus largement traité dans le livre de Cortier et Gaudry [CG22].

Examinons maintenant quelques méthodes de comptage classiques. Dans ce qui suit, on se concentre sur l'algorithmique. Les propriétés de telle ou telle fonction de choix social ne sont signalées que pour donner du contexte, et peuvent s'approfondir à partir de la table synthétique disponible à

`https://en.wikipedia.org/wiki/Schulze_method#Comparison_table`

5.4 Références bibliographiques

[CG22] Véronique Cortier and Pierrick Gaudry. *Le vote électronique*. Odile Jacob, 2022.

[Sen01] Arunava Sen. Another direct proof of the gibbard–satterthwaite theorem. *Economics Letters*, 70(3) :381–385, 2001.

[Tan04] Philippe Tanchoux. Les procédures électorales en france de la fin de l'ancien régime à la première guerre mondiale. 2004.

À retenir.

- Une méthode de comptage de votes se modélise par une fonction de choix/d'ordre social et se décrit par un algorithme.
- L'intérêt d'une méthode de comptage dépend de la complexité et des propriétés de son algorithme.

Chapitre 6

Bornes inférieures de complexité

Cette séance aborde un problème fondamental en théorie de la complexité : une fois que l'on dispose d'un algorithme résolvant un problème algorithmique, comment déterminer s'il est *optimal* au sens où il *n'existe pas* d'algorithme de meilleure complexité pour ce problème ? Cette question cache un changement de point de vue important : il faut raisonner non pas sur un algorithme particulier, mais sur *l'ensemble des algorithmes résolvant un problème donné*. Cette séance introduit à deux outils pour faire cela : les *arguments d'adversaires* et les modèles de calcul à base d'*arbres de décision*, variantes des arbres d'exécution étudiés au Chapitre 4. Nous présentons ces méthodes au travers de deux résultats emblèmatiques.

Objectifs. À l'issue de cette séance, il est attendu que vous

- compreniez la notion de complexité d'un problème,
- sachiez prouver une borne inférieure sur la complexité d'un problème par un argument d'adversaire sur le modèle des deux exemples traités (Divisible par 3 et Recherche dans un tableau trié), et
- sachiez prouver une borne inférieure sur la complexité d'un problème par analyse d'arbres de décision.

La formalisation de la notion d'adversaire n'est pas exigible.

6.1 Borne inférieure sur la complexité d'un problème

Les bornes fines permettent de comparer les complexités pire-cas de différents algorithmes. Ainsi, en RAM taille arbitraire, `tri fusion`, de complexité $\Theta(n \log n)$, est meilleur [1] que `tri_rapide` quand ce dernier utilise le premier élément comme pivot. Il est naturel de se poser la question suivante : *est-il possible de faire mieux que le meilleur de ces algorithmes, c'est-à-dire de trier un tableau en temps* $o(n \log n)$ *?* Une réponse négative à cette question prend généralement la forme suivante :

> Une fonction $n \mapsto f(n)$ est une **borne inférieure** pour un problème algorithmique P si **tout** algorithme $\mathcal{A}$ qui résout P est de complexité $\Omega(f(n))$.

Soulignons que par « tout algorithme », on entend « tous les algorithmes imaginables » et pas seulement « tous les algorithmes connus ». On a donc **deux notions distinctes** de bornes inférieures : pour un algorithme et pour un problème.

Dans quel modèle ? Toute borne inférieure, que ce soit pour un algorithme ou pour un problème, est relative à un modèle de calcul : une fonction peut être une borne inférieure pour un modèle de calcul et pas pour un autre. Lorsque l'on discute de complexité de problèmes, il est essentiel de préciser le modèle de calcul dans lequel on travaille.

1. Au sens de la complexité pire-cas. Rappelons que, comme discuté en séance 2, la complexité pire-cas n'a pas pour but de prédire les performances pratiques sur des instances particulières.

Pour quel problème ? Il est aussi important de bien préciser le problème algorithmique considéré. Il est par exemple possible, dans le modèle RAM taille arbitraire, de donner un algorithme de complexité *linéaire* pour la variante suivante du problème de tri :

> Tri $0-1$
> **Entrée :** Un tableau de n entiers valant chacun 0 ou 1.
> **Sortie :** Un tableau contenant ces entiers triés dans l'ordre croissant.

Il suffit en effet de compter le nombre k de 0 que contient le tableau d'entrée, puis de réécrire le contenu de ce tableau en commençant par k cases à 0 et en complétant par des cases à 1.

Argument de taille. Pour certains problèmes, la taille de la sortie (dans le pire cas) suffit à donner une borne inférieure intéressante. C'est souvent le cas pour les problèmes d'énumération. Il est par exemple facile de voir que le problème suivant [2] a une complexité $\Omega(n^2)$:

> Énumération d'inversions
> **Entrée :** $S[1..n]$ et $T[1..n]$ deux tableaux de permutation de taille n.
> **Sortie :** La liste des inversions entre $S[1..n]$ et $T[1..n]$, dans un ordre quelconque.

Cet argument élémentaire s'avère malheureusement inopérant sur des problèmes de comptage ou de décision, pour lesquels la sortie est de taille constante.

6.2 Arguments d'adversaire : quelle proportion de l'entrée est-il nécessaire de lire ?

Notre première technique, appelée « argument d'adversaire », permet de prouver que *tout* algorithme résolvant un problème donné *doit* lire une quantité minimum d'information de l'entrée.

6.2.1 Exemple introductif : divisibilité

Commençons par remarquer que certains problèmes peuvent être résolus *sans lire toute l'entrée*. On en a déja vu un exemple avec la résolution de Recherche dans un tableau trié (Section 3.2) par recherche dichotomique. Voici un autre exemple :

> Parité
> **Entrée :** La représentation binaire $x_n x_{n-1} \dots x_0$ d'un entier strictement positif x.
> **Sortie :** Vrai ou faux, x est divisible par 2 ?

En effet, il existe un algorithme de complexité $O(1)$ pour résoudre Parité : il suffit de renvoyer `vrai` si x_0 vaut 0, et `faux` sinon. Lire les bits x_1, x_2, …, x_n de l'entrée est inutile pour résoudre ce problème ! Considérons maintenant un problème qui peut sembler proche :

> Divisible par 3
> **Entrée :** La représentation binaire $x_n x_{n-1} \dots x_d$ d'un entier strictement positif x.
> **Sortie :** Vrai ou faux, x est divisible par 3 ?

2. Rappel : un tableau de permutation est un tableau de taille n qui contient exactement une fois chacun des entiers entre 1 et n. Une inversion entre deux tableaux de permutations est un couple (i, j) telle que i apparaît avant j dans l'un et après j dans l'autre.

On peut prouver qu'il est *impossible* de trouver un raccourci permettant, comme pour PARITÉ, de résoudre le problème sans lire essentiellement [3] toute l'entrée.

Proposition 6.1. *Pour tout algorithme $\mathcal{A}$ qui résout* DIVISIBLE PAR 3 *et pour tout entier n, il existe une entrée $x_n x_{n-1} \dots x_0$ pour laquelle $\mathcal{A}$ accède à au moins n bits de l'entrée.*

Démonstration. Soit $W = w_n w_{n-1} \dots w_0 = 11000 \dots 0 \in \{0,1\}^{n+1}$ et pour $0 \leq i < n-1$, notons $W^{\neg i}$ le mot binaire obtenu en inversant w_i dans W. Ainsi, on a par exemple $W^{\neg 0} = 11000 \dots 01$ et $W^{\neg n-1} = 10000 \dots 0$.

L'entier de représentation binaire W est $2^{n-1} + 2^{n-2} = 3 * 2^{n-2}$, et est donc divisible par 3. L'entier de représentation binaire $W^{\neg n-1}$ est 2^{n-1} et n'est pas divisible par 3 ; pour $0 \leq i < n-1$, l'entier de représentation binaire $W^{\neg i}$ est $3 * 2^{n-2} + 2^i$ et n'est pas non plus divisible par 3. Autrement dit, changer presque n'importe quel bit dans W change la réponse à DIVISIBLE PAR 3.

Supposons maintenant qu'il existe un algorithme $\mathcal{A}$ résolvant MULTIPLE DE 3 qui traite W sans lire chacun des bits w_i avec $0 \leq i < n$. Notons j un indice tel que $\mathcal{A}$ ne lit pas w_j. Dès lors, l'exécution de $\mathcal{A}$ sur W et sur $W^{\neg j}$ conduit à la même réponse, puisque ces deux entrées coincident sur les bits lus par $\mathcal{A}$. Par conséquent, $\mathcal{A}$ retourne la même réponse pour W et pour $W^{\neg j}$ et est donc incorrect pour l'une de ces entrées. □

La Proposition 6.1 implique que la complexité de MULTIPLE DE 3 en RAM 8 bits est $\Omega(n)$, puisqu'une même instruction élémentaire ne peut examiner que $O(1)$ bits de l'entrée. Les problèmes PARITÉ et DIVISIBLE PAR 3 ont donc des complexités différentes, puisqu'étant respectivement $O(1)$ et $\Omega(n)$.

6.2.2 Exemple avancé : recherche dans un tableau trié

Intéressons-nous maintenant au problème suivant, déjà étudié au Chapitre 3 :

RECHERCHE DANS UN TABLEAU TRIÉ

Entrée : Un tableau $T[1..n]$ d'entiers tel que $T[1] \leq T[2] \leq \dots \leq T[n]$ et un entier x.

Sortie : Un indice i tel que $T[i] = x$ ou -1 si cet indice n'existe pas.

En RAM taille arbitraire, la réponse est de taille $O(1)$, aussi les arguments de taille sont inopérants.

Proposition 6.2. *Pour tout algorithme $\mathcal{A}$ qui résout* RECHERCHE DANS UN TABLEAU TRIÉ *et pour tout entier n, il existe une entrée $(T[1..n], x)$ pour laquelle $\mathcal{A}$ accède à $\Omega(\log n)$ cases de $T[]$.*

À nouveau, remarquons que cela implique immédiatement que la complexité de RECHERCHE DANS UN TABLEAU TRIÉ en RAM taille arbitraire est $\Omega(\log n)$, puisqu'une même instruction élémentaire ne peut examiner que $O(1)$ cases de l'entrée. Comme ce problème peut être résolu en temps $O(\log n)$ par recherche dichotomique, sa complexité est $\Theta(\log n)$.

La preuve de la Proposition 6.1 a tiré parti du fait qu'il existe une entrée (notre W) pour laquelle tout algorithme résolvant le problème doit lire $\Omega(n)$ bits. Il s'avère qu'il n'existe aucune entrée simultanément défavorable pour *tous* les algorithmes résolvant RECHERCHE DANS UN TABLEAU TRIÉ. [4] La preuve de la Proposition 6.2 est donc plus délicate que celle de la Proposition 6.1, et nous amène à mieux expliciter le principe d'argument d'adversaire.

Interception des accès mémoire lors de l'exécution d'un algorithme

Informellement, un argument d'adversaire est une expérience de pensée dans laquelle on exécute un algorithme $\mathcal{A}$ en interceptant ses accès à la zone mémoire contenant l'entrée, et en les transmettant à un algorithme $\mathcal{B}$, conçu par nos soins et appelé *adversaire*, afin que $\mathcal{B}$ construise l'entrée (idéalement défavorable à $\mathcal{A}$) au fil des accès mémoire de $\mathcal{A}$. Détaillons cela !

3. Il se cache une subtilité ici : puisque l'entrée est la représentation binaire d'un entier strictement positif, $\mathcal{A}$ sait que $x_n = 1$ sans le lire et peut déterminer l'entrée complète en ne lisant que n bits.

4. Pour le prouver, il suffit de fixer une entrée $(T[1..n], x)$ arbitraire et de montrer qu'il existe un algorithme résolvant RECHERCHE DANS UN TABLEAU TRIÉ qui est efficace sur cette entrée. On fait cela à l'exercice 1 du TD6.

On formalise cela en modifiant le modèle de calcul RAM (8 bits ou taille arbitraire) comme suit : on divise l'unité mémoire en une *mémoire d'entrée*, contenant l'entrée et en lecture seule, et une *mémoire de travail* en lecture et écriture. On ajoute un troisième composant, appelons-le **moniteur**, qui fait l'interface entre l'unité de calcul et la mémoire d'entrée. Lors de l'exécution d'un algorithme $\mathcal{A}$ dans ce modèle, le moniteur peut au choix charger une entrée et la lire fidèlement, ou exécuter un algorithme $\mathcal{B}$, appelé **adversaire**, qui construit l'entrée à mesure qu'il reçoit les requêtes de $\mathcal{A}$.

L'adversaire

L'algorithme $\mathcal{B}$, l'adversaire, est **interactif**. Il consiste en deux sous-algorithmes :

- Un **bloc d'initialisation**, que l'on appelle avant l'exécution de $\mathcal{A}$. Ce bloc reçoit un argument n indiquant la taille de l'entrée à simuler, et ne retourne rien.
- Un **bloc requête**, que l'on appelle pendant l'exécution de $\mathcal{A}$ à chaque fois que $\mathcal{A}$ accède à la mémoire d'entrée. Au $i$ème appel de ce bloc, il reçoit un argument a_i indiquant la portion de l'entrée à laquelle $\mathcal{A}$ souhaite accéder (n° de bit, indice de tableau, ...) et retourne l'information v_i que $\mathcal{A}$ y lit.

Du point de vue de $\mathcal{A}$, l'adresse a_1 de la première requête est indépendante de l'entrée puisqu'au moment où $\mathcal{A}$ l'exécute, il ne « connaît » rien de l'entrée. En revanche, l'adresse a_2 de la deuxième requête de $\mathcal{A}$ peut varier selon la réponse v_1 qu'il a reçu. Plus généralement, l'adresse a_i de la $i$ème requête de $\mathcal{A}$ dépend de la séquence $(a_1, v_1, a_2, v_2, \ldots, a_{i-1}, v_{i-1})$. L'algorithme $\mathcal{B}$, l'adversaire, est dit **interactif** car v_i est choisi avant de connaitre a_{i+1}, autrement dit en ne connaissant que la séquence $(a_1, v_1, a_2, v_2, \ldots, a_{i-1}, v_{i-1}, a_i)$.

Illustration. Voici, à titre d'exemple, l'adversaire que l'on va utiliser pour prouver la Proposition 6.2 :

Initialisation. Créer deux variables g et d et les initialiser à $g = 1$ et $d = n$.

Traitement de la requête a_i.

```
si la requête demande la valeur de x répondre 1.
sinon
    si g=i=d répondre 1
    si i < g répondre 0
    si i > d répondre 2
    sinon
      si i-g > d-i
        d=i-1 et répondre 2
      sinon
        g=i+1 et répondre 0
```

Séquence d'accès et compatibilité

L'interaction entre un algorithme $\mathcal{A}$ et un adversaire $\mathcal{B}$ produit une séquence d'accès $(a_1, v_1, a_2, v_2, \ldots)$ qui se termine lorsque $\mathcal{A}$ termine. Une séquence d'accès $(a_1, v_1, a_2, v_2, \ldots, a_k, v_k)$ est **compatible** avec une entrée e si pour tout $1 \leq i \leq k$, la valeur d'adresse a_i dans e est v_i. Un algorithme interactif $\mathcal{B}$ est un **adversaire pour un problème** P si pour tout algorithme $\mathcal{A}$ résolvant P, la séquence d'accès formée par l'interaction entre $\mathcal{A}$ et $\mathcal{B}$ est compatible avec au moins une entrée.

Observation 6.2.1. *Soit P un problème algorithmique, $\mathcal{A}$ un algorithme résolvant P, et $\mathcal{B}$ un adversaire pour P. Il existe une réponse simultanément correcte pour toutes les entrées de P compatibles avec la séquence d'accès formée par l'interaction de $\mathcal{A}$ avec $\mathcal{B}$ jusqu'à terminaison de $\mathcal{A}$.*

Illustration. Fixons un entier n et notons C_n l'ensemble des entrées $(T[1..n], x = 1)$ où $T[1..n]$ est constitué d'un nombre quelconque de 0, d'un unique 1 et d'un nombre quelconque de 2 :

$$[1, 2, 2, 2, 2, \ldots, 2, 2, 2]$$
$$[0, 1, 2, 2, 2, \ldots, 2, 2, 2]$$
$$[0, 0, 1, 2, 2, \ldots, 2, 2, 2]$$
$$\ldots$$
$$[0, 0, 0, 0, 0, \ldots, 0, 1, 2]$$
$$[0, 0, 0, 0, 0, \ldots, 0, 0, 1]$$

Remarquons que chaque élément de C_n est une entrée de taille n valide du problème RECHERCHE DANS UN TABLEAU TRIÉ puisque le tableau $T[]$ est trié.

Ambiguïté

Une séquence d'accès $(a_1, v_1, a_2, v_2, \ldots, a_k, v_k)$ est **ambigüe pour un problème algorithmique** P si cette séquence d'accès est compatible avec deux entrées e et e' ne pouvant avoir une même réponse pour P, *i.e.* telles que $P(e) \cap P(e') = \emptyset$.

Observation 6.2.2. *Soit P un problème algorithmique, $\mathcal{A}$ un algorithme résolvant P, et $\mathcal{B}$ un adversaire pour P. Tant que la séquence d'accès produite par l'interaction de $\mathcal{A}$ et $\mathcal{B}$ est ambigüe pour P, $\mathcal{A}$ ne peut pas terminer.*

Illustration. Chaque élément de C_n est déterminé par la position du 1. Cette position du 1 se trouve aussi être la seule réponse possible au problème RECHERCHE DANS UN TABLEAU TRIÉ sur cette entrée. Il n'existe donc aucune réponse au problème qui est correcte pour plus d'une entrée de C_n.

Lemme 6.3. *Considérons une séquence d'accès $S = (a_1, v_1, a_2, v_2, \ldots, a_k, v_k)$ produite par l'interaction de l'adversaire avec un algorithme $\mathcal{A}$. Notons d_i et g_i les valeurs prises par les variables d et g immédiatement après le traitement de la requête a_i. Une entrée $e \in C_n$ est compatible avec S si et seulement si la position du 1 dans e appartient à l'intervalle $[g_k, d_k]$.*

Démonstration. Dès que $g > 1$ la case d'indice $g-1$ a fait l'objet d'une requête et d'une réponse valant 0. De même, dès que $d < n$ la case d'indice $d+1$ a fait l'objet d'une requête et d'une réponse valant 2. De plus, aucune case d'indice appartenant à $[g, d]$ n'a fait l'objet d'une requête. Comme g est croissant et d est décroissant au cours de l'interaction, l'énoncé en découle. □

On peut maintenant mener l'analyse :

Lemme 6.4. *Pour tout algorithme $\mathcal{A}$ et pour tout $0 \leq k \leq n$, la séquence $(a_1, v_1, a_2, v_2, \ldots, a_k, v_k)$ produite par l'interaction de $\mathcal{A}$ avec notre adversaire est compatible avec au moins $\lceil \frac{n}{2^k} \rceil$ entrées de C_n.*

Démonstration. L'assertion est vraie pour $k = 0$. D'après le Lemme 6.3, le nombre d'entrées candidates compatibles avec les réponses de l'adversaire est exactement $d - g + 1$. Ce nombre ne change qu'aux lignes 7 et 9 de l'adversaire, et est remplacé par $\max(j - g, d - j) \geq \frac{d-g+1}{2}$. Une récurrence immédiate permet de conclure. □

La preuve de la Proposition 6.2 est à ce stade facile. Si $\lceil \frac{n}{2^k} \rceil > 2$, le Lemme 6.4 assure que toute séquence de k requêtes et réponses est ambigüe pour RECHERCHE DANS UN TABLEAU TRIÉ. Tout algorithme $\mathcal{A}$ résolvant RECHERCHE DANS UN TABLEAU TRIÉ doit donc faire dans le pire cas au moins k accès avec

$$\frac{n}{2^k} < 2 \quad \Leftrightarrow \quad \log_2 n \leq k + 1,$$

soit $\Omega(\log n)$ accès.

6.2.3 Conception d'arguments d'adversaires

Les arguments d'adversaire permettent d'établir que tout algorithme résolvant un problème donné doit lire une portion minimum de l'entrée. Comme on l'a vu, cette portion se traduit en borne inférieure sur la complexité de l'algorithme en car toute instruction élémentaire lit $O(1)$ cases mémoire.

> Un argument d'adversaire permet au mieux de prouver une borne inférieure $\Omega(n)$.

Voici quelques conseils pour élaborer un argument d'adversaire pour un problème P donné :

1. Commencer par se demander s'il existe un algorithme qui résout P sans lire, dans le cas le pire, *essentiellement toute* l'entrée.
2. Si la réponse à la question 1. est négative, il est raisonnable de commencer par essayer de procéder comme pour DIVISIBLE PAR 3 : chercher une entrée W telle que pour toute portion non lue, il est possible de changer cette portion pour modifier la réponse à P.
3. Si la réponse à 1. est positive ou que les tentatives 2. sont infructueuses, on peut chercher à **restreindre** le problème P à un sous-ensemble d'entrées facile à contrôler et analyser, comme l'ensemble C_n dans notre traitement de RECHERCHE DANS UN TABLEAU TRIÉ.

6.3 Arbres de décision

Notre seconde technique opère dans le modèle des *arbres de décision*, proche des *arbres d'exécution* vus au Chapitre 4. Ce modèle est assez différent des modèles RAM 8 bits/taille arbitraire et s'avère bien adapté à la preuve de bornes inférieures. Nous allons l'illustrer sur le problème suivant : [5]

> TRI (VERSION PERMUTATION)
>
> **Entrée :** Un tableau $T[1..n]$ d'entiers deux à deux distincts.
>
> **Sortie :** Une permutation $P[1..n]$ telle que $T[P[1]] < T[P[2]] < \ldots < T[P[n]]$.

Il s'agit d'une reformulation du problème de tri standard, mais l'on demande de calculer non pas le tableau trié mais la permutation qui le réordonne. C'est un bon exercice de vérifier que l'on peut modifier l'algorithme classique de tri fusion de manière à ce qu'il résolve ce problème.

L'intuition

Intuitivement, un arbre de décision modélise la recherche d'un objet au travers d'une séquence de questions, selon le principe du jeu « *qui est-ce ?* ». [6] Chaque question fournit un indice sur l'objet cherché. Chaque réponse à une question détermine la question suivante. Les questions s'arrêtent dès que l'ensemble d'indices accumulé ne laisse qu'une seule possibilité de réponse. Ce processus est modélisé par un arbre tel que celui de la Figure 6.1.

Le modèle

Un **arbre de décision** est un arbre enraciné fini $\mathcal{D}(n)$. [7] Chaque nœud interne est étiqueté par une question, appelée **requête**, portant sur l'entrée e. Un nœud interne a autant de descendants qu'il y a de réponses possibles à sa requête, et chaque arête qui le joint à un fils est étiquetée par une réponse différente. Chaque feuille est étiquetée par un résultat. **Exécuter** un arbre $\mathcal{D}(n)$ sur une entrée e (de taille n) consiste à parcourir l'arbre en partant de la racine, et en suivant à chaque nœud interne l'arête étiquetée par la réponse à la requête de ce nœud sur l'entrée e. Le **résultat** de l'exécution de $\mathcal{D}(n)$ sur e est l'étiquette de la feuille à laquelle le parcours aboutit.

Formellement, un arbre $\mathcal{D}(n)$ **résout** un problème $P : E \to 2^S$ si pour toute entrée e de taille n, le résultat de l'exécution de $\mathcal{D}(n)$ sur e appartient à $P(e)$.

5. Rappel : un tableau $P[1..n]$ est une *permutation* s'il contient les valeurs $1, 2, \ldots, n$ exactement une fois chacune.
6. https://fr.wikipedia.org/wiki/Qui_est-ce_?
7. À nouveau, formellement, il s'agit d'une famille d'arbres paramétrée par n, c'est-à-dire une fonction de $\mathbb{N}$ dans l'ensemble des arbres enracinés.

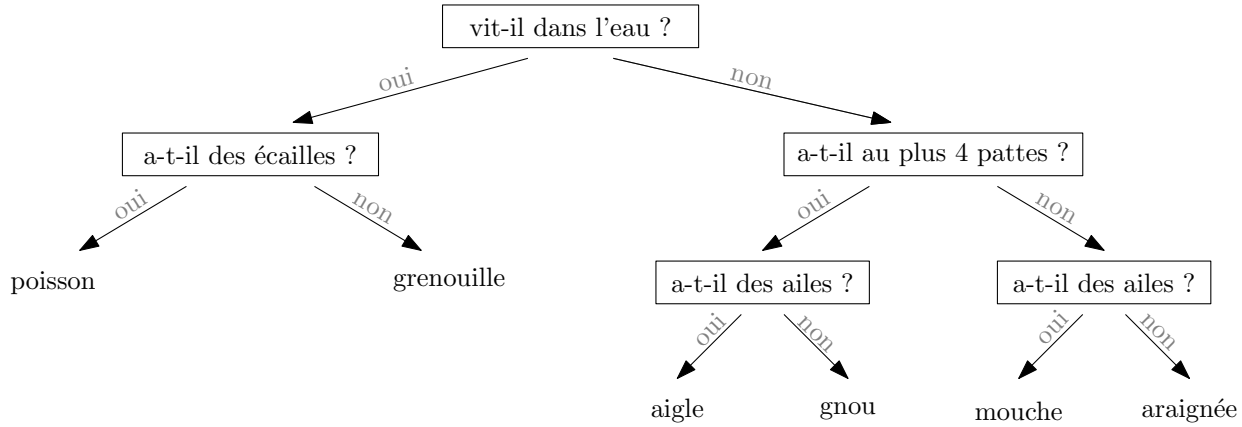


FIGURE 6.1 – Un exemple d'arbre représentant la recherche d'un animal parmi {aigle, araignée, gnou, grenouille, mouche, poisson}.

