Algorithmes et Complexité

Responsable du cours : Xavier Goaoc

FICM 1A Tronc commun scientifique 2022–2023

Mode d'emploi

On peut définir le **calcul** comme le traitement de l'information par un système physique. Cette définition soulève naturellement plusieurs questions :

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, de ses questionnements actuels (sécurité des systèmes, consommation énergétique, apprentissage automatique, informatique quantique, ...) aux changements profonds à venir.

Ce cours aborde deux principaux domaines : l'**algorithmique**, entendue comme la méthodologie de l'organisation efficace du calcul, et la **théorie de la complexité**, qui étudie les calculs qu'il est impossible d'organiser efficacement. Il touche aussi, plus ponctuellement, aux *mathématiques discrètes*, à la *théorie de la calculabilité* et à l'*architecture des ordinateurs*.

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, tandis que le polycopié fournit les détails. Lire le polycopié avant le cours ne présente aucun risque de divulgâchage (c'est même recommandé). 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. L'**évaluation** de ce module consiste en un partiel (1h) et d'un examen final (2h), comptant pour respectivement 1/3 et 2/3 de la note finale. Les compositions ne vous seront pas rendues, mais des séances de **consultation des copies** seront organisées.

Les seuls documents autorisés au partiel et à l'examen sont

- la **copie papier** du polycopié, qui peut être annotée, et
- une feuille A4 manuscrite personnelle.

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 Mathilde Bouvel, Fabienne Buffet, Véronique Cortier, Alexandre Guernut, Samuel Hornus, Emmanuel Jeandel et Florent Koechlin. Les erreurs, coquilles et autres approximations sont bien évidemment de ma responsabilité.

Table des matières

1 Cas d'étude : problèmes d'affectation et algorithme de Gale-Shapley 11
- 1.1 Problématique d'affectation de ressources . . . 11
  - 1.1.1 Contraintes de compatibilité et de préférence . . . 11
  - 1.1.2 Acceptabilité et instabilités . . . 12
- 1.2 Notion de problème algorithmique . . . 12
  - 1.2.1 Définition . . . 12
  - 1.2.2 Premiers exemples . . . 13
- 1.3 Le problème algorithmique des mariages stables . . . 13
- 1.4 L'algorithme de Gale-Shapley . . . 15
  - 1.4.1 Un mot sur la présentation . . . 15
  - 1.4.2 L'algorithme . . . 15
  - 1.4.3 Terminaison de l'algorithme . . . 16
  - 1.4.4 Correction de l'algorithme . . . 17
- 1.5 Une propriété de la solution : optimalité pour les demandeurs . . . 18
- 1.6 Adapter l'algorithme à des variantes du problème . . . 19
  - 1.6.1 Ensembles de tailles différentes . . . 19
  - 1.6.2 Capacités . . . 20
  - 1.6.3 Listes de préférences tronquées . . . 22
- 1.7 Après l'algorithmique, le déploiement : Parcoursup . . . 22
- 1.8 Prolongements . . . 23
- 1.9 Références bibliographiques . . . 24

2 Modèles de calcul classiques 27
- 2.1 Les multiples rôles d'un modèle de calcul . . . 27
- 2.2 Numérisation de l'information . . . 29
  - 2.2.1 Encodage d'entiers et de séquences d'entiers . . . 29
  - 2.2.2 Taille d'une donnée . . . 30
  - 2.2.3 Problèmes algorithmiques en mots binaires . . . 30
- 2.3 Intermezzo : principes d'architecture des systèmes de calcul . . . 30
  - 2.3.1 Stockage de l'information . . . 31
  - 2.3.2 Instruction élémentaire . . . 31
  - 2.3.3 Processeurs, assembleur et langages . . . 32
  - 2.3.4 Et les algorithmes dans tout cela ? . . . 32
- 2.4 Modèle RAM à taille de mot constante . . . 32
  - 2.4.1 Description du modèle . . . 33
  - 2.4.2 Stockage de mots de taille arbitraire en RAM taille constante . . . 33
  - 2.4.3 Algorithmique et modèle RAM taille constante . . . 34
  - 2.4.4 Un exemple : multiplication d'entiers . . . 35
- 2.5 Complexité asymptotique pire-cas . . . 36

2.5.1 L'esquisse . . . 36
2.5.2 Des lacunes.. que résout l'asymptotique . . . 36
2.5.3 Exemple : complexité de l'algorithme de Gale-Shapley . . . 37
2.5.4 Discussion : pertinence et limites de cette mesure de complexité . . . 38
2.6 Simplification : modèle RAM à taille de mots arbitraire . . . 38
2.6.1 Objectif de la simplification de modèle . . . 38
2.6.2 Description du modèle . . . 39
2.6.3 Exemple d'instruction abusive : multiplication (bis) . . . 40
2.6.4 Complexité numérique VS complexité arithmétique . . . 40
2.7 Prolongements . . . 41
2.8 Références bibliographiques . . . 41

3 Méthode récursive 45
3.1 Méthodologie récursive : simplifier et déléguer . . . 45
3.1.1 Les cas de base . . . 46
3.1.2 Apprendre à déléguer . . . 46
3.1.3 Premier exemple . . . 46
3.2 Recherche dichotomique (binary search) . . . 47
3.2.1 Exemple . . . 47
3.3 Diviser pour régner . . . 48
3.3.1 Exemple : tri fusion . . . 48
3.3.2 Exemple : tri rapide . . . 49
3.4 Exploration par retour arrière (backtracking) . . . 50
3.4.1 Exemple : problème des n reines . . . 50
3.4.2 Commentaires . . . 51
3.5 Prolongements . . . 52

4 Complexité asymptotique pire-cas d'algorithmes récursifs 55
4.1 Préliminaires : rappels sur les arbres . . . 55
4.2 Arbres d'exécution . . . 58
4.2.1 Arbre d'exécution d'une entrée . . . 58
4.2.2 Arbre d'exécution d'un algorithme . . . 58
4.2.3 Premiers exemples . . . 59
4.3 De l'arbre d'exécution à la complexité . . . 60
4.3.1 Le principe . . . 60
4.3.2 Cas des récursions uniformes . . . 61
4.4 Borne inférieure sur la complexité d'un algorithme . . . 62
4.5 Le « théorème maître » . . . 63
4.6 Prolongement : le théorème d'Akra-Bazzi . . . 63

5 Cas d'étude : Systèmes de vote 67
5.1 Problématique . . . 67
5.1.1 Un objectif à définir . . . 67
5.1.2 Discussion informelle de quelques méthodes de comptage . . . 68
5.2 Fonctions de choix/d'ordre social . . . 69
5.2.1 Formalisme . . . 69
5.2.2 Propriétés et résultats d'impossibilité . . . 70
5.3 L'apport d'un regard algorithmique . . . 71
5.4 Quelques méthodes de comptage . . . 71
5.4.1 Méthodes de Copeland et Minimax . . . 72

5.4.2 Méthode de Kemeny-Young . . . 72
5.4.3 Méthode des paires ordonnées . . . 73
5.4.4 Méthode de Schulze . . . 74
5.5 Prolongements . . . 75
5.6 Références bibliographiques . . . 76

6 Bornes inférieures de complexité **79**
6.1 Borne inférieure sur la complexité d'un problème . . . 79
6.2 Arguments d'adversaire . . . 80
6.2.1 Un exemple simple : Tous distincts . . . 80
6.2.2 Un exemple plus subtil : recherche dans un tableau trié . . . 81
6.3 Arbres de décision . . . 83
6.3.1 L'intuition . . . 83
6.3.2 Le modèle . . . 83
6.3.3 Exemple : tri par arbre de décision . . . 84
6.3.4 Un argument de théorie de l'information . . . 84
6.4 Prolongement . . . 85

7 Réduction **87**
7.1 Premier exemple : calcul d'enveloppe convexe . . . 87
7.1.1 L'algorithme de Graham . . . 88
7.1.2 Un exemple de réduction linéaire . . . 89
7.1.3 La réduction réciproque... . . . 89
7.2 Formalisation du mécanisme de réduction . . . 90
7.2.1 Problèmes algorithmiques généraux . . . 90
7.2.2 Problèmes de décision . . . 91
7.3 Deux problèmes de référence . . . 92
7.3.1 k-coloration de graphe . . . 92
7.3.2 k-satisfiabilité de formule booléenne . . . 93
7.4 Réductions entre 3-Col et 3-Sat . . . 94
7.4.1 Un premier exemple : réduction polynomiale de 3-col à 3-Sat . . . 94
7.4.2 Esquisse de réduction de 3-Sat à 3-Col . . . 95
7.5 Borne inférieure conditionnelle : problèmes 3-Sum-difficiles . . . 95
7.5.1 Algorithmes pour 3-Sum . . . 96
7.5.2 Exemples de problèmes 3-Sum-difficiles . . . 96
7.5.3 Réfutation de la conjecture 3-Sum et conjecture renforcée . . . 98
7.6 Prolongements . . . 98
7.7 Références bibliographiques . . . 98

8 Classes de complexité et NP-difficulté **101**
8.1 La classe de complexité P . . . 101
8.2 La classe de complexité NP . . . 102
8.2.1 Définition . . . 102
8.2.2 Premiers exemples . . . 103
8.2.3 Solvable implique vérifiable . . . 103
8.3 Problèmes NP-difficiles et NP-complets . . . 104
8.3.1 Problème NP-difficile et réduction . . . 104
8.3.2 $P \stackrel{?}{=} NP$ . . . 104
8.3.3 Un aperçu de la preuve du théorème de Cook-Levin . . . 105
8.4 Exemples de problèmes NP-difficiles . . . 106

8.5 Prolongements 109
8.6 Références bibliographiques 110

9 Modèle de calcul quantique **113**
9.1 Vue d'ensemble 113
9.2 Calcul sur un qubit 115
9.3 Calcul sur deux qubits 116
9.3.1 Définition d'un opérateur par référence au calcul booléen 117
9.3.2 Impossibilité du clonage quantique 117
9.4 Calcul sur m qubits 117
9.4.1 Calcul tensoriel sur $\mathbb{C}$-espaces vectoriels 118
9.4.2 Système à m qubits et algorithme quantique 119
9.5 Circuits 120
9.5.1 Représentation d'une porte agissant sur un ou deux qubits 120
9.5.2 Portes contrôlées et leur représentation 121
9.5.3 Quelques portes usuelles 121
9.6 Algorithme de Grover 122
9.6.1 L'idée géométrique 123
9.6.2 La convergence 124
9.6.3 La traduction algorithmique 124
9.7 Prolongements 125

A Comment présenter un algorithme dans ce cours **129**

B Rappels sur les notations asymptotiques **131**

C Complément : quelques notions d'architecture **135**
C.1 Stockage de l'information 135
C.2 Manipulation de l'information 136
C.3 Qu'est-ce qu'une instruction élémentaire ? 138
C.4 Qu'est-ce qu'un programme ? 139

D Compléments : machines de Turing, indécidabilité et thèse de Church-Turing **141**
D.1 Machine de Turing 142
D.2 Thèse de Church-Turing 143
D.3 Machine de Turing universelle et indécidabilité de l'arrêt 143
D.4 Et les langages dans tout ça ? 144

E Introduction aux structures de données et à leur analyse **145**
E.1 Définitions : type abstrait et structure de donnée 145
E.2 Premiers exemples : tableau et liste chaînée 145
E.3 Analyse de complexité d'une structure de donnée 146
E.4 Types de données d'un langage de programmation 147

F Preuve du théorème d'Akra-Bazzi **149**
F.1 Détour par une récurrence réelle exacte 149
F.2 Mise en place d'une récurrence 150
F.3 Fonctions à croissance polynomiale 150
F.4 Initialisation de la récurrence 151
F.5 Induction 151
F.6 Retour au Théorème F.0.1 152

G Complément : formalisation de la notion d'adversaire **155**

H Machines de Turing non-déterministe et classe NP **157**
H.1 Équivalence entre log-RAM et machine de Turing déterministe 157
H.2 Machine de Turing non-déterministe 158
H.3 Classe NP et machines de Turing non-déterministes 158

I Transformée de Fourier discrète, classique et quantique **161**
I.1 Problématique : retour sur le produit de polynômes 161
I.1.1 Représentation d'un polynôme 161
I.1.2 Représentation et opérations 162
I.1.3 Évaluation multi-points d'un polynôme – idées 162
I.1.4 Évaluation multi-points d'un polynôme – synthèse 164
I.2 Transformée de Fourier discrète 164
I.2.1 Définition de la transformée de Fourier discrète 164
I.2.2 TFD et conjugaison 165
I.2.3 TFD et convolution 165
I.2.4 Lien avec la multiplication (rapide) de polynômes 165
I.3 Algorithme de FFT classique 166
I.4 Algorithme quantique de transformée de Fourier discrète 167
I.4.1 La TFQ, version unitaire de la TFD 167
I.4.2 Factorisation tensorielle de la TFQ 167
I.4.3 L'algorithme quantique de calcul de la TFQ 168
I.4.4 Détail des preuves 168

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.

Nous commençons par poser une problématique (une affectation à effectuer) puis la formalisons en un *problème algorithmique* (le calcul d'un mariage stable) et présentons un algorithme qui le résout (« Gale-Shapley »). Nous établissons ensuite certaines propriétés de cet algorithme (terminaison, complexité, optimalité pour les demandeurs) et examinons son adaptation à des variantes de la problématique initiale. 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

- formaliser un problème algorithmique,
- modéliser une problèmatique applicative par un problème algorithmique,
- présenter un algorithme (sans nécessairement écrire du pseudocode),
- analyser des propriétés (du résultat) d'un algorithme.

1.1 Problématique d'affectation de ressources

Les problématiques d'affectation de ressources relèvent du domaine de la recherche opérationnelle. Elles mettent généralement 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]).

1.1.1 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. Par exemple, lors de l'affectation des étudiants en médecine au stage infirmier de début de 2ème année, chaque étudiant formule son ordre de préférence sur les stages proposés, et tous les stages ordonnent les étudiants par ordre de classement au concours PACES.

1.1.2 Acceptabilité et instabilités

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 étudiants (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.

1.2 Notion de problème algorithmique

La première étape pour résoudre une problématique comme celle décrite ci-dessus 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.

1.2.1 Définition

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

> Un **problème algorithmique** est une fonction $P : E \to 2^S$ où E et S sont deux ensembles dénombrables.

L'ensemble E est l'ensemble des *entrées* possibles du problème. On appelle aussi une entrée $e \in E$ une *instance* du problème. L'ensemble S est l'ensemble des réponses possibles au problème. Pour

chaque entrée $e \in E$ possible, $P(e)$ désigne l'ensemble des sorties acceptées comme réponse au problème sur cette entrée e.

1.2.2 Premiers exemples

Dans ce cours nous *présentons* les problèmes algorithmiques comme suit :

> CALCUL DU 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 $P([i_1, i_2, \ldots, i_k]) = \min\{i_1, i_2, \ldots, i_k\}$.

Dans CALCUL DU 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 :

> CALCUL DE LA 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 $P([i_1, i_2, \ldots, i_k]) = \{j : i_j = \min\{i_1, i_2, \ldots, i_k\}\}$.

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.

> Nous laissons pour l'instant de côté les questions d'**encodage** des entrées et sorties. Nous y reviendrons au Chapitre 2.

1.3 Le problème algorithmique des mariages stables

Voyons comment la problématique d'affectation sans instabilité décrite en Section 1.1 se formalise en problème algorithmique au sens de la Section 1.2.

On modélise les ensembles d'agents à apparier par deux ensembles finis 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 une paire 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 une paire 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 [1] ; autrement dit, M contient une paire (a, b') avec $b \prec_a b'$.

(iii) b préfère a à son appariement ; autrement dit, M contient une paire (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 :

Théorème 1.3.1. *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.

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 une paire 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. 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.

1.4 L'algorithme de Gale-Shapley

L'algorithme que nous présentons maintenant est attribué[2] au système d'affectation des internes dans le système hospitalier de Boston, et a été analysé par Gale et Shapley. Nous travaillerons à partir de leur article originel [GS62] qui reste très lisible.

L'algorithme de Gale-Shapley 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, l'essentiel de la difficulté résidant dans la preuve de correction et l'analyse de complexité. Comprendre ces preuves est indispensable pour pouvoir adapter l'algorithme à des variations du problème.

1.4.1 Un mot sur la présentation

Gale et Shapley présentent leur algorithme par un texte 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 aussi efficace que du « pseudo-code » pour faire comprendre le principe de l'algorithme.[3] Nous adaptons cependant le scénario au public et à l'époque en mettant ici en scène des étudiant·es postulant à des colocations. Nous avons donc deux ensembles de même taille : les étudiant·es (« students ») et les colocations (« houses »). Chaque étudiant·e cherche une place en colocation et chaque colocation a *exactement une* place à remplir. Chaque étudiant·e a connaissance de toutes les colocations et les a classées par ordre de préférence. Chaque colocation a connaissance de tous les étudiant·es et les a classé·es par ordre de préférence.

1.4.2 L'algorithme

Voici l'algorithme de Gale et Shapley :

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.*

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

Nous allons maintenant étudier les propriétés de cet algorithme. On raisonne pour cela sur une entrée générique à n étudiant·es et n colocations.

2. Ceci est un exemple du principe de Stigler : une notion est rarement attribuée à ses premiers inventeurs. Ceci s'applique au principe de Stigler.

3. L'Annexe A discute de la question de la présentation des algorithmes dans ce cours.

1.4.3 Terminaison de l'algorithme

Un algorithme **termine** si pour toute donnée d'entrée *finie*, l'algorithme s'arrête après un nombre *fini* de pas de calcul.[4] Voici comment Gale et Shapley justifient que « leur » algorithme termine :

> *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.*

Leur argument repose sur l'examen des *demandes* (*proposals*) mais laisse un certain nombre d'observations implicites[5]. On peut par exemple se demander ce que l'on doit faire si un·e étudiant·e a *toutes* ses demandes refusées. L'algorithme ne le précise pas car... cela est impossible. Encore faut-il le prouver.

On ne peut pas épuiser une liste. Appellons *phase* de l'algorithme une séquence consistant en l'émission d'une (nouvelle) demande par les étudiant·es 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.

Observation 1.4.1. *Le nombre d'étudiant·es ayant reçu un refus à la phase i égale le nombre de colocations n'ayant aucun·e étudiant·e en attente (*on a string*) à la fin de la phase i.*

Démonstration. Chaque étudiant·e est en attente dans 0 ou 1 colocation. Chaque colocation garde 0 ou 1 étudiant·e en attente. Il y a donc, en fin de phase, autant de colocations ayant un·e étudiant·e en attente que d'étudiant·es en attente. Rappelons qu'il y a autant de colocations de qu'étudiant·es. En passant au complémentaire, on obtient donc que le nombre de colocation sans étudiant·e en attente égale le nombre d'étudiant·es venant de recevoir un refus. □

Observation 1.4.2. *Une colocation qui n'a aucun·e étudiant·e en attente à la fin de la phase i n'a reçu aucune demande aux phases $\leq i$.*

Démonstration. À chaque phase, une colocation rassemble la demande qu'elle a mise en attente et les nouvelles demandes reçues, garde sa préférée qu'elle met en attente, et rejette les autres[6]. Ainsi, si une colocation reçoit au moins une demande à la $j$ème phase, elle a une demande en attente à la fin de la $i$ème phase pour tout $i \geq j$. □

Observation 1.4.3. *La dernière*[7] *demande d'un·e étudiant·e ne peut pas être rejetée.*

Démonstration. Supposons qu'à la phase i, l'étudiant Stanislas adresse sa dernière demande. À la fin de cette phase, chaque colocation aura reçu au moins une demande depuis le début de l'algorithme (ne serait-ce que celle de Stanislas). Ainsi, à la fin de la phase i, chaque colocation aura une demande en attente. Il n'y a donc aucune colocation sans demande en attente (Observation 1.4.2) et donc aucun refus (Observation 1.4.1). □

4. On formalisera ce qu'est un algorithme et un pas de calcul au chapître suivant.
5. Les auteurs ont une confiance (audacieuse ?) en leur lectorat.
6. En particulier, si une demande était en attente à la phase précédente et n'est plus la préférée, cette demande est désormais rejetée.
7. Précisons qu'il s'agit de la demande adressée au dernier élément de la liste de préférences d'un étudiant (et non pas de la dernière demande en date).