Exemple : tri par arbre de décision

Considérons l'algorithme de tri-fusion modifié pour résoudre TRI (VERSION PERMUTATION). Cet algorithme n'examine l'entrée $T[1..n]$ qu'au travers de *comparaisons* du type « $T[i]$ *est-il plus grand ou plus petit que* $T[j]$ *?* » et traite deux entrées distinctes de la même manière tant que les comparaisons produisent le même résultat. Ces deux observations suffisent à garantir que l'on peut le traduire en un arbre de décision $\mathcal{D}_{\texttt{fusion}}(n)$. Voici par exemple $\mathcal{D}_{\texttt{fusion}}(4)$:

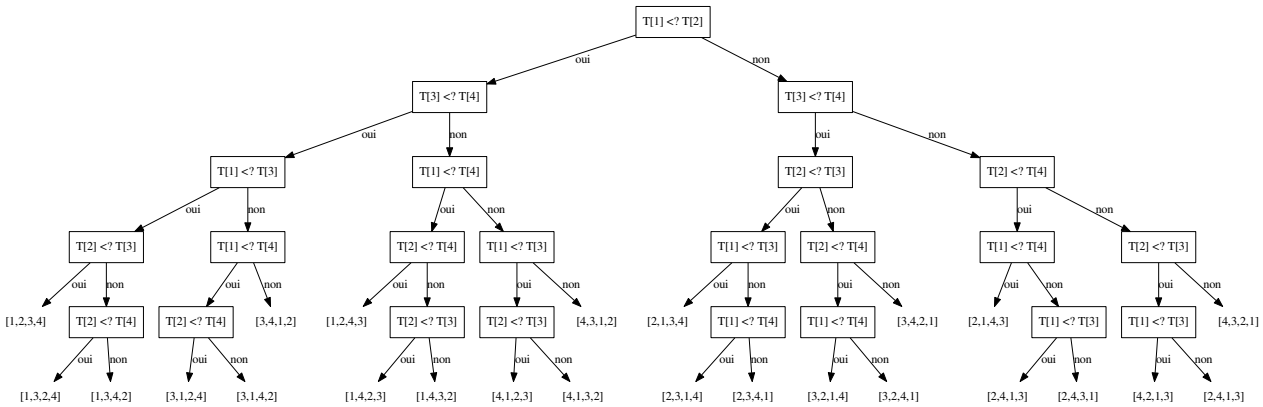


FIGURE 6.2 – Arbre $\mathcal{D}_{\texttt{fusion}}(4)$ décrivant l'algorithme de tri-fusion modifié pour résoudre TRI (VERSION PERMUTATION).

D'autres algorithmes de tri se traduisent en arbre de décision, par exemple l'algorithme *tri_ rapide* lorsque l'on choisit comme pivot un élément de position ou de rang donné. On désigne souvent de tels algorithmes comme étant « basés sur des comparaisons » (*comparison-based algorithms*). Remarquons que ce type de traduction a la propriété suivante :

> La complexité pire-cas d'un algorithme dans le modèle RAM taille arbitraire est supérieure ou égale à la hauteur de l'arbre de décision correspondant.

Un argument de théorie de l'information

On peut maintenant établir une borne inférieure sur la complexité de *tous* les algorithmes de tri basés sur des comparaisons en combinant trois observations :

a. Pour toute permutation σ de taille n, il existe une entrée $T_\sigma[1..n]$ du problème de TRI (VERSION PERMUTATION) pour laquelle la seule réponse acceptable est σ. Tout arbre de décision $\mathcal{D}(n)$ qui résout TRI (VERSION PERMUTATION) a donc au moins $n!$ feuilles.

b. Les comparaisons autorisent au plus 2 réponses (« inférieur » et « supérieur »). Ainsi, tout algorithme de tri basé sur des comparaisons se traduit en arbre de décision $\mathcal{D}(n)$ d'arité 2, c'est-à-dire en *arbre binaire*.

c. Un arbre binaire de hauteur h a au plus 2^h feuilles (c.f. Chapitre 4).

Ainsi, tout arbre de décision qui résout TRI (VERSION PERMUTATION) pour $n = 4$ a au moins $4! = 24$ feuilles, et donc une hauteur au moins 5. En particulier, l'arbre $D_{\texttt{fusion}}(4)$ de la Figure 6.2 est de hauteur optimale pour le tri de quatre éléments. On obtient ainsi :

Proposition 6.5. *Tout arbre de décision qui résout* TRI (VERSION PERMUTATION) *est de hauteur* $\Omega(n \log n)$.

Démonstration. Tout arbre de décision $\mathcal{D}(n)$ qui résout TRI (VERSION PERMUTATION) est de hauteur au moins $\log_2(n!)$. L'équivalent de Stirling implique que $\log_2(n!) = \Omega(n \log n)$. □

Lien entre arbres de décision et modèles RAM

De manière générale, une borne inférieure obtenues par arbres de décision **n'implique pas** une borne inférieure dans le modèle RAM 8 bits / taille arbitraire. On a par exemple vu à l'exercice 1 du TD 2 que la variante suivante du problème de tri peut être résolue en temps $O(n)$

INVERSION
Entrée : Un tableau de permutation $T[1..n]$.
Sortie : L'inverse $I[1..n]$ de $T[1..n]$.

(Une solution consiste à affecter `I[T[i]] =i` pour `i=1..n`.) En particulier, la Proposition 6.5 **n'implique pas** que la complexité de TRI VERSION PERMUTATION est $\Omega(n \log n)$ en RAM taille arbitraire. [8]

Néanmoins, **certains** algorithmes RAM peuvent se traduire en arbre de décision. De manière informelle, lors de l'exécution d'un algorithme $\mathcal{A}$ par un modèle RAM, on peut représenter l'**état du modèle** par un ensemble fini d'informations : le contenu de la mémoire et la position du pointeur d'instruction dans le programme. L'exécution d'une instruction modifie cet état du modèle ; le « degré de branchement » d'une instruction est le nombre d'états différents auxquels son exécution peut conduire. Les algorithmes du modèle RAM qui n'utilisent que des instructions de « degré de branchement » borné peuvent se modéliser par des arbres de décision ; ainsi, la Proposition 6.5 implique qu'*en RAM taille arbitraire, tout algorithme de tri qui n'accède aux données qu'au travers de comparaisons est de complexité* $\Omega(n \log n)$. [9] En revanche, les algorithmes du modèle RAM qui utilisent des instructions de degré non-borné, comme `I[T[i]] =i`, ne sont pas modélisables dans le modèle des arbres de décision. Formaliser cela demanderait des développements qui débordent du cadre de ce cours.

6.4 Prolongement

Les bornes inférieures dans le modèle des arbres de décision ne sont intéressantes que pour des problèmes dont **l'espace des réponses est de grande taille** puisque la borne est, *in fine*, un logarithme de cette taille. Ces bornes sont en particulier inopérantes sur un problème de décision comme TOUS DISTINCTS. Le modèle des **arbres de calcul algébrique** permet de dépasser ces deux problèmes au moyen d'outils de géométrie algébrique, et d'établir par exemple une borne inférieur de $\Omega(n \log n)$ sur TOUS DISTINCTS.

8. Le responsable du cours n'a pas connaissance d'une borne inférieure meilleure que $\Omega(n)$ pour ce problème dans ce modèle.

9. Cela nous apprend aussi que s'il existe un algorithme pour TRI VERSION PERMUTATION de complexité $o(n \log n)$, il doit utiliser des instructions de « degré de branchement » non borné.

Les arguments d'adversaires se contentent de compter le nombre d'accès mémoire réalisé et sont incapables de prouver des bornes inférieures meilleures que $\Omega(n)$. Il peut être tentant de vouloir rendre un adversaire plus puissant en lui permettant non seulement de gérer les accès mémoire d'un algorithme $\mathcal{A}$ générique, mais en lui permettant d'analyser une « description » de $\mathcal{A}$. Cela soulève naturellement des questions pratiques, par exemple le choix d'une présentation d'un algorithme, mais plus fondamentalement, cela risque surtout de se heurter au théorème d'indécidabilité de Rice. [10]

À retenir.

- La complexité d'un problème dans un modèle de calcul est la meilleure complexité d'un algorithme pour ce problème dans ce modèle.
- On peut minorer la complexité d'un problème par le pire-cas de la taille de la sortie.

- Un adversaire pour un problème algorithmique P est un algorithme $\mathcal{B}$ qui interagit avec n'importe quel algorithme $\mathcal{A}$ résolvant P. L'adversaire traite les requêtes de $\mathcal{A}$ d'accès à l'entrée, et y répond *en ligne* : la réponse à la $i$ème requête est envoyée avant que la $(i+1)$ème requête ne soit reçue. Tant que les réponses de $\mathcal{B}$ sont compatibles avec deux entrées n'admettant pas de réponse commune, $\mathcal{A}$ ne peut pas conclure.
- On peut minorer la complexité d'un problème par une traduction en arbre de décision. Cela ne s'applique qu'aux algorithmes « à branchement borné ». La borne obtenue est de l'ordre du logarithme du nombre de réponses possibles.

10. https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Rice

Chapitre 7

Réduction

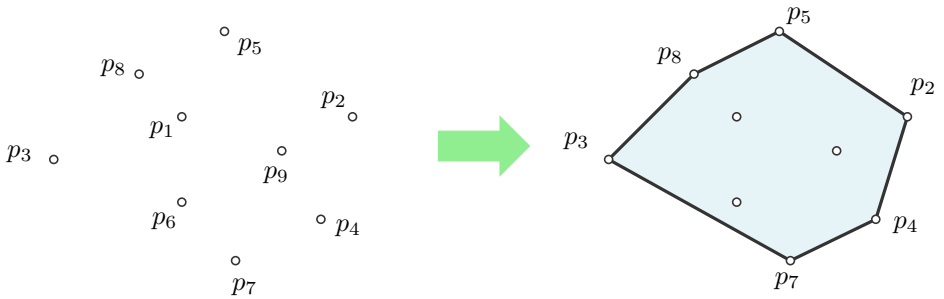
Cette séance introduit les réductions entre problèmes. On commence par quelques exemples d'utilisation de cet outil pour propager des bornes inférieures de complexité entre problèmes algorithmiques. On introduit ensuite l'idée de bornes inférieures conditionnelles, idée que l'on approfondira au chapitre suivant.

Objectifs. À l'issue de cette séance, il est attendu que vous

- sachiez réduire un problème à un autre dans des cas simples,
- sachiez propager une borne inférieure par réduction,
- compreniez le principe de borne inférieure conditionnelle,
- soyiez familier avec les problèmes de référence COLORATION, SATISFIABILITÉ et 3-SUM.

7.1 Premier exemple : calcul d'enveloppe convexe

Commençons par faire un peu de géométrie. Soit P un ensemble de points du plan $\mathbb{R}^2$. L'**enveloppe convexe** de P est définie comme[1] l'intersection de tous les demi-plans contenant P ; elle est notée $\text{conv}(P)$. Lorsque P est fini, $\text{conv}(P)$ est un polygone convexe dont les sommets sont dans P :



L'enveloppe convexe d'une séquence $P = [p_1, p_2, \ldots, p_n]$ peut être décrite par la liste circulaire[2] des indices de points apparaissant sur l'enveloppe convexe lorsqu'on la parcourt dans l'ordre trigonométrique. Dans l'exemple ci-dessus, l'enveloppe convexe des points $\{p_1, p_2, \ldots, p_9\}$ est donc $[3, 7, 4, 2, 5, 8]$. Appelons cela ici la *représentation par liste circulaire* de l'enveloppe convexe.

Le calcul de l'enveloppe convexe d'un ensemble de points donné est un problème classique de géométrie algorithmique. Il se formalise ainsi :

CALCUL D'ENVELOPPE CONVEXE

Entrée : Un tableau $P[1..n]$ où $P[i]$ est un point du plan.

Sortie : La représentation par liste circulaire de l'enveloppe convexe de $\{P[1], P[2], \ldots, P[n]\}$.

1. Si aucun demi-plan ne contient P alors l'enveloppe convexe est définie comme $\mathbb{R}^2$ tout entier. Dans la suite, ce cas ne se produit pas car on prendra P fini.
2. Par convention, on représente une liste circulaire par n'importe laquelle des listes obtenue en la coupant. Ainsi la liste circulaire $\ldots 3, 6, 9, 3, 6, 9, \ldots$ est représentable par $[3, 6, 9]$, $[6, 9, 3]$ ou $[9, 3, 6]$.

Chaque point est donné par ses coordonnées cartésiennes $P[i] = (x_i, y_i)$, chaque coordonnée étant encodée en un mot binaire fini et stocké dans une case mémoire.[3] Examinons la complexité de ce problème...

L'algorithme de Graham

Un algorithme classique pour Calcul d'enveloppe convexe peut se résumer[4] comme suit :

a. Déterminer le point de coordonnée x minimale, notons le p^*.

b. Trier les autres points p de P par ordre croissant des pentes des droites pp^* et noter $P^* = p[\sigma(1)], p[\sigma(2)], p[\sigma(3)], \ldots, p[\sigma(n)]$ la séquence obtenue. Remarquer que P^* forme un polygone sans auto-intersection mais pas nécessairement convexe.

c. Parcourir P^* en partant du début jusqu'à trouver une concavité

d. Si une concavité est trouvée, supprimer de P^* le sommet concave et reprendre le parcours du (c) au sommet qui précède le sommet supprimé.

e. Lorsque le parcours termine, retourner les indices des sommets restants dans P^*, dans l'ordre où ils apparaissent dans P^*.

Pour comprendre le déroulement des étapes (c), (d) et (e), l'animation disponible à

https://en.wikipedia.org/wiki/Graham_scan

vaut très certainement mieux que de longues explications.

Esquissons l'analyse de complexité de cet algorithme en RAM taille arbitraire. L'étape (a) peut se faire en $O(n)$ et l'étape (b) en $O(n \log n)$ si l'on utilise un algorithme comme le `tri_fusion`. Le coût du parcours (c-d-e) est $O(n)$ puisqu'on détecte $O(n)$ concavités (chacune conduit à supprimer un sommet, donc il y en a au plus $n-3$) et chacune est traitée en temps $O(1)$. Au final, l'algorithme de Graham est de complexité $O(n \log n)$ en RAM taille arbitraire.

Calcul d'enveloppe convexe est de complexité $O(n \log n)$ en RAM taille arbitraire.

Un examen attentif de la description ci-dessus révèle que la présentation de l'algorithme de Graham ne fait aucun usage abusif d'instruction décomposable.

Un exemple de réduction linéaire

Soulignons la manière dont l'algorithme de Graham utilise le tri comme sous-routine :

- L'étape (a) part d'une entrée de Calcul d'enveloppe convexe (une séquence de n points) et construit en temps $O(n)$ une entrée de[5] Tri (une séquence de n nombres, les pentes les droites $p^*P[i]$).
- Les étapes (c), (d),(e) partent d'une solution à ce problème de Tri et en déduisent, en temps $O(n)$, une solution à Calcul d'enveloppe convexe sur l'entrée initiale.

Cette construction permet ainsi de transformer tout algorithme pour Tri en un algorithme pour Calcul d'enveloppe convexe modulo un pré-traitement de l'entrée et un post-traitement de la sortie. Une telle transformation s'appelle une *réduction* (on en donnera une définition formelle ci-après). Comme les pré- et post-traitements se font en temps $O(n)$, on parle de *réduction linéaire*.

Cette réduction permet non seulement de construire un algorithme pour Calcul d'enveloppe convexe, mais aussi de comparer les complexités **des problèmes** Tri et Calcul d'enveloppe convexe :

3. En particulier, les coordonnées ne sont pas des réels arbitraires.

4. Cette description est faite pour un ensemble P en *position générale* : un seul point est d'abscisse minimale, aucun triplet de point n'est aligné, etc. C'est une pratique courante en géométrie algorithmique, l'idée étant qu'une fois l'algorithme compris dans ce cadre là il est facile de l'adapter. C'est souvent vrai (et c'est le cas ici).

5. On utilise ici Tri pour désigner la classe algorithmique de problèmes consistant à trier n nombres donnés en entrée. On n'explicite pas le type de nombre autorisé lorsque le contexte permet de le déterminer facilement (ici il découle du type de nombre autorisés pour l'entrée de Calcul d'enveloppe convexe).

S'il existe un algorithme de complexité $o(n \log n)$ pour TRI alors il existe un algorithme de complexité $o(n \log n)$ pour CALCUL D'ENVELOPPE CONVEXE.

En contraposant cette affirmation, on obtient :

Si $\Omega(n \log n)$ est une borne inférieure pour CALCUL D'ENVELOPPE CONVEXE alors c'est aussi une borne inférieure pour TRI.

Plus généralement, toute borne inférieure superlinéaire pour CALCUL D'ENVELOPPE CONVEXE s'étend à TRI. Tout cela (et les encadrés ci-dessus) est valide dans un modèle de calcul dans lesquels les algorithmes de pré- et post-traitement ont un sens et sont de complexités linéaires.

Cette réduction ne s'avère pas très utile puisque l'on « sait » déjà[6] que le problème de tri est de complexité $\Theta(n \log n)$. Il s'avère que l'on peut faire la réduction inverse...

La réduction réciproque...

Considérons une instance du problème TRI, c'est-à-dire un tableau $T[1..n]$ de n nombres. On le réduit à CALCUL D'ENVELOPPE CONVEXE comme suit.

Le *pré-traitement* consiste à calculer le tableau $P[1..n] = [p_1, p_2, \ldots, p_n]$ où $P[i]$ est le point de coordonnées $(T[i], T[i]^2)$. L'ensemble $P = \{p_1, p_2, \ldots, p_n\}$ a deux propriétés faciles à montrer :

- chaque point $P[i]$ apparaît sur l'enveloppe convexe,
- si on parcourt les sommets de l'enveloppe convexe de P dans l'ordre trigonométrique en partant du point le plus à gauche (x minimal), les points $P[i]$ apparaissent dans l'ordre de $T[i]$ croissants.

Supposons que l'on ait calculé une représentation par liste circulaire $R[1..n]$ de l'enveloppe convexe de P. Le *post-traitement* consiste dès lors à calculer l'indice i^* tel que $T[i]$ soit minimal, rechercher la position de i^* dans $R[1..n]$, puis opérer une permutation circulaire de $R[1..n]$ pour obtenir un tableau $R'[1..n]$ où i^* est en première position. Le tri du tableau $T[1..n]$ est alors $[T[R'[1]], T[R'[2]], \ldots, T[R'[n]]]$. On a donc :

TRI se réduit linéairement à CALCUL D'ENVELOPPE CONVEXE.

Ainsi, dans tout modèle de calcul qui permet les deux réductions en temps linéaire, le **problème** CALCUL D'ENVELOPPE CONVEXE a la même complexité que le problème TRI. Il semble donc difficile d'améliorer l'algorithme de Graham...

7.2 Formalisation du mécanisme de réduction

La formalisation de ce mécanisme de réduction fait apparaître quelques complications dans la manière dont il propage les bornes de complexité. Cela nous amène à restreindre notre attention à une sous-classe de problèmes algorithmiques, les problèmes de décision, pour lesquels ces complications disparaissent. On revient sur le cas général en prolongements.

Problèmes de décision et leurs réductions

Un **problème de décision** est un problème algorithmique $E \to 2^S$ qui admet exactement deux réponses possibles. Autrement dit, un problème de décision se ramène à une fonction $E \to S$ où E est dénombrable et $|S| = 2$. Pour simplifier la présentation, on suppose ici que $S = \{0, 1\}$ pour tous les problèmes de décision.[7]

Soient $D : E \to \{0, 1\}$ et $D' : E' \to \{0, 1\}$ deux problèmes de décision et supposons fixées des fonctions de taille sur E et sur E'. Une **réduction de D à D'** est une fonction $r : E \to E'$ telle que

6. Les « » renvoient au fait que les bornes inférieures et supérieures ne sont pas dans le même modèle de calcul.
7. Le cas général s'y ramène facilement en fixant une bijection $S \to \{0, 1\}$; cette bijection se calcule en temps $O(1)$ aussi elle n'a aucune conséquence sur notre discussion.

$D = D' \circ r$. L'idée est simple : une réduction permet, partant d'une entrée quelconque x de D, de la transformer en une entrée $y = r(x)$ de D' telle que la réponse à D sur x égale la réponse à D' sur y.

La **complexité d'une réduction** r est la complexité du calcul de cette fonction. On note $D \leq_{f(n)} D'$ le fait qu'il existe une réduction de D à D' de complexité $f(n)$.

Réductions et bornes de complexité

La restriction aux problèmes de décision permet d'avoir une propagation raisonnablement simple des bornes de complexités :

Proposition 7.1. *Soient D et D' deux problèmes de décision avec $D \leq_{f(n)} D'$, et g une fonction telle que $f = o(g)$. Les assertions suivantes sont vraies :*

(i) si D' est de complexité $O(g(n))$ alors D est de complexité $O(g \circ f(n))$,

(ii) si D est de complexité $\Omega(g \circ f(n))$ alors D' est de complexité $\Omega(g(n))$.

Démonstration. Fixons des fonctions de taille sur les ensembles d'entrées E et E' de, respectivement, D et D'. Notons r une réduction de complexité $O(f(n))$ de D à D' et $\mathcal{A}_r$ un algorithme de complexité $O(f(n))$ calculant r. Soit g une fonction telle que $f = o(g)$.

Pour (i), supposons qu'il existe un algorithme $\mathcal{A}_{D'}$ calculant D' et de complexité $O(g(n))$. Pour toute entrée $e \in E$, $\mathcal{A}_r$ calcule $e' = r(e)$ en temps $O(f(n))$ et la taille de e' est $O(f(n))$.[8] L'algorithme $\mathcal{A}_{D'}$ appliqué à e' fournit $D'(e') = D(e)$ en temps $O(g \circ f(n))$. Le problème D est bien de complexité $O(g \circ f(n) + f(n)) = O(g \circ f(n))$.

Pour (ii), raisonnons par contraposée et supposons que D' ne soit pas de complexité $\Omega(g(n))$. C'est donc que D' est de complexité $o(g)$. Choisissons h une fonction telle que D' est de complexité $O(h)$ tout en gardant $f = o(h)$; on utilise ici le fait que h n'est pas nécessairement égal à la complexité de D' mais la domine simplement.Le (i) assure que D est de complexité $O(h \circ f(n))$. Or

$$h = o(g) \text{ et } f \to \infty \quad \Rightarrow \quad h \circ f = o(g \circ f).$$

donc D n'est pas de complexité $\Omega(g \circ f(n))$. □

Ainsi, si $D \leq_{O(n)} D'$ alors toute borne inférieure superlinéaire pour D est une borne inférieure pour D', et toute borne supérieure superlinéaire pour D' est une borne supérieure pour D. Si $D \leq_{f(n)} D'$ avec f non-linéaire, les bornes se propagent similairement mais avec « un peu de perte ».

Composition de réductions

Soulignons que l'on peut composer les réductions :

Lemme 7.2. *Soient D, D' et D'' trois problèmes de décision. Si $D \leq_{f(n)} D'$ et $D' \leq_{g(n)} D''$ alors $D \leq_{f(n)+g\circ f(n)} D''$.*

Démonstration. Notons r la réduction de D à D' et r' la réduction de D' à D''. Pour toute entrée e de D on a

$$D(e) = D'(r(e)) = D''(r'(r(e)))$$

aussi $r' \circ r$ est une réduction de D a D''. Notons n la taille de e et remarquons que si r est de complexité $f(n)$ alors la taille de $r(e)$ est au plus $f(n)$ puisque le calcul de r requiert d'écrire la sortie $r(e)$.[9] Ainsi, le calcul de $r'(r(e))$ peut se faire en temps $O(f(n) + g \circ f(n))$. □

En particulier, si $D \leq_{O(n)} D'$ et $D' \leq_{O(n)} D''$ alors $D \leq_{O(n)} D''$.

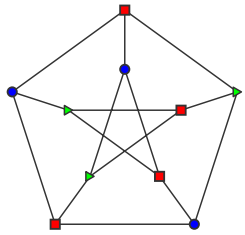
7.3 Deux problèmes de référence

Examinons maintenant deux problèmes de décision classiques (et pratiquons la réduction !).

8. Cette assertion suppose que les fonctions de taille sont compatibles en un certain sens ; c'est par exemple le cas quand on prend la taille d'encodage ou le nombre de case mémoires occupée par l'entrée.

9. Cette assertion suppose, à nouveau, que les notions de taille sont raisonnablement compatibles.

7.3.1 k-coloration de graphe



Le premier problème vient de l'optimisation combinatoire. Soit $G = (V, E)$ un graphe non orienté tel que défini en Section 4.1. Ici V désigne l'ensemble de sommets et E l'ensemble des arêtes. Fixons un entier $k \geq 2$. Une **coloration propre de G par k couleurs** est une fonction $c\colon V \to \{1, 2, \ldots, k\}$ telle que $c(u) \neq c(v)$ pour toute arête $\{u, v\} \in E$.

Considérons le problème suivant :

> k-Col
> **Entrée :** Un graphe $G = (V, E)$ donné par sa matrice d'adjacence $A \in \{0, 1\}^{n \times n}$.
> **Sortie :** 1 si le graphe G est k-coloriable, 0 sinon.

Il est facile de résoudre 2-Col comme suit :

- On peut supposer que le graphe G est connexe. En effet, un graphe non-connexe est 2-coloriable si et seulement si chacune de ses composantes connexes est 2-coloriable.
- Fixons un sommet $u \in V$. Il existe une 2-coloration propre de G si et seulement si il existe une 2-coloration propre de G dans laquelle u est de couleur 1.
- Dans toute 2-coloration propre $c : V \to \{1, 2\}$, pour tout sommet $v \in V$, pour tout voisin w de V, la couleur de w égale $3 - c(v)$.

On peut donc choisir un sommet arbitraire u, fixer sa couleur à 1 et propager cette coloration à ses voisins, puis à leurs voisins, etc. Cette propagation peut aboutir à une contradiction s'il faut propager la couleur $3 - c(v)$ d'un sommet v à un de ses voisins w qui a déjà reçu la couleur $c(v)$ d'un autre de ses voisins. Si cela se produit, alors G d'admet pas[10] de 2-coloration propre. Si aucune contradiction ne se produit, l'algorithme produit une 2-coloration propre de la composante connexe de u.[11]

L'existence d'un algorithme polynomial pour 3-Col est a ce jour un problème ouvert.

7.3.2 k-satisfiabilité de formule booléenne

Le second problème vient du calcul booléen. On se fixe un ensemble fini V, que l'on appelle ensemble des *variables*, et on note $\vee$ l'opérateur `ou`, $\wedge$ l'opérateur `et`, et $\neg$ l'opérateur `non`. Un *littéral* (sur V) est soit un élément de V, soit la négation d'un élément de V. L'ensemble des littéraux est donc $L \stackrel{\text{def}}{=} \cup_{x \in V} \{x, \neg x\}$. Une *clause* (sur V) est une disjonction finie de littéraux, c'est-à-dire une formule de la forme $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_k$ où $k \in \mathbb{N}^*$ et $\ell_i \in L$ pour tout i.[12] Une formule de logique propositionnelle (sur V) est dite en *forme normale conjonctive* (CNF) si c'est une conjonction finie de clauses, c'est-à-dire une formule de la forme $C_1 \wedge C_2 \wedge \ldots \wedge C_t$ où $t \in \mathbb{N}^*$ et chaque C_i est une clause (sur V).[13]

Une *affectation* des variables V est une fonction $V \to \{\texttt{vrai}, \texttt{faux}\}$. Partant d'une affectation des variables, on peut *évaluer* toute formule de logique au moyen des règles de calcul booléen, obtenant ainsi soit `vrai` soit `faux`. Une formule de logique propositionnelle est *satisfiable* s'il existe une affectation des variables pour laquelle elle s'évalue à `vrai`. Le problème qui nous intéresse est le suivant[14] :

10. En effet, les conditions listées ci-dessus sont nécessaires.
11. Exercice : prouvez qu'un graphe est 2-coloriable si et seulement si il ne contient pas de cycle de longueur impaire.
12. Par exemple, $C_1 \stackrel{\text{def}}{=} x \vee \neg y \vee z$ et $C_2 \stackrel{\text{def}}{=} x \vee \neg z$ sont deux clauses (sur $\{x, y, z\}$). En revanche, $C_3 \stackrel{\text{def}}{=} x \wedge y$ n'en est pas une.
13. Par exemple, $x \wedge y$ et $(x \vee \neg y \vee z) \wedge (x \vee \neg z)$ sont des formules de logique propositionnelle en CNF.
14. Précisons que pour toute formule de logique propositionnelle ϕ, il existe une formule de logique propositionnelle f en forme normale conjonctive de mêmes variables telle que f et ϕ ont la même évaluation pour toute affectation des variables. Ainsi, comme leur nom l'indique, les formes CNF sont une sorte de « normalisation ». Cela fait du problème k-Sat un problème fondamental.

k-SAT

Entrée : Une formule de logique propositionnelle f à au plus n variables en forme normale conjonctive ayant au plus n clauses, chacune comportant au plus k littéraux.

Sortie : 1 si f est satisfiable, 0 sinon.

Il existe un algorithme de complexité $O(n)$ pour résoudre 2-SAT [APT79, Théorème 1]. L'existence d'un algorithme polynomial pour 3-SAT est a ce jour un problème ouvert.

7.4 Réductions entre 3-COL et 3-SAT

Utilisons ces problèmes de référence pour pratiquer les réductions.

Un premier exemple : réduction polynomiale de 3-COL à 3-SAT

Réduire 3-COL à 3-SAT revient à se poser la question : *comment résoudrait-on* 3-COL *si l'on disposait d'une « boîte noire » résolvant* 3-SAT*, que l'on ne peut appeler qu'une seule fois ?* La réponse à cette question est essentiellement une tâche de modélisation.

Supposons donné en entrée un graphe $G = (V, E)$ et considérons que $V = \{1, 2, \ldots, n\}$. On définit $3n$ variables $\{x_{i,j} : 1 \leq i \leq n, 1 \leq j \leq 3\}$ et on code une coloration $c : V \to \{1, 2, 3\}$ par l'affectation $x_{i,c(i)} \stackrel{\text{def}}{=} \texttt{vrai}$ et $x_{i,\neq c(i)} \stackrel{\text{def}}{=} \texttt{faux}$. On définit pour $1 \leq i \leq n$

$$\begin{aligned}\psi_i \stackrel{\text{def}}{=} \quad & (x_{i,1} \vee x_{i,2} \vee x_{i,3}) \wedge (\neg x_{i,1} \vee \neg x_{i,2} \vee \neg x_{i,3}) \\ & \wedge (x_{i,1} \vee \neg x_{i,2} \vee \neg x_{i,3}) \wedge (\neg x_{i,1} \vee x_{i,2} \vee \neg x_{i,3}) \wedge (\neg x_{i,1} \vee \neg x_{i,2} \vee x_{i,3}),\end{aligned}$$

formule vraie si et seulement si exactement une des variables $\{x_{i,1}, x_{i,2}, x_{i,3}\}$ est vraie. On définit de même pour $1 \leq u, v \leq n$

$$\chi_{u,v} \stackrel{\text{def}}{=} \bigwedge_{j=1}^{3} \neg x_{u,j} \vee \neg x_{v,j},$$

formule vraie si et seulement si les variables associées à u et v ne sont pas vraies pour une même "couleur" j. Dès lors, il existe une 3-coloration propre de G si et seulement si

$$\psi_G \stackrel{\text{def}}{=} \left(\bigwedge_{i=1}^{n} \psi_i \right) \wedge \left(\bigwedge_{\{u,v\} \in E} \chi_{u,v} \right)$$

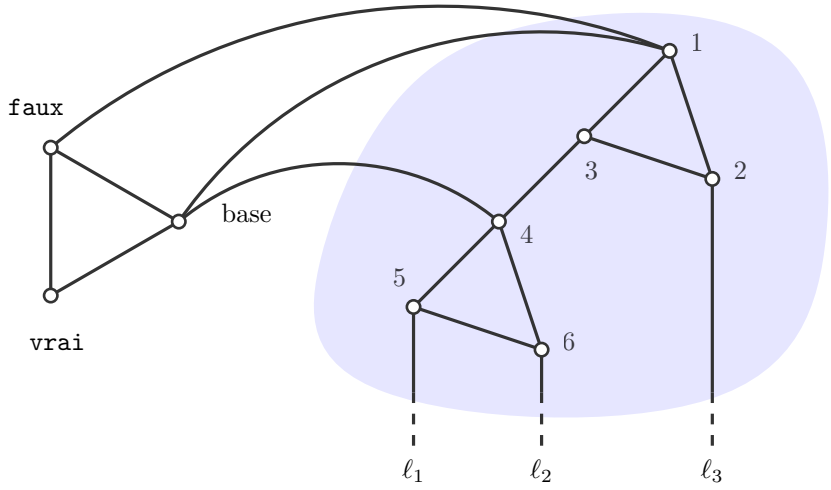
est satisfiable. Si on note $n = |V| + |E|$ la complexité du graphe G, la formule Ψ_G peut être calculée en temps $O(n)$. Ainsi, 3-COL $\leq_{O(n)}$ 3-SAT.

Réduction de 3-SAT à 3-COL

Esquissons une réduction dans l'autre sens. Supposons donnée une formule de logique propositionnelle f à au plus n variables en forme normale conjonctive ayant au plus n clauses, chacune comportant au plus 3 littéraux. On décrit un graphe G_f ayant un nombre polynomial de sommets et pouvant être construit en temps polynomial, tel que G_f est 3-coloriable si et seulement si f est satisfiable.

On commence par construire un triangle dont on nomme les sommets $\{\texttt{vrai}, \texttt{faux}, \texttt{base}\}$. Pour chaque variable x_i de f, on crée dans G_f deux sommets nommés x_i et $\neg x_i$, que l'on relie entre eux et à `base`. Remarquons que dans toute 3-coloration de G_f, exactement l'un des sommets x_i ou $\neg x_i$ est colorié comme le sommet `vrai`. Par abus de langage, on dit que le littéral correspondant (x_i ou $\neg x_i$) « est » vrai dans ce coloriage.

On construit ensuite un « gadget »pour chaque clause $C = \ell_1 \vee \ell_2 \vee \ell_3$ de f. Il s'agit du graphe suivant (les deux triangles reliés par une arête, dans la zone ombrée) :



Les sommets 1 et 4 du gadget sont reliés au sommet `base`. Le sommet 1 est de plus relié au sommet `faux`. Les sommets 5, 6 et 2 sont reliés aux sommets associes aux littéraux ($x_\bullet$ ou $\neg x_\bullet$) de la clause C. Ce gadget a la propriété suivante : *une 3-coloration du triangle* {`vrai`, `faux`, `base`} *et des variables apparaissant dans la clause C peut s'étendre en une 3-coloration du gadget si et seulement si un des littéraux de C a la même couleur que le sommet* `vrai`.

On construit un tel gadget pour chaque clause. Le graphe G_f ainsi obtenu est 3-coloriable si et seulement si f est satisfiable. Notons que G_f a $O(n)$ sommets et $O(n)$ arêtes et peut être calculé en temps $O(n)$. Ainsi, 3-Sat $\leq_{O(n)}$ 3-Col.

7.5 Borne inférieure conditionnelle : problèmes 3-Sum-difficiles

Les réductions permettent de définir des bornes inférieures de complexités *conditionnées* à des conjectures assez simples. Cette idée sera au cœur du Chapitre 8 ; on l'introduit ici sur l'exemple des problèmes 3-Sum-difficiles. Ce sont des problèmes auxquels se réduit le problème élémentaire suivant :

3-Sum

Entrée : $T[1..n]$ un tableau de n entiers positifs ou négatifs.

Sortie : 1 s'il existe $i < j < k$ tels que $T[i] + T[j] + T[k] = 0$, 0 sinon.

Algorithmes pour 3-Sum

En RAM taille arbitraire, on peut résoudre 3-Sum en temps $O(n^3)$ par un simple examen de chaque triplet. On peut le résoudre [15] en $O(n^2 \log n)$ comme suit :

```
solution_subcubique(T[1..n])
   calculer la liste L[1..m] des T[i]+T[j] avec i<j
   trier T et L par ordres croissants
   a,b = 1,m
   tant que (a<n+1 et m>0)
     si T[a] + L[b] = 0 retourner 1
     sinon, si T[a] + L[b] < 0 incrémenter a
     sinon décrémenter b
   retourner 0
```

Un peu d'effort permet d'améliorer cela en une solution quadratique :

15. Les deux algorithmes qui suivent sont incorrects en ce qu'ils ne vérifient pas que i, j et k sont deux à deux distincts. On laisse la correction (facile mais un peu laborieuse) en exercice.

```
solution_quadratique(T[1..n])
   trier T par ordre croissant
   pour k=1..n
     i,j = 1,n
     tant que (i<n+1 et j>0)
       si T[i] + T[j] + T[k] = 0 retourner 1
       sinon, si T[i] + T[j] + T[k] < 0 incrémenter i
       sinon, décrémenter j
   retourner 0
```

Pendant une vingtaine d'années, cette solution a été conjecturée comme indépassable. Autrement dit, on conjecturait que $\Omega(n^2)$ était une borne inférieure pour le problème 3-Sum (dans le modèle RAM taille arbitraire). Cette conjecture a été réfutée depuis (on y reviendra).

Exemples de problèmes 3-Sum-difficiles

Il s'avère que 3-Sum se réduit à des problèmes très divers. En voici quatre exemples :

Alignement

Entrée : Un tableau $P[1..n]$ où $P[i]$ est un point du plan.

Sortie : 1 s'il existe $i < j < k$ tels que $P[i]$, $P[j]$ et $P[k]$ sont alignés, 0 sinon.

X+Y

Entrée : Deux tableaux $X[1..n]$ et $Y[1..n]$ d'entiers.

Sortie : 1 si l'ensemble $\{X[i] + Y[j] : 1 \leq i, j \leq n\}$ est-il de cardinal n^2, 0 sinon.

Geombase

Entrée : Dans le plan, des points $A[1..n]$ sur la ligne $y = 0$, $B[1..n]$ sur la ligne $y = 1$ et $C[1..n]$ sur la ligne $y = 2$

Sortie : 1 s'il existe $1 \leq i, j, k \leq n$ tels que $A[i]$, $B[j]$ et $C[k]$ sont alignés, 0 sinon.

Planification de trajectoire

Entrée : Dans le plan, deux segments p_1p_2 et p_3p_4 de même longueur et un tableau $T[1..n]$ de polygones ayant chacun $O(1)$ sommets.

Sortie : 1 s'il est possible de déplacer continûment p_1p_2 en p_3p_4 sans qu'il rencontre l'intérieur d'aucun polygone $T[i]$, 0 sinon.

Examinons le premier exemple, Alignement. Comme pour 3-Sum, il est trivial de résoudre ce problème en $O(n^3)$, il est facile de le résoudre en $O(n^2 \log n)$ et possible de le résoudre en $O(n^2)$. [16]

Proposition 7.3. 3-Sum $\leq_{O(n)}$ Alignement.

Démonstration. Pour $a < b < c$ trois réels, les points (a, a^3), (b, b^3) et (c, c^3) sont alignés si et seulement si $a + b + c = 0$. En effet, l'alignement des points peut se caractériser par l'annulation du déterminant

$$0 = \begin{vmatrix} b-a & c-a \\ b^3-a^3 & c^3-a^3 \end{vmatrix} = (b-a)(c^3-a^3) - (c-a)(b^3-a^3) = (b-a)(c-a)(c-b)(a+b+c).$$

On peut donc réduire 3-Sum à Alignement en transformant le tableau $T[1..n]$ d'entiers positifs ou négatifs en un tableau $P[1..n]$ de points où $P[i] = (T[i], T[i]^2)$. □

16. C'est bien plus compliqué que la solution quadratique de 3-Sum et cela déborde largement de ce cours : l'algorithme standard pour le faire utilise de la dualité point-droite et un argument d'amortissement appelé *théorème de la zone*.

Le problème 3-SUM se réduit aussi en temps linéaire à chacun des trois autres exemples (on omet les preuves). Pour chacun de ces problèmes on connaît une solution de complexité $O(n^2)$; les réductions annoncées assurent les *bornes inférieures conditionnelles* suivantes :

Si 3-SUM est de complexité $\Omega(n^2)$, alors chacun de ces problèmes est de complexité $\Omega(n^2)$.

On appelle un problème auquel 3-SUM se réduit linéairement un problème **3-SUM-difficile**. Si on souhaite prouver qu'un problème 3-SUM-difficile est de complexité $\Omega(n^2)$, alors on a intérêt à concentrer nos efforts sur le problème 3-SUM : non seulement il semble être le plus simple (car plus « épuré »), mais une preuve pour ce problème impliquerait le résultat pour *tous* les problèmes 3-SUM-difficiles.

Réfutation de la conjecture 3-SUM et conjecture renforcée

Des années 1980 jusqu'en 2014, la meilleure solution connue pour 3-SUM était l'algorithme quadratique ci-dessus. La simplicité de ce problème rendait plausible la conjecture que cette borne était optimale, et 3-SUM a été réduit en temps $o(n^2)$ à des dizaines de problèmes pour lesquels les meilleurs algorithmes connus étaient aussi en $O(n^2)$. Chacune de ces réductions établissait une borne inférieure quadratique conditionnelle sur le problème cible, suggérant l'optimalité de la solution connue. Lorsqu'un algorithme de complexité [17] $o(n^2)$ a été trouvée pour 3-SUM [GP18], toutes ces bornes inférieures s'en sont trouvées invalidées... Le travail fait pour prouver que des dizaines de problèmes sont 3-SUM-difficiles n'est pas perdu pour autant. La réfutation de la conjecture que 3-SUM est $\Omega(n^2)$ a fait émerger une nouvelle conjecture, à laquelle les bornes inférieures conditionnelles se réfèrent désormais :

Conjecture 7.4. *Pour tout $\epsilon > 0$, le problème* 3-SUM *est de complexité* $\Omega(n^{2-\epsilon})$.

7.6 Prolongements

L'algorithme donné ci-dessus pour 2-COL ne se généralise pas pour k-COL avec $k \geq 3$. En effet, la couleur d'un sommet u dans une k-coloration propre ne détermine pas celle de ses voisins. À ce jour, l'algorithme de meilleure complexité connu pour le problème 3-colorabilité est dû à Beigel et Eppstein [BE05] et est de complexité $O\left(1.3289^n\right)$.

L'étude des problèmes 3-SUM-difficiles est l'amorce de ce que l'on appelle FINE-GRAINED COMPLEXITY THEORY, et qui identifie quelques problèmes « barrière ». Comme 3-SUM, chacun de ces problèmes barrière est très épuré et il existe un écart sensible entre les meilleures bornes inférieure et supérieure connues.

Revenons sur la question des réductions entre problèmes algorithmiques généraux, Considérons deux problèmes algorithmiques $P : E \to 2^S$ et $P' : E' \to 2^{S'}$. Une **réduction de P à P'** est un couple de fonctions $e : E \to E'$ et $s : S' \to S$ telles que

$$P = s \circ P' \circ e.$$

L'idée est simple : une réduction permet, partant d'une entrée quelconque x de P, de la transformer en une entrée $y = e(x)$ de P' telle que de toute solution $z \in P'(y)$ on puisse déduire une solution $s(z) \in P(x)$.

Fixons des fonctions de taille sur E et sur E'. Les calculs des fonctions e et s sont eux-mêmes des problèmes algorithmiques (qui ont la particularité que pour toute entrée il existe une unique sortie). La **complexité d'une réduction** (e, s) est la complexité de ces problèmes. Ainsi, une réduction est de complexité $O(n)$ si chacune des fonctions e et s peut être calculée par un algorithme de complexité linéaire. On note $P \leq_{f(n)} P'$ le fait qu'il existe une réduction de P à P' de complexité $f(n)$.

Ainsi, la Section 7.1 a établi que lorsque l'on mesure la taille d'un tableau par son nombre n de cases, CALCUL D'ENVELOPPE CONVEXE $\leq_{O(n)}$ TRI et TRI $\leq_{O(n)}$ CALCUL D'ENVELOPPE CONVEXE.

Examinons la manière dont $\leq_{f(n)}$ permet de propager des bornes de complexité $O()$ et $\Omega()$. Soient P et P' deux problèmes algorithmiques tels que $P \leq_{f(n)} P'$. Notons (e, s) une réduction (e, s) de P à P' et $\mathcal{A}_e$ et $\mathcal{A}_s$ des algorithmes, chacun de complexité $O(f(n))$, calculant e et s, respectivement.

17. De complexité $O\left(n^2 \left(\frac{\log\log n}{\log n}\right)^{2/3}\right)$ pour être précis.

Si P' est de complexité $O(g(n))$ alors P est de complexité $O(f(n) + g \circ f(n) + f \circ g \circ f(n))$.

En effet, supposons qu'il existe un algorithme $\mathcal{A}$ calculant P' et de complexité $O(g(n))$. Considérons une entrée x pour P de taille n. L'algorithme $\mathcal{A}_e$ calcule $y = e(x)$ en temps $O(f(n))$; en particulier la taille de y est $O(f(n))$. L'algorithme $\mathcal{A}$ appliqué à y fournit une sortie z en temps $O(g \circ f(n))$ et cette sortie est de taille $O(g \circ f(n))$. Dès lors, l'algorithme $\mathcal{A}_s$ appliqué à z fournit, en temps $O(f \circ g \circ f(n))$ une solution à P sur x. Le problème P est bien de complexité $O(f(n) + g \circ f(n) + f \circ g \circ f(n))$.

Si P est de complexité $\Omega(g(n))$ alors P est de complexité $\Omega(f \circ g \circ f(n))$ sous condition sur f...

Tentons de contraposer le raisonnement précédent pour propager les bornes inférieures de complexités. Supposons que P' ne soit pas de complexité $\Omega(g(n))$. C'est donc que P' est de complexité $O(h(n))$ avec $h = o(g)$. Le raisonnement précédent nous donne que P est de complexité $O(f(n)+h\circ f(n)+f\circ h\circ f(n))$. On aimerait avoir

$$h = o(g) \quad \stackrel{?}{\Rightarrow} \quad f(n) + h \circ f(n) + f \circ h \circ f(n) = o(f(n) + g \circ f(n) + f \circ g \circ f(n)).$$

mais cela est faux en général (prendre par exemple $g(n) = \log^2 n$ et $h(n) = f(n) = \log n$). Cette implication est vraie pour certaines fonctions f... mais il est souhaitable de simplifier le cadre d'étude des réductions, d'où la focalisation sur le cas des problèmes de décision.

7.7 Références bibliographiques

[APT79] Bengt Aspvall, Michael F Plass, and Robert E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. processing letters*, 8(3) :121–123, 1979.

[BE05] Richard Beigel and David Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms*, 54(2) :168–204, 2005.

À retenir.

- Un **problème de décision** est un problème algorithmique où seules deux réponses sont possibles, 0 ou 1, et où chaque entrée a exactement une réponse.
- Nos **réductions** sont entre problèmes de décision ; une telle réduction est un algorithme qui transforme toute entrée d'un problème A en une entrée d'un problème B de même réponse.
- Une réduction permet de comparer les complexités de problèmes, ce que l'on note $\leq_{f(n)}$.
- Les réductions permettent de formuler des bornes inférieures conditionnées à la validité d'autres bornes inférieures.
- k-Col demande si un graphe non orienté donné admet une coloration propre par k couleurs.
- k-Sat demande si une formule de logique propositionnelle est satisfiable, cette formule étant donnée en forme normale conjonctive (CNF) où chaque clause est de taille k.

Chapitre 8

Classes de complexité et NP-difficulté

Il est courant de classer les problèmes algorithmiques en deux catégories : ceux qui sont de complexité polynomiale (et donc « faciles ») et ceux qui sont de complexité super-polynomiale (et donc « difficiles »). Cette séance se focalise sur cette frontière entre polynomial et non-polynomial, en examinant la question suivante :

> Comment déterminer si un problème algorithmique donné admet une solution de complexité polynomiale ?

On retrouve le même type de difficulté qu'au Chapitre 6 : prouver qu'un problème algorithmique n'a pas de solution polynomiale amène à discuter de l'*ensemble* des algorithmes qui résolvent ce problème. À ce jour, la principale méthode pour cela consiste à établir une *borne inférieure conditionnelle* au moyen de l'outil de *réduction* introduit au Chapitre 7. La NP-difficulté est l'exemple le plus connu de borne inférieure conditionnelle.

Dans ce chapitre on travaille sur des problèmes de décision **normalisés** au sens où l'ensemble des sorties est $\{0,1\}$. Tout problème de décision normalisé $D : E \to \{0,1\}$ a un **problème complémentaire** $D' : E \to \{0,1\}$ défini par $D'(e) = 1 - D(e)$.

Objectifs. À l'issue de cette séance, il est attendu que vous...

- compreniez les notions de classe de complexité P et NP,
- sachiez effectuer une réduction polynomiale entre deux problèmes,
- connaissiez quelques exemples de problèmes NP-difficiles,
- sachiez prouver qu'un problème est NP-difficile pour des exemples simples.

8.1 La classe de complexité P

Formellement, une **classe de complexité** est un ensemble de problèmes de décisions [1]. Une classe de complexité est généralement définie comme l'ensemble des problèmes d'une complexité donnée dans un modèle de calcul donné. La première classe qui nous intéresse est la suivante :

> La **classe P** est l'ensemble des problèmes de décision normalisés qu'il est possible de résoudre par un algorithme en **RAM 8 bits** de complexité polynomiale relativement à la **taille d'encodage**.

Pour être dans la classe P, il ne suffit pas qu'un problème admette « un algorithme polynomial » : il faut que cet algorithme soit dans le bon modèle (RAM 8 bits) et que sa complexité soit polynomiale relativement à la bonne fonction de taille (la taille d'encodage). Soulignons cependant que :

- Si une fonction de taille t est polynomiale en la taille d'encodage, au sens où il existe un polynôme p tel que $t(e) \leq p(\ell)$ pour toute entrée e de taille d'encodage ℓ, alors tout algorithme polynomial relativement à t est polynomial relativement à la taille d'encodage.

1. Ou, de manière équivalente, un ensemble de langages tels que définis en Complément B.

- Si un algorithme en RAM taille arbitraire est polynomial relativement à la taille d'encodage et ne fait aucun usage abusif[2] d'instruction décomposable, alors cet algorithme en RAM 8 bits est polynomial relativement à la taille d'encodage.

Ainsi, l'algorithme discuté en Section 7.3.1 qui résout 2-Col en RAM taille arbitraire en $O(n)$ atteste que 2-Col est dans P : en effet, n est majoré par la taille d'encodage et les instructions décomposables de cet algorithme sont toutes raisonnables. De la même manière, on peut s'assurer que le problème Tous distincts (étudié au Chapitre 6, Section 6.2) appartient aussi à la classe P.