L'algorithme progresse... et termine. Nous allons maintenant, comme le suggèrent Gale et Shapley, examiner la progression de l'algorithme sur une instance quelconque (que l'on suppose fixée au long de l'analyse) par un décompte des demandes. On note d_i le nombre cumulées de demandes faites, tou·te·s étudiant·es et colocations confondus, au cours des phases $1, 2, \ldots, i$. Notons T le numéro de la dernière phase, si elle existe, et $T = \infty$ si l'algorithme ne s'arrête pas.

D'après le paragraphe (a), $d_1 = n$ puisqu'à la première phase chaque étudiant adresse exactement une demande. Par ailleurs, pour qu'une phase ne soit pas la dernière il est nécessaire qu'au moins une colocation n'ait personne en attente, cf paragraphe (d). Avec l'Observation 1.4.1, on en déduit qu'à chaque phase au moins une nouvelle demande est émise. Ainsi,

$$\forall i \in \{2, 3, \ldots, T\}, \qquad d_i \geq d_{i-1} + 1.$$

Par récurrence, on a donc que $d_i \geq n + i - 1$ pour tout $i \in \{1, 2, \ldots, T\}$. Cependant, chaque étudiant progresse dans sa liste sans revenir en arrière, aussi chaque demande est adressée au plus une fois. Comme il y a n^2 demandes distinctes possibles, le nombre de demandes adressées est au plus n^2 et on a donc

$$n + T - 1 \leq d_T \leq n^2,$$

soit $T \leq n^2 - n + 1$. Autrement dit, l'algorithme ne peut pas exécuter plus de $n^2 - n + 1$ phases.

La borne ci-dessus est améliorable, comme l'annoncent Gale et Shapley. On reprendra cela en exercice.

1.4.4 Correction de l'algorithme

Un algorithme **résout** un problème algorithmique $P : E \to 2^S$ si, partant d'une entrée quelconque[8] $e \in E$, 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.

Observation 1.4.4. *L'algorithme de Gale-Shapley retourne un mariage.*

Démonstration. Le résultat de l'algorithme est l'ensemble des paires (a, b) où a est une colocation et b est l'étudiant·e en attente chez a au moment où l'algorithme termine. Chaque colocation appartient à au moins une paire puisque l'algorithme ne termine que quand toutes les colocations ont un étudiant en attente. Chaque colocation appartient à au plus une paire puisqu'une colocation ne garde jamais plus d'un·e étudiant·e en attente. L'algorithme retourne donc exactement n paires et chaque colocation appartient à exactement une de ces paires. De plus, chaque étudiant·e appartient à au plus une paire car un·e étudiant·e n'a jamais plus d'une demande en attente. Comme il y a autant d'étudiant·es que de colocations, chaque étudiant·e appartient à exactement une paire. L'algorithme retourne donc bien un mariage. □

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

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 :

Observation 1.4.5. *À la fin de chaque phase, l'étudiant·e mis·e en attente par une colocation est son préféré parmi tou·t·es les candidat·es qui lui ont 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 étudiant·e 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'étudiant mis en attente à la phase i. D'une part, e_{i+1} est préféré ou égal à chacune des demandes reçues à la phase $i + 1$. D'autre part, e_{i+1} est préféré ou égal à e_i, qui est lui-même (par hypothèse de récurrence) le préféré parmi tous les candidats qui lui ont adressé une demande dans les phases $\leq i$. □

On peut maintenant conclure la preuve de correction :

Observation 1.4.6. *Le mariage retourné par l'algorithme de Gale-Shapley est stable pour le système de préférence donné en entrée.*

Démonstration. Rappelons que pour qu'une paire, disons (Stéfanie, Blandan), constitue une instabilité pour le mariage M retourné par l'algorithme, cette paire doit remplir trois conditions :

(i) (Stéfanie, Blandan) n'est pas dans M,

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

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

Supposons (i) et (ii) vraies et montrons que (iii) est faux. Puisque Stéfanie est appariée à Commanderie dans M, elle a adressé une demande à Commanderie. Comme Stéfanie adresse ses demandes par préférences décroissantes, (ii) implique qu'elle a adressé une demande à Blandan. D'après l'Observation 1.4.5, l'appariement final de Blandan (ici Charles) est son préféré parmi toutes les demandes que Blandan a reçu. En particulier, Blandan préfère Charles à Stéfanie et (iii) est faux. □

1.5 Une propriété de la solution : optimalité pour les demandeurs

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. Pour formaliser cela, 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 1.5.1. *L'algorithme de Gale-Shapley apparie chaque élément de l'ensemble des demandeurs à l'arbitreur qu'il préfère parmi ceux qui lui sont accessibles.*

On prouve cela à l'exercice 3 du TD.

1.6 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.

1.6.1 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 [9], 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 de A et B est dit **maximal** (ou **maximal pour l'inclusion**) s'il n'est pas contenu strictement dans un autre mariage 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.

De manière générale, la condition d'être de cardinal maximum implique que l'on est maximal pour l'inclusion mais la réciproque peut être fausse. [10] Dans le cas des mariages stables, maximalité pour la taille et pour l'inclusion s'avèrent équivalentes [11]. 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.

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 :

9. Au sens où il permet de modéliser des problématiques intéressantes.
10. Intuitivement, la « maximalité pour l'inclusion » est une condition d'optimalité locale, tandis que le « cardinal maximal »est une condition d'optimalité globale.
11. On le prouve à l'exercice 4 du TD.

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 paires 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 1.4.5 et 1.4.6 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 1.5.1 d'optimalité pour les demandeurs. On traite ces deux points aux exercices 4 et 5 du TD.

Il est remarquable que cet algorithme produit *le même résultat* que l'algorithme de la Section 1.4.2 dans le cas où A et B sont de même taille. Il le *généralise*.

1.6.2 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[12] ou les deux types[13] 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 l'agent x. Les définitions de système de préférences et de tableau de préférences

12. Par exemple l'affectation de bacheliers dans des établissements d'enseignement supérieur permet à chaque établissement d'accueillir plusieurs bacheliers mais n'affecte chaque bachelier que dans au plus une affectation.
13. 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.

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 $c(x)$ paires.[14] Soulignons qu'un mariage de capacité c est un *ensemble*, et contient donc 0 ou 1 copie de chaque paire.[15] 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* **x** *propose to his favorite* **c(x)** *house**s**. Each house* **y** *who receives more than* **c(y)** *proposal rejects all but its* **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* **x** *who* ***was*** *rejected now proposes to* ***its next*** *choices* ***so as to have*** **c(x)** ***proposals out****. Each house* **y** *receiving proposals chooses its favorite* **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* **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 1.4.5 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 1.6.1 (traité en exercice).

14. En particulier, un mariage au sens de la Section 1.6.1 est un mariage de capacité c pour la fonction c constante égale à 1.

15. 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.

1.6.3 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 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...

1.7 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'étudiant·es et des milliers de formations [16] 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 1.5.1) 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. [17] Cela est apparent dans le fait qu'au fil de l'exécution

16. D'après wikipedia, ∼660 000 étudiant·es et ∼15 500 formations en 2020.

17. Soulignons que la question n'est pas de savoir si les étudiant·es 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.

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 étudiant·es consulté·es de décider et communiquer leurs arbitrages. Cela ralentit donc l'algorithme et impose aux étudiants de consulter régulièrement l'état interne courant.
- 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 1.4.5), 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.

1.8 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 largement 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].

1.9 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 de Gale-Shapley, aussi appelé *algorithme de choix retardé*, résout de nombreuses variantes du calcul d'un mariage stable.
- Cet algorithme consiste à ce que l'un des types d'agents (les *proposeurs*) 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 termine en $O(nm)$ phases (propositions + refus) où n et m sont les nombres d'agents des deux types. La solution calculée est optimale pour les demandeurs.

Notes personnelles

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 les notions d'algorithme et de complexité. Cela nous demande de préciser le modèle de calcul dans lequel on travaille.

Nous faisons cela en quatre temps. Nous commençons par aborder la *numérisation* de l'information et quelques principes *d'architecture des ordinateurs*, notamment la notion d'*instruction élémentaire* qui est fondamentale. Nous présentons ensuite deux modèles de calcul, en fait deux variantes issues d'une même famille (les modèles RAM). La première modélise plus fidèlement l'architecture des ordinateurs mais est d'usage laborieux, la seconde est plus permissive mais peut autoriser des algorithmes absurde du point de vue des systèmes de calcul réels. Nous introduisons la notion d'*instruction abusive* pour relier ces deux modèles. Nous précisons enfin la notion de complexité asymptotique pire-cas. Elle se décline en deux versions, dans le modèle restrictif (où elle est appelée complexité *numérique*) et dans le modèle permissif (où elle est appelée *arithmétique*).

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

- sachiez coder de l'information par des mots binaires et mesurer la taille du codage d'une information (nombre de bits, à $O()$ près),
- compreniez le modèle *RAM à taille de mot constante*, la notion d'algorithme associée et la complexité asymptotique d'un tel algorithme,
- compreniez le modèle *RAM à taille de mot arbitraire* et sachiez identifier une instruction abusive dans ce modèle,
- sachiez déterminer les complexité *numérique* et *arithmétique* d'un algorithme simple.

2.1 Les multiples rôles d'un modèle de calcul

Un *modèle de calcul* décrit la manière dont un système *calcule*, c'est-à-dire traite de l'information. Il en existe une grande variété, organisée en familles : séquentiels (machine de Turing, automate fini, 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**.

Un modèle de calcul séquentiel pour quoi faire ?

Les premiers modèles de calcul ont été inventés aux débuts de **la théorie de la calculabilité** pour décrire précisément *ce qui est calculable et 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 existant 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 ces *problèmes indécidables*.

Comme l'intitulé de ce cours le laisse entendre, nous laissons de côté la théorie de la calculabilité, et n'en donnons en Complément D qu'une très rapide introduction à seules fins culturelles.

Un modèle de calcul sert aussi en **algorithmique** pour formuler des *algorithmes*, c'est-à-dire des manières d'organiser un calcul donné. Le modèle de calcul guide donc la *présentation* des algorithmes, c'est-à-dire sa communication à un lecteur, supposé intelligent. Profitons de cette occasion pour dissiper un malentendu courant :

Il est préférable de **ne pas** présenter un algorithme sous la forme d'un programme informatique dans un langage de programmation (`python`, `C`, `OCaml`, ...).

Une première raison pour cela est que le niveau de détail imposé par un langage de programmation permet rarement une communication efficace, qui doit se concentrer sur les étapes signifiantes [1]. Une seconde raison est que les algorithmes, les langages de programmation et les architectures matérielles évoluent conjointement pour rendre le calcul plus facile et plus efficace [2], or penser les algorithmes au travers d'un langage particulier entrave cette évolution.

Un modèle de calcul permet aussi, en **théorie de la complexité**, d'étudier l'efficacité des calculs et les limites à cette efficacité. Cette analyse associe généralement à chaque opération un coût, reflétant les ressources qu'elle consomme. L'addition des coûts des opérations réalisées par un algorithme donne sa *complexité*. Cela permet aussi d'étudier la *complexité d'un problème*, définie comme la complexité minimale d'un algorithme résolvant ce problème.

L'algorithmique et la complexité imposent des exigences contradictoires sur le modèle de calcul. Un modèle permissif simplifie la présentation d'*algorithmes individuels* en algorithmique, mais complique le raisonnement sur des *ensembles d'algorithmes*, qui est l'essence des théories de la complexité et de la calculabilité. Il est donc raisonnable d'adapter le modèle selon les circonstances :

Dans ce cours, on définit deux modèles : *RAM taille constante* pour la théorie de la complexité et *RAM taille arbitraire* pour l'algorithmique.

D'autres modèles de calcul viendront s'ajouter ultérieurement : arbres de décision au Chapitre 6, équivalents de machine de Turing non-déterministe au Chapitre 8, et circuits quantiques au Chapitre 9.

1. Par exemple le sens de parcours des tableaux, l'ordre d'affectation des variables, *etc.* ne doivent être précisés que s'ils ont de l'importance.

2. Par exemple, l'usage intensif de certaines opérations dans des algorithmes beaucoup utilisés conduit à créer des instructions dédiées au niveau des processeurs. Cf par exemple les *tensor processing unit.*

2.2 Numérisation de l'information

Une caractéristique fondamentale d'un système physique traitant de l'information est la manière dont il mémorise cette information. Dans la majorité des systèmes actuels, cette mémorisation se fait par la combinaison d'un très grand nombre de *cellules mémoires élémentaires*. Chaque cellule mémoire élémentaire peut être dans deux[3] états ; selon que la cellule est dans l'un ou l'autre état, on considère qu'elle « mémorise » le chiffre 0 ou le chiffre 1. En associant plusieurs mémoires élémentaires, on peut stocker un **mot binaire**, c'est-à-dire une suite *finie* d'éléments de $\{0, 1\}$. Cette restriction est fondamentale : l'informatique classique ne calcule que sur des mots binaires.

> 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*).[4] La **longueur** d'un mot binaire w est son nombre de bits, et est généralement notée $|w|$. On note $\{0, 1\}^n$ l'ensemble des mots binaires de longueur n et $\mathbf{\{0,1\}}^* \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}}\{0, 1\}^n$ l'ensemble des mots binaires. Lorsque l'on a besoin d'expliciter les bits d'un mot binaire w, il est courant[5] de les noter $w = w_1w_2 \ldots w_{|w|}$.

2.2.1 Encodage d'entiers et de séquences d'entiers

La représentation d'une information par un mot binaire est appelée **encodage** de cette information. Certaines informations admettent des encodages naturels. Par exemple, tout entier naturel $a \in \mathbb{N}$ peut s'écrire $a = \sum_{i=0}^{k-1} a_i 2^i$ avec $a_i \in \{0, 1\}$ et cette écriture est unique si l'on impose $a_{k-1} \neq 0$ ou, pour $x = 0$, que $k = 1$. Le mot binaire $a_{k-1}a_{k-2} \ldots a_0$ est appelé la **représentation binaire** de a. Il est naturel d'encoder les entiers positifs par leur représentation binaire. Soulignons que la taille de la représentation binaire d'un entier positif x est $\lceil\log_2(1+x)\rceil$. C'est immédiat pour $n = 0$. Pour $n \geq 1$, les entiers positifs ayant une représentation binaire de taille n sont les entiers compris entre 2^{n-1} (représenté par $100 \ldots 0$) et $2^n - 1$ (représenté par $11 \ldots 1$). Ainsi, la représentation binaire de x est de taille n si et seulement si

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

On peut facilement encoder une séquence d'entiers par *un seul* mot binaire. On peut par exemple commencer par encoder chaque entier par sa représentation binaire, puis appliquer à chacun de ces mots la transformation $0 \to 00$ et $1 \to 11$, et enfin concaténer[6] les mots obtenus en insérant entre deux mots consécutifs le mot 01. Par exemple, ce processus appliquée à la séquence $(8, 5)$ donne

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

Cette transformation est réversible puisque les séparations 01 sont facilement identifiables, et elle est donc injective. Plus généralement, on peut encoder toute information *discrétisée* (texte, image, son, séquence vidéo, ...) par *un seul* mot binaire.

3. Les systèmes de calcul quantiques constituent une exception notable. Nous y reviendrons au Chapitre 9.
4. Ainsi, dans le mot binaire 01101 le premier bit est à 0 et deuxième est à 1.
5. Lorsque w est la représentation binaire d'un entier on utilise souvent une convention différente, cf ci-après.
6. La **concaténation** de deux mots binaires w et w' est le mot binaire $w \cdot w' \stackrel{\text{def}}{=} w_1w_2 \ldots w_{|w|}w'_1w'_2 \ldots w'_{|w'|}$ obtenu en « accolant » w' à la fin de w. Cette opération est associative (mais pas commutative).

2.2.2 Taille d'une donnée

La **taille** d'une donnée x est définie comme la taille d'un mot binaire encodant x. Cela suppose de fixer une convention d'encodage, mais le principe suivant rends souvent cette convention d'encodage assez secondaire :

On s'intéresse généralement à la taille d'une donnée 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 $O(n^2 \log n)$.

Dans ce cours, le terme **borné** signifie majoré par une constante absolue.

Ainsi, si n désigne la taille de l'entrée d'un algorithme, $2n$ n'est pas borné mais $42 + \frac{(-1)^n}{n}$ l'est.

2.2.3 Problèmes algorithmiques en mots binaires

Au Chapitre 1, on a défini un *problème algorithmique* comme une fonction $P : E \to 2^S$ où E et S sont des ensembles dénombrables. Cette définition laisse le choix de la présentation des ensembles d'entrées et de sorties (E et S). Une alternative naturelle consiste à *normaliser* ces ensembles en imposant $E = S = \{0,1\}^*$. Formellement, un **problème algorithmique en mots binaires** est une fonction $P : \{0,1\}^* \to 2^{\{0,1\}^*}$.

Considérons un problème algorithmique $P : E \to 2^S$ « ensembliste » au sens où E et S sont des ensembles dénombrables arbitraires. Comme tout ensemble dénombrable s'injecte dans $\{0,1\}^*$, il existe une injection $c : E \to \{0,1\}^*$. De même, il existe une surjection $d : \{0,1\}^* \to S$ et un problème algorithmique en mots binaires $P' : \{0,1\}^* \to 2^{\{0,1\}^*}$ tels que [7] $P = d \circ P' \circ c$. Cela rend les problèmes algorithmiques en mots binaires essentiellement [8] équivalents aux problèmes algorithmiques.

Dans l'étude des problèmes algorithmiques, les points de vue « ensembliste » et « en mots binaires » sont complémentaires :

- Raisonner sur des problèmes algorithmiques « ensemblistes »permet d'exprimer les idées dans un langage plus direct et plus naturel. C'est d'un grand confort lors de la conception et l'étude d'algorithmes.
- Raisonner sur des problèmes algorithmiques en mots binaires fournit une notion claire de *taille* d'une entrée. C'est essentiel pour étudier la complexité d'un algorithme ou, en théorie de la complexité, définir des *réductions polynomiales* entre problèmes algorithmiques.

Ainsi, même quand on travaille sur des problèmes algorithmiques « ensemblistes », leur traduction en mots binaires, même implicite, est rarement loin...

2.3 Intermezzo : principes d'architecture des systèmes de calcul

Avant de définir nos modèles de calcul, examinons quelques principes généraux d'architecture des ordinateurs. 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·es trouveront plus de détails et d'exemples en Complément C.

7. Ici, on a implicitement étendu d en une fonction $d : 2^{\{0,1\}^*} \to 2^S$ par $d(X) \stackrel{\text{def}}{=} \{d(x) : x \in X\}$.
8. Cette affirmation laisse de côté le calcul des fonctions c et d, ce qui cache d'importantes subtilités.

2.3.1 Stockage de l'information

Le dispositif de mémorisation d'information d'un ordinateur comporte plusieurs niveaux. La construction d'une *cellule mémoire élémentaire*, déjà évoquée en Section 2.2, permet de mémoriser un nombre 1-bit. L'assemblage de b cellules mémoire élémentaires en une *case mémoire* permet de mémoriser un nombre b-bits. [9] L'organisation de M cases mémoire b-bits en une *unité mémoire*, qui permet ainsi de mémoriser M octets. Les deux premiers niveaux ont peu d'influence sur la définition du modèle de calcul, si ce n'est dans le choix du paramètre b. Nous les laissons de côté et renvoyons les élèves intéressé·es à l'ouvrage de Nisan et Schocken [NS21, §3.1].

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. On peut donc considérer la mémoire comme un tableau de mots binaires, typiquement des *octet* (i.e. des mots binaires 8-bits).

2.3.2 Instruction élémentaire

Sur un ordinateur, toute information est représentée par un mot binaire. Tout calcul est donc une transformation d'un mot binaire (l'entrée du calcul) en un autre mot binaire (la sortie du calcul), c'est-à-dire une fonction de $\{0,1\}^*$ dans $\{0,1\}^*$. Le point de départ 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 calculant f.

Autrement dit, 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 admet ici ce principe fondamental, mais pour les élèves intéressé·es, on en donne une explication en Complément C.

Une **instruction élémentaire** dans un système de calcul est une opération transformant une donnée d'entrée en une donnée de sortie, et pour laquelle

(i) la donnée d'entrée détermine complètement la donnée de sortie, et

(ii) le nombre de données d'entrées possible est borné.

La propriété (ii) assure que l'on peut coder les entrées par un mot binaire de taille ℓ constante. Avec la propriété (i), on a aussi un nombre borné de données de sortie possibles et on peut donc les coder par un mot binaire de taille m constante. L'exécution d'une instruction élémentaire se réduit donc au calcul de la fonction qui envoie chaque mot de $\{0,1\}^\ell$ codant une donnée d'entrée sur le mot de $\{0,1\}^m$ codant la donnée de sortie associée.

Ainsi, l'addition de deux entiers **n'est pas** une instruction élémentaire (l'ensemble des entrées est infini) mais l'addition de deux entiers modulo 2^{100} **est** une instruction élémentaire. De même, la comparaison de deux mots binaires pour l'ordre lexicographique **n'est pas** une instruction élémentaire, mais le tri d'un tableau de 100 entiers, chacun à valeur dans $\{0, 1, \ldots, 2000\}$, **est** une instruction élémentaire.

9. Généralement $b = 8$ mais d'autres choix sont possibles.

2.3.3 Processeurs, assembleur et langages

Un **processeur** est un système physique qui calcule. À la conception d'un processeur, on choisit les instructions élémentaires qu'il est capable d'exécuter. Ces instructions élémentaire constituent son **langage assembleur** et peuvent, en principe, varier d'un processeur à l'autre. [10] 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'instructions, 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 [11] 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 le circuit à activer, activation du circuit en question.

2.3.4 Et les algorithmes dans tout cela ?

Formellement, un **algorithme** est la décomposition d'un calcul en une séquence d'instructions élémentaires. Le choix des instructions élémentaires utilisées pour décrire un algorithme est laissée libre, aussi un algorithme peut ne pas être directement interprétable dans l'assembleur d'un processeur, 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 [12], raison de plus pour distinguer la conception d'algorithmes de leur programmation.

2.4 Modèle RAM à taille de mot constante

Cette section présente un premier modèle de calcul, un second venant en Section 2.6. Ces deux modèles appartiennent à la famille des modèles RAM, acronyme de *Random Access Machine*. Les modèles RAM s'inspirent des principes d'architecture exposés ci-dessus mais les épurent pour faciliter la description de algorithmes et permettre l'analyse de leur complexité.

10. 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).

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

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

2.4.1 Description du modèle

Le **modèle RAM** comporte deux parties, l'unité mémoire et l'unité de calcul.

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 $0, 1, 2, \dots$ 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 [13].

L'**unité de calcul** peut exécuter des *instructions élémentaires* sur des données contenues dans des cases mémoire dont on connait les adresses. [14] 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 à taille de mot constante** 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 **constante**, fixée à 8 mais toute autre valeur fixe ferait l'affaire,
- est acceptée comme **instruction élémentaire** toute opération transformant une donnée d'entrée, à choisir dans un ensemble de taille bornée, en une donnée de sortie, déterminée uniquement par la donnée d'entrée.

On abrégera le nom de ce modèle en **RAM taille constante**.

Remarquons qu'avec cette définition, toute séquence de longueur bornée d'instructions élémentaires constitue elle-même une instruction élémentaire.

2.4.2 Stockage de mots de taille arbitraire en RAM taille constante

Dans le modèle RAM à taille de mots constante, 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 :

$$\underbrace{01011011101111101111101111101111}_{\text{mot 32 bits}} \quad \to \quad \underbrace{01011011}_{\text{8 bits}}\ \underbrace{10111101}_{\text{8 bits}}\ \underbrace{11110111}_{\text{8 bits}}\ \underbrace{11101111}_{\text{8 bits}}.$$

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 :

$$\underbrace{10110111011110111 1}_{\text{mot 17 bits}} \quad \to \quad \underbrace{000000010110111011111011 1}_{\text{mot 24 bits}} \quad \to \quad \underbrace{00000001}_{\text{8 bits}}\ \underbrace{01101110}_{\text{8 bits}}\ \underbrace{111101111}_{\text{8 bits}}.$$

13. Par exemple, les cases mémoire d'adresses 5 et 12 peuvent toutes deux contenir le mot « 101010 ».
14. Autrement dit, 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.

Ainsi, dans le modèle RAM à taille de mot constante, on représente un mot binaire par un *tableau* d'octets. Ainsi

$$w = 101101110111101111 \quad \rightsquigarrow \quad w[1..3] = [00000001, 01101110, 111101111].$$

Le *stockage d'entiers* 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.) Ainsi l'entier $w = 187887$ de représentation binaire 101101110111101111 est codé par le tableau $w[1..3] = [111101111, 01101110, 00000001]$, 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**.

2.4.3 Algorithmique et modèle RAM taille constante

Formellement, un **algorithme** est une séquence d'instructions élémentaires. Dans ce cours...

> Toute instruction non-élémentaire utilisée pour présenter un algorithme doit être clairement décomposable en une séquence d'instructions élémentaires.

(Cette condition sera renforcée en Section 2.5.3.) Pour voir que cette condition ne va pas toujours de soi, et 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
    étudiant i demande à sa colocation préférée
```

mais pour le le modèle RAM à taille de mots constante, ces instructions **ne sont pas** élémentaires. En effet, si on décompose [15] le `pour` en

```
1 i = 1
2 étudiant i demande à sa colocation préférée
3 i = i+1
4 si i != n+1 retourner à la ligne 2
```

on voit qu'il cache un `i=i+1` qui **n'est pas** une instruction élémentaire : l'ensemble des données d'entrée (les entiers de $\mathbb{N}$) 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...

15. 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.

2.4.4 Un exemple : multiplication d'entiers

Illustrons la présentation d'un algorithme dans le modèle RAM taille constante sur 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 [16] et fondamental ; on y reviendra plusieurs fois au cours du module. En voici une formulation adaptée au modèle RAM à taille de mots constante :

> 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 $a * b$.

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

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

à évaluer chaque terme $D[k]$ naïvement pour $k = 1..2n-1$, puis à "propager les retenues" entre les $D[k]$ pour en déduire $C[1..2n-1]$. Cela devrait ressembler à la manière dont vous avez appris à multiplier en petite classe [17]. On peut le faire par exemple comme suit :

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

(La division de la ligne 7 est entière.) Peu d'instructions de cet algorithme sont élémentaires, mais chacune se décompose aisément en instructions élémentaires (décomposition qu'il serait laborieux d'expliciter).

16. Les derniers progrès en date sur ce problème datent de 2019.
17. 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.

2.5 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**.

2.5.1 L'esquisse

Considérons un algorithme $\mathcal{A}$ qui résout un problème algorithmique d'espace d'entrées 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. La fonction $c_{\mathcal{A}} : E \to \mathbb{N}$ ainsi définie est généralement difficile à utiliser, ne serait-ce que parce que l'ensemble E peut être compliqué à décrire. Rappelons que E est muni d'une notion de taille, cf Section 2.2.2, et notons $t(e)$ la taille d'une entrée $e \in E$. On peut agréger les valeurs de $c_{\mathcal{A}}$ taille par taille. Pour l'analyse **pire-cas**, l'agrégation des valeurs de $c_{\mathcal{A}}$ d'une taille donnée se fait par la fonction [18] :

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

Autrement dit, $C_{\mathcal{A}}(n)$ est le nombre maximum d'instructions élémentaires exécutées lors du traitement d'une entrée de taille n.

2.5.2 Des lacunes.. que résout l'asymptotique

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

- La taille d'une donnée x est définie comme la taille d'un mot binaire encodant x, or ce mot peut varier d'une convention d'encodage à l'autre. La définition précise de $C_{\mathcal{A}}$ demanderait de préciser cet encodage, ce qui peut s'avérer laborieux.
- 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 [19].

Il s'avère que ces deux problèmes, et d'autres [20], disparaissent essentiellement [21] 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 $n \to \infty$, du nombre d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter une entrée de taille n.

Autrement dit, lorsque l'on mesure la complexité d'un algorithme on ignore les facteurs additifs et multiplicatifs. Lorsque le contexte est clair, on peut abréger « complexité asymptotique pire-cas de $\mathcal{A}$ » en **complexité** de $\mathcal{A}$.

18. On peut vérifier que la définition de taille assure que l'ensemble $\{e \in E \colon t(e) = n\}$ est fini pour tout n, ce qui assure que l'opération max apparaissant dans l'Équation (2.1) est bien définie.
19. Ce phénomène est décrit plus généralement par le *Théorème d'accéleration linéaire*.
20. Par exemple, le caractère discutable du choix d'assigner un coût unité à chaque instruction élémentaire.
21. Tous les problèmes ne disparaissent pas. Ainsi, par exemple, certaines données n'ont pas d'encodage optimal (on dit *entropique*) connu.

On va assez systématiquement estimer les complexités asymptotique pire-cas des algorithmes que l'on décrit. Cela nous permet de préciser la condition [22] donné en Section 2.4.3 d'usage d'instructions non-élémentaires :

> Lorsqu'une instruction élémentaire est utilisée pour présenter un algorithme, elle doit être clairement décomposable en une séquence d'instructions élémentaires et la complexité asymptotique pire-cas de cette décomposition doit être facile à établir.

Soulignons que le terme « borné » tel qu'on l'a utilisé jusqu'ici est équivalent à « $O(1)$ ».

2.5.3 Exemple : complexité de l'algorithme de Gale-Shapley

Prenons un exemple concret : évaluons la complexité de l'algorithme Gale-Shapley dans le modèle RAM à taille de mots constants. On considère la version symétrique de cet algorithme, avec n demandeurs et n arbitreurs. 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 non-élémentaires 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 établi au Chapitre 1 qu'il y a $O(n^2)$ phases. La complexité d'une phase est $O(n \log n)$ par les mêmes arguments que ceux utilisés pour la 1ère phase et en majorant par n le nombre de proposeurs participants. 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 taille constante est $O(n^3 \log n)$. Cette complexité se décompose en $O(n^3)$ instructions non-élémentaires, impliquant chacune des entiers de taille $O(\log n)$.

22. Rappellons la première version : *Toute instruction non-élémentaire utilisée pour présenter un algorithme doit être clairement décomposable en une séquence d'instructions élémentaires.*

2.5.4 Discussion : pertinence et limites de cette mesure de complexité

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. Par exemple, pour une fonction $f : \mathbb{N} \to \mathbb{N}$ polynomiale, le *ratio* $f(2n)/f(n)$ peut être majoré, pour n assez grand, en fonction de son seul comportement asymptotique :

- si $f = O(n)$ alors $f(2n)/f(n) \lesssim 2$,
- si $f = O(n^2)$ alors $f(2n)/f(n) \lesssim 4$,
- si $f = O(n^k)$ alors $f(2n)/f(n) \lesssim 2^k$,

Pour des fonctions à la croissance plus brutale, par exemple exponentielle, on peut de même majorer le ratio $f(n+1)/f(n)$ à partir de la seule complexité asymptotique pire-cas.

2.6 Simplification : modèle RAM à taille de mots arbitraire

Le modèle RAM taille constante modélise assez fidèlement les systèmes calculants mais il induit un découpage de l'information en tableau d'octets qui peut s'avérer fastidieux à l'usage. Nous allons ici le simplifier en un modèle *RAM taille arbitraire*, tout en installant quelques garde-fous pour assurer que ce nouveau modèle reste « compatible » avec le modèle RAM taille constante.

2.6.1 Objectif de la simplification de modèle

L'analyse de la complexité de l'algorithme de Gale-Shapley en Section 2.5.3 a consisté à majorer le nombre d'instructions non-élémentaires, ici $O(n^3)$, à majorer la complexité de chacune d'entre elles, ici $O(\log n)$, puis à multiplier ces deux bornes. On peut remarquer qu'ici, le nombre d'instructions non-élémentaires est déjà une approximation intéressante [23] de la complexité de l'algorithme. Ce phénomène dépends en fait du niveau de granularité adopté pour décrire l'algorithme : si l'on admettait comme instruction non-élémentaire "une phase de l'algorithme de Gale-Shapley", le nombre d'instructions non-élémentaires deviendrait $O(n^2)$, ce qui s'éloigne sensiblement de la complexité de l'algorithme. Une question naturelle se pose :

> Peut-on simplifier le modèle RAM taille constante pour que l'analyse de complexité soit facilitée mais donne un résultat proche de la complexité dans le modèle RAM taille constante ?

23. Autrement dit, si une complexité de $O(n^3)$ semble acceptable, il est probable qu'une complexité de $O(n^3 \log n)$ le soit aussi. Inversement, si une complexité de $O(n^3 \log n)$ apparaît prohibitive, il est vraisemblable qu'une complexité de $O(n^3)$ le soit déjà.

La simplification que l'on va proposer est le modèle RAM à taille de mots arbitraire. La complexité dans ce modèle approxime celle du modèle RAM taille constante à un facteur multiplicatif poly-logarithmique.[24] Cela impose des restrictions que l'on décrit en terme d'*instructions abusives*.

2.6.2 Description du modèle

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.

Le **modèle RAM à taille de mot 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**,
- est acceptée comme **instruction élémentaire** toute opération transformant une donnée d'entrée, décrite par un nombre borné de cases mémoire, en une sortie, décrite par un nombre borné de cases mémoire et déterminée uniquement par la donnée d'entrée.

On abrégera le nom de ce modèle en **RAM taille arbitraire**.

La **taille** d'une entrée est mesurée par le nombre de cases mémoire qu'elle occupe. La **complexité (asymptotique pire-cas)** d'un algorithme $\mathcal{A}$ dans le modèle RAM taille arbitraire est l'ordre de grandeur asymptotique, pour $n \to \infty$, du nombre d'instructions élémentaires exécutées par $\mathcal{A}$ pour traiter une entrée de taille n. Une instruction est **raisonnable** si

(i) elle peut se traduire en une séquence finie d'instructions élémentaires au sens du modèle RAM taille constante, et

(ii) il existe une constante c telle que lors du traitement d'une entrée de l'algorithme de taille n *au sens du modèle RAM taille constante*, cette traduction exécute $O(\log^c n)$ instructions élémentaires au sens du modèle RAM taille constante.

Une instruction qui n'est pas raisonnable est dite **abusive**. Avec ces définitions, la propriété suivante est immédiate :

Proposition 2.6.1. *Soit $\mathcal{A}$ un algorithme dans le modèle RAM taille arbitraire et notons $f(n)$ sa complexité dans ce modèle. Si toutes les instructions de $\mathcal{A}$ sont raisonnables, alors il existe une constante c et un algorithme $\mathcal{B}$ dans le modèle RAM taille constante telles que $\mathcal{B}$ résout les mêmes problèmes algorithmiques que $\mathcal{A}$ et la complexité de $\mathcal{B}$ dans le modèle RAM taille constante est $O\left(f(n)\log^c n\right)$.*

Dans ce cours, lorsqu'une instruction abusive est utilisée dans un algorithme pour le modèle RAM taille arbitraire, elle doit être signalée et sa complexité au sens du modèle RAM taille constante doit être explicitée.

Autrement dit, toute instruction abusive doit être traitée comme une "sous-fonction" dont il convient d'expliciter l'algorithme.

24. Autrement dit, tout algorithme $\mathcal{A}$ de complexité $O\left(f(n)\right)$ en RAM taille arbitraire se raffine en un algorithme $\mathcal{B}$ de complexité $O\left(f(n)\log^c n\right)$ pour une constante c.

2.6.3 Exemple d'instruction abusive : multiplication (bis)

Revenons sur le problème de multiplication d'entiers

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

et considérons à nouveau sa formulation compatible avec le modèle RAM taille constante :

> 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 dans le modèle RAM taille arbitraire :

```
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. 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. La complexité de cet algorithme dans le modèle RAM taille arbitraire est $O(n)$. Pour autant, cet algorithme ne donne pas lieu à un algorithme quasi-linéaire dans le modèle RAM taille constante ; de fait, il déroge déjà à la condition que *chaque instruction soit facilement décomposable en instructions élémentaires pour le modèle RAM constante.*

Considérons en effet l'instruction `vc = va*vb` de la ligne 7. Cette instruction ne demande rien de moins que de résoudre le problème MULTIPLICATION D'ENTIERS. Cela demande au minimum de lire l'entrée de taille $2n$, ce qui **ne peut pas** être fait en $O(\log^c n)$ instructions élémentaires dans le modèle RAM taille constante. Cette instruction est donc abusive. Du point de vue de la présentation, il faut expliciter par quel algorithme (au sens du modèle RAM taille constante) le produit `va * vb` est calculé (par exemple `multiplication_naive`).

2.6.4 Complexité numérique VS complexité arithmétique

Les règles de présentation d'algorithme que l'on a données autorisent à ce qu'un même algorithme soit valide pour les deux modèles RAM, taille constante et taille arbitraire. On peut donc être amené à définir deux complexité pour un même algorithme :

- La complexité dans le modèle RAM taille arbitraire. C'est ce que l'on appelle l'*arithmetic complexity*, que l'on traduit ici en **complexité arithmétique**.
- La complexité dans le modèle RAM taille constante. C'est ce que l'on appelle la *bit complexity*, que l'on traduit ici en **complexité numérique**.

Ainsi, la complexité arithmétique de l'algorithme `multiplication_naive` est $O(n^2)$ et sa complexité numérique est $O(n^2 \log n)$. De même, la complexité arithmétique de l'algorithme de Gale-Shapley est $O(n^3)$ et sa complexité numérique est $O(n^3 \log n)$.

2.7 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 D mais ne l'utilisons pas dans ce cours, lui préférant le modèle RAM à taille de mots constante, 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 dans le modèle RAM taille constante, 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 E propose une rapide introduction au sujet (prolongement non-exigible).

2.8 Références bibliographiques

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity : a modern approach*. Cambridge University Press, 2009.

[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.

[KSVV02] Alexei Yu Kitaev, Alexander Shen, Mikhail N Vyalyi, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.

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

[Pao10] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation*, 123 :170, 2010.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1) :230–265, 1937.

À retenir.

- Un problème algorithmique est dit **en mots binaires** si ses espaces d'entrées et de sortie sont $\{0,1\}^*$. La **taille** d'une donnée est la taille d'un mot binaire permettant de l'encoder.
- Les deux modèles de calcul utilisés dans ce cours sont basés sur le modèle RAM, et comportent donc chacun une *unité mémoire* et une *unité de calcul*.
- Le modèle **RAM taille constante** limite chaque case mémoire à stocker un octet et accepte comme instruction élémentaire toute opération transformant une entrée choisie dans un ensemble de taille bornée en une sortie qui ne dépend que de l'entrée.
- Le modèle **RAM taille arbitraire** autorise chaque case mémoire à stocker un mot binaire de taille arbitraire (mais finie). Il accepte comme instruction élémentaire toute opération transformant une donnée d'entrée, décrite par un nombre borné de cases mémoire, en une sortie, décrite par un nombre borné de cases mémoire et déterminée uniquement par la donnée d'entrée.
- La **complexité** d'un algorithme $\mathcal{A}$ relativement à un modèle de calcul est l'ordre de grandeur asymptotique, pour $n \to \infty$, du nombre d'instructions élémentaires, pour ce modèle, exécutées par $\mathcal{A}$ pour traiter une entrée de taille n.
- La complexité relative au modèle RAM taille arbitraire elle est appelée **complexité arithmétique**. La complexité relative au modèle RAM taille constante elle est appelée **complexité numérique**.
- Une instruction (élémentaire) dans un algorithme $\mathcal{A}$ pour le modèle RAM taille arbitraire est dite **abusive** si cette instruction ne peut pas être traduite dans le modèle RAM taille constante en un algorithme $\mathcal{B}$ de complexité polylogarithmique en la taille de la donnée d'entrée de $\mathcal{A}$.
- Lorsqu'un algorithme n'utilise pas d'instruction abusive, ses complexités numérique et arithmétique diffèrent au plus d'un facteur multiplicatif polylogarithmique.