La définition de la classe P donnée ci-dessus fait explicitement référence au modèle RAM 8 bits. Remplacer ce modèle par la machine de Turing, définie en complément, donne une définition équivalente car RAM 8 bits et machine de Turing peuvent se simuler l'un l'autre en temps polynomial(cf Section 8.5). Plus généralement, la classe P reste invariante[3] lorsque l'on change le modèle de calcul pour un modèle « polynomialement équivalent ». Cela renforce sa pertinence théorique, et permet un peu de flexibilité dans son étude (on peut choisir le modèle qui se prête le mieux à la question).

8.2 La classe de complexité NP

Informellement, la classe de complexité NP est l'ensemble des problèmes de décision qui peuvent être *vérifiés* en temps polynomial. Commençons par donner l'intuition de ce qu'est la vérification sur l'exemple suivant :

Sous-sommes

Entrée : Un tableau $X[1..n]$ d'entiers positifs ou nuls et un entier R.

Sortie : 1 s'il existe un sous-ensemble des entiers de $X[1..n]$ dont la somme égale R, 0 sinon.

Il est facile de mettre en place une procédure qui permette à quelqu'un de prouver que la réponse de Sous-sommes sur une entrée donnée $(X[1..n], R)$ est « oui » : il suffit de décrire un sous-ensemble Y de X dont la somme égale C. La description, appelée *certificat*, peut simplement être un mot binaire $w = w_1 w_2 \dots w_n$, et la *vérification* peut simplement sommer les entrées $X[i]$ pour les indices i tels que $w_i = 1$ et comparer le résultat à R. Soulignons quelques particularités de ce « protocole » :

- le certificat w est de taille polynomiale en n,
- l'algorithme de vérification prend en entrée une instance $(X[1..n], R)$ et un certificat w, et décide en temps polynomial si le certificat atteste bien que la réponse sur l'instance est « oui »,
- si la réponse est « oui », il existe un certificat qui l'atteste,
- si la réponse est « non », aucun certificat ne peut attester qu'elle est « oui ».

La question **n'est pas** ici de savoir comment calculer un certificat valide pour une instance donnée (ce qui reviendrait à *résoudre* Sous-sommes), mais simplement de permettre une telle vérification. Remarquons qu'il n'est pas évident que la réponse « non » soit elle aussi vérifiable en temps polynomial...

Certificats et vérification

Pour formaliser cela, considérons un problème de décision $D : E \to \{0, 1\}$. Pour $e \in E$ on note $|e|$ sa taille d'encodage. Supposons fixé un ensemble dénombrable $\mathcal{C}$, appelé **espace des certificats**. Un algorithme $\mathcal{V}$ est un **vérificateur polynomial** d'ensemble de certificats $\mathcal{C}$ pour D s'il existe un polynôme p tel que :

(i) $\mathcal{V}$ prend en entrée un couple $(e, c) \in E \times \mathcal{C}$ arbitraire et retourne `vrai` ou `faux` en temps polynomial en la taille d'encodage de (e, c).

(ii) Si $D(e) = 1$, *il existe* $c \in \mathcal{C}$ de taille d'encodage au plus $p(|e|)$ tel que $\mathcal{A}$ retourne `vrai` sur (e, c).

2. Au sens de la Section 2.4.3.

3. Cela n'est par exemple pas le cas pour la classe des problèmes de décision qui peuvent être résolus en temps *linéaire* : reconnaître un palindrome peut se faire en temps linéaire en RAM 8 bits mais prend $\Omega(n^2)$ sur une machine de Turing disposant d'un seul ruban.

(iii) Si $D(e) = 0$, *pour tout* $c \in \mathcal{C}$ de taille d'encodage au plus $p(|e|)$, $\mathcal{A}$ retourne `faux` sur (e, c).

> La **classe NP** est l'ensemble des problèmes de décision normalisés qui admettent un vérificateur polynomial dans le modèle **RAM 8 bits**.

L'exemple ci-dessus montre que Sous-sommes est dans NP. Soulignons deux subtilités :

- Les valeurs 0 et 1 jouent des rôles différents, cf les propriétés (ii) et (iii). Lorsque l'on énonce qu'un problème de décision est dans NP il est donc important d'expliciter quelle sortie joue le rôle de 1, c'est-à-dire est *certifiée*.
- Savoir si un problème de décision normalisé est dans NP et savoir si son problème complémentaire est dans NP sont deux questions distinctes. (Alors que pour P, les deux questions sont équivalentes.)

Premiers exemples

Revisitons les problèmes 3-Col et 3-Sat sous forme normalisée...

> k-Col
>
> **Entrée :** Un graphe $G = (V, E)$ donné par sa matrice d'adjacence $A \in \{0,1\}^{n \times n}$.
>
> **Sortie :** 1 si le graphe G est k-coloriable, 0 sinon.

Construisons un vérificateur polynomial pour 3-Col. On choisit comme espace de certificats $\mathcal{C} = \bigcup_{n \geq 1} \{1,2,3\}^n$. Le vérificateur $\mathcal{A}$ prend en entrée un graphe $G = (\{1,2,\ldots,n\}, E)$ et un certificat $c = (c_1, c_2, \ldots, c_\ell) \in \mathcal{C}$. L'algorithme $\mathcal{A}$ accepte (G, c) si $\ell = n$ et que pour tout $\{i,j\} \in E$ on a $c_i \neq c_j$. On a bien qu'il existe un certificat accepté par le vérificateur si et seulement si le graphe donné en entrée est 3-coloriable, et ce certificat est de taille $|V|$. En RAM taille arbitraire, la vérification se fait en temps linéaire en la taille $|V| + |E|$ du graphe ; puisque $|V| + |E|$ est majoré par la taille d'encodage et que dans l'algorithme ci-dessus toutes les instructions décomposables sont raisonnables, 3-Col est dans NP.

> k-Sat
>
> **Entrée :** Une formule de logique propositionnelle f à au plus n variables en forme normale conjonctive ayant au plus n clauses, chacune comportant au plus k littéraux.
>
> **Sortie :** 1 si f est satisfiable, 0 sinon.

Construisons un vérificateur polynomial pour 3-Sat. On prend $\mathcal{C} = \{0,1\}^*$ comme espace de certificats. Le vérificateur $\mathcal{A}$ prend en entrée une formule de logique propositionnelle f CNF de variables $(x_1, x_2, \ldots, x_n)$ et à au plus n clauses, chacune comportant au plus 3 littéraux, et un mot $c \in \mathcal{C}$. L'algorithme $\mathcal{A}$ accepte (f, c) si l'affectation $x_i \stackrel{\text{def}}{=} \texttt{vrai}$ si $c_i = 1$ et $x_i \stackrel{\text{def}}{=} \texttt{faux}$ sinon satisfait la formule f. On a bien qu'il existe un certificat accepté par le vérificateur si et seulement si la formule donnée en entrée est satisfiable. En RAM taille arbitraire, l'évaluation se fait en temps constant par clause, soit en temps linéaire dans l'ensemble ; cela se traduit bien par un algorithme RAM 8 bits de complexité polynomiale en la taille d'encodage de l'entrée.

Solvable implique vérifiable

Intuitivement, il ne semble pas plus difficile de *vérifier* une solution à un problème que de *résoudre* ce problème. C'est donc sans surprise que l'on a :

Proposition 8.1. *$P \subseteq NP$.*

Démonstration. Soit $D : E \to \{0,1\}$ un problème de la décision qui appartient à la classe P. Il existe donc un algorithme $\mathcal{A}$ en RAM 8 bits qui résout D et est de complexité polynomiale relativement à la taille d'encodage de l'entrée. On peut utiliser $\mathcal{A}$ comme un vérificateur, avec un espace de certificats vide. Par conséquent, $D \in$ NP. □

8.3 Problèmes NP-difficiles et NP-complets

L'inclusion P $\subseteq$ NP étant établie, il est naturel de se demander si ces deux classes sont égales. Comme le problème 3-SAT est dans NP, s'il n'admet pas de solution polynomiale alors P $\neq$ NP. Il s'avère que la réciproque est vraie : si 3-SAT $\in$ P alors P=NP. Cela découle du théorème suivant, établi indépendamment par Cook et par Levin :

Théorème 8.2. *Tout problème de la classe NP admet une réduction polynomiale à* 3-SAT.

Avant de discuter de la preuve de ce théorème, examinons certaines de ses conséquences.

Problème NP-difficile et réduction

Le Théorème 8.2 suscite la définition suivante :

> Un problème de décision A est **NP-difficile** si pour tout problème $B \in$ NP il existe une réduction polynomiale de B à A.

Ainsi, le Théorème 8.2 énonce que 3-SAT est NP-difficile. Pour des problèmes de décision A et B, notons $\mathbf{A \leq_P B}$ pour signifier qu'il existe une réduction polynomiale de A à B. Puisque la composition de réductions polynomiales est une réduction polynomiale (Lemme 7.2), on a :

> Si A est NP-difficile et $A \leq_P B$ alors B est NP-difficile.

La manière standard d'établir qu'un problème de décision D est NP-difficile consiste à montrer qu'un problème déjà connu comme NP-difficile se réduit polynomialement à D. Ainsi, la réduction polynomiale de 3-SAT à 3-COL esquissée en Section 7.4 assure que 3-COL est NP-difficile. On connaît à ce jour des *centaines*[4] de problèmes NP-difficiles.

Un problème de décision est dit **NP-complet** si il appartient à la classe NP *et* est NP-difficile. Ainsi, les problèmes 3-SAT et 3-COLORABILITÉ sont NP-complets.

P$\stackrel{?}{=}$NP

L'intérêt principal de la définition de problème NP-difficile réside dans une conséquences immédiate de cette définition :

> Si *l'un des* problèmes NP-difficiles admet une solution polynomiale, alors P=NP.

Ainsi trouver une solution polynomiale à *un*[5] problème NP-difficile assure que *tout* problème de NP admet une solution polynomiale.[6] La réciproque est tout aussi utile :

> Si P$\neq$NP alors *aucun* problème NP-difficile n'admet de solution polynomiale.

Ainsi, prouver qu'un problème est NP-difficile peut s'envisager comme une **borne inférieure conditionnelle** sur sa complexité : conditionnellement au fait que P$\neq$NP, ce problème est de complexité super-polynomiale.

La question de savoir si les classes P et NP coïncident ou sont distinctes est centrale en théorie de la complexité depuis les années 1970. Elle suscite énormément d'intérêt[7] et de travaux, et bien qu'elle reste ouverte à ce jour, de nombreuses avancées ont été faites au fil des décennies. La conjecture qui prédomine actuellement est que P$\neq$NP, et donc qu'aucun problème NP-difficile n'a de solution polynomiale.

4. Voir **https://en.wikipedia.org/wiki/List_of_NP-complete_problems** pour un premier aperçu. Une liste spécifique à l'optimisation est consultable à **https://www.csc.kth.se/~viggo/wwwcompendium/**.
5. N'importe lequel !
6. Autrement dit, tout problème dont on sait *vérifier une solution* en temps polynomial peut être *résolu* en temps polynomial.
7. Par exemple, elle a été distinguée comme un des *problèmes du millénaire*.

Un aperçu de la preuve du théorème de Cook-Levin

Examinons quelles idées se cachent derrière la preuve du théorème de Cook-Levin. Pour simplifier, on normalise ici les problèmes et les certificats en mots binaires. Cela se fait sans perte de généralité.

Considérons un problème de décision $D : \{0,1\}^* \to \{0,1\}$ appartenant à la classe NP. Par définition, il existe un vérificateur $\mathcal{A}$ et un polynôme $P(n)$ tels que pour tout mot $w \in \{0,1\}^*$:

- si $D(w) = 1$, alors il existe $c \in \{0,1\}^*$ de taille $|c| \leq P(|w|)$ tel que $\mathcal{A}$ accepte (w, c),
- si $D(w) = 0$, alors pour tout $c \in \{0,1\}^*$ de taille $|c| \leq P(|w|)$, $\mathcal{A}$ rejette (w, c).

Le certificat c peut être codé par $n = P(|w|)$ variables booléennes.[8] La preuve du Théorème 8.2 construit, étant donné un mot $w \in \{0,1\}^*$, une formule de logique propositionnelle $\phi_{\mathcal{A},w}$ sur n variables telle que *l'évaluation de $\phi_{\mathcal{A},w}$ pour un assignement de ces n variables encodant un certificat c égale* `vrai` *si $\mathcal{A}$ accepte (w, c) et* `faux` *sinon.* La construction doit se faire en temps polynomial en $|w|$, ce qui implique que la taille de la formule $\phi_{w,\mathcal{A}}$ est elle-même polynomiale en $|w|$. Il s'avère que l'on peut faire cela, et que la formule produite peut même être en forme normale conjonctive avec des clauses de taille 3. Voyons cela...

Le modèle RAM 8 bits décompose l'exécution d'un algorithme en pas de calcul discrets. Définissons un *instantané* comme étant l'information décrivant l'état du modèle entre deux pas de calcul. On peut décrire chaque instantané par un ensemble de variables booléennes, car il suffit d'encoder un ensemble fini de mots binaires : contenu des cases mémoire *utilisées*, valeurs des registres, position dans le programme, ... Comme l'algorithme $\mathcal{A}$ est polynomial, chaque instantané est formé d'un nombre polynomial de mots binaires. Puisque l'on travaille en RAM 8 bits, chacun de ces mots binaires peut être codé par $O(1)$ variables booléennes.

La formule $\phi_{\mathcal{A},w}$ que l'on construit modélise la suite des instantanés décrivant le système après chaque pas de calcul lors de l'exécution de l'algorithme $\mathcal{A}$ sur (w, c). On introduit un groupe de variable pour chaque instantané, et on ajoute dans la formule $\phi_{\mathcal{A},w}$ des conditions exprimant le fait que le $i$ème instantané décrit bien l'état dans lequel se trouve le modèle lorsque l'on applique au $(i-1)$ème instantané l'instruction qu'il pointe.[9] Une fois toutes les étapes du calcul déroulé, on ajoute la condition que l'instantané final traduit le fait que $\mathcal{A}$ a accepté le mot (w, c).

Il s'avère que l'on peut retravailler la formule $\phi_{\mathcal{A},w}$ de manière à ce qu'elle n'utilise que les $P(|w|)$ variables décrivant le certificat c – l'idée étant que les seules valeurs possibles des autres variables découle de ces $P(|w|)$ « variables libres ». Résoudre le problème D sur le mot w, ce qui est équivalent à décider s'il existe un certificat c tel que $\mathcal{A}$ accepte (w, c), est donc réduit au problème de décider si la formule retravaillée est satisfiable ou pas. Et voilà...

... à un détail près : cette réduction n'est, en l'état, pas polynomiale ! Dans cette description, le nombre $t_i(|w|)$ de variables booléennes utilisées pour coder le $i$ème instantané est certes polynomial en $|w|$. Cependant, le nombre d'instantané est lui-même un polynôme $T(|w|)$, aussi le nombre total de variables booléennes utilisées, à savoir $P(|w|) + \prod_{i=0}^{T(|w|)} t_i(|w|)$ croît plus vite que tout polynôme.

Il convient donc de revisiter cette première ébauche de preuve en remplaçant ce codage naïf des instantanés par quelque chose de plus parcimonieux. Cela peut se faire au moyen d'instantanés incrémentaux, qui au lieu de coder l'intégralité du $i$ème instantané ne code que les éléments ayant changés par rapport au $(i-1)$ème instantané. La réalisation qu'un tel codage peut ne prendre qu'un nombre *constant* de variables par pas de calcul est un des apports fondamentaux des preuves de Cook et de Levin. Cette idée est souvent exprimée par le fait que *le calcul est local.*

Pour préciser cette ébauche, quelque peu impressionniste, on peut se référer par exemple au chapitre 2 du livre d'Arora et Barak [AB09, §2.3] (NB : leur modèle de référence n'est pas la RAM 8 bits mais la machine de Turing).

8. On utilise ici l'identification habituelle `vrai` $\leftrightarrow$ 1, `faux` $\leftrightarrow$ 0.

9. Cela est possible car toute instruction peut s'exprimer comme une fonction booléenne d'un nombre constant de mots binaires de taille constante.

8.4 Exemples de problèmes NP-difficiles

Concluons ce chapitre par une liste de problèmes algorithmiques qui sont tous NP-difficiles. [10] Les noms de problèmes renvoient parfois à des problèmes d'optimisation bien connus (PLNE, voyageur de commerce, k-means, ...) ; il est à chaque fois implicite que c'est la **version décision** du problème qui est NP-difficile. [11] Précisons que la liste est plus fournie dans la version électronique du polycopié.

Commençons par (les versions décision) de problèmes d'optimisations que vous retrouverez en recherche opérationnelle :

PROGRAMMATION LINÉAIRE EN NOMBRE ENTIERS (PLNE)

Entrée : Une matrice $A \in \mathbb{Q}^{n \times m}$, des vecteur $b, c \in \mathbb{Q}^n$ et un nombre $s \in \mathbb{Q}$.

Sortie : Vrai ou faux, il existe $x \in \mathbb{Z}^n$ tel que $Ax \leq b$ et $c^T x \geq s$.

PROGRAMMATION LINÉAIRE 0-1

Entrée : Une matrice $A \in \mathbb{Q}^{n \times m}$, des vecteur $b, c \in \mathbb{Q}^n$ et un nombre $s \in \mathbb{Q}$.

Sortie : Vrai ou faux, il existe $x \in \{0,1\}^n$ tel que $Ax \leq b$ et $c^T x \geq s$.

SAC À DOS

Entrée : Des paires d'entiers $(v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)$ et des entiers V et W.

Sortie : Vrai ou faux, il existe un sous-ensemble $I \subset \{1, 2, \ldots, n\}$ tel que $\sum_{i \in I} v_i \geq V$ et $\sum_{i \in I} w_i \leq W$.

Continuons par de l'optimisation sur graphes [12]. Pour les deux premiers problèmes, notez que l'entier k doit faire partie de l'entrée (le problème est trivialement polynomial si k est constant).

CLIQUE

Entrée : Un graphe $G = (V, E)$ où $|V| = n$ et un entier k.

Sortie : Vrai ou faux, il existe une clique de taille k dans G.

VERTEX COVER

Entrée : Un graphe $G = (V, E)$ où $|V| = n$ et un entier k.

Sortie : Vrai ou faux, il existe un sous-ensemble $I \subset V$ de taille $|I| \leq k$ tel que chaque arête de E ait au moins un sommet dans I.

CYCLE HAMILTONIEN

Entrée : Un graphe $G = (V, E)$ où $|V| = n$.

Sortie : Vrai ou faux, G contient un cycle qui passe par chaque sommet une et une seule fois.

Un autre classique...

10. Pour la plupart il est facile de vérifier qu'ils sont aussi dans NP et donc NP-complets.

11. C'est bien entendu plus fort : un algorithme qui résoudrait le problème d'optimisation en temps polynomial résoudrait aussi le problème de décision en temps polynomial : il suffit d'ajouter une comparaison.

12. Ces problèmes peuvent facilement se formuler comme des PLNE. Attention, **cela ne prouve pas** qu'ils sont NP-difficiles, car la réduction est dans le mauvais sens. Il faut travailler un peu plus...

VOYAGEUR DE COMMERCE

Entrée : Un nombre $s \in \mathbb{Q}$ et une matrice $A \in \mathbb{Q}^{n \times n}$ symmétrique, de diagonale nulle et satisfaisant $A_{i,j} + A_{j,k} \geq A_{i,k}$ pour tous $1 \leq i, j, k \leq n$.

Sortie : Vrai ou faux, il existe un tableau de permutation $T[1..n]$ tel que $\sum_{i=1}^{n-1} A_{T[i],T[i+1]} \leq s$.

... qui reste NP-difficile quand les distances sont euclidiennes :

VOYAGEUR DE COMMERCE EUCLIDIEN

Entrée : Un nombre $s \in \mathbb{Q}$ et une séquence de points $p_1, p_2, \ldots, p_n \in \mathbb{Q}^2$.

Sortie : Vrai ou faux, il existe un tableau de permutation $T[1..n]$ tel que $\sum_{i=1}^{n-1} \|p_{T[i]}p_{T[i+1]}\|_2 \leq s$, où $\|\cdot\|_2$ est la distance euclidienne.

Voici un problème de partitionnement fondamental en apprentissage automatique [13]...

k-MEANS

Entrée : Un nombre $s \in \mathbb{Q}$ et une séquence de points $p_1, p_2, \ldots, p_n \in \mathbb{Q}^2$.

Sortie : Vrai ou faux, il existe $c_1, c_2, \ldots, c_k \in \mathbb{Q}^2$ et une partition $[n] = I_1 \sqcup I_2 \sqcup \ldots \sqcup I_k$ tels que $\sum_{\ell=1}^{k} \sum_{j \in I_\ell} {\|p_j c_\ell\|_2}^2 \leq s$.

... et quelques autres problèmes d'optimisation géométrique [14] :

COUVERTURE PAR DES DEMI-ESPACES

Entrée : Un entier s, un ensemble P de n points de $\mathbb{Q}^3$ et une séquence de demi-espaces $H_1, H_2, \ldots, H_n$ de $\mathbb{R}^3$ (H_i est représenté par un point $a_i \in \mathbb{Q}^3$ et un vecteur $\overrightarrow{v_i} \in \mathbb{Q}^3$).

Sortie : Vrai ou faux, il existe un sous-ensemble $I \subseteq \{1, 2, \ldots, n\}$ tel que $|I| \leq s$ et $P \subset \bigcup_{i \in I} H_i$.

ISOMÉTRIE

Entrée : Une matrice $M \in \mathbb{Q}^{n \times n}$.

Sortie : Vrai ou faux, il existe une séquence $p_1, p_2, \ldots, p_n \in \mathbb{R}^3$ tels que $\|p_i p_j\|_\infty = M_{i,j}$.

DEMI-ESPACE DENSE

Entrée : Des entiers d et s et une séquence de points $p_1, p_2, \ldots, p_n \in \mathbb{Q}^d$.

Sortie : Vrai ou faux, il existe $x \in \mathbb{R}^d$ tel que $|\{i : p_i \cdot x \geq 0\}| \geq s$.

Voici trois problèmes d'algèbre :

13. Autrement dit, un problème de *clustering* fondamental en *machine learning*.

14. Les analogues 2D de COUVERTURE PAR DES DEMI-ESPACES et ISOMÉTRIE sont polynomiaux ; c'est un peu comme k-SAT ou k-COL, qui sont polynomiaux pour $k = 2$ et NP-difficile pour $k \geq 3$. Quant à HEMISPHÈRE DENSE, le problème est polynomial en toute dimension fixée, mais est NP-difficile si la dimension fait partie de l'entrée du problème.

COPOSITIVITÉ

Entrée : Une matrice $M \in \mathbb{Q}^{n\times n}$.

Sortie : Vrai ou faux, M est-elle copositive ? Autrement dit, est-ce vrai que pour tout $x \in (\mathbb{R}^+)^n$ on a $x^T M x \geq 0$?

NMF EXACTE *(non-negative matrix factorization)*

Entrée : Un entier k et une matrice $M \in \mathbb{Q}^{m\times n}$ de rang k.

Sortie : Vrai ou faux, existe-t-il deux matrices $W \in \mathbb{R}^{m\times k}$ et $H \in \mathbb{R}^{k\times n}$ à coefficients positifs ou nuls tels que $M = WH$?

VALEUR PROPRE TENSORIELLE

Entrée : Un tenseur $T = (t_{i,j,k}) \in \mathbb{Q}^{n\times n\times n}$.

Sortie : Vrai ou faux, 0 est-il valeur propre de T ? Autrement dit, existe-t-il $x \in \mathbb{R}^n$ tel que $\sum_{1\leq i,j\leq n} a_{i,j,k} x_i x_j = 0$ pour tout k ?

Et terminons par un jeu [15] :

DÉMINEUR

Entrée : Une grille $n \times n$ dont certaines cases sont étiquetées par des entiers.

Sortie : Vrai ou faux, existe-t-il un ensemble de mines cohérent avec ces étiquettes pour les règles du jeu *démineur* ?

8.5 Prolongements

Au-delà de P et NP, de nombreuses classes de complexité examinent comment la puissance de calcul est affectée par d'autres facteurs comme l'espace mémoire (classe PSPACE) ou l'usage du hasard (classe AM). Le site https://complexityzoo.net donne un aperçu de nombreuses classes et de ce que l'on comprend de leurs relations.

La notion de NP-difficulté n'est pas utile que pour les problèmes de décision, mais s'applique aussi par exemple aux problèmes d'optimisation ou encore d'approximation. Un résultat célèbre dans cette direction, dû à Lund et Yannakakis, est qu'il existe un réel $\epsilon > 0$ tel que si P≠NP, alors aucun algorithme polynomial ne peut approximer le nombre chromatique d'un graphe à n sommet à un facteur multiplicatif inférieur à n^ϵ. Ce nombre est donc difficile non seulement à calculer exactement, mais aussi à approximer.

Si la conjecture que $P \neq NP$ est avérée, il ne peut exister d'algorithme polynomial pour 3-SAT. Il existe des conjectures plus précises. Définissons, pour $k \geq 2$, le réel s_k comme l'infimum des réels s tels qu'il existe un algorithme de complexité $O(2^{sn})$ qui résout K-SAT. En particulier $s_2 = 0$. [16] La conjecture suivante est appelée ETH, pour *Exponential Time Hypothesis* :

Conjecture 8.3 (ETH). *$s_k > 0$ pour tout $k \geq 3$.*

Il y a même une version forte de la conjecture ETH, appelée SETH, pour *Strong Exponential Time Hypothesis*. La suite $s_3 \leq s_4 \leq \ldots$ est croissante et majorée [17] par 1, aussi elle admet une limite s_∞.

15. De nombreux jeux ont des version NP-difficiles, voir le cours http://courses.csail.mit.edu/6.892/spring19/ pour quelques exemples.

16. Si $f(n) = O(n)$ alors pour tout $s > 0$, $f(n) = O(2^{sn})$. L'infimum est donc 0.

17. Il suffit de tester les 2^n affectations possible. Chaque test est fait en temps $O(n^t)$ pour une constante t. Au total, cela prend un temps $O\left(2^{n+t\log_2 n}\right)$, ce qui est $O\left(2^{(1+\epsilon)n}\right)$ pour tout $\epsilon > 0$. L'infimum est donc bien 1.

Conjecture 8.4 (SETH). $s_\infty = 1$.

Il existe des théories analogues à la NP-difficulté, identifiant des bornes inférieures conditionnelles à diverses conjectures. Par exemple, si la conjecture ETH est vraie, alors il n'existe pas d'algorithme de complexité $n^{o(k)}$ pour, étant donné un graphe à n sommets et un entier k, calculer une clique de taille k ou reporter qu'aucune n'existe. [18]

Un sondage récent (2019) de William Gasarch auprès de la communauté étudiant la théorie de la complexité indique qu'à peu près 80% des sondés pensent que P≠NP. Ce pourcentage monte à 99% lorsque l'on ne prend en compte que les réponses de gens que Gasarch estime « avoir beaucoup réfléchi au problème ». Pour apprécier ce que signifie « avoir beaucoup réfléchi au problème », ou tout simplement pour se faire une idée de l'état de nos connaissances sur le problème $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$, on peut se référer au panorama récent [Aar16]. La section 6 discute notamment les progrès de ces dernières décennies sur la question : preuves que certaines approches ne peuvent pas marcher (barrière de la relativisation par exemple), programmes proposés et en cours de développement (théorie géométrique de la complexité par exemple), etc.

Historiquement, la classe NP se définit au travers d'une variante *non-déterministe* des machines de Turing. On ne discutera pas plus avant ce point de vue dans ce cours, mais le Complément E le présente de manière succinte à fins culturelles.

8.6 Références bibliographiques

[Aar16] Scott Aaronson. $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$. In *Open problems in mathematics*, pages 1–122. Springer, 2016. Disponible à https://www.scottaaronson.com/papers/pnp.pdf.

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity : a modern approach*. Cambridge University Press, 2009.

[APT79] Bengt Aspvall, Michael F Plass, and Robert E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. processing letters*, 8(3) :121–123, 1979.

[GP18] Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *Journal of the ACM (JACM)*, 65(4) :1–25, 2018.

À retenir.

- P est l'ensemble des problèmes solvables en temps polynomial.
- NP est l'ensemble des problèmes vérifiables en temps polynomial.
- Un problème de décision est NP-difficile si tout problème de NP s'y réduit polynomialement.
- Le théorème de Cook-Levin énonce que 3-Sat est NP-difficile.
- Si $A \leq_P B$ et A est NP-difficile alors B est NP-difficile.

18. Cet énoncé n'est intéressant que si k dépend de n.

Chapitre 9

Modèle de calcul quantique

Ce chapitre présente un modèle de calcul quantique. Ce type de modèles est apparu dans les années 1990 et se démarque des modèles classiques vus au Chapitre 2 notamment par le fait que l'information n'est pas manipulée au travers de mots binaires. Ces modèles suscitent beaucoup d'intérêt car ils sont susceptibles à la fois d'être réalisables par des systèmes physiques et d'offrir certaines accélérations « exponentielles » par rapport aux modèles de calcul classiques. Nous allons tâcher de préciser cela.

La définition du modèle de calcul occupe l'essentiel du chapitre, et laisse donc peu de place à l'algorithmique quantique proprement dite. (En cela, ce chapitre a un peu la même saveur que le Chapitre 2.) On détaille toutefois un algorithme quantique important, dû à Grover, qui permet de résoudre un problème de recherche en un nombre d'instructions sensiblement inférieur à ce que peut faire tout algorithme en RAM 8 bits.

Soulignons que ce modèle de calcul quantique se formule dans le langage du calcul tensoriel (d'espaces vectoriels complexes) que vous avez pratiqué en cours de mécanique des milieux continu. Le point de vue étant un peu différent, nous en redonnons les principes. Soulignons que l'objectif dans cette séance n'est pas de maîtriser ce calcul tensoriel, mais d'appréhender la nouvelle manière de traiter l'information qu'il permet d'envisager.

Objectifs. À l'issue de cette séance, il est attendu que vous...

- compreniez la manière dont un modèle de calcul quantique manipule l'information : notion de m-qubit, d'opérateur unitaire et de mesure,
- compreniez comment le formalisme tensoriel permet d'exprimer un qubit comme sous-système d'un m-qubit et de définir la notion d'opérateur élémentaire,
- sachiez construire un programme quantique élémentaire sur ≤ 3 qubits,
- sachiez dérouler l'exécution d'un algorithme quantique simple.

9.1 Vue d'ensemble

Pour poser le décor, commençons par définir ce que signifie calculer dans le modèle quantique.

L'espace $\mathbb{C}^n$, en tant que $\mathbb{C}$-espace vectoriel, est muni du produit scalaire (dit *Hermitien*) $\langle x, y\rangle \stackrel{\text{def}}{=} x_1\overline{y_1} + x_2\overline{y_2} + \ldots + x_n\overline{y_n}$. On note $\|\cdot\|_2$ la norme associée, c'est-à-dire que $\|v\|_2 = \sqrt{\langle v, v\rangle}$. Une application linéaire $M : \mathbb{C}^n \to \mathbb{C}^n$ est *unitaire* si elle préserve la norme, c'est-à-dire si $\|Mv\|_2 = \|v\|_2$ pour tout $v \in \mathbb{C}^n$. Cela est équivalent au fait que les colonnes de M forment une base orthonormale de $\mathbb{C}^n$.

Un **m-qubit** est un vecteur unité de $\mathbb{C}^{2^m}$, c'est-à-dire un élément de $\{v \in \mathbb{C}^{2^m} : \|v\|_2 = 1\}$. Un **opérateur** sur un m-qubit est une application linéaire unitaire $M \in \mathbb{C}^{2^m \times 2^m}$. **Mesurer** un m-qubit $(q_0, q_2, \ldots, q_{2^m-1}) \in \mathbb{C}^{2^m}$ consiste à tirer aléatoirement un entier $r \in \{0, 1, \ldots, 2^m - 1\}$, l'entier i étant choisi avec probabilité $|q_i|^2$. On considère dans ce cours que mesurer un m-qubit le détruit. [1]

1. C'est une hypothèse simplificatrice et un peu réductrice. Si certains systèmes physiques détruisent effectivement un m-qubit pour le mesurer, d'autres lui appliquent une projection aléatoire et permettent, en principe, de continuer à l'utiliser. Dans tous les cas, l'état interne est modifié et l'ancienne valeur ne peut pas être re-mesurée directement.

Un calcul quantique sur un m-qubit consiste à (i) initialiser un m-qubit, généralement avec l'entrée du problème considéré, (ii) appliquer à ce m-qubit un opérateur, (iii) mesurer le m-qubit obtenu et tirer de cette mesure une réponse à un problème (par exemple des diviseurs de cet entier).

Un peu d'intuition...

On peut envisager un m-qubit comme une information décrivant l'état interne d'un objet physique tel qu'un électron ou un photon. Un opérateur peut s'envisager comme le changement d'état qui résulte d'une action physique sur cet objet. L'observateur que nous sommes n'a pas directement accès à cet état interne, mais peut réaliser une mesure. Cette mesure donne un résultat probabiliste influencé par l'état interne de l'objet (et détruit ou modifie l'objet).

On peut aussi envisager le calcul quantique sous un angle probabiliste. Un 1-qubit (a, b) représente la loi de probabilité $|a|^2\delta_0 + |b|^2\delta_1$, où δ_x désigne la loi de probabilité produisant l'objet x avec probabilité 1. Cette loi de probabilité peut être modifiée par des opérateurs (unitaires) avant d'être, *in fine*, mesurée par *un* échantillonnage. Ce point de vue masque certaines subtilités importantes (c.f. le dernier paragraphe de la Section 9.2) mais donne une première intuition.

Ce qu'il nous reste à faire

Dans cet aperçu, il faut envisager l'application d'un opérateur (à un m-qubit) comme l'exécution non pas d'une instruction, mais d'un programme (sur une entrée). Autrement dit, un opérateur $\mathbb{C}^{2^m} \to \mathbb{C}^{2^m}$ est arbitrairement compliqué. Pour développer une algorithmique ou une théorie de la complexité de tels calculs, il est nécessaire de fixer des règles de décomposition d'un opérateur arbitraire en opérateurs « élémentaires ».

Un opérateur sur un m-qubit est considéré comme « élémentaire » s'il n'agit que sur un nombre constant de « qubits du m-qubit ». C'est, en principe, assez similaire à la notion d'*instruction élémentaire* dans le modèle RAM vu au Chapitre 2. Dans le détail, la notion de « qubit d'un m-qubit », délicate à définir, va nécessiter d'envisager un m-qubit comme un tenseur et d'user d'un système de notations bien pensé. On introduit tout cela graduellement au fil du chapitre, avant de conclure par une présentation de l'algorithme de Grover, un des rares exemples d'algorithme quantique aux performances inégalables dans les modèles classiques.

9.2 Calcul sur un qubit

Commençons par reprendre, commenter et illustrer le modèle dans le cas $m = 1$.

> En calcul classique, l'unité élémentaire d'information est le *bit*, à valeur dans $\{0, 1\}$.

Formellement, un **qubit** (ou **1-qubit**) est un vecteur de $\mathbb{C}^2$ de norme 1. L'ensemble des valeurs possibles d'un qubit est donc

$$Q \stackrel{\text{def}}{=} \{(a, b) \in \mathbb{C}^2 \colon |a|^2 + |b|^2 = 1\}.$$

Le terme qubit peut à la fois désigner une valeur de Q, ou une variable à valeurs dans Q. On adopte trois conventions usuelles en calcul quantique :

a. On note un qubit q sous la forme $|q\rangle$; insistons que $|\cdot\rangle$ n'est pas une opération, mais une simple notation signalant un vecteur.

b. On munit $\mathbb{C}^2$ d'une base canonique notée $(|0\rangle, |1\rangle)$. Le qubit (a, b) est donc noté $a|0\rangle + b|1\rangle$.

c. On s'autorise à omettre les facteurs de normalisation, et à noter par exemple $|0\rangle + |1\rangle$ pour désigner $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. Plus généralement, un vecteur $a|0\rangle + b|1\rangle$ de $\mathbb{C}^2 \setminus \{(0, 0)\}$ désigne le qubit $\frac{1}{\sqrt{|a|^2+|b|^2}}(a|0\rangle + b|1\rangle)$.

> En calcul classique, on peut appliquer à un bit toute application $\{0, 1\} \to \{0, 1\}$.

Les opérateurs permis en calcul quantique sont les *transformations unitaires de* $\mathbb{C}^2$, c'est-à-dire les matrices de $\mathbb{C}^{2\times 2}$ qui préservent la norme[2]. Voici quelques exemples de transformations unitaires de $\mathbb{C}^2$:

$$X \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y \stackrel{\text{def}}{=} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad H \stackrel{\text{def}}{=} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Par exemple, $X(a\,|0\rangle + b\,|1\rangle) = b\,|0\rangle + a\,|1\rangle$. Cet opérateur est aussi défini par son action sur la base $\{|0\rangle, |1\rangle\}$, à savoir $|0\rangle \mapsto |1\rangle, |1\rangle \mapsto |0\rangle$, ce qui lui vaut d'être décrite comme l'application qui « inverse » le qubit (par analogie avec le calcul booléen). Remarquons que l'opérateur H a notamment pour effet

$$|0\rangle \mapsto |0\rangle + |1\rangle \quad \text{et} \quad |0\rangle + |1\rangle \mapsto |0\rangle,$$

et transforme donc un qubit $|0\rangle$ (dit « neutre ») en qubit $|0\rangle + |1\rangle$ (dit « uniforme »), et vice-versa.

Soulignons que tout opérateur est par définition *réversible*, contrairement au modèle classique où par exemple l'instruction $0 \mapsto 0$ et $1 \mapsto 0$ est possible.

> En calcul classique, on peut lire la valeur d'un bit sans le détruire.

Une fois l'opérateur souhaité appliqué, l'information se trouve dans le modèle sous la forme d'un qubit $a\,|0\rangle + b\,|1\rangle$. On ne peut pas accéder aux valeurs a et b, mais on peut mesurer ce qubit. La **mesure** du qubit $a\,|0\rangle + b\,|1\rangle$ est une expérience aléatoire dont le résultat vaut $|0\rangle$ avec probabilité $|a|^2$ et $|1\rangle$ avec probabilité $|b|^2$. Mesurer un qubit le détruit et peut ne donner qu'un peu d'information sur sa valeur. Soulignons que si un qubit $a\,|0\rangle + b\,|1\rangle$ est obtenu par un calcul reproductible, alors on peut faire plusieurs mesures de ce qubit (en répétant le calcul à chaque mesure) et ainsi obtenir une estimation arbitrairement bonne de $|a|^2$ et $|b|^2$.

Même des mesures répétées[3] d'un qubit $a\,|0\rangle + b\,|1\rangle$ ne donnent pas accès à l'état (a, b) mais à son « reflet » $(|a|^2, |b|^2)$. Des qubits différents peuvent avoir le même « reflet » et être donc indistinguables par mesure, bien que se comportant différemment sous l'action d'opérateurs. Par exemple, les mesures des qubits $|0\rangle + |1\rangle$ et $|0\rangle - |1\rangle$ produisent le même résultat (un bit valant 0 ou 1 avec équiprobabilité), mais

$$H\,(|0\rangle + |1\rangle) = |0\rangle \quad \text{et} \quad H\,(|0\rangle - |1\rangle) = |1\rangle,$$

deux qubits distinguables avec certitude en une seule mesure.

9.3 Calcul sur deux qubits

Passons maintenant à $m = 2$.

Formellement, un **2-qubit** est un vecteur unitaire de $\mathbb{C}^4$. Il est usuel de munir $\mathbb{C}^4$ d'une base canonique notée $(|00\rangle, |01\rangle, |10\rangle, |11\rangle)$ et de noter $a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle$ le 2-qubit (a, b, c, d). Comme pour les valeurs d'un qubit, il est commode d'utiliser un vecteur de $\mathbb{C}^4 \setminus \{(0,0,0,0)\}$ arbitraire pour représenter le 2-qubit obtenu après normalisation (par exemple $|00\rangle + |11\rangle$ pour $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \left(\frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\right)$. Les opérateurs permis sur un 2-qubit sont les transformations unitaires de $\mathbb{C}^4$, par exemple :

$$\text{CNOT} \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

> En calcul classique, un système 2-bits se décompose en deux systèmes 1-bit, autrement dit $\{0,1\}^2 = \{0,1\} \times \{0,1\}$.

2. Le fait qu'une matrice $M \in \mathbb{C}^{2\times 2}$ préserve la norme peut se reconnaître au fait que ses colonnes forment une base orthonormée de $\mathbb{C}^2$.

3. Au sens où l'on répète le calcul produisant ce qubit.

Soient $|\phi\rangle = (a, b)$ et $|\psi\rangle = (c, d)$ deux qubits. Remarquons que $\chi \stackrel{\text{def}}{=} (ac, ad, bc, bd)$ est un 2-qubit puisque

$$|ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 = (|a|^2 + |b|^2)(|c|^2 + |d|^2) = 1.$$

On dit que le 2-qubit $|\chi\rangle$ est le *produit tensoriel* des qubits $|\phi\rangle$ et $|\psi\rangle$, ce que l'on note $|\chi\rangle = |\phi\rangle \otimes |\psi\rangle$.

Un 2-qubit $|\chi\rangle$ est dit **séparable** (ou **factorisable**) si il peut s'écrire comme produit tensoriel de deux qubits $|\chi\rangle = |\phi\rangle \otimes |\psi\rangle$. Un 2-qubit qui n'est pas séparable est dit **intriqué**. Par exemple, $|00\rangle + |11\rangle$ est intriqué puisque l'écrire comme $(a, b) \otimes (c, d)$ force $ad = bc = 0$ et $ac = bd \neq 0$, ce qui est impossible sur $\mathbb{C}$.

Définition d'un opérateur par référence au calcul booléen

On a vu que l'opérateur X sur un qubit peut s'interpréter comme inversant un qubit. Cette analogie s'étend à certains opérateurs sur 2-qubits car tout 2-qubit de la base canonique de $\mathbb{C}^4$ est séparable en produit tensoriel des qubits $|0\rangle$ et $|1\rangle$:

$$|00\rangle = |0\rangle \otimes |0\rangle\,, \quad |01\rangle = |0\rangle \otimes |1\rangle\,, \quad |10\rangle = |1\rangle \otimes |0\rangle\,, \quad |11\rangle = |1\rangle \otimes |1\rangle\,.$$

On peut donc associer à toute formule booléenne[4] $f : \{0,1\}^2 \to \{0,1\}$ une application linéaire $g : \mathbb{C}^4 \to \mathbb{C}^2$ définie par

$$\forall (x, y) \in \{0,1\}^2, \quad g(|xy\rangle) \stackrel{\text{def}}{=} |f(x, y)\rangle\,, \tag{9.1}$$

dont on peut se servir pour donner des interprétations intuitives d'opérateurs.

Par exemple, l'action de CNOT sur un 2-qubit de base inverse le second qubit si et seulement si le premier qubit égale $|1\rangle$. On résume cela par la notation $\text{CNOT}(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |x \oplus y\rangle$. Cette interprétation, qui donne à cet opérateur son nom de *Controlled NOT*, n'a bien sûr de sens que sur les vecteurs de la base canonique de $\mathbb{C}^4$.

Cette notation est à utiliser avec discernement car toute fonction ainsi construite n'est pas un opérateur permis en calcul quantique. Par exemple, l'application linéaire $|xy\rangle \mapsto |x \vee y\rangle \otimes |x \wedge y\rangle$ n'est pas unitaire (elle n'est même pas inversible puisque $|01\rangle$ et $|10\rangle$ ont la même image).

Impossibilité du clonage quantique

> En calcul classique, on peut copier un bit sur un autre.

Nous pouvons maintenant énoncer et prouver une première propriété simple mais fondamentale du modèle de calcul quantique : il est *impossible* de copier un qubit. Cela se formalise par l'énoncé suivant, connu sous le nom de principe d'impossibilité du clonage quantique (*no-cloning theorem*) :

Proposition 9.1. *Il n'existe pas de transformation unitaire $T : \mathbb{C}^4 \to \mathbb{C}^4$ tel que pour tout qubit $|\phi\rangle$ on ait $T(|\phi\rangle \otimes |0\rangle) = |\phi\rangle \otimes |\phi\rangle$.*

On traite cette preuve en exercice.

9.4 Calcul sur m qubits

Revenons maintenant à un m arbitraire.

Un **m-qubit** est un vecteur unité de $\mathbb{C}^{2^m}$ et un opérateur sur un m-qubit est une application unitaire de $\mathbb{C}^{2^m}$ dans lui-même. La décomposition de (certains) m-qubits se fait en généralisant le recours au produit tensoriel. Cette décomposition s'avère essentielle pour définir la notion d'opérateur élémentaire, notion fondamentale pour examiner le calcul quantique du point de vue de l'algorithmique et de la théorie de la complexité.

4. Rappelons que $\vee$, $\wedge$, $\neg$ et $\oplus$ désignent respectivement les opérateurs `ou`, `et`, `non` et `ou exclusif`.

Calcul tensoriel sur $\mathbb{C}$-espaces vectoriels

Le **produit tensoriel** de deux $\mathbb{C}$-espaces vectoriels $E = \mathbb{C}^k$ et $F = \mathbb{C}^\ell$ est le $\mathbb{C}$-espace vectoriel $\mathbb{C}^{\ell k}$, noté $E \otimes F$ et muni d'une application *bilinéaire* $(E, F) \to E \otimes F$ définie comme suit.

a. On fixe des bases $(e_1, e_2, \ldots, e_k)$ de E, $(f_1, f_2, \ldots, f_\ell)$ de F et $(t_1, t_2, \ldots, t_{k\ell})$ de $E \otimes F$.

b. On note ϕ la bijection entre $\{(e_i, f_j) \colon 1 \leq i \leq k, 1 \leq j \leq \ell\}$ et $\{t_i : 1 \leq i \leq k\ell\}$ qui envoie[5] (e_1, f_1) sur t_1, (e_1, f_2) sur t_2, …, (e_1, f_ℓ) sur t_ℓ, (e_2, f_1) sur $t_{\ell+1}$, …, (e_k, f_ℓ) sur $t_{k\ell}$.

c. On étend ϕ par bilinéarité :

$$\phi : \left\{ \begin{array}{ccc} E \times F & \to & E \otimes F \\ \left(\sum_{i=1}^{k} a_i e_i, \sum_{j=1}^{\ell} b_j f_j\right) & \mapsto & \sum_{i=1}^{k}\sum_{j=1}^{\ell} a_i b_j \phi(e_i, f_j). \end{array} \right.$$

On note généralement l'application ϕ sous la forme d'un produit, c'est-à-dire que pour $u \in E$ et $v \in F$ on note $u \otimes v \stackrel{\text{def}}{=} \phi(u, v)$. Cela signifie que $(e_1 \otimes f_1, e_1 \otimes f_2, \ldots e_1 \otimes f_\ell, e_2 \otimes f_1, \ldots, e_k \otimes f_\ell)$ est une base de $E \otimes F$; c'est un simple renommage de la base $(t_1, t_2, \ldots, t_{k\ell})$.

Pour tous $\mathbb{C}$-espaces vectoriels E, F et G, il existe un isomorphisme canonique entre $(E \otimes F) \otimes G$ et $E \otimes (F \otimes G)$ qui identifie $(e_i \otimes f_j) \otimes g_k$ et $e_i \otimes (f_j \otimes g_k)$. Il est donc naturel d'identifier ces deux produits tensoriels en un même espace $E \otimes F \otimes G$. Autrement dit, le produit tensoriel est associatif. On note $E^{\otimes n}$ le produit tensoriel de n copies de E. En particulier,

$$\mathbb{C}^{2^m} = \left(\mathbb{C}^2\right)^{\otimes m} = \underbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2}_{m \text{ fois}}$$

> On peut envisager l'espace $\mathbb{C}^{2^m}$ qui contient les m-qubits comme le produit tensoriel $\left(\mathbb{C}^2\right)^{\otimes m}$ de m espaces contenant chacun un qubit.

Rappelons que le produit Hermitien sur $\mathbb{C}^n$ est $\langle x, y\rangle \stackrel{\text{def}}{=} x_1\overline{y_1} + x_2\overline{y_2} + \ldots + x_n\overline{y_n}$. Pour $u \in \mathbb{C}^k$ et $v \in \mathbb{C}^\ell$ on a

$$\|u \otimes v\|_2{}^2 = \langle u \otimes v, u \otimes v\rangle = \sum_{i=1}^{k}\sum_{j=1}^{\ell} u_i v_j \ \overline{u_i v_j} = \sum_{i=1}^{k} u_i\overline{u_i} \sum_{j=1}^{\ell} v_j\overline{v_j} = \|u\|_2{}^2 \|v\|_2{}^2.$$

En particulier, le produit tensoriel de deux vecteurs unité est un vecteur unité.

> Le produit tensoriel d'un m-qubit et d'un n-qubit est un $(m+n)$-qubit.

Formellement, le *produit tensoriel d'applications linéaires* $f_1 : E_1 \to F_1$, $f_2 : E_2 \to F_2$, …, $f_n : E_n \to F_n$ est une application linéaire allant de $E_1 \otimes E_2 \otimes \ldots \otimes E_n$ dans $F_1 \otimes F_2 \otimes \ldots \otimes F_n$. Elle est notée $f_1 \otimes f_2 \otimes \ldots \otimes f_n$ et définie par

$$\underbrace{(f_1 \otimes f_2 \otimes \ldots \otimes f_n)}_{\text{nom de l'application}} \colon \underbrace{(e_1 \otimes e_2 \otimes \ldots \otimes e_n)}_{\text{élément de } E_1 \otimes E_2 \otimes \ldots \otimes E_n} \mapsto \underbrace{f_1(e_1) \otimes f_2(e_2) \otimes \ldots \otimes f_n(e_n)}_{\text{élément de } F_1 \otimes F_2 \otimes \ldots \otimes F_n}.$$

Exemples

Reprenons les applications X et Y opérant sur un qubit :

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \text{et} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

5. Autrement dit, $\phi((e_i, f_j)) \stackrel{\text{def}}{=} t_{\ell(i-1)+j}$.

Leur produit tensoriel $X \otimes Y$ est l'application de $\mathbb{C}^2 \otimes \mathbb{C}^2$ dans lui-même, définie par

$$(X \otimes Y) : \begin{cases} |0\rangle \otimes |0\rangle & \mapsto & X(|0\rangle) \otimes Y(|0\rangle) = |1\rangle \otimes i\,|1\rangle & = & i\,|11\rangle \\ |0\rangle \otimes |1\rangle & \mapsto & X(|0\rangle) \otimes Y(|1\rangle) = |1\rangle \otimes -i\,|0\rangle & = & -i\,|10\rangle \\ |1\rangle \otimes |0\rangle & \mapsto & X(|1\rangle) \otimes Y(|0\rangle) = |0\rangle \otimes i\,|1\rangle & = & i\,|01\rangle \\ |1\rangle \otimes |1\rangle & \mapsto & X(|1\rangle) \otimes Y(|1\rangle) = |0\rangle \otimes -i\,|0\rangle & = & -i\,|00\rangle \end{cases}$$

Cette application est donc de matrice

$$X \otimes Y = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix} \quad \text{et l'on reconnaît} \quad X \otimes Y = \begin{pmatrix} 0 \cdot Y & 1 \cdot Y \\ 1 \cdot Y & 0 \cdot Y \end{pmatrix} = \begin{pmatrix} 0 & Y \\ Y & 0 \end{pmatrix}.$$

Il existe des opérateurs de $\mathbb{C}^4$ **indécomposables** en produit tensoriel d'applications de $\mathbb{C}^2$ dans lui-même. Par exemple, si l'opérateur CNOT (défini en Section 9.3) devait s'écrire $A \otimes B$ avec $A : \mathbb{C}^2 \to \mathbb{C}^2$ et $B : \mathbb{C}^2 \to \mathbb{C}^2$, alors le bloc 2×2 haut-gauche force B à être un multiple de l'identité, tandis que le bloc 2×2 bas-droite force B à être nul sur la diagonale.