Notes personnelles

Chapitre 3

Méthode récursive

Dans cette séance, nous allons passer en revue les principes de conception d'algorithmes *récursifs*. Il ne s'agit pas de construire une théorie de la récursion, mais de donner une vision cohérente et unifiée d'un ensemble de notions que vous avez pu aborder dans votre scolarité, 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 oeuvre les trois principales méthodes de conception d'algorithme récursif (recherche dichotomique, diviser pour régner, et exploration par retour arrière). L'analyse de complexité des algorithmes récursifs sera 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.

On peut mesurer la **simplicité** d'une entrée par sa taille [1], une entrée A étant plus simple qu'une entrée B si la taille de A est inférieure à la taille de B. Pour certains problèmes on sera amené à mesurer la simplicité de manière plus *ad hoc*, le principe étant :

> 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 $i \in \mathbb{N}$.

Ce principe est vérifié lorsque l'on utilise les notions de tailles associées au modèle de calcul car il n'existe pas de suite infinie strictement décroissante dans $\mathbb{N}$.

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.

1. Rappel du Chapitre 2 : dans le modèle RAM taille constante c'est la taille d'un mot binaire encodant l'entrée, dans le modèle RAM taille arbitraire c'est le nombre de cases mémoire occupées par l'entrée.

3.1.1 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 appel récursif. 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. Dans ce cours, le choix des cas de base doit satisfaire deux propriétés :

- toute séquence de simplifications mène à un cas de base, [2] et
- l'ensemble des cas de base est fini, [3]

Un traitement efficace des cas de base est une étape importante de la conception d'un algorithme récursif opérationnel. Cette étape est souvent non-triviale et spécifique à chaque problème. Pour séparer les difficultés, nous focalisons notre attention dans ce cours sur la *méthodologie* générale des algorithmes récursifs, c'est-à-dire l'étude des mécanismes de simplification-délégation. Par conséquent, on **ne détaillera pas** la manière dont les cas de base sont traités.

> Dans ce cours, pour décrire la manière dont les cas de base sont traités, il suffit de fixer une constante (par exemple 42) et d'utiliser l'instruction élémentaire `traiter directement les entrées de taille au plus 42`.

Le traitement des cas de base étant une instruction élémentaire, la manière dont il est réalisé n'affecte pas la complexité de l'algorithme.

3.1.2 Apprendre à déléguer

Soulignons que lorsque l'on conçoit un algorithme récursif, il *faut 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*.

3.1.3 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 :

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

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. [4] Pour voir ceci comme une simplification, on peut mesurer

2. Cette propriété est standard. On parle de récursion « bien fondée ».
3. Cette propriété est plus discutable. On pourrait imaginer un problème prenant en entrée un entier pour lequel les entrées paires soient faciles à traiter. C'est une hypothèse de confort afin de garantir que le traitement des cas de base peut être fait par une instruction élémentaire.
4. Une variante de cet algorithme simplifie (a,b) en $(a \mod b, b)$ si $a \geq b$ et en $(b \mod a, a)$ sinon.

la « simplicité » d'une entrée (a, b) par le nombre[5] $\max(a, b)$: 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 : par exemple, les entrées (a, b) telles que $\max(a, b) \leq 15$ sont elles traitées directement.

3.2 Recherche dichotomique (binary search)

Passons maintenant aux familles de mises en œuvre de cette méthodologie récursive. Voici la première :

> 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.

Comme annoncé, si la taille de l'entrée est inférieure à une constante t_0 donnée, on la résout directement.

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

3.2.1 Exemple

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

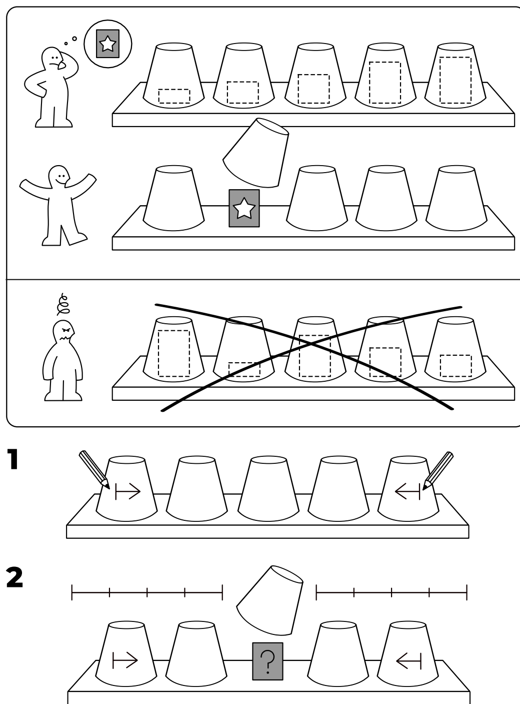
> RECHERCHE DANS UN TABLEAU TRIÉ
>
> **Entrée :** Un tableau $T[1..n]$ d'entiers tel que $T[1] < T[2] < \ldots < T[n]$ et un entier x.
>
> **Sortie :** L'indice i tel que $T[i] = x$ ou -1 si cet indice n'existe pas

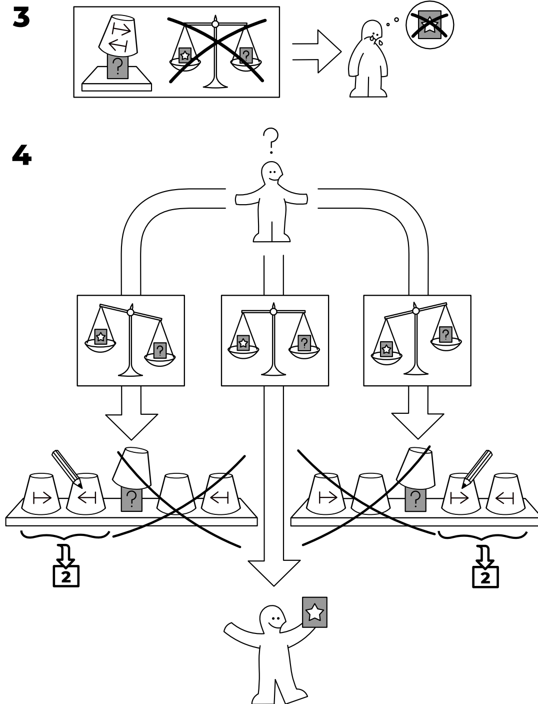
La solution classique de ce problème exploite le fait que pour tout indice p tel que $1 \leq p \leq n$, comparer x à $T[p]$ permet de déterminer que x n'est pas dans une partie du tableau, soit $T[1..p]$ soit $T[p..n]$. On a donc :

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

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.

5. Le nombre $a * b$ fonctionnerait tout aussi bien.





3.3 Diviser pour régner

Voici notre deuxième famille de mises en œuvre de la méthodologie récursive.

> Un algorithme de type **diviser pour régner** est un algorithme récursif qui
> (i) divise l'entrée donnée en plusieurs parties *plus petites* et *indépendantes* du même problème,
> (ii) délègue la résolution du problème sur chacune de ces parties, et
> (iii) combine les solutions à chacune de ces parties en une solution à l'entrée donnée.

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

3.3.1 Exemple : tri fusion

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

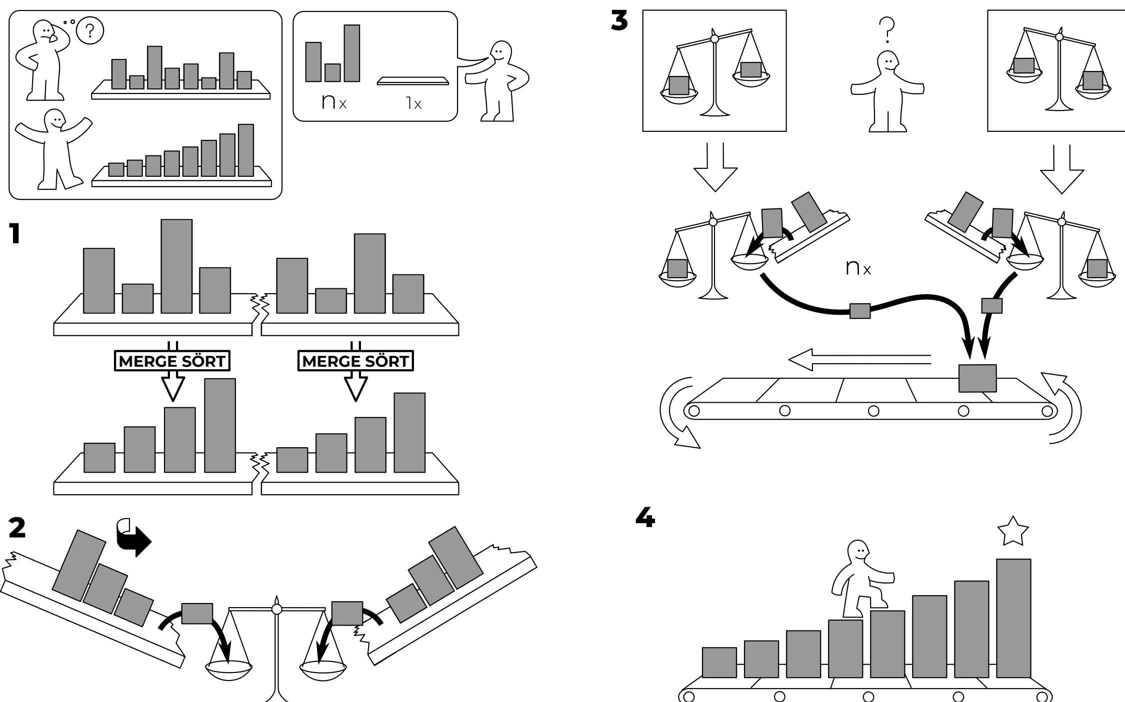
Tri

Entrée : Un tableau $T[1..n]$ d'entiers distincts.

Sortie : Un tableau $S[1..n]$ contenant les mêmes entiers que $T[1..n]$ mais dans l'ordre croissant.

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

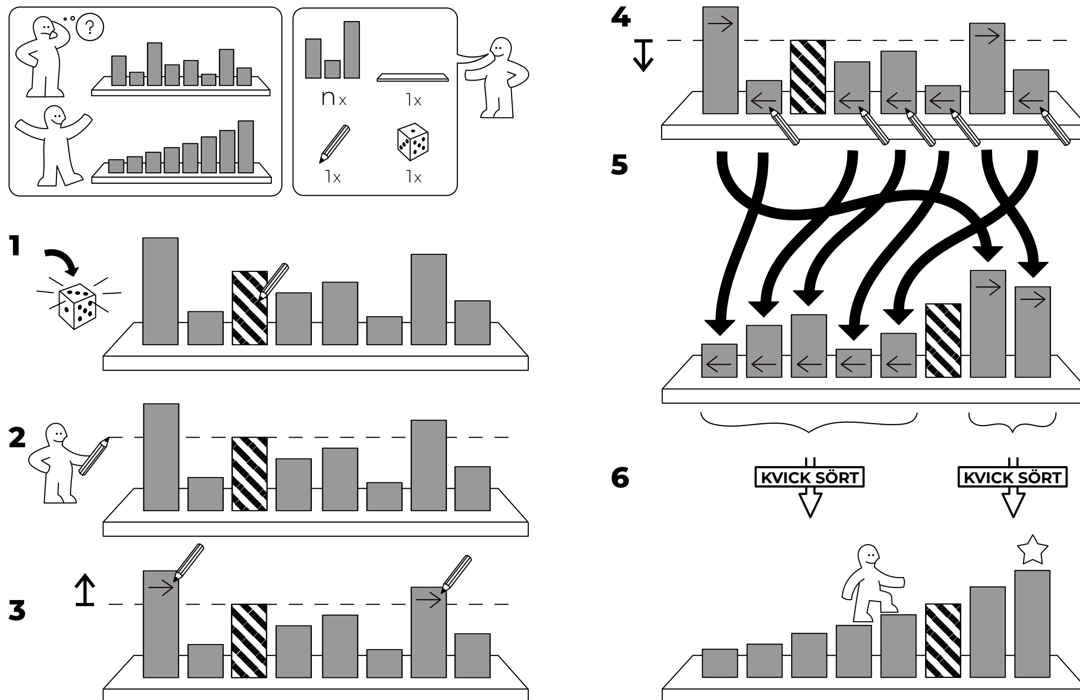
```
i=0, j=n/2 et k=0
tant que (i < n/2 et j < n):
    si T[i]<T[j]
    alors R[k] = T[i] et i = i+1
    sinon R[k] = T[j] et j = j+1
    k = k+1
si i == n/2 compléter R avec T[j..n-1]
sinon compléter R avec T[i..n/2-1]
copier R dans T
```



3.3.2 Exemple : tri rapide

Le **tri rapide** est un autre exemple classique d'algorithme « diviser pour régner ». Il résout le problème TRI en choisissant [6] un pivot p, divise $T[1..n]$ en deux parties, l'une contenant les éléments $\leq p$ et l'autre contenant les éléments $> p$. Il délègue le tri de ces parties, puis les concaténe.

6. 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 connaissez.



3.4 Exploration par retour arrière (backtracking)

Voici la troisième et dernière famille de mises en œuvre que nous étudions :

> Un algorithme d'**exploration par retour arrière** construit une solution à un problème algorithmique en
>
> (i) identifiant une décision élémentaire permettant de progresser dans la construction de la solution,
>
> (ii) identifiant la liste des choix possibles pour cette décision, et
>
> (iii) déléguant récursivement l'exploration de *chacun* de ces choix.

L'algorithme termine à l'étape (i) s'il ne reste plus de décision à prendre (auquel cas la solution est trouvée) ou à l'étape (ii) si aucun choix n'est possible pour la prochaine décision (auquel cas la séquence de décisions prises jusque là ne permet pas de construire une solution).

3.4.1 Exemple : problème des n reines

Voici un exemple classique de problème pour lequel une exploration par retour arrière est adaptée :

n REINES

Entrée : Un entier n.

Sortie : 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 $n = 4$.

Efforçons-nous de construire une solution à ce problème par une séquence de décisions élémentaires. Dans toute solution valide, chaque ligne contient *exactement une* reine. On peut donc choisir comme r-ième décision élémentaire « le placement de la reine de la r-ième ligne ». L'étape (ii) consiste alors à déterminer parmi les n cases de la r-ième ligne celles qui sont compatibles avec les choix effectués pour les $r - 1$ lignes précédentes ; l'algorithme doit, pour cela, connaître ces choix passés. Ensuite, pour chaque choix identifié comme possible à l'étape (ii), l'étape (iii) procède à un appel récursif. Cet appel récursif transmet l'information de toutes les décisions prises jusque là, *y compris* la position choisie sur la r-ième ligne dans le choix que cet appel récursif explore.