Système à m qubits et algorithme quantique

On munit $\left(\mathbb{C}^2\right)^{\otimes m}$ de la base canonique $\{|w_1\rangle \otimes |w_2\rangle \otimes \ldots \otimes |w_m\rangle : w \in \{0,1\}^m\}$. Pour $w \in \{0,1\}^m$ on écrit $|w\rangle \stackrel{\text{def}}{=} |w_1\rangle \otimes |w_2\rangle \otimes \ldots \otimes |w_m\rangle$. L'associativité du produit tensoriel assure que pour tous mots binaires w, w' on a $|w\rangle \otimes |w'\rangle = |ww'\rangle$, où ww' désigne la concaténation de w et w'.

Un m-qubit est **séparable** (ou factorisable) si il peut s'écrire comme un produit tensoriel d'un m_1-qubit et d'un m_2-qubit avec $m_1, m_2 > 0$ (on a alors forcément $m_1 + m_2 = m$). Un m-qubit qui n'est pas séparable est dit **intriqué**. Tout vecteur de la base canonique de $\left(\mathbb{C}^2\right)^{\otimes m}$ est séparable, tandis que le m-qubit $|0^m\rangle + |1^m\rangle$ est intriqué.

Opérateur agissant sur un ou deux qubits

Un opérateur unitaire $T : \left(\mathbb{C}^2\right)^{\otimes m} \to \left(\mathbb{C}^2\right)^{\otimes m}$ **agit sur un qubit** s'il existe un opérateur unitaire $T' : \mathbb{C}^2 \to \mathbb{C}^2$ et un entier $p \in \{1, 2, \ldots, m\}$ tel que $T = \mathrm{id}_{(\mathbb{C}^2)^{\otimes(p-1)}} \otimes T' \otimes \mathrm{id}_{(\mathbb{C}^2)^{\otimes(m-p)}}$. On dit que T agit sur le $p$ième qubit. [6]

Un opérateur unitaire $T : \left(\mathbb{C}^2\right)^{\otimes m} \to \left(\mathbb{C}^2\right)^{\otimes m}$ **agit sur deux qubits** s'il existe un opérateur unitaire $T' : \mathbb{C}^4 \to \mathbb{C}^4$ et des entiers $1 \leq i < j \leq m$ tels que

$$T = T_{i+1,j} \circ \left(\mathrm{id}_{(\mathbb{C}^2)^{\otimes(i-1)}} \otimes T' \otimes \mathrm{id}_{(\mathbb{C}^2)^{\otimes(m-i-1)}}\right) \circ T_{i+1,j}$$

où $T_{i,j}$ est l'opérateur qui « échange les $i$ème et $j$ème qubits ». Formellement, on peut définir $T_{i,j}$ comme l'unique opérateur satisfaisant $T_{i,j}(|w_1 x w_2 y w_3\rangle) = |w_1 y w_2 x w_3\rangle$ pour tous $w_1 \in \{0,1\}^{i-1}, x \in \{0,1\}, w_2 \in \{0,1\}^{j-i-1}, y \in \{0,1\}, w_3 \in \{0,1\}^{m-j}$. L'application $T_{i,j}$ permute les vecteurs de la base canonique de $\left(\mathbb{C}^2\right)^{\otimes m}$ et est par conséquent unitaire. Autrement dit, un opérateur qui agit sur les $i$ème et $j$ème qubits revient à échanger les $(i+1)$ème et $j$ème qubits, à agir sur les $i$ème et $(i+1)$ème qubits, puis à échanger à nouveau les $(i+1)$ème et $j$ème qubits. On dit qu'un tel opérateur agit sur les $i$ème et $j$ème qubits.

Porte et algorithme quantiques

> En calcul classique, toute fonction booléenne (et donc toute instruction) est calculable au moyen de portes `et`, `ou` et `non`.

Une **porte quantique** est un opérateur $T : \left(\mathbb{C}^2\right)^{\otimes m} \to \left(\mathbb{C}^2\right)^{\otimes m}$ qui agit sur un ou deux qubits. Il s'avère que les portes quantiques engendrent tous les opérateurs unitaires.

6. Ici, id_E désigne l'identité sur l'espace E.

Théorème 9.2. *Pour tout opérateur unitaire* $T : \left(\mathbb{C}^2\right)^{\otimes m} \to \left(\mathbb{C}^2\right)^{\otimes m}$ *il existe une suite finie* T_1, T_2, …, T_k *de portes quantiques de* $\left(\mathbb{C}^2\right)^{\otimes m}$ *dans* $\left(\mathbb{C}^2\right)^{\otimes m}$ *telle que* $T = T_k \circ T_{k-1} \circ \ldots \circ T_1$.

Prouver ce théorème dépasserait le cadre de cette séance, mais on renvoie les élèves intéressé·es à [KSVV02, Théorème 8.1]. Soulignons que comme l'ensemble des opérateurs unitaires est non-dénombrable, toute famille génératrice est infinie. On peut cependant se ramener à des familles génératrices finies par des méthodes d'approximation.

> En calcul classique, (i) une instruction est élémentaire si elle agit sur un nombre borné de cases mémoire, et (ii) un algorithme est une suite finie d'instructions élémentaires.

En calcul quantique, un opérateur est **élémentaire** s'il est égal à la composition d'un nombre borné de portes quantiques. Les opérateurs élémentaires sont ceux qui peuvent s'exprimer comme compositions et produits tensoriels de matrices creuses.

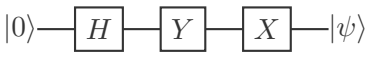
Un **algorithme quantique** est une suite finie d'opérateurs élémentaires.

9.5 Circuits

Un algorithme quantique peut être représenté par un diagramme similaire à un circuit booléen classique. On décrit dans cette section les conventions d'écriture de circuit et on donne quelques exemples supplémentaires de portes quantiques, notamment les *portes contrôlées*.

Représentation d'une porte agissant sur un ou deux qubits

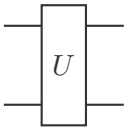
Dans un circuit quantique, chaque qubit est représenté par un fil. Les opérateurs agissant sur ce qubit sont représentés par des boites étiquetées par l'opérateur. Dans ce cours[7], les portes sont appliquées successivement de gauche à droite. La valeur indiquée à gauche du fil indique la valeur initiale du qubit, la valeur à droite indique sa valeur finale. Ainsi, le circuit



indique que le qubit est initialisé à $|0\rangle$, puis qu'on lui applique un opérateur H, puis un opérateur Y, puis un opérateur X, pour produire le qubit $|\psi\rangle$. On a ainsi

$$|\psi\rangle = XYH\,|0\rangle = XY(|0\rangle + |1\rangle) = X(i\,|1\rangle - i\,|0\rangle) = i\,|0\rangle - i\,|1\rangle\,.$$

Une porte agissant sur deux qubits est, de même, représentée par une boite ayant deux entrées et deux sorties. Ainsi, un opérateur noté U agissant sur deux qubits serait représenté par :



Portes contrôlées et leur représentation

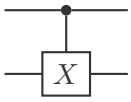
Pour tout opérateur $G\colon \mathbb{C}^{2^m} \to \mathbb{C}^{2^m}$ sur un m-qubit, on définit l'opérateur $\Lambda(G)$ sur un $(m+1)$-qubit par[8]

$$\Lambda(G)\colon \begin{cases} |0\rangle \otimes |\psi\rangle \mapsto |0\rangle \otimes |\psi\rangle\,, \\ |1\rangle \otimes |\psi\rangle \mapsto |1\rangle \otimes G(|\psi\rangle). \end{cases}$$

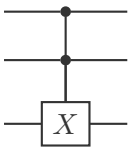
7. Attention, certaines sources adoptent la convention d'une lecture de droite à gauche, par exemple [KSVV02].
8. On décrit ici l'action de $\Lambda(G)$ sur un $(m+1)$-qubit séparable $|\phi\rangle \otimes |\psi\rangle$, où $|\psi\rangle$ est un m-qubit. L'action sur un $(m+1)$-qubit intriqué est obtenue en étendant cette définition par linéarité.

La matrice de $\Lambda(G)$ est donc $\begin{pmatrix} \mathrm{id}_{(\mathbb{C}^2)^{\otimes m}} & 0 \\ 0 & G \end{pmatrix}$. Le premier qubit auquel s'applique $\Lambda(G)$ est appelé **qubit de contrôle**.

On peut vérifier que l'opérateur CNOT défini ci-dessus n'est autre que l'opérateur $\Lambda(X)$, appliquant X sur le second qubit, contrôlé par le premier qubit. Pour tout opérateur G sur un qubit, on représente $\Lambda(G)$ par l'opérateur G sur le fil du second qubit, rattaché au qubit de contrôle par un point. Pour $\Lambda(X)$, cela donne :



On peut « itérer » le contrôle, c'est-à-dire définir $\Lambda(\Lambda(X))$. La convention de représentation consiste à rattacher l'operateur X à chacun des qubits qui le controle. Pour $\Lambda(\Lambda(X))$, on obtient :

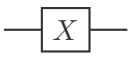


dont l'action est $\begin{cases} |00\psi\rangle \mapsto |00\psi\rangle\,, \\ |01\psi\rangle \mapsto |01\psi\rangle\,, \\ |10\psi\rangle \mapsto |10\psi\rangle\,, \\ |11\psi\rangle \mapsto |11\rangle \otimes X\,|\phi\rangle\,. \end{cases}$

Quelques portes usuelles

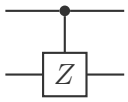
Donnons (ou rappelons) les matrices et représentations de quelques portes usuelles.

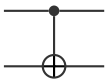
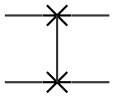
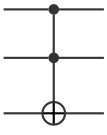
- La porte *de Hadamard*  a pour matrice $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.

- La porte  a pour matrice $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

- La porte  a pour matrice $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$.

- La porte  a pour matrice $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.

- La porte *controlled Z*  a pour matrice $CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$.

- La porte *controlled NOT*  a pour matrice $\mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$.
- La porte *SWAP*  a pour matrice $\mathrm{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.
- La porte *Toffoli*  a pour matrice $\mathrm{CCNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$.

9.6 Algorithme de Grover

Considérons le problème algorithmique suivant :

RECHERCHE UNIVOQUE

Entrée : Une fonction booléenne $f : \{0,1\}^n \to \{0,1\}$ telle que $f(w^*) = 1$ pour exactement un mot $w^* \in \{0,1\}^n$.

Sortie : Le mot w^* tel que $f(w^*) = 1$.

On suppose que l'on ne connaît la fonction f qu'en tant que boîte noire. Autrement dit, on peut calculer $f(w)$ pour tout mot $w \in \{0,1\}^n$ qui nous chante, mais on ne peut étudier f qu'au travers des valeurs qu'elle prend. Cela revient à travailler dans un modèle de calcul usuel (classique ou quantique) augmenté d'une instruction (non élémentaire, mais de coût unité) qui calcule la valeur de f pour un mot donné en entrée. On appelle cette instruction supplémentaire un *oracle*.

On peut montrer par un argument d'adversaire que dans le modèle RAM 8 bits augmenté de l'oracle f, le problème RECHERCHE UNIVOQUE est de complexité $\Omega\left(2^n\right)$; plus précisément, tout algorithme qui résout ce problème fait, dans le pire cas, au moins $2^n - 1$ appels à l'oracle f. L'algorithme quantique de Grover résout ce problème en $O\left(2^{n/2}\right)$ opérateurs élémentaires sur[9] $\approx n$ qubits. Autrement dit, si on note $N \stackrel{\text{def}}{=} 2^n$, le problème RECHERCHE UNIVOQUE de complexité $\Omega(N)$ dans le modèle classique peut être résolu par un algorithme quantique de complexité $O(\sqrt{N})$; on parle d'un *gain quadratique*.

L'idée géométrique

L'algorithme de Grover peut s'expliquer assez facilement en termes géométriques. Notons a le mot solution (que l'on ne connaît pas), $|a\rangle$ le vecteur correspondant de la base standard, et $|u\rangle$ le mot « uniforme » $|u\rangle = \frac{1}{2^{n/2}} \sum_{w\in\{0,1\}^n} |w\rangle$. Posons[10] $|e\rangle \stackrel{\text{def}}{=} |u\rangle - \frac{1}{2^{n/2}} |a\rangle$.

$|u\rangle$, $|a\rangle$ et $|e\rangle$ sont contenus dans $\text{vect}_{\mathbb{R}}\left(|u\rangle, |a\rangle\right)$; de plus, $|a\rangle$ et $|e\rangle$ sont orthogonaux.

L'algorithme de Grover calcule[11] une suite $|g_1\rangle$, $|g_2\rangle$, ...de n-qubits dans le plan $\text{vect}\left(|u\rangle, |a\rangle\right)$. Le premier terme est initialisé à $|g_1\rangle \stackrel{\text{def}}{=} |u\rangle$. Le passage de $|g_i\rangle$ à $|g_{i+1}\rangle$ se fait en deux étapes :

- étant donné $|g_i\rangle$, on définit un vecteur auxiliaire $|g_i'\rangle$ comme l'image de $|g_i\rangle$ par la réflexion d'axe $|e\rangle$ dans le plan $\text{vect}_{\mathbb{R}}\left(|u\rangle, |a\rangle\right)$,
- puis on définit $|g_{i+1}\rangle$ comme l'image de $|g_i'\rangle$ par la réflexion d'axe $|u\rangle$, toujours dans le plan $\text{vect}_{\mathbb{R}}\left(|u\rangle, |a\rangle\right)$.

Une itération de ce calcul est représentée Figure 9.1. Il s'avère que la suite de n-qubits $\{|g_i\rangle\}$ converge rapidement vers $|a\rangle$, et que les réflexions par rapport à $|e\rangle$ et à $|u\rangle$ peuvent être calculées en temps polynomial par un algorithme quantique.

La convergence

Commençons par la **convergence** de la suite $\{|g_i\rangle\}$ vers $|a\rangle$. Puisque les opérations géométriques se déroulent dans $\text{vect}_{\mathbb{R}}\left(|u\rangle, |a\rangle\right)$, le coefficient de $|a\rangle$ dans $\{|g_i\rangle\}$ est un réel. Montrons que ce réel vaut au moins $\frac{\sqrt{2}}{2}$ pour $i \approx 2^{n/2}$, ce qui assure qu'une mesure donne a avec probabilité au moins $\frac{1}{2}$.

Notons $\frac{\pi}{2} - \theta$ l'angle entre $|a\rangle$ et $|u\rangle$. Puisque les vecteurs $|a\rangle$ et $|u\rangle$ sont réels et unitaires, on a

$$\cos\left(\frac{\pi}{2} - \theta\right) = |a\rangle \cdot |u\rangle = \frac{1}{2^{n/2}} > 0,$$

9. On revient sur la question du nombre de qubits nécessaires en fin de section.
10. Rappelons que l'utilisation de vecteurs non-unité n'est qu'une facilité de notation, la normalisation étant implicite. On a explicité cette normalisation pour $|u\rangle$ par anticipation, ce vecteur intervenant explicitement dans des calculs.
11. Pour simplifier, on omet ici les qubits ancillaires. On revient sur cette question dans les prolongements.

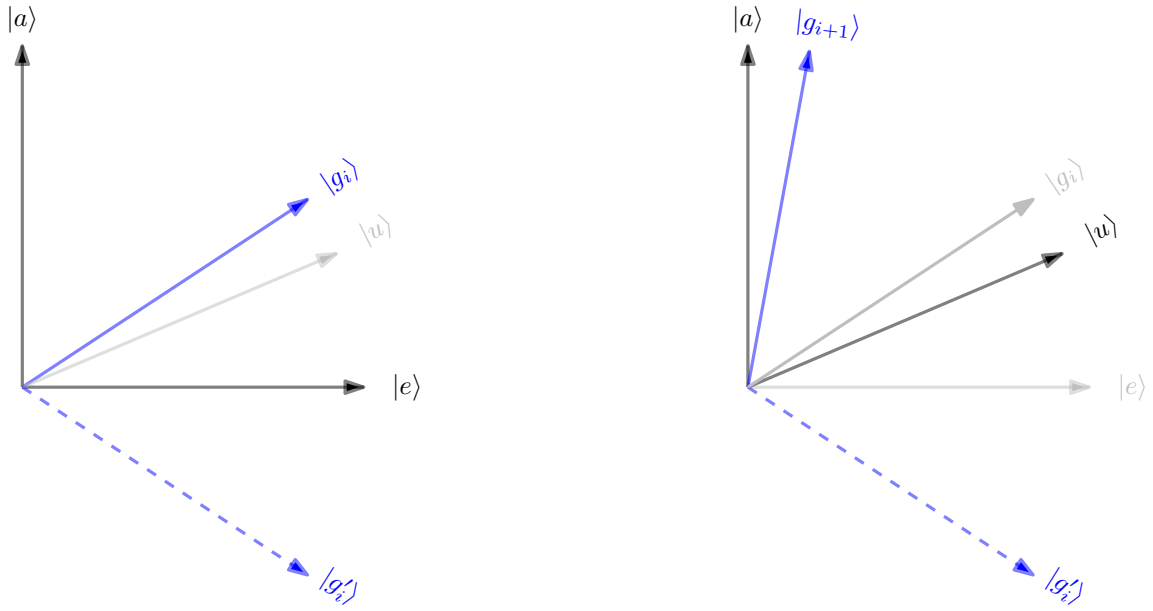


FIGURE 9.1 – Une itération de l'algorithme de Grover. Gauche : $|g'_i\rangle$ est obtenu en appliquant à $|g_i\rangle$ une réflexion d'axe $|e\rangle$. Droite : $|g_{i+1}\rangle$ est obtenu en appliquant à $|g'_i\rangle$ une réflexion d'axe $|u\rangle$.

d'où $0 < \theta < \frac{\pi}{2}$ pour $n \geq 1$. Notons $\frac{\pi}{2} - \alpha_i$ l'angle entre $|a\rangle$ et $|g_i\rangle$. On a donc $\alpha_1 = \theta$, puisque $|g_1\rangle = |u\rangle$, et un peu de trigonométrie révèle que

$$\frac{\pi}{2} - \alpha_{i+1} = \frac{\pi}{2} - \alpha_i - 2\theta = \ldots = \frac{\pi}{2} - (2i-1)\theta.$$

Posons $i^* \stackrel{\text{def}}{=} \lceil \frac{\pi}{8\theta} \rceil + 1$, c'est-à-dire le premier indice i tel que $(2i-1)\theta \geq \frac{\pi}{4}$. Remarquons que $|a\rangle \cdot |g_{i^*}\rangle \geq \frac{\sqrt{2}}{2}$, ce qui assure que le coefficient de $|a\rangle$ dans $|w\rangle$ est au moins $\frac{\sqrt{2}}{2}$ comme annoncé. De plus,

$$i^* \leq \frac{1}{\theta} \leq \frac{1}{\sin\theta} = \frac{1}{\cos\left(\frac{\pi}{2} - \theta\right)} = 2^{n/2}.$$

La traduction algorithmique

Voyons maintenant comment les réflexions par rapport à $|u\rangle$ et à $|e\rangle$ peuvent se traduire algorithmiquement.

Réflexion par rapport à $|e\rangle$. Considérons un n-qubit $|q\rangle = \sum_{w\in\{0,1\}^n} q_w |w\rangle$ dans le plan vectoriel engendré par $|u\rangle$ et $|a\rangle$, $\text{vect}_{\mathbb{R}}(|u\rangle, |a\rangle)$. Notons $|q'\rangle$ l'image de $|q\rangle$ par la réflexion d'axe $|e\rangle$ dans ce plan. Puisque $|a\rangle$ est un vecteur de base et que $|e\rangle = \sum_{w\in\{0,1\}^n\setminus\{a\}} |w\rangle$, on a $|q'\rangle = |q\rangle - q_a |a\rangle$. Cette transformation peut être calculée comme suit :

- On commence par calculer l'opérateur $|x\sigma\rangle \mapsto |x(\sigma \oplus f(x))\rangle$. Cet opérateur agit sur un $(n+1)$-qubit (x étant un n-qubit). Ici on se sert de l'hypothèse que f est une fonction booléenne calculable.[12] Cette transformation envoie $|x0\rangle$ sur $|x0\rangle$ si $x \neq a$ et sur $|a1\rangle$ si $x = a$.
- On applique ensuite une porte Z au dernier qubit de notre $(n+1)$-qubit. L'ensemble envoie donc $|x0\rangle$ sur $|x0\rangle$ si $x \neq a$ et sur $-|a1\rangle$ si $x = a$.
- On applique à nouveau l'application $|x\sigma\rangle \mapsto |x(\sigma \oplus f(x))\rangle$. L'ensemble envoie donc $|x0\rangle$ sur $|x0\rangle$ si $x \neq a$ et sur $-|a0\rangle$ si $x = a$.

12. Ce calcul peut mettre en jeu des qubits ancillaires, aspect que l'on ne discute pas ici pour simplifier. Voir les prolongements.

Réflexion par rapport à $|u\rangle$. La méthode précédente permet de calculer rapidement une réflexion par rapport à n'importe quel vecteur $|x\rangle$ de la base standard. En effet, il suffit de l'appliquer en remplaçant f par la fonction booléenne (facilement calculable) qui vaut 1 pour x et 0 pour tout autre mot. On peut se remener à ce calcul en appliquant l'opérateur de Hadamard H à chaque qubit. D'une part, cela ne change pas le produit scalaire entre nos deux qubits [13]. D'autre part, cela transporte $|u\rangle$ en $|0^n\rangle$, vecteur de la base standard. Une fois la réflexion par rapport à $|0^n\rangle$ effectuée, on réapplique $H^{-1} = H$ à chaque qubit pour annuler la transformation.

9.7 Prolongements

L'algorithme de Grover résout un problème plus général que RECHERCHE UNIVOQUE : il permet de déterminer en temps $O(2^{n/2}\text{poly}(n))$ si une formule CNF de complexité n est satisfiable. Pour ce problème, les meilleurs algorithmes classiques connus sont de complexité $\Omega(2^n\text{poly}(n))$, aussi pour ce problème plus général l'accélération est elle aussi quadratique. Soulignons qu'un grand nombre d'algorithmes quantiques se résument à appliquer (parfois très astucieusement) l'algorithme de Grover.

Le calcul quantique ne permet de calculer que des fonctions inversibles. Cela amène parfois à « se donner de la place » en ajoutant des qubits au vecteur de travail. Ainsi, le calcul du $\wedge$ logique ne peut se faire sur 2-qubits puisque $|00\rangle$, $|01\rangle$ et $|10\rangle$ doivent être envoyés sur un 2-qubit ayant un même qubit à 0.

Tout comme en calcul classique, on peut être amené en calcul quantique à introduire des variables auxiliaires destinés à réaliser un calcul intermédiaire. Ces qubits supplémentaires sont appelés *ancillaires*. Un principe général en calcul quantique consiste à *annuler les calculs intermédiaires*, c'est-à-dire à remettre à un état standard (typiquement $|0\rangle$) chaque qubit ancillaire une fois qu'il n'est plus utilisé. Cette pratique est guidée par l'économie : ces qubits n'étant pas utiles au résultat, il n'est pas utile de les mesurer, et les remettre dans un état standard permet leur réutiliser dans d'autres calculs. Remarquons que dans l'algorithme de Grover, le calcul de réflexion fait intervenir un qubit ancillaire (σ) qui est initialement à 0 et que l'on remet à 0.

9.8 Références bibliographiques

[KSVV02] Alexei Yu Kitaev, Alexander Shen, Mikhail N Vyalyi, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.

À retenir.

- Le modèle de calcul quantique change la manière dont on représente l'information : une cellule mémoire élémentaire ne stocke pas un élément de $\{0, 1\}$ lisible à loisir et de manière déterministe, mais un élément de $\mathbb{C}^2$, mesurable une seule fois et de manière probabiliste.
- Un m-qubit vit dans l'espace vectoriel engendré par le produit tensoriel de m qubits.
- Un m-qubit est décomposable s'il peut s'écrire comme produit tensoriel d'un m_1-qubit et d'un m_2-qubit avec $m_1, m_2 > 0$.
- Une porte quantique est un opérateur unitaire qui agit sur au plus deux qubits.
- Un calcul quantique est un opérateur unitaire. Un circuit quantique réalise un calcul s'il le décompose en produit de portes quantiques.
- L'algorithme de Grover résout en temps $O(2^{n/2}\text{poly}(n))$ un problème de complexité $\Omega(2^n)$ dans le modèle RAM 8 bits.

13. Vérifier cela qubit par qubit grâce à l'identité $ac + bd = \frac{1}{2}\left((a + b)(c + d) + (a - b)(c - d)\right)$.

Annexes

Les chapitres qui suivent rappellent les principes de notation asymptotique et précisent les conventions de présentation des algorithmes dans ce cours.

Annexe A

Conventions de présentation d'un algorithme dans ce cours

Dans ce cours, la description d'un algorithme a pour but de communiquer à un lecteur ou une lectrice, supposé·e intelligent·e, les principes de son fonctionnement, et ce de la manière la plus efficace possible. Un algorithme est généralement intéressant car il résout un problème algorithmique **tout en en assurant certaines propriétés** : faible complexité, optimalité en un certain sens, ou encore garantie que la solution satisfait certaines conditions. [1]

> La présentation d'un algorithme doit décrire efficacement comment cet algorithme résout ce problème en ayant les propriétés qui le rendent intéressant.

A.1 Un algorithme n'est pas un code

Commençons par clarifier ce qu'il **ne faut pas** faire... On voit parfois des algorithmes présentés sous la forme de programmes rédigés dans un langage de programmation (`python`, `C`, `OCaml`, ...).

> La présentation d'un algorithme par un programme est **interdite** dans ce cours.

On autorise cependant l'usage de pseudocode, cf ci-après. Voici quelques unes des raisons motivant cette interdiction :

- C'est *laborieux* car les langages de programmation ont une syntaxe rigide qu'il est inutile de mémoriser pour réfléchir aux algorithmes.
- C'est *trompeur* car les langages de programmation fournissent des primitives sophistiquées dont le comportement dépend de la manière dont le langage les implante. Par exemple, la complexité asymptotique pire-cas de l'insertion, de la recherche et de la suppression dans un `set` de `python` est régulièrement mal comprise. [2]
- C'est *inefficace* car le niveau de détail imposé par un langage de programmation oblige à expliciter beaucoup de détails sans aucune importance.
- C'est *restrictif* car les algorithmes, les langages de programmation et les architectures matérielles évoluent conjointement pour rendre le calcul plus facile et plus efficace [3] et penser les algorithmes au travers d'un langage particulier entrave cette évolution.

> En algorithmique, on étudie les algorithmes indépendamment de leur implantation afin de monter en abstraction et ainsi gagner en **simplicité** et en **généralité**.

1. Par exemple, l'algorithme de Karatsuba résout la multiplication d'entiers longs n-bits en temps $o(n^2)$, l'algorithme de Gale-Shapley a la propriété d'être « optimal pour les demandeurs » (Théorème 1.6), et l'algorithme des paires ordonnées (Chapitre 5), qui décide du résultat d'une élection, fournit un résultat respectant la règle de Condorcet.

2. Une table de hachage peut être implantée de plusieurs manières, selon par exemple que l'on gère les collisions par liste chaînée, par adressage ouvert linéaire/quadratique, par hachage coucou, par re-hachage...La complexité de l'insertion, de la suppression et de la recherche peut varier selon l'implantation mais **n'est jamais** $O(1)$.

3. Par exemple, l'usage intensif de certaines opérations dans des algorithmes beaucoup utilisés conduit à créer des instructions dédiées au niveau des processeurs ; les *tensor processing unit* (TPU) sont un exemple récent.

L'implantation efficace des algorithmes est un sujet important et fait l'objet d'une discipline à part entière, **l'ingénierie algorithmique**, qui combine algorithmique, architecture des ordinateurs et principes des langages de programmation.

A.2 Présentation par pseudocode et présentation textuelle

Dans ce cours, la présentation des algorithmes se fera par une combinaison de **présentation par pseudocode** et de **présentation textuelle**.

Présentation par pseudocode

Un **pseudocode** est une approximation d'un code informatique qui se veut facilement compréhensible par toute personne qui maitrise un langage de programmation d'une famille donnée. Voici par exemple une présentation par pseudocode d'un algorithme de calcul du minimum d'une liste de n entiers donnés en entrée :

```
fonction min(L[1..n])
   record = L[1]
   pour i = 2..n
     si record > L[i]
       record = L[i]
   retourner record
```

On trouve sur la wikipédia de nombreux exemples de pseudocodes facilement compréhensibles par toute personne familière avec au moins un langage impératif (C, python, Pascal, ...).

Présentation textuelle

Un algorithme peut aussi être présenté par un texte décrivant ce qu'il fait. Voici par exemple une présentation textuelle de l'algorithme présenté par pseudocode ci-dessus :

> Parcourir la liste élément par élément, en maintenant dans une variable auxiliaire la plus petite valeur examinée jusque là. Une fois le parcours terminé, retourner la valeur de cette variable auxiliaire.

La présentation de leur algorithme par Gale et Shapley, reproduite au Chapitre 1, est un autre exemple moins élémentaire.

Comparaison et combinaison

Dans l'exemple du calcul du minimum d'une liste, la présentation textuelle communique plus efficacement l'idée de l'algorithme que le pseudocode car : (i) elle omet des détails sans importance (le sens de parcours du tableau, le nom de la variable intermédiaire), et (ii) elle explicite l'idée de l'algorithme (maintenir la plus petite valeur examinée lors du parcours). Pour d'autres algorithmes, c'est la présentation par pseudocode qui s'avère la plus efficace (par exemple pour l'algorithme de Floyd-Warshall, abordé au Chapitre 5). Les présentations textuelle et par pseudocode ont ainsi chacune leurs avantages.

> Dans ce cours, on décrit un algorithme en combinant présentations par pseudocode et textuelle.

Plus précisément, on autorise à utiliser une description textuelle au sein d'un pseudocode. Cela s'avère pratique pour présenter un algorithme dont certaines parties se prêtent plutôt à une présentation par pseudocodes et d'autres sont plus facilement présentées par texte. C'est par exemple le cas de l'étape de fusion du *tri fusion* (cf Section 3.3) :

```
i=0, j=n/2 et k=0
tant que (i < n/2 et j < n)
   si T[i]<T[j] alors R[k] = T[i] et i = i+1
   sinon R[k] = T[j] et j = j+1
   k = k+1
si i == n/2 alors compléter R avec T[j..n-1]
sinon compléter R avec T[i..n/2-1]
copier R dans T
```

Remarquer que dans ce pseudocode, certaines opérations sont décrites textuellement (par exemple « compléter R avec T[i..n/2-1] », ou encore « copier R dans T »).

A.3 Conventions et recommandations

Concluons par quelques conventions et recommandations relatives à la présentation des algorithmes.

Les tableaux sont le moyen privilégié de représenter les entrées et sorties.

L'organisation en tableau reproduit l'organisation de la mémoire, telle qu'expliquée au Chapitre 2. On décrit un tableau nommé T dont les indices peuvent varier de a à b par $T[a..b]$; le nombre d'éléments d'un tel tableau est donc $b-a+1$. On laisse libre les plages d'indices utilisés par les tableaux [4] : $T[1..n]$ est un tableau de taille n dont les indices commencent à 1, tandis que $A[0..m]$ est un tableau de taille $m+1$ dont les indices commencent à 0. (L'exercice 3 du TD 2 apporte quelques explications sur les raisons de cette liberté.)

Qu'a-t-on le droit d'utiliser comme opération lorsque l'on décrit un algorithme ?

Cette question se pose à la fois pour la présentation par pseudocode (quelles pseudoinstructions sont autorisées ?) et pour la présentation textuelle (à quel niveau de détail faut-il descendre dans la description ?). C'est une question délicate qui n'est pas très éloignée d'une question profonde : qu'est-ce qu'un algorithme ? La réponse est donnée au Chapitre 2 : on peut utiliser toute opération qui est facilement traduisible en opérations élémentaires pour le modèle de calcul considéré. Définir le modèle de calcul et ses opérations élémentaires est non-trivial et occupe l'essentiel de ce chapitre...

Comment déterminer si un détail est important lorsque l'on présente un algorithme ?

Un détail d'un algorithme est important s'il joue un rôle dans le fait que l'algorithme a les propriétés qui le rendent intéressant. On peut par exemple vérifier que dans l'algorithme de Gale-Shapley l'ordre dans lequel, à un round donné, les demandes des élèves sont transmises aux colocations n'est pas important : changer cet ordre n'affecte ni le résultat ni le nombre de phases. Inversement, le fait que chaque colocation garde exactement une demande en attente est important.

Donner des détails non-importants lorsqu'on présente un algorithme peut être pénalisé à l'examen.

Une erreur dans la présentation d'un algorithme est pénalisée, y compris lorsqu'elle porte sur un détail non-important (et quand bien même ne pas donner ce détail ne serait pas pénalisé). Soulignons cependant que ne pas donner un détail important serait là aussi pénalisé. Autrement dit, il convient de donner tous les détails importants et ajouter des détails non-importants augmente inutilement le risque d'erreur.

Par ailleurs, le barême de l'examen alloue un bonus/malus de rédaction afin de prendre en compte la clarté et la présentation. Fournir correctement des détails non-importants peut contribuer à compliquer inutilement la présentation, et donc être pénalisé par ce bonus/malus.

4. En particulier, dans ce cours, les indices d'un tableau peuvent ne pas commencer à 0.

Un algorithme peut utiliser comme instruction tout autre algorithme considéré comme acquis.

Ce peut être parce que cet autre algorithme est un classique (par exemple, le *tri fusion*), ou parce qu'on l'a étudié en cours (par exemple, l'algorithme de Gale-Shapley vu au Chapitre 1), ou parce qu'on l'a étudié en exercice, etc. Ainsi, à mesure que le cours progresse, la panoplie des outils disponibles s'agrandit.

Notes personnelles

Compléments

Les chapitre qui suivent proposent divers compléments qui approfondissent les thèmes abordés par le cours. Ces chapitres sont proposés pour les plus curieux·ses et leur lecture est totalement optionnelle.

Annexe B

Compléments : machines de Turing, indécidabilité et thèse de Church-Turing

Le discours « sur les problèmes futurs des mathématiques » prononcé par Hilbert au congrès international des mathématiques de 1900 présente une liste de 23 questions qui ont profondément influencé le développement des mathématiques au XXème siècle. Les questions No 2 et No 10 marquent aussi des jalons importants en théorie de la calculabilité. La 10ème question est en fait un problème algorithmique :

RÉSOLUTION D'ÉQUATION DIOPHANTIENNE

Entrée : Un polynôme P à coefficients entiers et $n < \infty$ indéterminées.

Sortie : Vrai ou faux, $P(x_1, x_2, \ldots, x_n) = 0$ admet une solution entière.

La 2ème question porte sur la possibilité de prouver que les axiomes de l'arithmétique sont non-contradictoires. Cette question a stimulé la formalisation de systèmes de déduction, notamment la *logique du premier ordre*, et a conduit à la formulation par Hilbert et Ackermann en 1928 d'un *problème de décision* proprement posé :

ENTSCHEIDUNGSPROBLEM

Entrée : Un énoncé de la logique du 1er ordre.

Sortie : La validité (vrai ou faux) de cet énoncé.

Le problème ENTSCHEIDUNGSPROBLEM a été résolu par Turing en 1935 : il a établi qu'il *n'existe pas*[1] d'algorithme qui résout ce problème. La preuve de Turing doit discuter de *l'ensemble des algorithmes possibles*, ce qui requiert de définir précisément ce qu'est un algorithme. Turing fait cela en introduisant ce que l'on appelle aujourd'hui les *machines de Turing.*[2]

B.1 Machine de Turing

La *machine automatique* a été introduite par Turing dans un article fondateur [**?**], et est désormais appelée *machine de Turing*. C'est une abstraction mathématique qui modélise un système *calculant* une fonction de $\{0,1\}^*$ dans $\{0,1\}^*$.[3] Les détails de la définition peuvent varier d'une source à l'autre. Nous suivons ici la présentation du livre d'Arora et Barak [AB09]. (Voir par exemple le livre de Kitaev, Shen et Vyalyi [KSVV02] pour une autre présentation classique.)

1. Comprendre : il est impossible de concevoir un algorithme qui résoudrait ce problème.
2. Quant au 10ème problème de Hilbert, il a lui aussi été prouvé comme indécidable à la fin des années 1960 par Matiyasevich (s'appuyant sur des travaux de Robinson). Mais c'est une autre histoire...
3. Comme discuté en Section **??**, un problème comme ENTSCHEIDUNGSPROBLEM peut se normaliser en un problème algorithmique en mots binaires $P : \{0,1\}^* \to 2^{\{0,1\}^*}$. Une solution à un tel problème prend la forme d'une fonction $f : \{0,1\}^* \to \{0,1\}^*$ telle que $f(w) \in P(w)$ pour tout $w \in \{0,1\}^*$. Turing s'intéresse aux fonctions de ce type dont le calcul peut être automatisé.

Définition formelle

On appelle *alphabet* l'ensemble $\Gamma \stackrel{\text{def}}{=} \{0, 1, \text{␣}\}$, où ␣ est appelé *symbole blanc*. Mathématiquement, une *machine de Turing* est un couple $M = (Q, \delta)$ où

- Q est un ensemble fini, appelé *ensemble des états*, comportant deux éléments distingués q_0 et q_f, appelés respectivement l'*état initial* et l'*état final*,
- δ est une fonction de $Q \times \Gamma$ dans $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$, appelée *fonction de transition*.

Interprétation

Une machine de Turing modélise un système constitué de deux parties, un **ruban** et un **processeur**.[4] Le ruban est une succession infinie de cases, indexées par les entiers naturels ($\mathbb{N}$) ; chaque case contient un symbole de l'alphabet Γ, le symbole blanc étant le seul autorisé à être contenu dans une infinité de cases. Le processeur est caractérisé par un état, *i.e.* un élément de Q, et l'indice d'une case du ruban ; on dit que le processeur est *positionné* sur cette case.

Déroulement d'un calcul

Un calcul **commence** avec le processeur dans l'état initial q_0 et positionné sur la première case du ruban. Le contenu du ruban avant le premier pas de calcul est libre, sous réserve que ␣ soit le seul symbole occupant une infinité de cases, et est appelé *entrée* du calcul.

Le calcul se déroule en pas discrets. Le déroulement d'un pas de calcul est déterminé par l'état e dans lequel se trouve le processeur et par le symbole s contenu dans la case sur laquelle il est positionné. Si on note $(e', s', D) \stackrel{\text{def}}{=} \delta(e, s)$ l'image de (e, s) par la fonction de transition δ, le pas de calcul consiste à

(i) écrire s' dans la case sur laquelle le processeur est positionné, puis

(ii) à déplacer la position du processeur sur le ruban en la diminuant (si $D = \leftarrow$) ou en l'augmentant (si $D = \rightarrow$), et enfin

(iii) à définir l'état interne du processeur comme valant désormais e'.

Le pas de calcul se termine une fois ces trois opérations réalisées.

Si à l'issu d'un pas de calcul le processeur se trouve dans l'état final q_f, alors le calcul **termine**. Sinon, un nouveau pas de calcul commence. Le contenu du ruban au moment où le calcul termine est appelé *sortie* du calcul. (Si le calcul ne termine pas il n'a pas de sortie.)

Fonction calculée

Une machine de Turing **calcule** une fonction $f : \{0,1\}^* \to \{0,1\}^*$ si pour tout mot binaire w, lorsque le calcul est initié avec w comme entrée, il termine après un nombre fini de pas de calcul et a pour sortie $f(w)$.

Une fonction $f : \{0,1\}^* \to \{0,1\}^*$ est **calculable** s'il existe une machine de Turing qui la calcule.

Une machine de Turing **résout un problème algorithmique** P si elle calcule une fonction f telle que $f(w) \in P(w)$ pour tout $w \in \{0,1\}^*$.

4. Le processeur était conçu comme un homme dans le texte original de Turing, mais cela peut aussi être un système physique.

B.2 Thèse de Church-Turing

Si on sait décrire une machine de Turing qui calcule une fonction f, alors on sait donner un ensemble de règles élémentaires pour passer de tout mot binaire w au mot binaire $f(w)$. Ces règles ne demandent aucune prise d'initiative et de nombreux systèmes physiques permettent aujourd'hui de les exécuter automatiquement. La **thèse de Church-Turing** affirme que la réciproque est vraie :

> Si une fonction $f : \{0,1\}^* \to \{0,1\}^*$ peut être évaluée par une « méthode effective », alors cette fonction est calculable par une machine de Turing.

Ici, « méthode effective » est à prendre au même sens, assez large, que dans l'énoncé par Hilbert de son 10ème problème : un procédé suffisamment décomposé pour qu'aucune étape ne requière la moindre initiative.

La thèse de Church énonce essentiellement que le formalisme des machines de Turing décrit « ce qui est calculable ». Corollaire immédiat : toute fonction qui ne serait pas calculable par une machine de Turing **ne serait tout simplement pas calculable**. Cette thèse est à envisager comme une loi fondamentale du calcul, du même type que la loi de conservation de l'énergie en physique. Elle a été vérifiée par tous les systèmes de calcul que l'on a su construire jusqu'ici. Sa remise en question, sans être impossible, préfigurerait une évolution profonde de notre compréhension de ce que signifie *calculer*.

B.3 Machine de Turing universelle et indécidabilité de l'arrêt

Il peut sembler surprenant qu'il existe des fonctions qu'aucune machine ne Turing ne puisse calculer. Nous allons maintenant en esquisser [5] un exemple aux conséquences spectaculaires.

Il est possible d'encoder une machine de Turing par un simple mot binaire. [6] On peut, plus systématiquement, associer à tout mot binaire w une machine de Turing M_w. Naturellement, ce mot w qui encode la machine M_w peut tout à fait servir de mot d'entrée à un calcul mené sur une *autre* machine de Turing M'. [7] On peut dès lors envisager le calcul de fonctions dont l'entrée *s'interprête* comme une machine de Turing. [8] La fonction suivante est particulièrement intéressante :

$$H : (w, e) \mapsto \begin{cases} 1 & \text{si le calcul de } M_w \text{ sur l'entrée } e \text{ termine} \\ & \text{après un nombre fini de pas de calculs,} \\ 0 & \text{sinon.} \end{cases}$$

Le *problème de l'arrêt* consiste à déterminer une machine de Turing qui calcule H. Le théorème suivant est dû à Church et à Turing indépendamment :

Théorème B.1. *La fonction H n'est pas calculable.*

Explicitons une conséquence pratique du Théorème B.1. Un langage de programmation est dit *Turing-complet* si toute fonction calculable par une machine de Turing peut être calculée par un programme dans ce langage. (La réciproque va de soi d'après la thèse de Church.) De très nombreux langages sont Turing-complets, par exemple `Python`. [9] L'indécidabilité du problème de l'arrêt signifie donc qu'il est impossible à un programme d'examiner le code source d'un programme python et de décider si son exécution va terminer en un temps fini ou boucler indéfiniment. Cela ne laisse que peu d'espoir de traiter systématiquement les questions plus sophistiquées d'analyse de code [10].

5. Pour un traitement plus détaillé de ces notions, se reporter à [AB09, §1.4].

6. On a vu qu'il suffit d'encoder une machine de Turing par une séquence d'entiers. On peut encoder l'ensemble des états par trois nombres (a, b, c) via l'identification $Q \simeq \{1, 2, \ldots, a\}$ avec $a = |Q|$; les entiers b et c sont ici les numéros des états initiaux et finaux. Une fois cela fait, on définit une convention d'énumération de $Q \times \Gamma$, et on encode les images par δ dans cet ordre.

7. Cela ne devrait pas plus vous surprendre que le fait que sur un ordinateur moderne, on peut écrire un programme dans un fichier, puis fournir ce fichier comme donnée à un autre programme.

8. Une construction importante est la *machine de Turing universelle* [?], qui prend en entrée un couple de mots binaires (w, e) et calcule *la sortie calculée par la machine M_w sur l'entrée e*. (Rappelez-vous qu'on a expliqué comment encoder sans ambiguité une concaténation.) Autrement dit, cette machine simule toute autre machine de Turing à partir de son encodage. Un précurseur de l'ordinateur programmable !

9. Il devrait être facile de se convaincre que l'on peut écrire un simulateur de machine de Turing en `python`. D'autres systèmes Turing-complets sont plus exotiques, par exemple les donjons du jeu `Dwarf fortress`.

10. Par exemple : le programme donné en entrée est-il un virus ?

B.4 Et les langages dans tout ça ?

La théorie de la calculabilité est parfois abordée sous l'angle de « langages » qu'il s'agit de « reconnaître ». Précisons comment cela s'articule avec le formalisme donné ci-dessus. Lorsque l'on travaille sur un problème de décision D, il est naturel d'oublier la fonction D et de simplement s'intéresser à l'ensemble $D^{-1}(\{1\}) = \{w \in \{0,1\}^* : D(w) = 1\}$ des « entrées acceptées ». Cet ensemble est ce que l'on appelle le langage associé à D. Formellement, un **langage** est un ensemble de mots, c'est-à-dire un sous-ensemble de $\{0,1\}^*$. Le problème de décision associé à un langage revient à décider, ou *reconnaître*, si un mot donné en entrée appartient ou pas au langage.

> La théorie de la calculabilité étudie les langages $L \subset \{0,1\}^*$ pour lesquels l'appartenance peut être décidée par un algorithme.

D'une certaine manière, la théorie de la calculabilité s'efforce d'identifier de la structure dans l'ensemble des langages.

Annexe C

Introduction aux structures de données et à leur analyse

C.1 Définitions : type abstrait et structure de donnée

Un **type abstrait** est une description mathématique d'une organisation d'un ensemble de données et d'opérations qui peuvent être effectuées sur cette organisation. Un type abstrait décrit le comportement (on dit la *sémantique*) de l'accès aux données, du point de vue de l'utilisateur. Autrement dit, c'est la description d'une interface d'accès à des données.

Une **structure de donnée** décrit une réalisation d'un type abstrait dans un modèle de calcul. La structure de donnée spécifie l'organisation des données en mémoire ainsi que les algorithmes réalisant chacune des opérations supportées par le type abstrait. La relation entre *type abstrait* et *structure de donnée* est donc similaire à celle entre *problème algorithmique* et *algorithme*.

C.2 Premiers exemples : tableau et liste chaînée

Voyons deux exemples simples de types abstraits et de structures de données associées.

Tableau. Le type abstrait *tableau* représente une séquence de N mots binaires, où N est fixé. Les opérations supportées sont la lecture (qui, étant donné un entier $1 \leq i \leq N$, retourne le $i$ème mot de la séquence) et l'écriture (qui, étant donné un entier $1 \leq i \leq N$ et un mot binaire w, définit le $i$ème mot de la séquence comme valant w).

On peut réaliser un tableau en stockant l'information dans n cases mémoire consécutives, réservées à la création du tableau et libérées à sa destruction. Si on note t l'adresse de la première de ces cases mémoire, pour lire le $i$ème mot de la séquence on lit la valeur contenue à la case mémoire d'adresse $t + i - 1$; de même, pour écrire w dans le $i$ème mot de la séquence, on écrit w à la case mémoire d'adresse $t + i - 1$.

Liste. Le type abstrait *liste* représente une *séquence finie* de mots binaires, de longueur variable. Les opérations supportées incluent généralement [1] l'ajout (qui, étant donné un mot binaire w, ajoute ce mot en fin de séquence), la suppression (qui, étant donné un entier i, supprime le $i$ème mot de la séquence, le mot anciennement en position $i + 1$ se retrouvant ainsi en position i, etc.), la lecture (qui, étant donné un entier i, retourne le $i$ème mot de la séquence), et l'écriture (qui, étant donné un entier i et un mot binaire w, change le $i$ème mot de la séquence en w).

On peut réaliser une liste par une structure de données appelée *liste chainée*, définie comme suit :

- La liste chaînée est constituée de maillons, chaque maillon étant formé de deux cases mémoire consécutives. Notons a_i l'adresse de la première case du $i$ème maillon. La case mémoire d'adresse a_i contient le mot binaire en $i$ème position. La case mémoire d'adresse $a_i + 1$ contient a_{i+1}, autrement dit l'adresse mémoire du maillon suivant. On peut indiquer qu'il n'y a pas de maillon suivant en y écrivant par exemple -1 ou toute autre valeur spéciale fixée par convention.

1. La liste complète des opérations supportées par une liste varie un peu selon les sources.

- Pour ajouter un mot binaire w en fin de liste, on commence par trouver l'adresse a_{fin} du dernier maillon (par exemple en partant du premier maillon et en passant au maillon suivant jusqu'à trouver -1). On réserve ensuite deux nouvelles cases mémoire consécutives, disons à l'adresse $a_{nouveau}$. On écrit w à l'adresse $a_{nouveau}$, on écrit $a_{nouveau}$ à l'adresse $a_{fin}+1$, et on écrit -1 à l'adresse $a_{nouveau}+1$

Les autres opérations peuvent se faire de manière similaire

Dans les structures de données définies ci-dessus, le nombre d'éléments contenus dans un tableau est fixée à sa création et ne peut pas être augmenté (la case mémoire d'adresse $t+n$ peut ne pas être disponible). En revanche, le nombre d'éléments contenus dans une liste chaînée varie au fil des opérations, sans limitation autre que la mémoire globalement disponible (qui n'est pas limitée dans le modèle RAM). La contrepartie de cette flexibilité est que certaines opérations (par exemple la lecture) sont plus coûteuses dans une liste que dans un tableau. Pour discuter cela, il faut préciser le modèle d'analyse de complexité que l'on considère...