Pour mettre en œuvre cela, il reste à préciser la manière dont l'information des décisions prises est transmise. Dans le cas du problème 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
```

3.4.2 Commentaires

La résolution d'un problème au moyen d'une exploration par retour arrière suppose d'*identifier* une *séquence* de décisions **aussi élémentaires que possible** menant à la construction d'une solution. Plusieurs notions de « décision élémentaire » sont généralement envisageables, et 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 une 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**. Par confort, dans ce cours on se limitera à des algorithmes utilisant un nombre **borné** de cas de base, de sorte qu'ils 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 adaptée à la construction d'un objet déterminé par une séquence de décisions élémentaires. Un algorithme par backtracking choisit une décision élémentaire et, pour chaque choix possible, délègue la recherche d'un objet compatible avec ce choix ; chaque délégation décrémente ainsi le nombre de décisions élémentaires restant à prendre.

Notes personnelles

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 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 à cette séance que l'analyse de ce type de récurrence peut s'automatiser, mais que le traitement rigoureux de certaines petites complications (parties entières, petites fluctuations, analyse asymptotique, etc.) demande un petit effort.

Les analyses de complexité réalisées dans cette section s'appliquent à la fois à la complexité arithmétique (modèle RAM taille arbitraire) et à la complexité numérique (modèle RAM taille constante).

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

- utiliser des arbres enracinés pour modéliser,
- 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.

1. Précisons que ce point de vue sera légèrement différent de celui sous lequel vous avez abordé les arbre dans votre cours de premier semestre, celui de *type abstrait* d'une structures de données au sens du Complément E.

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 d'éléments de V. Un **graphe**[2] est une paire (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.

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 une paire $(V, \vec{E})$ 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)

2. Un graphe est parfois appelé **graphe non orienté**.

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.

est une arête orientée. La **longueur** d'un chemin $s_0, s_1, \dots, s_\ell$ 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 a au plus k^h nœuds et au plus k^{h-1} 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.

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 $A' \stackrel{\text{def}}{=} \star_{v_0}(A_1, A_2, \dots, A_k)$ l'arbre enraciné $(V', \vec{E}')$ avec

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

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
- $\star_{v_0}(A_1, A_2, \dots, A_k)$ pour tous $k \geq 1$, $v_0 \in V$ et $A_1, A_2, \dots, A_k \in A(V)$ 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 s'écrit $\star_{v_0}(A_1, A_2, \dots, A_k)$. Dans le second cas, on appelle $A_1, A_2, \dots, A_k$ 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.

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.

4.2 Arbres d'exécution

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

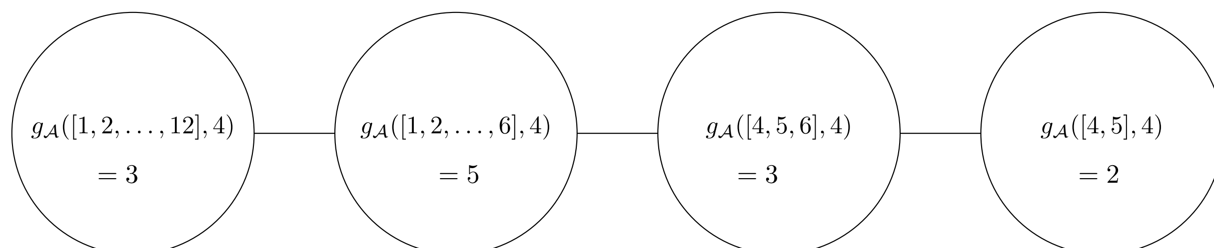
4.2.1 Arbre d'exécution d'une entrée

Considérons un algorithme récursif $\mathcal{A}$. Notons e une entrée et $t(e)$ sa taille[8]. 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 vus au Chapitre 3 : l'algorithme `rechercher` de recherche dichotomique d'un élément dans un tableau trié et l'algorithme `tri fusion` de tri d'un tableau d'entiers par fusion.

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 :



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 taille $\Theta(\log_2 t(e))$. La fonction $g_{\texttt{tri_fusion}}(e)$ est, elle, plus laborieuse à expliciter.

4.2.2 Arbre d'exécution d'un algorithme

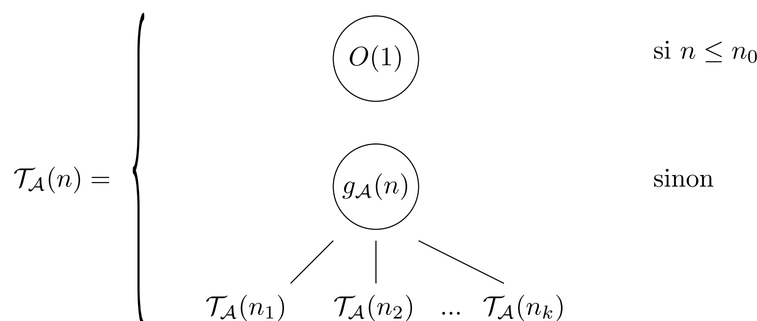
Les arbres que l'on vient de définir constituent une application de l'ensemble des entrées de l'algorithme dans l'ensemble des arbres (à nœuds étiquetés par $\mathbb{N}$). Comme pour la fonction de complexité, il est souhaitable de simplifier cet objet en (i) agrégeant les entrées en paquets en considérant le pire cas parmi les entrées d'une taille n donnée, puis (ii) en passant d'un comptage précis des instructions élémentaires à une estimation asymptotique pour $n \to \infty$.

8. Rappellons que la notion de taille dépend du modèle : taille d'encodage en RAM taille constante et nombre de cases mémoire occupées en RAM taille arbitraire.

Commençons par la simplification (i), 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 les cas de base). [9] 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 (ii), on remplace, dans l'étiquetage, les fonctions précises $\max\{g_\mathcal{A}(\cdot) \colon \ldots\}$ par leur comportement asymptotique, que l'on note $g_\mathcal{A}(n)$.



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

4.2.3 Premiers exemples

Commençons par l'algorithme `rechercher` de recherche dichotomique d'un élément dans un tableau trié de taille n. Quelque soit l'entrée, la récursion se fait par un seul appel, donc $\mathcal{T}_{\texttt{rechercher}}(n)$ 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 $p \approx \log_2(n/n_0) = O(\log_2 n)$. Ainsi, $\mathcal{T}_{\texttt{rechercher}}(n)$ est un chemin de longueur $O(\log_2 n)$. À chaque étape de la récursion, le coût de simplification est $O(1)$. Chaque nœud de ce chemin est donc étiqueté $O(1)$.

Considérons maintenant l'algorithme `tri_fusion` de tri d'un tableau d'entiers. Quelque soit l'entrée (un tableau de taille n), l'algorithme la coupe 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 $O(\log_2 n)$, 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 $O(n)$.

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`,

9. Cette approximation se fait sans perte de généralité, puisque (i) on avait posé comme condition, de confort, que le nombre de cas de base est borné, et (ii) toute entrée de taille bornée peut être traitée en nombre borné d'instructions élémentaires.

10. C'est là que 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, 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.

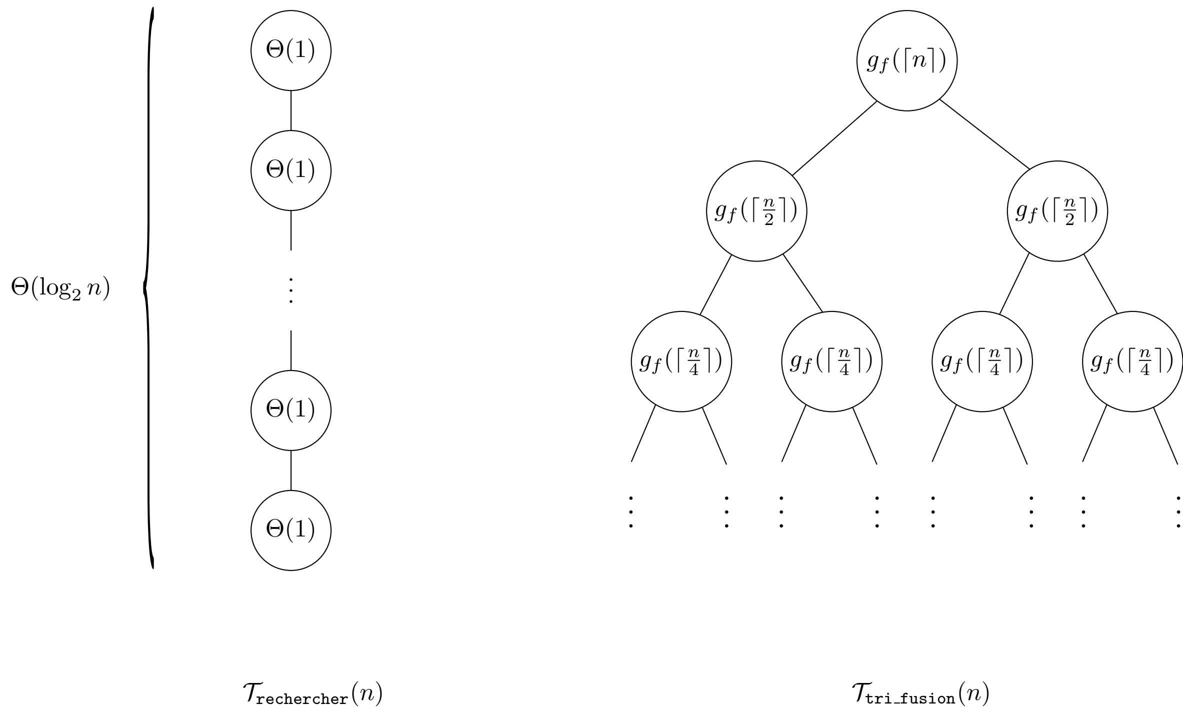


FIGURE 4.1 – Arbres d'exécution pour la recherche dichotomique (à droite) et le tri fusion (à gauche).

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.

4.3 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. Cette section présente une première analyse pas complètement rigoureuse[13]. Cela nous conduira à formuler un premier théorème de structure (le théorème maître) dont on établira rigoureusement une généralisation (le théorème d'Akra-Bazzi).

4.3.1 Le principe

Notons $\mathrm{val}(v)$ 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

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.

13. Ce qui est affirmé dans cette section est vrai mais certaines affirmations ne sont pas complètement justifiées.

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 1 à $h = O(\log n)$, où h est la hauteur de l'arbre. Pour $1 \leq p \leq h$, il y a 2^p nœuds de profondeur p (cela se prouve par récurrence). Chaque nœud de profondeur p a 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.* [14]

4.3.2 Cas des récursions uniformes

Dans beaucoup d'algorithmes récursifs, le nombre k d'appels récursifs est le même à chaque niveau de la récursion. De même, il est courant que chaque appel récursif porte sur une fraction *constante* de l'entrée.

> 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.

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 $g_{\mathcal{A}}(\lceil \alpha^p n \rceil)$.

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.*

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

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 : $C_{\mathcal{A}}(n) = O\left(k^h\right) = O\left(k^{\log_{\frac{1}{\alpha}} n}\right) = O\left(n^{\log_{\frac{1}{\alpha}} k}\right)$.

4.4 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 ? Commençons par examiner cela sur l'exemple de l'algorithme de `tri_rapide` introduit au Chapitre 3.

Cet algorithme dépend d'un choix de pivot ; supposons 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. [15] 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 constante K. [16] 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.

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 :

15. 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.

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

Un algorithme $\mathcal{A}$ est de complexité $\Omega(f(n))$ s'il existe une suite $\{i_j\}_{j\in\mathbb{N}}$ strictement croissante d'entiers positifs et une suite d'entrées $\{e_{i_j}\}_{j\in\mathbb{N}}$ du problème résolu par $\mathcal{A}$ tels que (i) chaque entrée e_{i_j} est de taille i_j, et (ii) le traitement de e_{i_j} par $\mathcal{A}$ exécute au moins $K \cdot f(i_j)$ instructions élémentaires, où K est une constante indépendante 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 $\mathcal{A}$ (souvent abrégé en *une 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.

4.5 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 logarithmique ». On fait cela au moyen de la notation $\tilde{O}()$:

- $f = \tilde{O}(g)$ signifie qu'il existe une constante c telle que $f(n) = O(g(n)\log^c n)$,
- $f = \tilde{\Omega}(g)$ signifie qu'il existe une constante c telle que $f(n) = \Omega(g(n)\log^c n)$,
- $f = \tilde{\Theta}(g)$ signifie qu'il existe une constante c telle que $f(n) = O(g(n)\log^c n)$ et une constante d telle que $f(n) = \Omega(g(n)\log^d n)$. Ces constantes c et d peuvent être différentes.

Ces notations étant posées, voici le **théorème maître** :

Théorème 4.5.1. *Soit k un entier positif, $\alpha \in (0,1)$ un réel et d un réel positif. Si $\mathcal{A}$ traite toute entrée de taille n suffisamment grande par k appels récursifs, chacun sur une entrée de taille $\lceil \alpha n \rceil$, avec $g_{\mathcal{A}}(n) = \tilde{\Theta}\left(n^d\right)$, alors*

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

Le principe de la démonstration est que de de $g_{\mathcal{A}}(n) = \tilde{\Theta}\left(n^d\right)$ on tire $k^p g_{\mathcal{A}}\left(\lceil \alpha^p n \rceil\right) = \left(k\alpha^d\right)^p \cdot \tilde{\Theta}\left(n^d\right)$. La comparaiton entre $k\alpha^d$ et 1 permet alors d'identifier l'un des trois régimes précédents (décroissance exponentielle, quasi-constance et croissance exponentielle). On ne détaille pas cette preuve plus avant, mais on en prouve une version plus forte.

4.6 Prolongement : le théorème d'Akra-Bazzi

Concluons ce chapitre par un aperçu d'un outil actuel d'analyse d'algorithme. Cette section n'est proposée que comme un prolongement non-exigible, et peut être librement ignorée (sous réserve d'accepter le théorème maître sans preuve).

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. L'esquisse de preuve du théorème maître semble bien évidemment laisser de la place à un peu de souplesse... mais jusqu'à quel point ? Une des meilleures réponses connues à ce jour est le théorème suivant, dû à Akra et Bazzi :

Théorème 4.6.1. *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

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

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, que l'on considère comme sortant du cadre de ce cours, est un solide exercice d'analyse réelle. On la présente en Complément F 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 $\leq k^h$ nœuds et $\leq k^{h-1}$ 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.
- 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 un majorant 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**.

Notes personnelles

Chapitre 5

Cas d'étude : Systèmes de vote

Cette séance constitue un intermède entre les séances méthodologiques d'algorithmique (3 et 4) et celles de théorie de la complexité (6, 7 et 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

- modéliser une méthode de comptage comme une fonction de choix social
- analyser une propriété d'une fonction de choix social, y compris quand cette fonction est donnée par un algorithme qui la calcule,
- adapter les algorithmes de calcul efficace de fonctions de choix social décrites ici à des variantes de ces fonctions.

5.1 Problématique

On appelle ici *système électoral* une méthode permettant de déterminer une préférence collective à partir de l'expression des préférences individuelles de plusieurs individus. Les problématiques qui nous intéressent sont la définition et l'analyse d'un système électoral.

5.1.1 Un objectif à définir

Supposons à titre d'exemple que l'on organise une consultation des habitant·es 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 restreint le champ à trois options :

(a) Supprimer des voies de circulation motorisée pour créer des autoroutes pour vélo [1].

(b) Supprimer des espaces de stationnement pour créer des autoroutes pour vélo.

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

1. Double-voies vélo séparées des voies de circulation motorisée.

Supposons que l'on ait acté de mettre en œuvre l'une de ces trois options, et que la consultation ait pour objectif de déterminer laquelle.

Une solution naturelle pour mettre en place cette consultation consiste à demander à chaque votant·e de choisir une option, puis de retenir l'option qui obtient le plus grand nombre de suffrages. C'est ce que l'on appelle un **vote majoritaire**. Cette solution peut conduire à une décision mal acceptée[2], aussi voyons comment on peut envisager d'améliorer cela.

Supposons que l'on demande à chaque votant· d'ordonner les trois choix par ordre de préférence décroissante (sans autoriser d'ex æquo : chaque classement doit être strict). Cela autorise donc 6 votes possibles. Supposons que 100 000 votant·s se soient exprimé·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

Ainsi, aucun choix ne recueille de majorité absolue.

Un **vainqueur de Condorcet** est un choix (x) tel que pour tout autre choix (y), une majorité des votant·es préfère (x) à (y). Lorsqu'un vainqueur de Condorcet existe, il est unique et est généralement considéré comme un résultat acceptable. Soulignons qu'un tel vainqueur peut ne pas exister : dans notre exemple, 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). Autrement dit, les comparaisons des choix deux à deux peuvent ne pas produire un ordre ; ce phénomène est connu sous le nom de **paradoxe de Condorcet**.

Examinons quelques méthodes définissant, à partir d'une expression de préférences individuelles, que l'on appelle dorénavant **suffrage**, une préférence collective, que l'on appelle dorénavant **résultat**.[3] À ce stade, si certaines méthodes sont données par des algorithmes qui calculent efficacement le résultat à partir du suffrage, d'autres sont moins effectives. On ne se souciera de ces questions d'algorithmes, de complexité et d'effectivité qu'à partir de la Section 5.3.

5.1.2 Discussion informelle de quelques méthodes de comptage

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.[4] 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), d'une courte avance.

La **méthode de Dodgson**[5] compte, pour chaque choix, le nombre minimum de transpositions[6] 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

2. 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.

3. Bien entendu, mieux vaut décider de la définition *avant* de procéder au vote...

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

5. Mieux connu sous son nom de plume, Lewis Caroll.

6. Changer un vote « a>b>c » en « c>b>a » requiert deux transpositions.

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.

Ces méthodes ont des propriétés *induites* assez différentes. Par exemple, lorsqu'un suffrage admet un vainqueur de Condorcet, c'est aussi le résultat de la méthode de Dodgson. Ce n'est pas le cas pour la méthode de Borda.[7] On peut de même se demander si ces différentes méthodes de comptage peuvent, dans certains cas, avoir pour résultat un *perdant de Condorcet*, c'est-à-dire un choix qui perd tous ses « duels » ; c'est possible pour la méthode de Borda, mais pas pour celle de Dodgson.

Plus généralement, on peut s'interroger sur le fait que telle ou telle méthode incite les votant·es à mentir sur leurs préférences afin d'améliorer le résultat de leur point de vue : dans le comptage de l'exemple ci-dessus par la méthode de Borda, si 1 001 votant·es préférant « a>b>c » votaient « b>a>c », le résultat basculerait de (c) à (b), ce qui serait mieux de leur point de vue. On peut considérer qu'un tel phénomène de « vote utile » dénature la consultation. On va cependant voir qu'il est mathématiquement inévitable.

5.2 Fonctions de choix/d'ordre social

Examinons maintenant une formalisation mathématique des systèmes électoraux et quelques propriétés que l'on peut en déduire.

5.2.1 Formalisme

> On modélise les **choix** entre lesquels les votants doivent se prononcer par un ensemble fini, que l'on note C.

Ainsi, dans notre exemple introductif cycliste, $C = \{a, b, c\}$. Pour le premier tour de l'élection présidentielle 2022 en France l'ensemble des choix était

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

> On modélise un **vote** par un ordre total sur C. On note V l'ensemble des votes possibles.

Ainsi, dans notre exemple introductif cycliste, les valeurs possibles d'un vote sont

$$V = \{a < b < c, a < c < b, b < a < c, b < c < a, c < a < b, c < b < a\}.$$

Dans le cas de l'élection présidentielle 2022 en France, puisqu'il y a 12 candidat·es, l'ensemble V est de taille $12!$. Étant donné un vote $v \in V$ et deux choix x et y, on note $x \prec_v y$ si le vote v préfère le choix x au choix y. On note $\max(v)$ le choix préféré du vote v.

7. Par exemple, si on remplace le suffrage de la Section 5.1.1 par $[20, 12, 8, 25, 17, 18]$, dans l'ordre, alors (b) devient un vainqueur de Condorcet et le résultat de la méthode de Borda reste (c).

Un **profil de vote** (aussi appelé un **suffrage**) est un vecteur $\Pi \in V^n$, pour un entier $n \geq 2$. L'entier n est le nombre de votes du profil Π.

Un profil de vote modélise l'expression de n votant·es. Par exemple, le profil de vote associé à notre exemple introductif cycliste est

$$\Pi = (\underbrace{a < b < c, \ \dots, \ a < b < c}_{22\ 000 \text{ copies}}, \underbrace{a < c < b, \ \dots, \ a < c < b}_{12\ 000 \text{ copies}}, \underbrace{b < a < c, \ \dots, \ b < a < c}_{6\ 000 \text{ copies}},$$
$$\underbrace{b < c < a, \ \dots, \ b < c < a}_{27\ 000 \text{ copies}}, \underbrace{c < a < b, \ \dots, \ c < a < b}_{17\ 000 \text{ copies}}, \underbrace{c < b < a, \ \dots, \ c < b < a}_{16\ 000 \text{ copies}}).$$

Étant donné $\Pi \in V^n$, on note Π_i l'ordre total exprimé par le $i$ème vote. Pour les considérations algorithmiques, on représentera un profil de vote par un tableau $\Pi[1..n]$ dont chaque entrée est un vote.

Une **fonction de choix social** f est une fonction $f : V^n \to C$.

Autrement dit, une fonction de choix social associe à tout suffrage un, et un seul, des choix proposés au vote. Un exemple de fonction de choix social est la fonction qui associe à tout profil de vote le vainqueur par la méthode de Borda vue ci-dessus. Remarquons que l'on a ici défini une fonction (de choix social) au moyen d'un algorithme qui la calcule (la méthode de Borda).

Une **fonction d'ordre social** est une fonction $f : V^n \to V$.

Autrement dit, une fonction d'ordre social associe à tout suffrage un ordre sur les choix proposés au vote. Un exemple est donné par la fonction qui associe à tout suffrage la liste ordonnée des choix, classés par ordre de score de Borda décroissants[8].

5.2.2 Propriétés et résultats d'impossibilité

Pour décider de la fonction de choix/d'ordre social à utiliser, il peut être utile de comprendre les propriétés des unes et des autres. Voyons quelques exemples de propriétés facilement exprimables dans ce formalisme. On se concentre ici sur le cas des fonctions de choix social mais cela s'appliquerait de même aux fonctions d'ordre social.

Une fonction de choix social f est **dictatoriale** s'il existe un entier $1 \leq i \leq n$ tel que pour tout $\Pi \in V^n$ et tout $x \in C$, si $f(\Pi_1, \Pi_2, \dots, \Pi_n) = x$ alors $x = \max \Pi_i$. Autrement dit, le résultat de l'élection coïncide[9] systématiquement avec la préférence d'un·e votant·e prédéfini·e.

Une fonction de choix social f est **manipulable** s'il existe $\Pi \in V^n$, un entier $1 \leq i \leq n$, et $v \in V$ tels que le ou la $i$ème votant·e aurait intérêt à voter $v \neq \Pi_i$ pour améliorer le résultat de son point de vue, c'est-à-dire tels que

$$f(\Pi_1, \Pi_2, \dots \Pi_{i-1}, v, \Pi_{i+1}, \dots, \Pi_n) \prec_{\Pi_i} f(\Pi). \tag{5.1}$$

Il est utile de pouvoir décrire de quelle manière une fonction f est manipulable ; on dit que **remplacer Π_i par v dans Π manipule f** pour signifier que l'équation (5.1) est vraie.

8. Pour que ce soit un ordre total il convient de préciser comment sont départagés les ex æquos.
9. Soulignons que c'est un constat purement descriptif. On ne cherche pas à modéliser de lien de causalité.

Une fonction de choix social est **unanime** si pour tout $\Pi \in V^n$, si $\max \Pi_1 = \max \Pi_2 = \ldots = \max \Pi_n = x \in C$ alors $f(\Pi) = x$. Autrement dit, quand tous les votant·es classent le même choix premier, c'est ce choix qui est élu.

On peut remarquer que les fonctions (modélisant les méthodes) de Borda et de Dodgson ont les avantages d'être non-dictatoriales et unanimes, mais ont l'inconvénient d'être manipulables. Il s'avère que ces trois avantages sont incompatibles :

Théorème 5.2.1 (Gibbard–Satterthwaite)**.** *Pour $|C| \geq 3$ choix et des votes sans indifférence, toute fonction de choix social qui est unanime et non-manipulable est dictatoriale.*

Un énoncé similaire s'applique aux fonctions d'ordre social (Théorème d'Arrow). Ces résultats admettent des preuves élémentaires, qui exploitent astucieusement l'unanimité et la non-manipulabilité de la fonction pour déterminer ses valeurs. On renvoit à l'article de Sen [Sen01] pour une preuve simple du Théorème 5.2.1. On en tire un principe important en théorie du choix social :

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

5.3 L'apport d'un regard algorithmique

L'algorithmique et la complexité, nos sujets principaux, éclairent ces questions (fonctions d'ordre et de choix social, propriétés, résultats d'impossibilité) de plusieurs manières :

- Formellement, toute fonction de choix/d'ordre social est un problème algorithmique. [10] On peut donc se poser, sur ces fonctions, les questions que l'on se pose sur les problèmes algorithmiques usuels : quels algorithmes pour les calculer ? Avec quelle complexité ? . . . [11]
- Il arrive qu'une fonction de choix/d'ordre social soit donnée sous la forme d'un algorithme qui la calcule. C'est par exemple le cas de notre description de la méthode de Borda. Dès lors, étudier les propriétés de cette fonction s'apparente à de l'analyse d'algorithmes. [12]
- La théorie de la complexité peut offrir des manières de contourner les résultats d'impossibilité. Par exemple, le théorème de Gibbard–Satterthwaite n'exclut pas l'existence d'une fonction de choix social sur $|C| \geq 3$ choix qui soit unanime, non-dictatoriale et *difficile* à manipuler, au sens où il n'existe pas d'algorithme efficace pour calculer une manipulation.

Nous allons désormais examiner ces fonctions d'un œil plus algorithmique.

5.4 Quelques méthodes de comptage

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

10. L'espace des entrées est l'ensemble des profils de votes et l'espace de sorties est l'ensemble des choix / votes.
11. Par exemple, la fonction de Dodgson est inintéressante en pratique car son calcul est NP-difficile (cf remarque en début de la Section 5.4).
12. Dans le même esprit que le Théorème 1.5.1 d'optimalité du mariage calculé par l'algorithme de Gale-Shapley.

Nous anticipons par ailleurs sur les deux séances qui suivent, et signalons certains problèmes comme *NP-difficiles*. En attendant de définir cette notion au Chapitre 8, elle est à comprendre informellement comme synonyme de « n'admettant vraisemblablement pas d'algorithme de complexité polynomiale ».

On considère dans ce qui suit une élection d'ensemble de choix $C = \{c_1, c_2, \ldots, c_m\}$ et un profil de votes $\Pi \in V^n$. 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 $1 \leq i, j \leq m$, $i \neq j$, on définit $d(i, j)$ comme le nombre de votant·es préférant c_i à c_j. Autrement dit,

$$d(i,j) \stackrel{\text{def}}{=} |\{s \colon 1 \leq s \leq n \text{ et } c_i \prec_{\Pi_s} c_j\}|\,.$$

On note $D = (d(i,j))_{1 \leq i,j \leq n}$ la **matrice des duels**. On note G_Π le graphe orienté ayant un sommet pour chaque choix de C et une arête orientée de chaque sommet i à chaque sommet j étiquetée par $d(ii, j)$. En particulier, le graphe non-orientié sous-jacent à G_Π est complet. La matrice D et le graphe G_Π sont calculables en temps $O(m^2 n)$.

5.4.1 Méthodes de Copeland et Minimax

La **règle de Copeland** est une fonction de choix social. Elle associe à tout choix c_i un score défini comme le nombre de choix c_j, $j \neq i$, tels que $d(i,j) > d(j,i)$. Le résultat est le choix de score maximal, s'il existe. Un algorithme naïf calcule cette fonction en temps $O(m^2)$ à partir de la matrice D. Un de ses inconvénients majeurs est qu'elle produit souvent des ex æquos.

Les **méthodes minimax** sont des fonctions de choix social qui examinent pour chaque choix c_i son moins bon duel, que l'on peut définir de plusieurs manières :

- $\text{def}(i) \stackrel{\text{def}}{=} \min\{d(i,j) \colon j \neq i\}$, ou
- $\text{def}(i) \stackrel{\text{def}}{=} \min\{0\} \cup \{d(i,j) \colon j \neq i, d(i,j) < d(j,i)\}$, ou encore
- $\text{def}(i) \stackrel{\text{def}}{=} \min\{d(i,j) - d(j,i) \colon j \neq i\}$.

Le résultat est alors le choix c_i qui maximise $\text{def}(i)$. Un algorithme naïf calcule ces fonctions en temps $O(m^2)$ à partir de la matrice D. Ces méthodes minimax échouent à satisfaire de nombreuses propriétés.

5.4.2 Méthode de Kemeny-Young

La **règle de Kemeny-Young** est une fonction d'ordre social. Son résultat est déterminé comme l'ordre 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.[13] Autrement dit, à tout ordre strict r sur C on associe le désaccord

$$\text{des}(r) \stackrel{\text{def}}{=} \sum_{k=1}^{n} \sum_{1 \leq i \neq j \leq m} \mathbb{1}_{c_i \prec_{\Pi_k} c_j \text{ et } c_j \prec_r c_i} = \sum_{1 \leq i \neq j \leq m} \sum_{k=1}^{n} \mathbb{1}_{c_i \prec_{\Pi_k} c_j \text{ et } c_j \prec_r c_i} = \sum_{i,j \colon c_j \prec_r c_i} d(i,j).$$

Le résultat est l'ordre de désaccord minimal, s'il est unique.

Calculer séparément les nombres d'inversions pour chaque ordre r et chaque vote Π_k donne, avec l'algorithme récursif de calcul de nombre d'inversions vu en TD, une complexité totale de $O(nm!m \log m)$. Calculer d'abord la matrice des duels puis l'utiliser comme indiqué ci-dessus

13. Rappelons qu'une *inversion* entre deux ordres $r, s \in V$ est une paire $x, y \in C$ telle que $x \prec_r y$ et $y \prec_s x$. La distance entre permutation définie par le nombre d'inversions est parfois appelée *distance de Kendal-τ*.

donne une complexité totale de $O\left(m^{O(1)}n + m!m^{O(1)}\right)$, la détermination des facteurs polynomiaux étant laissée en exercice.

La méthode de Kemeny-Young a l'inconvénient d'être difficile à calculer exactement quand le nombre de candidats est élevé. Ces deux complexités sont linéaires en le nombre de votant·es mais sont plus qu'exponentielles en le nombre de choix. On peut améliorer cela [14] mais il est peu probable qu'il existe un algorithme de complexité polynomiale en le nombre de choix, même à nombre de votants constant. [15]

5.4.3 Méthode des paires ordonnées

La **méthode des paires ordonnées** est une fonction de choix social. Elle est calculée à partir du sous-graphe orienté G' de $G \stackrel{\text{def}}{=} G_\Pi$ obtenu par le procédé suivant :