C.3 Analyse de complexité d'une structure de donnée

Comme pour les algorithmes, on peut définir un modèle d'analyse asymptotique pire-cas pour une structure de données. Ce modèle diffère de l'analyse d'algorithmes classiques en deux points.

D'une part, on donne une fonction de complexité par algorithme, c'est-à-dire pour chacune des opérations que supporte le type abstrait réalisé par la structure de données. Ainsi, dans le cas d'une structure de donnée réalisant un tableau, on a deux fonctions de complexité : une pour la lecture et une pour l'écriture.

D'autre part, et c'est le point important, le temps pris par un algorithme sur une entrée donnée peut dépendre non pas de la seule entrée de cet algorithme, mais de la *la séquence complète* d'opérations faites *avant* l'opération analysée. Dans l'exemple de réalisation d'une liste donnée ci-dessus, le temps mis pour *ajouter* un mot binaire w à la liste dépend principalement du nombre d'éléments déjà contenus dans la liste.

Formellement, on définit donc le **modèle d'analyse asymptotique pire cas** d'une structure de donnée comme suit :

- On choisit un type d'opération supporté par le type abstrait, et on note $\mathcal{A}$ l'algorithme de la structure de données qui réalise ce type d'opérations.
- On considère ensuite une *séquence finie* S d'opérations dont la dernière, notons la s_{fin}, est du type choisi. Notons S' la séquence obtenue en supprimant s_{fin} de S.
- On définit la *taille* de S comme l'espace mémoire occupé par la structure une fois traitée S'. On définit $C_{\mathcal{A}}(S)$ comme le nombre d'opérations élémentaires de calcul faites par $\mathcal{A}$ pour traiter s_{fin} après avoir traité S'.
- Pour tout entier n, on définit ensuite $C_{\mathcal{A}}(n)$ comme le maximum de $C_{\mathcal{A}}(S)$ pour toutes les séquences S de requêtes ont la taille vaut n et qui terminent par une opération du type choisi. On ne retient de $C_{\mathcal{A}}(n)$ que son comportement asymptotique.

C.4 Types de données d'un langage de programmation

La description des types de données proposés par un langage de programmation (par exemple `python`) présente géneralement le type abstrait, puisque c'est « l'interface programmeur ». Pour connaître la complexité des différentes opérations supportées, il faut examiner de quelle manière ces types abstraits sont implémentés, autrement dit par quelles *structures de données* ils sont réalisés.

Par exemple, le type `list` en python est réalisé[2] par des tableaux de taille fixe ; lorsque l'ajout d'un nouvel élément « déborde » du tableau utilisé, `python` réserve un nouveau tableau plus grand.

2. cf https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython

Ainsi, si l'ajout du $n$ème élément prend souvent $O(1)$ opérations élémentaires de calcul, la complexité pire-cas est $O(n)\ldots$

Terminons par un point sur la complexité des *tables de hachage*, qui est souvent source de confusion. Tout d'abord, une *table de hachage* est plutôt un type abstrait (souvent appelé *ensemble*) tandis que la complexité est associée à une structure de donnée implémentant ce type abstrait. Or il existe de nombreuses manières fort différentes d'implémenter une table de hachage : table de hachage chaînée, table de hachage à adressage ouvert linéaire, hachage coucou, ... Ces implémentations diffèrent essentiellement sur la manière dont elles gèrent les collisions. De ce point de vue, le principe de tiroirs de Dirichlet est impitoyable : toutes ces structures de données ont des complexité d'insertion, de suppression et d'accès $O(n)$.[3]

3. Quant à l'approche consistant à rehacher dès qu'une collision se présente, elle permet effectivement de garantir suppression et accès en temps $O(1)$, mais l'algorithme d'ajout est de complexité indéfinie : l'ajout d'un élément peut prendre un temps arbitrairement grand !

Annexe D

Preuve du théorème d'Akra-Bazzi

L'esquisse de preuve du théorème maître donnée au Chapitre 4 est formellement incomplète, ne serait-ce que parce que les arrondis et les parties entières ne sont pas traités soigneusement. Plutôt que de les compléter, nous proposons ici une preuve du théorème d'Akra-Bazzi, que l'on rappelle :

Théorème D.1. *Soit $k \geq 1$ et n_0 des entiers positifs, $\alpha_1, \alpha_2, \ldots, \alpha_k$ des réels tels que $1 > \alpha_i > 0$, et $h_1, h_2, \ldots, h_k$ des fonctions. On suppose qu'il existe $\epsilon > 0$ tel que $h_i(n) \leq \frac{n}{\log^{1+\epsilon} n}$ pour $n \geq n_0$. Notons p le réel tel que $\sum_{i=1}^{k} {\alpha_i}^p = 1$.*

Si un algorithme $\mathcal{A}$ traite une entrée de taille $n \geq n_0$ par des appels récursifs à k sous-entrées, de taille respectives $\alpha_i n + h_i(n)$ pour $1 \leq i \leq k$, alors

$$C_{\mathcal{A}}(n) \leq \begin{cases} O\left(n^p\right) & \text{si } g_{\mathcal{A}}(n) = O\left(n^{p-\epsilon}\right) \text{ pour une constante } \epsilon > 0, \\ O\left(g_{\mathcal{A}}(n) \log n\right) & \text{si } g_{\mathcal{A}}(n) = \tilde{O}\left(n^p\right), \\ O\left(g_{\mathcal{A}}(n)\right) & \text{sinon.} \end{cases}$$

L'indice p existe et est unique (nous y reviendrons).[1] La preuve du Théorème d'Akra-Bazzi est un solide exercice d'analyse réelle. On considère cette preuve comme sortant du cadre de ce cours, mais on la présente néanmoins ci-après (par étapes) pour les étudiant.e.s souhaitant disposer d'un outil d'analyse d'algorithme puissant et rigoureux.

D.1 Détour par une récurrence réelle exacte

Le cœur de la preuve d'Akra et Bazzi est la résolution d'une récurrence *exacte* sur une fonction *d'une variable réelle*. Considérons une fonction $f : [1, \infty) \to \mathbb{R}$ définie par une récurrence de la forme

$$f(x) = \begin{cases} h(x) & \text{pour } 1 \leq x \leq x_0 \\ \displaystyle\sum_{i=1}^{k} f\left(\alpha_i x + h_i(x)\right) + g(x) & \text{pour } x > x_0, \end{cases} \tag{D.1}$$

dont nous allons préciser les paramètres (x_0, g, h, k, α_i, et h_i) au fil de l'analyse. Sans surprise, on suppose

(a) k est un entier positif et $1 > \alpha_i > 0$ pour $1 \leq i \leq k$.

On note p l'unique réel positif ou nul tel que $\sum_{i=1}^{k} {\alpha_i}^p = 1$. L'existence de p découle de l'application du théorème des valeurs intermédiaires à la fonction $\phi : t \mapsto \sum_{i=1}^{k} {\alpha_i}^t$, qui est continue, tend vers k pour $t \to 0$ et vers 0 pour $t \to \infty$. L'unicité de p découle du fait que ϕ est strictement décroissante.

1. Remarquons que $p \leq 1$ dès que $\alpha_1 + \alpha_2 + \ldots + \alpha_k \leq 1$, c'est-à-dire que la somme des tailles des appels récursifs n'excède pas sensiblement la taille de l'entrée.

D.2 Mise en place d'une récurrence

Nous allons prouver qu'il existe des constantes positives c_5 et c_6 telles que pour tout $x > x_0$,

$$c_5 x^p \left(1 + \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u\right) \leq f(x) \leq c_6 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u\right). \quad \text{(D.2)}$$

On divise $[1,\infty)$ en intervalles en posant $I_0 \stackrel{\text{def}}{=} [1, x_0]$ et $I_j \stackrel{\text{def}}{=} (x_0 + j - 1, x_0 + j]$. Nous allons prouver l'encadrement (D.2) sur I_j pour tout $j \geq 1$, par récurrence sur j ; soulignons que les constantes c_5 et c_6 doivent être les mêmes pour tous les intervalles. On suppose les conditions suivantes vérifiées :

(b) $\alpha_i + \frac{1}{\log^{1+\epsilon} x_0} < 1$

(c) $\left(1 - \alpha_i - \frac{1}{\log^{1+\epsilon} x_0}\right) x_0 > 1$

(d) il existe $\epsilon > 0$ tel que $|h_i(x)| \leq \frac{x}{\log^{1+\epsilon} x}$ pour $x \geq x_0$ et $k \geq i \geq 1$.

Ces conditions vont nous permettre d'utiliser la relation (D.1) pour propager des encadrements de $\cup_{j'<j} I_{j'}$ à I_j :

Lemme D.2. *Sous les conditions (b), (c) et (d), pour tous $j \geq 1$ et $x \in I_j$, on a $\alpha_i x + h_i(x) \in I_{j'}$ avec $j' < j$.*

Démonstration. Les conditions posées assurent que pour tous $j \geq 1$ et $x \in I_j$,

$$\begin{aligned}
\alpha_i x + h_i(x) &\leq \left(\alpha_i + \frac{1}{\log^{1+\epsilon} x}\right) x \\
&\leq \left(\alpha_i + \frac{1}{\log^{1+\epsilon}(x_0 + j)}\right)(x_0 + j) \\
&\leq \underbrace{\left(\alpha_i + \frac{1}{\log^{1+\epsilon}(x_0 + j)}\right)}_{<1 \text{ par (b) et croissance de log}} j + x_0 - \underbrace{\left(1 - \alpha_i - \frac{1}{\log^{1+\epsilon}(x_0 + j)}\right)}_{>1 \text{ par (c) et croissance de log}} x_0 < j + x_0 - 1,
\end{aligned}$$

soit $\alpha_i x + h_i(x) < \inf I_j$. □

D.3 Fonctions à croissance polynomiale

Avant d'entreprendre la preuve de l'encadrement (D.2), établissons un préliminaire technique. Une fonction $g : \mathbb{R}^+ \to \mathbb{R}$ est *à croissance polynomiale* si elle est à valeurs positives et que pour tout réel $\alpha \in (0,1)$ il existe des réels c_1, c_2 strictement positifs tels que

$$\forall x \geq 1, \forall u \in [\alpha x, x], \qquad c_1 \cdot g(x) \leq g(u) \leq c_2 \cdot g(x).$$

Remarquons que toute fonction de la forme $\Theta\left(x^v \log^w x\right)$ est à croissance polynomiale, pour tous $v, w \in \mathbb{R}$.

Lemme D.3. *Pour tout réel $t \geq 0$ et toute fonction g à croissance polynomiale, il existe des constantes c_3 et c_4 telles que*

$$\forall 1 \leq i \leq k, \quad \forall x \geq 1, \qquad c_3 g(x) \leq x^t \int_{\alpha_i x + h_i(x)}^x \frac{g(u)}{u^{t+1}}\mathrm{d}u \leq c_4 g(x).$$

Démonstration. Remarquons qu'il suffit d'établir l'encadrement pour $x \geq x_0$, puisque la fonction

$$x \mapsto \frac{x^t}{g(x)} \int_{\alpha_i x + h_i(x)}^x \frac{g(u)}{u^{t+1}}\mathrm{d}u$$

envoie l'intervalle $[1, x_0]$ sur un intervalle $[{c_3}', {c_4}']$ avec $0 < {c_3}' \leq {c_4}' < \infty$.

Les conditions (b) et (d) assurent qu'il existe $1 > \beta_1 > \beta_0 > 0$ tels que pour tout $1 \leq i \leq k$ et pour tout $x \geq x_0$ on ait $\beta_0 \leq \alpha_i + \frac{h_i(x)}{x} \leq \beta_1$. Puisque g est à valeurs positives, on a alors

$$\forall 1 \leq i \leq k, \quad \forall x \geq x_0, \qquad \int_{\beta_0 x}^x \frac{g(u)}{u^{t+1}}\mathrm{d}u \leq \int_{\alpha_i x + h_i(x)}^x \frac{g(u)}{u^{t+1}}\mathrm{d}u \leq \int_{\beta_1 x}^x \frac{g(u)}{u^{t+1}}\mathrm{d}u.$$

Puisque g est à croissance polynomiale, il existe c_1 et c_2 tels que

$$\forall x \geq x_0, \forall u \in [\beta_0 x, x], \qquad c_1 g(x) \leq g(u) \leq c_2 g(x).$$

Ainsi,

$$\forall x \geq x_0, \qquad c_1 g(x) \int_{\beta_1 x}^{x} \frac{1}{u^{t+1}} \mathrm{d}u \leq \int_{\alpha x}^{x} \frac{g(u)}{u^{t+1}} \mathrm{d}u \leq c_2 g(x) \int_{\beta_0 x}^{x} \frac{1}{u^{t+1}} \mathrm{d}u.$$

Le résultat annoncé s'en suit par simple intégration. □

D.4 Initialisation de la récurrence

Pour assurer l'encadrement (D.2) sur I_1, nous supposons que les conditions suivantes sont vérifiées :

(e) g est à croissance polynomiale,
(f) $\log^{\epsilon/2} x_0 \geq 2$
(g) Il existe $0 < a_1 \leq a_2 < \infty$ telles que $h(I_0) \subseteq [a_1, a_2]$.

D'une part cela assure que $f(I_1) \subseteq [ka_1 + c_1 g(x_0), ka_2 + c_2 g(x_0+1)]$. D'autre part, pour $x_0 \leq x \leq x_0+1$, les expressions

$$\left(1 + \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} \mathrm{d}u\right) \quad \text{et} \quad \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} \mathrm{d}u\right)$$

sont à valeurs dans un compact de $(0, \infty)$. Cela assure donc que l'encadrement est vrai pour tout c_5 suffisamment petit et c_6 suffisamment grand. Nous fixerons ces constantes ultérieurement car elles seront soumises à une autre contrainte chacune.

D.5 Induction

Pour assurer la propagation de l'encadrement il nous reste à formuler deux conditions. On suppose x_0 suffisamment grand pour que

(h) $\forall x \geq x_0, \left(1 + \frac{1}{\alpha_i \log^{1+\epsilon} x}\right)^p \left(1 - \frac{1}{\log^{\epsilon/2}(\alpha_i x + h_i(x))}\right) \leq 1 - \frac{1}{\log^{\epsilon/2} x}$, et

(i) $\forall x \geq x_0, \left(1 - \frac{1}{\alpha_i \log^{1+\epsilon} x}\right)^p \left(1 + \frac{1}{\log^{\epsilon/2}(\alpha_i x + h_i(x))}\right) \geq 1 + \frac{1}{\log^{\epsilon/2} x}$.

(On laisse en exercice de vérifier que (h) et (i) sont vérifiées pour tout x_0 suffisamment grand.)

Fixons maintenant $j \geq 2$ et supposons l'encadrement (D.2) vérifié sur tout intervalle $I_{j'}$ avec $j' < j$. En utilisant la définition de f et l'hypothèse de récurrence, on obtient donc pour tout $x \in I_j$,

$$\begin{aligned}
f(x) &= \sum_{i=1}^{k} f\left(\alpha_i x + h_i(x)\right) + g(x) \\
&\leq \sum_{i=1}^{k} c_6 \left(\alpha_i x + h_i(x)\right)^p \left(1 - \frac{1}{\log^{\epsilon/2}\left(\alpha_i x + h_i(x)\right)}\right)\left(1 + \int_1^{\alpha_i x + h_i(x)} \frac{g(u)}{u^{p+1}} \mathrm{d}u\right) + g(x) \\
&\leq c_6 \sum_{i=1}^{k} {\alpha_i}^p x^p \left(1 + \frac{1}{\alpha_i \log^{1+\epsilon} x}\right)^p \left(1 - \frac{1}{\log^{\epsilon/2}\left(\alpha_i x + h_i(x)\right)}\right)\left(1 + \int_1^{\alpha_i x + h_i(x)} \frac{g(u)}{u^{p+1}} \mathrm{d}u\right) + g(x)
\end{aligned}$$

cette dernière inégalité utilisant la condition (d). D'après le Lemme D.3, on a

$$\begin{aligned}
\int_1^{\alpha_i x + h_i(x)} \frac{g(u)}{u^{p+1}} \mathrm{d}u &= \int_1^x \frac{g(u)}{u^{p+1}} \mathrm{d}u - \int_{\alpha_i x + h_i(x)}^x \frac{g(u)}{u^{p+1}} \mathrm{d}u \\
&\leq \int_1^x \frac{g(u)}{u^{p+1}} \mathrm{d}u - c_3 g(x).
\end{aligned}$$

et en utilisant la condition (h), on obtient

$$f(x) \leq c_6 \sum_{i=1}^{k} {\alpha_i}^p x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u - c_3 g(x)\right) + g(x)$$
$$= c_6 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u - c_3 g(x)\right) + g(x),$$

la seconde égalité résultant du fait que dans la somme intermédiaire, seul α_i dépend de i et que $\sum_{i=1}^{k} {\alpha_i}^p = 1$. Cela se réécrit en

$$f(x) \leq c_6 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u\right) + \left(1 - c_6 c_3 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\right) g(x).$$

D'après (f), il suffit que $2c_3c_6 > 1$ pour s'assurer que le dernier terme soit négatif, et que l'on ait

$$f(x) \leq c_6 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u\right)$$

comme annoncé. Le raisonnement pour la minoration est similaire et utilise la condition (i) à la place de la condition (h) ; il conduit à imposer $2c_4c_5 < 1$. Cela conclut la preuve de l'encadrement (D.2).

D.6 Retour au Théorème D.1

Il reste à déduire le théorème d'Akra-Bazzi de l'encadrement (D.2). Considérons donc un algorithme $\mathcal{A}$ satisfaisant les hypothèses du Théorème D.1. Soit $f : [1, \infty] \to \mathbb{R}$ la fonction d'une variable réelle définie par $f(x) \stackrel{\text{def}}{=} C_{\mathcal{A}}(\lceil x \rceil)$. La fonction f satisfait la récursion (D.1) en posant $g(x) \stackrel{\text{def}}{=} g_{\mathcal{A}}(\lceil x \rceil)$, en prolongeant chaque h_i de $\mathbb{N}$ dans $\mathbb{R}$ par $h_i(x) \stackrel{\text{def}}{=} h_i(\lceil x \rceil)$ et en prenant $x_0 \geq n_0$. Remarquons que g est bien à croissance polynomiale. De plus. Pour x_0 suffisamment grand, les conditions (a)–(i) sont satisfaites et l'encadrement (D.2) assure que

$$f(x) = \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}}\mathrm{d}u\right)\right).$$

Si $g_{\mathcal{A}}(n) = O(n^{p-\epsilon})$ pour $\epsilon > 0$, on a alors $g(x) = O(x^{p-\epsilon})$ et

$$f(x) = O\left(x^p \left(1 + \int_1^x \frac{1}{u^{1+\epsilon}}\mathrm{d}u\right)\right) = O\left(x^p\right) + o(1).$$

Ainsi, $f(x) = O(x^p)$ et $C_{\mathcal{A}}(n) = O(n^p)$. Les deux autres cas de figure sur la fonction $g_{\mathcal{A}}$ se traitent de manière analogue.

Annexe E

Machines de Turing non-déterministe et classe NP

Revenons sur la manière dont les questions de complexité traitées au Chapitre 8 se rattachent aux modèles des machines de Turing introduites en Complément B.

E.1 Équivalence entre log-RAM et machine de Turing déterministe

Le modèle *log-RAM* est la variante du modèle word-RAM dans lequel ce que peut contenir une case mémoire dépend de la taille de l'entrée traitée : lors du traitement d'une entrée de taille n, chaque case mémoire peut contenir un mot binaire de taille $O(\log n)$. On utilise le modèle log-RAM dans ce cours en raison de la propriété suivante :

Lemme E.1. *La machine de Turing et le modèle RAM 8 bits peuvent se simuler l'un l'autre en temps polynomial.*

Donner une preuve détaillée de ce résultat n'est pas difficile mais s'avère (très) laborieux. On se contente donc d'en donner les grandes lignes.

Ébauche de preuve. Il est facile de concevoir un algorithme dans le modèle log-RAM qui prend en entrée un encodage d'une machine de Turing et d'une de ses entrées, et simule chaque pas du calcul de la machine de Turing sur cette entrée via $O(1)$ opérations élémentaires.

Réciproquement, pour tout algorithme du modèle log-RAM, il existe une machine de Turing qui prend en entrée un encodage de cet algorithme et une de ses entrées et simule chaque opération élémentaire via un nombre polynomial de pas de calcul. Cela se fait en deux étapes : on montre que l'on peut simuler tout circuit combinatoire de taille constante par une machine de Turing, puis on montre que l'on peut simuler l'unité mémoire grace au ruban. Ce deuxième point cache deux subtilités. D'une part, une case du ruban ne peut stocker qu'un bit d'information (0 ou 1, voire rien) donc il faut plusieurs cases du ruban, $O(\log n)$ en l'occurrence, pour simuler une case mémoire. D'autre part, l'accès à une case mémoire d'adresse donnée se fait en temps constant dans le modèle log-RAM, mais requiert de repositionner la tête de lecture dans le modèle de machine de Turing. On peut majorer le nombre de pas de déplacement de la tête par le nombre d'informations stockées en mémoire par l'algorithme simulé, qui est au plus le nombre d'opérations élémentaires. Il devrait être clair qu'une telle simulation n'est pas possible pour le modèle RAM, d'où l'usage du modèle log-RAM pour définir la classe P. □

E.2 Machine de Turing non-déterministe

Spécialisons maintenant la définition d'une *machine de Turing* vue en Section B.1 au cas des problèmes de décision. On appelle *alphabet* l'ensemble $\Gamma \stackrel{\text{def}}{=} \{0, 1, \textvisiblespace\}$, où $\textvisiblespace$ est appelé *symbole blanc*. Mathématiquement, une *machine de Turing* est un couple $M = (Q, \delta)$ où

- Q est un ensemble fini, appelé *ensemble des états*, comportant deux éléments distingués q_0 et q_f, appelés respectivement l'*état initial* et l'*état final*,

- δ est une fonction de $Q \times \Gamma$ dans $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$, appelée *fonction de transition*.

Le calcul commence avec le processeur dans l'état initial q_0 et positionné sur la première case du ruban, lequel contient l'entrée. Le déroulement d'un pas de calcul est *complètement déterminé* par l'état courant $e \in Q$ du processeur et le symbole s contenu dans la case du ruban sur laquelle le processeur est positionné : on lit $(e', s', D) \stackrel{\text{def}}{=} \delta(e, s)$, on écrit s' dans la case sur laquelle le processeur est positionné, on déplace la position du processeur sur le ruban en la diminuant (si $D = \leftarrow$) ou en l'augmentant (si $D = \rightarrow$), et on définit l'état interne du processeur comme valant désormais e'. Une telle machine est dite **déterministe** pour cette raison. Si le calcul termine et que la première case du ruban contient 1, l'entrée est dite *acceptée*, c'est-à-dire que la décision est `vrai`.

La **machine de Turing non-déterministe** se distingue par le fait qu'elle dispose de *deux* fonctions de transition. Formellement, c'est un triplet $M = (Q, \delta_0, \delta_1)$ où

- Q est un ensemble fini, appelé *ensemble des états*, comportant deux éléments distingués q_0 et q_f, appelés respectivement l'*état initial* et l'*état final*,
- δ_0 et δ_1 sont deux fonctions de $Q \times \Gamma$ dans $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$, appelées *fonctions de transition*.

Le calcul commence avec le processeur dans l'état initial q_0 et positionné sur la première case du ruban. Le déroulement d'un pas de calcul commence par le choix, arbitraire, d'une des deux fonctions de transition (δ_0 ou δ_1), puis l'application de cette fonction de transition comme dans le modèle déterministe. L'entrée est acceptée **s'il existe** une séquence de choix pour lesquels le calcul termine avec la première case du ruban contenant 1. Si **chaque** séquence de choix conduit soit à un calcul sans fin, soit à un calcul terminant avec la première case du ruban contenant 0, alors l'entrée est rejetée. Autrement dit, on peut envisager une machine de Turing non-déterministe comme explorant *simultanément* tous les choix possibles.

Les machines de Turing déterministes sont des cas particuliers de machines de Turing non-déterministe (il suffit de prendre $\delta_0 = \delta_1 = \delta$). Il s'avère que si un langage peut être reconnu par une machine de Turing non-déterministe, alors il existe aussi une machine de Turing déterministe qui le reconnait. Autrement dit, du point de vue de la seule *calculabilité*, les machines de Turing déterministes sont équivalentes aux non-déterministes.

E.3 Classe NP et machines de Turing non-déterministes

La classe NP peut se définir comme l'ensemble des problèmes de décision qu'il est possible de résoudre en temps polynomial par une machine de Turing non-déterministe. [1] Ici, la complexité en temps mesure le nombre de pas de calculs que fait la machine de Turing avant de terminer. Avant de travailler avec cette classe, reformulons la de manière plus pratique.

Les définitions de la classe NP par certificats (que l'on vient de donner) et par machines de Turing non-déterministes (que l'on a esquissé à la section précédente) sont en fait équivalentes. L'idée clef pour voir cela consiste à décrire les calculs de longueur ℓ possibles sur une machine de Turing non-déterministe par les mots binaires de longueur ℓ : le $k$ème bit du mot binaire décrit simplement la fonction, δ_0 ou δ_1, qu'il appliquée au $k$ème pas de calcul. Ainsi, un certificat décrit simplement la séquence de choix à faire pour dérouler un calcul acceptant.

1. NP est l'acronyme de Non-deterministic Polynomial.