```
G'= graphe orienté de sommets {1,2, ..., m} 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
```

Rappelons que le poids d'une arête (i, j) de G est le nombre $d(i, j)$ de votant·es qui préfèrent le choix c_i au choix c_j. En cas d'égalité entre $d(i, j)$ et $d(i', j')$, il est courant de privilégier l'arête ayant la plus petite inverse (c'est-à-dire (i, j) si $d(j, i) < d(j', i')$, et (i', j') sinon). On suppose dans la suite que les $d(i, j)$ sont deux à deux distincts.

Cette construction assure que G' est acyclique. En particulier, G' contient *au moins une* source. [16] 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.

On a vu en exercice un algorithme décidant 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 $O(m^4)$, ce à quoi il faut ajouter la complexité $O(m^2 n)$ 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 $O(m^4 + m^2 n)$.

14. En particulier, reformuler l'évaluation du résultat de la règle de Kemeny-Young comme le calcul d'un ensemble d'arcs de retours (« *feedback arc set* ») dans un tournoi, puis utiliser un algorithme de programmation dynamique comme l'algorithme de Held-Karp, donne une complexité de $O(m2^m + n)$.

15. Formellement : il est NP-difficile (relativement à m) de décider, étant donné un profil de 4 votes et un seuil S, si le résultat r de la règle de Kemeny-Young satisfait des$(r) \leq S$.

16. 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.

5.4.4 Méthode de Schulze

La **méthode de Schulze** est une fonction de choix social qui amende les méthodes minimax. On construit le sous-graphe orienté G' de G_Π obtenu en ne gardant que les arêtes $i \to j$ telles que $d(i,j) > d(j,i)$. La *force* $\delta(i,j)$ de l'arête $i \to j$ peut être définie de plusieurs manières :

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

(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 :

Lemme 5.4.1. *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 $a \to c$. $\square$

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.

Examinons cette méthode sur le plan algorithmique. Le calcul du graphe G' et des poids $\delta(\cdot,\cdot)$ peut facilement se faire en temps $O(m^2)$. Le calcul de G'' demande, en revanche, de calculer la force maximale des chemins entre *toute* paire de sommets. On peut calculer les $\binom{m}{2}$ forces en temps $O(m^3)$ par une adaptation simple de l'*algorithme de Floyd-Warshall* de calcul de distances dans un graphe pondéré.

Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un exemple d'application d'une méthodologie récursive appelée *programmation dynamique*. On a en entrée un graphe orienté G'' de sommets $[m] \stackrel{\text{def}}{=} \{1, 2, \ldots, m\}$ 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 $[m]^2$,

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

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

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

Comme $f(i,j) = g(i,j,m)$, 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 :

$$\forall k \geq 2, \quad g(i,j,k) = \max\{g(i,j,k-1), \min(g(i,k,k-1), g(k,j,k-1))\}. \tag{5.2}$$

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

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

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

5.5 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 5.2.1 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].

5.6 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.

- Les méthodes de comptage de votes peuvent se modéliser par des fonctions de choix/d'ordre social.
- Les propriétés de fonctions de choix/d'ordre social ont été très largement étudiées. Certaines propriétés sont mutuellement exclusives.
- L'intérêt pratique d'une méthode de comptage dépend de ses propriétés et de la complexité de son calcul.

Notes personnelles

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 d'exemples.

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 simple, 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, dans le modèle 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 demander 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 $\Omega(f(n))$.

On a donc **deux notions distinctes** de bornes inférieures : pour un algorithme et pour un problème.

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.

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.

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

Pour prouver qu'une fonction $f(n)$ est une borne inférieure pour un problème P, une première technique consiste à supposer donné un algorithme $\mathcal{A}$ résolvant le problème considéré en temps $o(f(n))$, et à prouver que cet algorithme ne peut pas être correct sur toutes les entrées.

6.2.1 Un exemple simple : TOUS DISTINCTS

Considérons un premier exemple [3] pour lequel l'entrée incorrecte est facile à construire.

TOUS DISTINCTS

Entrée : Un tableau $T[1..n]$ d'entiers.

Sortie : Vrai ou faux, ces entiers sont deux à deux distincts.

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 une paire (i, j) telle que i apparaît avant j dans l'un et après j dans l'autre.

3. Ce problème apparaît naturellement lorsque l'on souhaite vérifier qu'une liste ne contient pas de doublon, ou qu'une entrée n'apparaît qu'une seule fois dans une série d'enregistrements.

Un algorithme naïf résout ce problème en temps $\Theta(n^2)$. Une meilleure solution consiste à trier $T[1..n]$ par `tri_fusion`, puis à examiner toutes les paires d'indices *consécutifs* à la recherche d'éléments égaux. Cet algorithme est de complexité [4] $\Theta(n \log n)$. Peut-on faire mieux ?

Il semble naturel que la complexité de TOUS DISTINCTS dans le modèle RAM taille arbitraire soit $\Omega(n)$: on conçoit mal qu'un algorithme qui n'examine pas toutes les cases du tableau d'entrée puisse conclure correctement dans tous les cas. Cela peut se formaliser ainsi :

Proposition 6.2.1. *Le problème* TOUS DISTINCTS *est de complexité* $\Omega(n)$ *dans le modèle RAM taille arbitraire.*

Démonstration. Supposons donné un algorithme $\mathcal{A}$ dans le modèle RAM taille arbitraire tel que $\mathcal{A}$ résout le problème TOUS DISTINCTS en temps $o(n)$. Montrons qu'il existe une entrée sur laquelle $\mathcal{A}$ se trompe.

Fixons un entier $n \geq 2$ et posons $U[1..n] = [1, 2, \ldots, n]$. Exécutons $\mathcal{A}$ sur l'entrée $U[1..n]$ et notons $L(n)$ l'ensemble des indices de cases de U lues par $\mathcal{A}$ au cours de son exécution : [5]

$$L(n) \stackrel{\text{def}}{=} \{i \in [n] \colon \text{ l'exécution de } \mathcal{A} \text{ sur } U[1..n] \text{ accède à } U[i]\}.$$

Notons $\ell(n)$ la taille de l'ensemble $L(n)$.

Notons $c_{\mathcal{A}}(U[1..n])$ le nombre d'instructions élémentaires réalisées par $\mathcal{A}$ pour traiter l'entrée $U[1..n]$. Chaque instruction élémentaire lit $O(1)$ cases mémoire, aussi $\ell(n) = O(c_{\mathcal{A}}(U[1..n]))$. L'hypothèse que $\mathcal{A}$ est de complexité $o(n)$ assure que $\ell(n) = o(n)$. Par conséquent, il existe n_0 tel que pour tout $n \geq n_0$, on a $\ell(n) < n$. Ainsi, il existe une case de $U[1..n_0]$ que $\mathcal{A}$ n'a pas lu au cours du traitement de cette entrée ; notons i_0 l'indice de cette case. Définissons alors un tableau $T[1..n_0]$ par

$$T[i] = \begin{cases} i & \text{si } i \neq i^* \\ i_0 + 1 & \text{si } i = i_0 \text{ et } i_0 < n \\ i_0 - 1 & \text{si } i = i_0 \text{ et } i_0 = n \end{cases}$$

Les tableaux $U[1..n_0]$ et $T[1..n_0]$ diffèrent uniquement en la case d'indice i_0. L'exécution de $\mathcal{A}$ sur ces deux entrées produit donc le même résultat (puisque $\mathcal{A}$ ne lit pas la case d'indice i_0). Or la réponse doit être différente. L'algorithme se trompe donc sur au moins une de ces entrées. □

6.2.2 Un exemple plus subtil : RECHERCHE DANS UN TABLEAU TRIÉ

Intéressons-nous maintenant au problème :

RECHERCHE DANS UN TABLEAU TRIÉ

Entrée :	Un tableau $T[1..n]$ d'entiers tel que $T[1] \leq T[2] \leq \ldots \leq T[n]$ et un entier x.
Sortie :	Un indice i tel que $T[i] = x$ ou -1 si cet indice n'existe pas

On a vu au Chapitre 3 qu'il peut être résolu par recherche dichotomique. [6] Il devrait être clair, au minimum depuis le Chapitre 4, que cet algorithme est de complexité $O(\log n)$. Est-ce possible de faire mieux ?

4. Ici on utilise que le coût de traitement par `tri_fusion` de n'importe quelle entrée de taille n est $\Omega(n \log n)$.
5. Soulignons qu'on énonce ici juste l'existence d'un tel ensemble $L(n)$, et qu'on n'a pas besoin que notre modèle de calcul permette de le *calculer*.
6. Le problème du Chapitre 3 était légèrement différent en ce qu'il demandait que le tableau soit strictement croissant. L'algorithme de recherche dichotomique s'adapte sans difficulté.

Contrairement à l'exemple précédent, un algorithme peut tout à fait résoudre Recherche dans un tableau trié sans lire toute l'entrée (c'est ce que fait la recherche dichotomique). Pour montrer qu'aucun algorithme $\mathcal{A}$ ne peut résoudre ce problème en $o(\log n)$, on va réaliser l'expérience de pensée suivante : [7]

> Imaginons qu'à chaque fois que $\mathcal{A}$ accède à une case de l'entrée (un $T[i]$), il adresse une demande (« *que vaut* $T[15]$ *?* ») à un autre algorithme $\mathcal{B}$ qui lui répond (« 42 »).

Cette expérience de pensée donne la possibilité à $\mathcal{B}$ de *choisir* l'entrée en fonction des requêtes reçues. Bien entendu, ces choix doivent être cohérents au sens où il doit exister au moins une entrée de l'algorithme cohérente avec les réponses faites par $\mathcal{B}$. Ainsi, lors de cette expérience de pensée, $\mathcal{B}$ guide l'algorithme $\mathcal{A}$ de la même manière que le ferait toute entrée cohérente avec ses choix.

Illustrons cette idée sur la recherche de $x = 1$ dans un tableau trié ne comportant que trois valeurs possibles (0, 1 et 2). Supposons qu'un algorithme $\mathcal{A}$ résout Recherche dans un tableau trié et est de complexité $o(\log n)$. On initialise $\mathcal{B}$ avec un tableau $U[0..n+1] = [0, -1, -1, \dots, -1, 2]$ et on lance l'exécution de $\mathcal{A}$ avec $x = 1$. Chaque fois que $\mathcal{A}$ accède en lecture à une case de l'entrée, disons $T[i]$, l'indice $i \in \{1, \dots, n\}$ est transmis à $\mathcal{B}$ qui répond comme suit :

- Si $U[i] \neq -1$ alors $\mathcal{B}$ répond que $T[i]$ égale $U[i]$. (Ce cas se produit lorsque $\mathcal{A}$ a déjà accédé à $T[i]$; $\mathcal{B}$ ne fait que reproduire la réponse qu'il a déjà donné par le passé.)
- Sinon, soit $[a, b]$ le plus grand intervalle à bornes entières [8] contenant i et tel que $T[j] = -1$ pour tout $a \leq j \leq b$. L'algorithme $\mathcal{B}$ détermine la valeur α

$$\alpha = \begin{cases} U[a-1] & \text{si } i - a \leq b - i, \\ U[b+1] & \text{sinon,} \end{cases}$$

  l'affecte à $U[i]$ et la renvoit à $\mathcal{A}$ comme valeur de $T[i]$.

On peut montrer par récurrence sur le nombre d'accès que les valeurs de $U[]$ qui ne sont pas à -1 sont monotones : pour tous $i < j$, si $-1 \notin \{U[i], U[j]\}$ alors $U[i] \leq U[j]$. Cela assure qu'il existe au moins une entrée du problème cohérente avec les choix faits par $\mathcal{B}$.

À tout moment, il y a *exactement un* intervalle $[a, b]$ tel que $U[a] = 0$, $U[b] = 2$ et $U[i] = -1$ pour tout $a < i < b$; appelons les entrées $\{a+1, a+2, \dots, b-1\}$ l'*intervalle incertain*. Tant que l'intervalle incertain est non-vide, les réponses faites par $\mathcal{B}$ sont compatible avec une entrée sans 1 (remplir l'intervalle incertain de 0) et avec une entrée avec 1 (remplir l'intervalle incertain de 1). La taille de l'intervalle incertain est initialement n, et cette taille est divisée par au plus 2 à chaque requête. Comme $\mathcal{A}$ est de complexité $o(\log n)$, pour n assez grand $\mathcal{A}$ répond après avoir fait strictement moins de $\log_2 n$ requêtes, et donc avec un intervalle incertain non vide. L'algorithme $\mathcal{A}$ ne peut donc être correct.

Les élèves intéressé·e trouveront en Complément G une formalisation de la notion d'adversaire. Cette formalisation n'est pas exigible dans ce cours.

---

7. Dans la mesure où toute lecture d'une donnée correspond à un échange entre l'unité de calcul et l'unité mémoire, on peut simplement imaginer que $\mathcal{B}$ intercepte ces échanges et répond à la place de l'unité mémoire.
8. Par exemple, à la première requête $a = 1$ et $b = n$.

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 taille constante/arbitraire et s'avère bien adapté à la preuve de bornes inférieures. Nous allons l'illustrer sur le problème suivant : [9]

> 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.

6.3.1 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 ?* ». [10] 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.

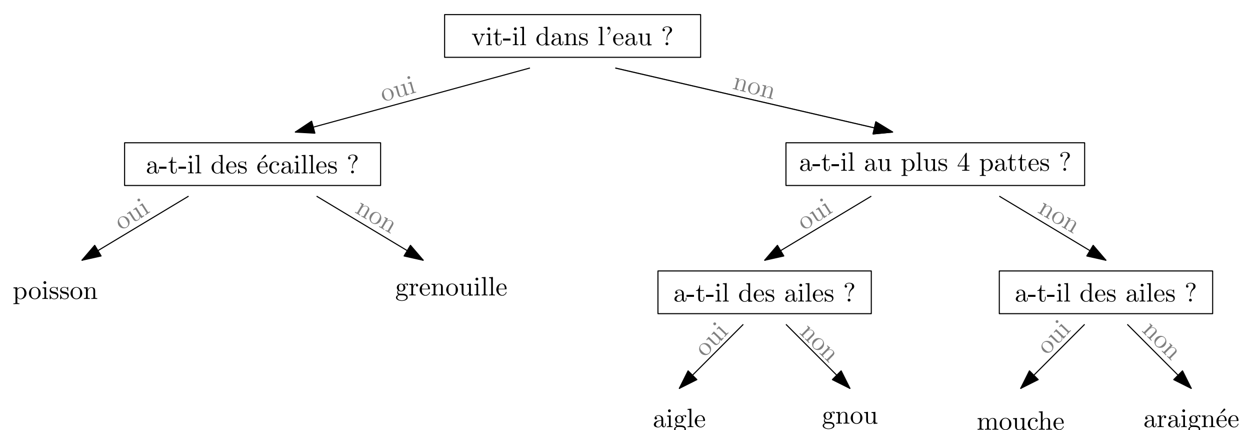


FIGURE 6.1 – Un exemple d'arbre représentant la recherche d'un animal parmi {aigle, araignée, gnou, grenouille, mouche, poisson}.

6.3.2 Le modèle

Un **arbre de décision** est un arbre enraciné fini $\mathcal{D}(n)$. [11] Chaque nœud interne est étiqueté par une question, appelée **requête**, portant sur l'entrée e. Un nœud interne a autant de des-

9. Rappel : un tableau $P[1..n]$ est une *permutation* s'il contient les valeurs $1, 2, \ldots, n$ exactement une fois chacune.

10. https://fr.wikipedia.org/wiki/Qui_est-ce_?

11. 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.

cendants 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 $P(e)$.

6.3.3 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 $\mathcal{D}_{\texttt{fusion}}(4)$:

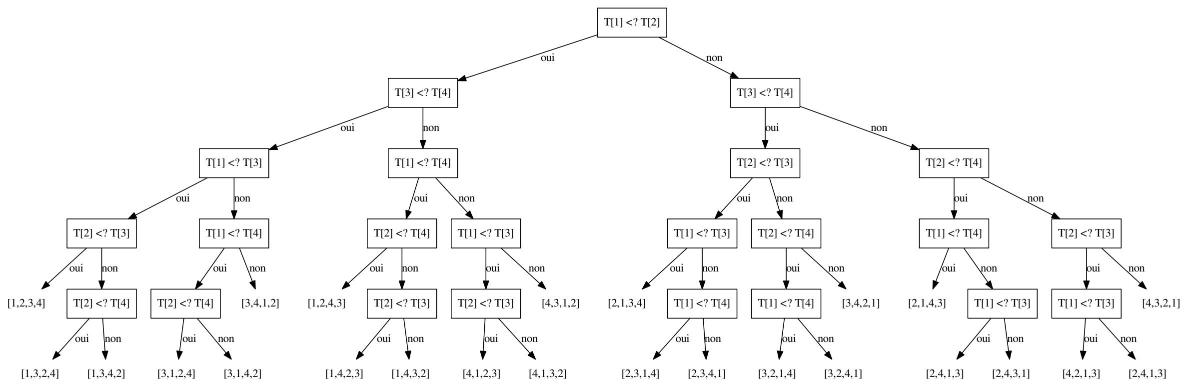


Figure 6.2 – Arbre $\mathcal{D}_{\texttt{fusion}}(4)$ décrivant l'algorithme de tri-fusion modifié pour résoudre Tri (version permutation).

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.

6.3.4 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 6.3.1. *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 $\log_2(n!) = \Omega(n \log n)$. ☐

6.4 Prolongement

Les bornes inférieures obtenues par arbres de décision ont deux limitations fondamentales. D'une part, une borne inférieure dans le modèle des arbres de décision **n'implique pas** une borne inférieure dans le modèle RAM taille constante/arbitraire. En effet, il existe des algorithme pour le modèle RAM taille constante/arbitraire qui ne sont pas traduisibles dans le modèle des arbres de décision. [12] [13] D'autre part, 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.

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.
- On peut minorer la complexité d'un problème par des arguments d'adversaire, qui prouvent que tout algorithme doit lire un minimum d'information de l'entrée.
- On peut minorer la complexité d'un problème par une traduction en arbres 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.

12. Certaines opérations du modèle RAM n'ont aucun sens dans le modèle des arbres de décision, par exemple de calcul de la moyenne $\frac{1}{n}(T[1] + T[2] + \ldots + T[n])$ et son utilisation dans une comparaison. Ainsi, un `tri_rapide` utilisant ce nombre comme pivot n'est pas traduisible en arbre de décision.

13. En particulier, la Proposition 6.3.1 n'implique pas que tout algorithme qui résout Tri (version permutation) dans le modèle RAM est de complexité $\Omega(n \log n)$. En revanche, elle nous assure que s'il existe un algorithme de complexité $o(n \log n)$, cet algorithme doit être intraduisible en arbre de décision.

Notes personnelles

Chapitre 7

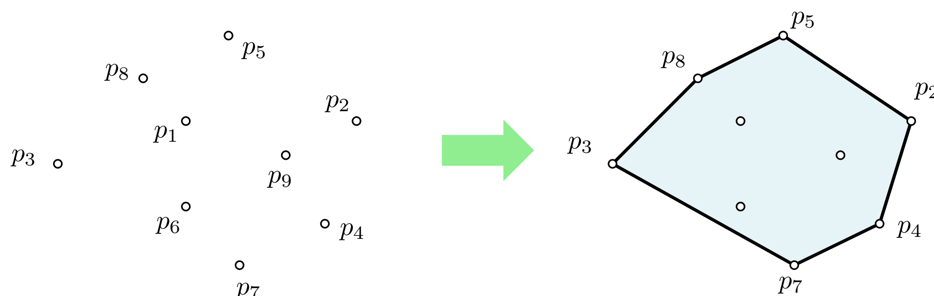
Réduction

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 approfondiraau 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 réduciton,
> - 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 $\mathrm{conv}(P)$. Lorsque P est fini, $\mathrm{conv}(P)$ 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 donc être décrite par la liste circulaire[2] des indices de points apparaissant sur l'enveloppe convexe lorsqu'on la parcourt dans

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 $[9, 3, 6]$.

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 $P = [p_1, p_2, \ldots, p_n]$ où p_i est un point du plan.

Sortie : la représentation par liste circulaire de l'enveloppe convexe de P.

Chaque point est donné par ses coordonnées cartésiennes $p_i = (x_i, y_i)$. Les valeurs possibles d'une coordonnée sont restreinte de manière à ce que chacune puisse être décrite par $O(\log n)$ bits.[3] Cela rend le problème Calcul d'enveloppe convexe adapté au modèle RAM taille arbitraire et déjà intéressant.

Examinons la complexité de ce problème...

7.1.1 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 précédant 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. 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 $O(n \log n)$.

3. Par exemple : les coordonnées sont des entiers compris entre 0 et n^{100}. Utiliser les nombres flottants du C (`float` ou `double`) serait bien plus restrictif...

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).

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 l'algorithme de Graham n'utilise pas d'instruction abusive.

7.1.2 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 complexité **des problèmes** Tri et 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 complexité linéaire.

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...

7.1.3 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 :

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).

6. Les « » renvoient au fait que les bornes inférieures et supérieures ne sont pas dans le même modèle de calcul.

- 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.

7.2.1 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 une paire 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 $s(z) \in P(x)$.

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 problème. Ainsi, une réduction est de complexité $O(n)$ si chacune des fonctions e et s peuvent être calculées 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, dans la Section 7.1, l'algorithme de Graham a établi que

$$\text{Calcul d'enveloppe convexe} \quad \leq_{O(n)} \quad \text{Tri}$$

et la Section 7.1.3 a prouvé que

$$\text{Tri} \quad \leq_{O(n)} \quad \text{Calcul d'enveloppe convexe}.$$

Examinons la manière dont $\leq_{f(n)}$ permet de propager des bornes de complexité $O()$ et $\Omega()$. Considérons deux problèmes algorithmiques P et P' 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.

> Si P' est de complexité $O(g(n))$ alors P est de complexité $O(f \circ g \circ f(n))$.

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 $O(f \circ g \circ f(n))$.

> 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 complexité. 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 $O(f \circ h \circ f(n))$. On aimerait avoir

$$h = O(g) \quad \overset{?}{\Rightarrow} \quad f \circ h \circ f = o(f \circ g \circ f).$$

mais cela est faux en général (prendre par exemple $g(n) = n^2$, $h(n) = n$ et $f(n) = \log n$). Cette implication est vraie pour certaines fonctions f, par exemple si $f = \Theta(n)$.

7.2.2 Problèmes de décision

Les complications identifiées par l'analyse précédente conduisent à simplifier le cadre d'étude des réductions.

Un **problème de décision** est un problème algorithmique admettant exactement deux réponses possibles.[7] Un **problème de décision en mots binaires** est[8] une fonction $D : \{0,1\}^* \to \{0,1\}$. On dit qu'un algorithme qui résout D **accepte** les entrées $w \in \{0,1\}^*$ telles que $D(w) = 1$ et **rejette** les entrées $w \in \{0,1\}^*$ telles que $D(w) = 0$.

Considérons deux problèmes de décisions D et D'. Une **réduction de D à D'** est une fonction $r : \{0,1\}^* \to \{0,1\}^*$ telle que $D(w) = D'(r(w))$ pour tout $w \in \{0,1\}^*$. 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 $f(n)$.

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

Proposition 7.2.1. *Si D et D' sont deux problèmes de décision avec $D \leq_{f(n)} D'$, alors,*

(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 $\Omega(g(n))$.

---

7. Autrement dit, non seulement l'ensemble S des réponses satisfait $|S| = 2$, mais l'image d'une entrée par un problème de décision ne peut prendre que deux valeurs possibles (et pas toute partie de S, puisque $|2^S| = 4$).

8. Rappelons que l'on note $\{0,1\}^* \overset{\text{def}}{=} \bigcup_{n\in\mathbb{N}}\{0,1\}^n$ l'ensemble des mots binaires finis et 2^x l'ensemble des parties d'un ensemble X.

Démonstration. 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 $f = O(g)$.

En effet, supposons qu'il existe un algorithme $\mathcal{A}_{D'}$ calculant D' et de complexité $O(g(n))$. Pour tout mot $w \in \{0,1\}^n$, $\mathcal{A}_r$ calcule $w' = r(w)$ en temps $O(f(n))$ et la taille de w' est $O(f(n))$. L'algorithme $\mathcal{A}_{D'}$ appliqué à w' fournit $D'(w') = D(w)$ en temps $O(g \circ f(n))$. Le problème D est bien de complexité $O(g \circ f(n))$.

Raisonnons par contraposée. Supposons que D' ne soit pas de complexité $\Omega(g(n))$. C'est donc que D' est de complexité $O(h(n))$ avec $h = o(g)$. 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 pour D est une borne inférieure pour D', et toute borne supérieure 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 ». Soulignons que l'on peut composer les réductions :

Lemme 7.2.2. *Soient D, D' et D'' trois problèmes de décision. Si $D \leq_{f(n)} D'$ et $D' \leq_{g(n)} D''$ alors $D \leq_{g \circ f(n)} D''$.*

Démonstration. Notons r la réduction de D à D' et r' la réduction de D' à D''. Pour tout mot $w \in \{0,1\}$ on a

$$D(w) = D'(r(w)) = D''(r'(r(w)))$$

aussi $r' \circ r$ est une réduction de D a D''. Remarquons que si r est de complexité $f(n)$ alors $|r(w)| \leq f(|w|)$ puisque le calcul de r requiert d'écrire la sortie $r(w)$. Ainsi, le calcul de $r'(r(w))$ peut se faire en temps $O(f(|w|) + g \circ f(|w|)) = O(g \circ f(|w|))$. □

7.3 Deux problèmes de référence

Examinons maintenant deux problèmes classiques qui fournissent de nombreuses occasion de réductions.

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.[9] 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 :	Vrai ou faux, le graphe G est k-coloriable.

9. En particulier le graphe n'a ni boucle ni arête multiple.

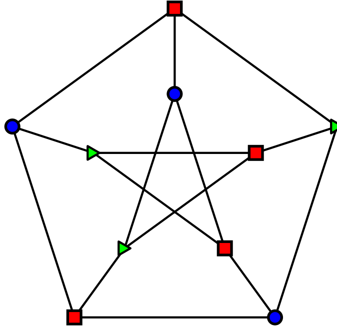


FIGURE 7.1 – Le *graphe de Petersen* est 3-coloriable mais pas 2-coloriable. Ici, la coloration c est représentée en donnant la même forme et couleur aux sommets de même image par c.

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 arbitrairement à 1, et propager cette coloration à ses voisins, puis à leurs voisins, etc. Une telle propagation peut aboutir à une contradiction : on doit propager la couleur $3 - c(v)$ d'un sommet v à un de ses voisins w, or w 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, déjà évoqué à la Section 2.3 et discuté plus en détail en Complément C. 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]

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.

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 [14], obtenant ainsi soit `vrai` soit `faux`. Une formule de logique propositionnelle est *satisfiable* s'il existe au moins une affectation des variables pour laquelle elle s'évalue à `vrai`. Le problème qui nous intéresse est le suivant [15] :

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 :	Vrai ou faux, f est satisfiable.

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.

7.4.1 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 [16].

Supposons donné en entrée un graphe $G = (V, E)$ et considérons que $V = \{1, 2, \dots, n\}$. 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'assignement $x_{i,c(i)} \stackrel{\text{def}}{=} \texttt{vrai}$ 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.

14. On peut formaliser cela par une définition récursive similaire à celle donnée pour l'ensemble des formules de logique propositionnelle en Section 2.3.

15. 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.

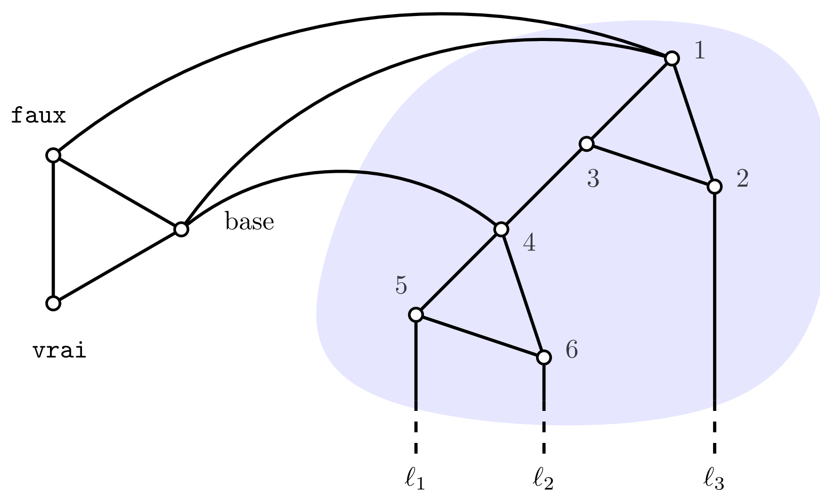
16. C'est assez similaire à ce que vous avez pratiqué en recherche opérationnelle.

7.4.2 Esquisse de 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* $\{\texttt{vrai}, \texttt{faux}, \texttt{base}\}$ *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 complexité *conditionnées* à des conjectures assez simples. On introduit cette idée ici sur l'exemple des problèmes 3-SUM-difficiles. On y revient au Chapitre 8 avec l'exemple emblématique de bornes inférieures conditionnelles : les problèmes NP-difficiles.

Le problème 3-SUM demande de décider si parmi n entiers donnés il en existe trois dont la somme est nulle :

3-SUM

Entrée : $T[1..n]$ un tableau de n entiers positifs ou négatifs.

Sortie : Vrai ou faux, existe-t-il $i < j < k$ tels que $T[i] + T[j] + T[k] = 0$?

Nous allons voir que 3-SUM se réduit à beaucoup de problèmes variés et en apparence plus compliqués.

7.5.1 Algorithmes pour 3-SUM

Examinons la complexité de ce problème dans le modèle RAM taille arbitraire. On peut résoudre 3-SUM en temps $O(n^3)$ par un simple examen de chaque triplet. On peut le résoudre [17] en $O(n^2 \log n)$ comme suit :

```
solution_subcubique(T[1..n])
   trier T par ordre croissant
   calculer la liste L[1..m] des T[i]+T[j] avec i<j
   trier L par ordre croissant
   a,b = 1,m
   tant que (a<n+1 et m>0)
      si T[a] + L[b] = 0 terminer en retournant Vrai
      sinon, si T[a] + L[b] < 0 incrémenter a
      sinon décrémenter b
   retourner Faux
```

On laisse en exercice la preuve que cet algorithme résout effectivement 3-SUM. Un peu d'effort permet d'améliorer cela en une solution quadratique :

```
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 terminer en retournant Vrai
         sinon, si T[i] + T[j] + T[k] < 0 incrémenter i
         sinon, décrémenter j
```

À nouveau, on laisse en exercice la preuve que cet algorithme résout effectivement 3-SUM. 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).

7.5.2 Exemples de problèmes 3-SUM-difficiles

Il s'avère que 3-SUM se réduit à des problèmes très divers. On en donne quatre exemples.

Notre premier exemple demande de décider si parmi n points du plan donnés en entrée il en existe trois d'alignés.

---

17. 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.

ALIGNEMENT

Entrée : Un tableau $P = [p_1, p_2, \dots, p_n]$ où p_i est un point du plan.

Sortie : Vrai ou faux, existe-t-il $i < j < k$ tels que p_i, p_j et p_k sont alignés ?

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[18] de le résoudre en $O(n^2)$.

Proposition 7.5.1. 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 transformer le tableau $T[1..n]$ d'entiers positifs ou négatifs, l'entrée de 3-SUM, en un tableau $P[1..n]$ de points où $P[i] = (T[i], T[i]^2)$. ☐

Le problème 3-SUM se réduit aussi en temps linéaire à chacun des trois exemples suivant (on omet les preuves).

X+Y

Entrée : Deux tableaux $X[1..n]$ et $Y[1..n]$ d'entiers.

Sortie : Vrai ou faux, l'ensemble $\{X[i] + Y[j] \colon 1 \leq i, j \leq n\}$ est-il de cardinal n^2 ?

GEOMBASE

Entrée : Dans le plan, des points $A[1..n]$ sur la ligne $y = 0$, $B[1..n]$ sur la ligne $y = 1$ et $B[1..n]$ sur la ligne $y = 2$

Sortie : Vrai ou faux, il existe une droite passant par un point de A, un point de B et un point de C

PLANIFICATION DE TRAJECTOIRE

Entrée : Dans le plan, deux segments p_1p_2 et p_3p_4 et un tableau $T[1..n]$ de polygones ayant chacun $O(1)$ sommets.

Sortie : Vrai ou faux, il est possible de déplacer continûment p_1p_2 en p_3p_4 sans qu'il rencontre l'intérieur d'aucun polygone $T[i]$.

Pour chacun de ces problèmes on connaît une solution de complexité $O(n^2)$; le fait que 3-SUM s'y réduise assure que si cet algorithme n'est pas optimal, alors 3-SUM peut être résolu en $o(n^2)$. Autrement dit,

18. C'est bien plus compliqué que la solution quadratique de 3-SUM et cela déborde largement de ce cours : l'algorithme standard pour faire utilise de la dualité point-droite et un argument d'amortissement appelé *théorème de la zone*.

Conditionellement au fait que 3-SUM est de complexité $\Omega(n^2)$, 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.

7.5.3 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 [19] $o(n^2)$ 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 (ouverte à ce jour) :

Conjecture 7.5.2. *Pour tout $\epsilon > 0$, le problème* 3-SUM *est de complexité* $\Omega(n^{2-\epsilon})$.

Autrement dit, la nouvelle conjecture est qu'il n'est pas possible de résoudre 3-SUM en temps *substantiellement* sous-quadratique.

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 $O(1.3289^n)$.

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.

7.7 Références bibliographiques

[BE05] Richard Beigel and David Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms*, 54(2) :168–204, 2005.

19. De complexité $O\left(n^2\left(\frac{\log\log n}{\log n}\right)^{2/3}\right)$ pour être précis.

À retenir.

- Un problème de décision est un problème algorithmique où seules deux réponses sont possibles, et où chaque entrée a exactement une réponse.
- Une réduction est un algorithme qui transforme toute entrée d'un problème A en une entrée d'un problème B de même réponse.
- Les réductions permettent de comparer les complexité de problèmes, ce que l'on note $\leq_{f(n)}$.
- Cette comparaison est simplifiée pour les seuls problèmes de décisions.
- 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.

Notes personnelles

Chapitre 8

Classes de complexité et NP-difficulté

Comme discuté en Section 2.5 du Chapitre 2, les ressources consommées par un algorithme polynomial augmentent d'un facteur multiplicatif constant quand on double la taille de l'entrée. Cela invite à 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.

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

- compreniez les notions de classe de complexité P et NP,
- compreniez les problèmes de satisfiabilité et de coloration de graphe,
- sachiez effectuer une réduction polynomiale entre deux problèmes,
- compreniez le principe d'une borne inférieure conditionnelle,
- connaissiez quelques exemples de problèmes NP-difficiles.

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 :

1. Ou, de manière équivalente, un ensemble de langages tels que définis dans le Complément D.

La **classe P** est l'ensemble des problèmes de décision qu'il est possible de résoudre dans le modèle **RAM taille constante** par un algorithme de complexité polynomiale.

La Proposition 2.6.1 assure que la classe P contient notamment tout problème algorithmique qui peut être résolu par un algorithme polynomial dans le modèle RAM taille arbitraire **et** sans instruction abusive.

La Section 7.3.1 donne un algorithme de complexité $O(n)$ pour 2-Col dans le modèle RAM taille arbitraire. On peut vérifier que cet algorithme ne contient pas d'instruction abusive, et donc que 2-Col peut être résolu en temps polynomial dans le modèle RAM taille constante. Ce problème est donc dans la classe P. 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 taille constante. Soulignons que remplacer ce modèle par la machine de Turing ne change rien à la définition car modèle RAM taille constante 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[2] 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).

La principale question qui nous intéresse est donc :

Comment prouver qu'un problème de décision n'appartient pas à la classe P ?

8.2 La classe de complexité NP

La classe de complexité NP se définit, comme la classe P, comme l'ensemble des problèmes de décision qui peuvent être résolus en temps polynomial dans un certain modèle de calcul. La définition de ce modèle demande un certain soin car il introduit une propriété peu intuitive de *non-déterminisme.*[3]

8.2.1 Définition

Un **vérificateur** d'espace d'entrée E et d'espace de certificats C est un algorithme $\mathcal{A}$ qui prend en entrée une paire $(e, c) \in E \times C$ et répond `Vrai` ou `Faux`. Dans le premier cas on dit que $\mathcal{A}$ **accepte** (e, c), dans le second cas on dit qu'il le **refuse**. Si $\mathcal{A}$ accepte (e, c) on dit que c **certifie** e **pour** $\mathcal{A}$.

Illustrons cela. Notons E l'ensemble des entrées du problème 3-Sum, c'est à dire des tableaux finis d'entiers positifs et négatifs. Notons C l'ensemble des triplets (i, j, k) d'entiers positifs. Appelons $\mathcal{A}$ l'algorithme qui prend en entrée une paire $(T[1..n], (i, j, k)) \in E \times C$ et qui l'accepte si (i) $1 \leq i, j, k \leq n$ et (ii) $T[i] + T[j] + T[k] = 0$, et la refuse sinon.

2. 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 taille constante mais prend $\Omega\left(n^2\right)$ sur une machine de Turing disposant d'un seul ruban.

3. On revient en Complément H sur le lien entre la définition que l'on donne et le non-déterminisme.

Un problème de décision $D\colon E \to \{0,1\}$ est **vérifiable en temps polynomial** (dans un modèle de calcul) s'il existe un polynôme P et un vérificateur de complexité polynomiale et d'espace d'entrée E tels que

- pour toute entrée $e \in E$ telle que $D(e) = 1$, il existe un certificat[4] de taille au plus $P(n)$ qui certifie e pour $\mathcal{A}$,
- pour toute entrée $e \in E$ telle que $D(e) = 0$, aucun certificat de taille au plus $P(n)$ ne certifie e pour $\mathcal{A}$.

On appelle alors $\mathcal{A}$ un **vérificateur polynomial** pour D. On peut vérifier que dans l'exemple précédent, $\mathcal{A}$ est un vérificateur polynomial pour 3-Sum.

La **classe NP** est l'ensemble des problèmes de décision vérifiables en temps polynomial dans le modèle **RAM taille constante**.

8.2.2 Premiers exemples

Montrons que les problèmes 3-Col et 3-Sat sont dans NP.

Construisons un vérificateur polynomial pour 3-Col. On prend $C = \bigcup_{n\geq 1}\{1,2,3\}^n$ comme espace de certificats. Le vérificateur $\mathcal{A}$ prend en entrée un graphe $G = (\{1,2,\ldots,n\}, E)$ et un certificat $c = (c_1, c_2, \ldots, c_\ell) \in C$. L'algorithme $\mathcal{A}$ accepte (G,c) si $\ell = n$ et pour tout $\{i,j\} \in E$ on a $c_i \neq c_j$. La vérification se fait en temps linéaire en la taille $|V| + |E|$ du graphe. On a bien qu'il existe un certificat accepté par le vérificateur si et seulement si le graphe donné en entrée est 3-coloriable.

Construisons un vérificateur polynomial pour 3-Sat. On prend $C = \{0,1\}^*$ 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 $c \in C$. L'algorithme $\mathcal{A}$ accepte (f,c) si l'affectation $x_i \overset{\text{def}}{=} \texttt{vrai}$ si $c_i = 1$ et $x_i \overset{\text{def}}{=} \texttt{faux}$ sinon satisfait la formule f. L'évaluation se fait en temps constant par clause, soit en temps linéaire dans l'ensemble. On a bien qu'il existe un certificat accepté par le vérificateur si et seulement si la formule donnée en entrée est satisfiable.

8.2.3 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 8.2.1. *$P \subseteq NP$.*

Démonstration. Soit $D : E \to \{0,1\}$ un problème de la décision qui appartient à la classe P. Il existe donc un algorithme $\mathcal{A}$ de complexité polynomiale dans le modèle RAM taille constante qui résout D. On peut utiliser $\mathcal{A}$ comme un vérificateur, avec un espace de certificats vide. Par conséquent, $D \in$ NP. □

4. La taille de c fait référence à la taille d'un mot binaire l'encodant, cf Chapitre 2.

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 8.3.1. *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.

8.3.1 Problème NP-difficile et réduction

Le Théorème 8.3.1 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 8.3.1 énonce que 3-SAT est NP-difficile. Pour des problèmes de décision A et B, notons $A \leq_P B$ pour signifier qu'il existe une réduction polynomiale de A à B. Puisque la composition de réductions polynomiales est une réduction polynomiale (Lemme 7.2.2), 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 7.4.2 assure que 3-COL est NP-difficile. On connaît à ce jour des *centaines*[5] 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.

8.3.2 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*[6] problème NP-difficile assure que *tout* problème de NP admet une solution polynomiale.[7] La réciproque est tout aussi utile :

> Si P$\neq$NP alors *aucun* problème NP-difficile n'admet de solution polynomiale.

5. 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/`.
6. N'importe lequel !
7. Autrement dit, tout problème dont on sait *vérifier une solution* en temps polynomial peut être *résolu* en temps polynomial.

Ainsi, prouver qu'un problème est NP-difficile peut s'envisager comme une **borne inférieure conditionnelle** sur sa complexité : conditionnellement au fait que P≠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 [8] 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≠NP, et donc qu'aucun problème NP-difficile n'a de solution polynomiale.

8.3.3 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. [9] La preuve du Théorème 8.3.1 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 taille constante 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 taille constante, 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. [10] 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

8. Par exemple, elle a été distinguée comme un des *problèmes du millénaire*.
9. On utilise ici l'identification habituelle `vrai` $\leftrightarrow 1$, `faux` $\leftrightarrow 0$.
10. Cela est possible car toute instruction peut s'exprimer comme une fonction booléenne d'un nombre constant de mots binaires de taille constante.

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 instantané 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, quelques 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 taille constante mais la machine de Turing).

8.4 Exemples de problèmes NP-difficiles

Concluons ce chapitre par une liste de problèmes algorithmiques qui sont tous NP-difficiles. [11] 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. [12]

Commençons par (les version décision) de problèmes d'optimisations qui devraient être familiers :

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 $c^T x \geq s$.

11. Pour la plupart il est facile de vérifier qu'ils sont aussi dans NP et donc NP-complets.

12. 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.

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 quelques problèmes d'optimisation sur les graphes[13]. Pour les deux premiers problèmes, noter qu'il est nécessaire que l'entier k fasse 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...

Voyageur de commerce

Entrée : Un nombre $s \in \mathbb{Q}$ et une matrice $A \in \mathbb{Q}^{n \times n}$ symétrique, 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 :

13. 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 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[14]...

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[15] :

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 de points $p_1, p_2, \ldots, p_n \in \mathbb{R}^3$ 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 $|\{i \colon p_i \cdot x \geq 0\}| \geq s$.

Voici trois problèmes d'algèbre :

14. Autrement dit, un problème de *clustering* fondamental en *machine learning*.

15. 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 [16] :

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$\neq$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 $s_2 = 0$. [17] La conjecture suivante est appelée ETH, pour *Exponential Time Hypothesis* :

16. De nombreux jeux ont des version NP-difficiles, voir le cours http://courses.csail.mit.edu/6.892/spring19/ pour quelques exemples.
17. Si $f(n) = O(n)$ alors pour tout $s > 0$, $f(n) = O(2^{sn})$. L'infimum est donc 0.

Conjecture 8.5.1 (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 [18] par 1, aussi elle admet une limite s_∞.

Conjecture 8.5.2 (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. [19]

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 $P \stackrel{?}{=} NP$, 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 H le présente de manière succinte à fins culturelles.

8.6 Références bibliographiques

[Aar16] Scott Aaronson. $P \stackrel{?}{=} NP$. 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 Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information 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. 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.

19. Cet énoncé n'est intéressant que si k dépend de n.

Notes personnelles

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 taille constante.

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]

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 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.

---

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.

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 $\mathbb{C}^{2\times2}$ 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

$$0 \mapsto 0 \quad \text{et} \quad 1 \mapsto 0$$

est possible.

> En calcul classique, on peut lire la valeur d'un bit sans le détruire.

2. Le fait qu'une matrice $M \in \mathbb{C}^{2\times2}$ préserve la norme peut se reconnaître au fait que ses colonnes forment une base orthonormée de $\mathbb{C}^2$.

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 $\{0,1\}^2 = \{0,1\} \times \{0,1\}$.

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 $\mathbb{C}$.

3. Au sens où l'on répète le calcul produisant ce qubit.

9.3.1 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 interprétation 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 $\mathrm{CNOT}(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |x \oplus y\rangle$. 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).

9.3.2 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 9.3.1. *Il n'existe pas de transformation unitaire* $T\colon \mathbb{C}^4 \to \mathbb{C}^4$ *tel que pour tout qubit* $|\phi\rangle$ *on ait* $T\left(|\phi\rangle \otimes |0\rangle\right) = |\phi\rangle \otimes |\phi\rangle$.

Démonstration. Supposons qu'un tel opérateur T existe. Pour $|\phi\rangle$ valant $|0\rangle$, $|1\rangle$ et $|0\rangle + |1\rangle$, la condition donne respectivement

$$\begin{aligned} T(|00\rangle) &= T(|0\rangle \otimes |0\rangle) = |00\rangle\,, \\ T(|10\rangle) &= T(|1\rangle \otimes |0\rangle) = |11\rangle\,, \\ T(|00\rangle + |10\rangle) &= T\left((|0\rangle + |1\rangle) \otimes |0\rangle\right) = (|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) = |00\rangle + |01\rangle + |10\rangle + |11\rangle\,, \end{aligned}$$

ce qui contredit la linéarité de T. □

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`.

9.4.1 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 $(e_1, e_2, \dots, e_k)$ de E, $(f_1, f_2, \dots, f_\ell)$ de F et $(t_1, t_2, \dots, t_{k\ell})$ de $E \otimes F$.

b. On note ϕ la bijection entre $\{(e_i, f_j) : 1 \leq i \leq k, 1 \leq j \leq \ell\}$ 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 :

$$\phi : \begin{cases} E \times F & \to \quad E \otimes F \\ \left(\sum_{i=1}^{k} a_i e_i, \sum_{j=1}^{\ell} b_j f_j\right) & \mapsto \quad \sum_{i=1}^{k} \sum_{j=1}^{\ell} a_i b_j \phi(e_i, f_j). \end{cases}$$

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 $(e_1 \otimes f_1, e_1 \otimes f_2, \dots e_1 \otimes f_\ell, e_2 \otimes f_1, \dots, e_k \otimes f_\ell)$ est une base de $E \otimes F$; c'est un simple renommage de la base $(t_1, t_2, \dots, t_{k\ell})$.

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,

$$\mathbb{C}^{2^m} = \left(\mathbb{C}^2\right)^{\otimes m} = \underbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \dots \otimes \mathbb{C}^2}_{m \text{ fois}}$$

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 $\langle x, y\rangle \stackrel{\text{def}}{=} x_1\overline{y_1} + x_2\overline{y_2} + \dots + x_n\overline{y_n}$. 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 $E_1 \otimes E_2 \otimes \dots \otimes E_n$ dans $F_1 \otimes F_2 \otimes \dots \otimes F_n$. Elle est notée $f_1 \otimes f_2 \otimes \dots \otimes f_n$ et définie par

$$\underbrace{(f_1 \otimes f_2 \otimes \dots \otimes f_n)}_{\text{nom de l'application}} : \underbrace{(e_1 \otimes e_2 \otimes \dots \otimes e_n)}_{\text{élément de } E_1 \otimes E_2 \otimes \dots \otimes E_n} \mapsto \underbrace{f_1(e_1) \otimes f_2(e_2) \otimes \dots \otimes f_n(e_n)}_{\text{élément de } F_1 \otimes F_2 \otimes \dots \otimes F_n}.$$

5. Autrement dit, $\phi((e_i, f_j)) \stackrel{\text{def}}{=} t_{\ell(i-1)+j}$.

Exemples

Reprenons les applications X et Y opérant sur un qubit :

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \text{et} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

Leur produit tensoriel $X \otimes Y$ est l'application de $\mathbb{C}^2 \otimes \mathbb{C}^2$ dans lui-même, définie par

$$(X \otimes Y) \colon \begin{cases} |0\rangle \otimes |0\rangle & \mapsto & X(|0\rangle) \otimes Y(|0\rangle) = |1\rangle \otimes i\,|1\rangle & = & i\,|11\rangle \\ |0\rangle \otimes |1\rangle & \mapsto & X(|0\rangle) \otimes Y(|1\rangle) = |1\rangle \otimes -i\,|0\rangle & = & -i\,|10\rangle \\ |1\rangle \otimes |0\rangle & \mapsto & X(|1\rangle) \otimes Y(|0\rangle) = |0\rangle \otimes i\,|1\rangle & = & i\,|01\rangle \\ |1\rangle \otimes |1\rangle & \mapsto & X(|1\rangle) \otimes Y(|1\rangle) = |0\rangle \otimes -i\,|0\rangle & = & -i\,|00\rangle \end{cases}$$

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.

9.4.2 Système à m qubits et algorithme quantique

On munit $(\mathbb{C}^2)^{\otimes m}$ 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 $(\mathbb{C}^2)^{\otimes m}$ 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 $T : (\mathbb{C}^2)^{\otimes m} \to (\mathbb{C}^2)^{\otimes m}$ **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 $T : (\mathbb{C}^2)^{\otimes m} \to (\mathbb{C}^2)^{\otimes m}$ **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 $w_1 \in$

6. Ici, id_E désigne l'identité sur l'espace E.

$\{0,1\}^{i-1}, x \in \{0,1\}, w_2 \in \{0,1\}^{j-i-1}, y \in \{0,1\}, w_3 \in \{0,1\}^{m-j}$. 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.

Théorème 9.4.1. *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 $T_1, T_2, \ldots, T_k$ 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 (cf Section 9.7).

> 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.

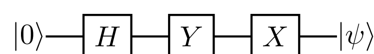
Un **algorithme quantique** est une suite finie d'opérateurs élémentaires.

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*.

9.5.1 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

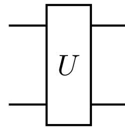


7. Attention, certaines sources adoptent la convention d'une lecture de droite à gauche, par exemple [KSVV02].

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 :



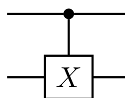
9.5.2 Portes contrôlées et leur représentation

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]

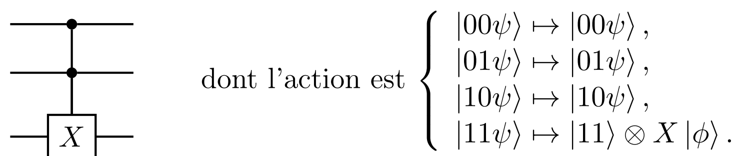
$$\Lambda(G)\colon \begin{cases} |0\rangle \otimes |\psi\rangle \mapsto |0\rangle \otimes |\psi\rangle\,, \\ |1\rangle \otimes |\psi\rangle \mapsto |1\rangle \otimes G(|\psi\rangle). \end{cases}$$

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 :



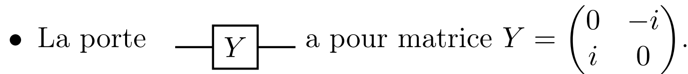
$$\begin{cases} |00\psi\rangle \mapsto |00\psi\rangle\,, \\ |01\psi\rangle \mapsto |01\psi\rangle\,, \\ |10\psi\rangle \mapsto |10\psi\rangle\,, \\ |11\psi\rangle \mapsto |11\rangle \otimes X\,|\phi\rangle\,. \end{cases}$$

9.5.3 Quelques portes usuelles

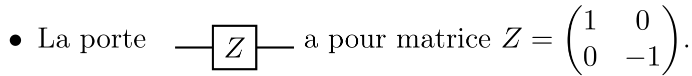
Donnons (ou rappelons) les matrices et représentations de quelques portes usuelles.

- La porte *de Hadamard* —[H]— a pour matrice $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.
- La porte —[X]— a pour matrice $X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

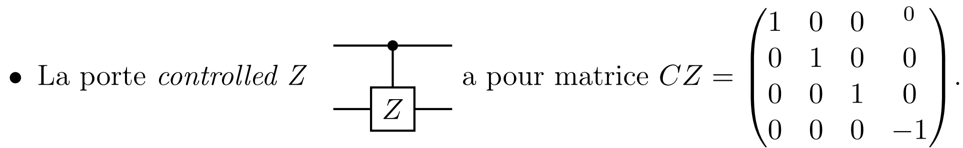
8. On décrit ici l'action de $\Lambda(G)$ sur un $(m+1)$-qubit séparable $|\phi\rangle \otimes |\psi\rangle$, où $|\psi\rangle$ est un m-qubit. L'action sur un $(m+1)$-qubit intriqué est obtenue en étendant cette définition par linéarité.



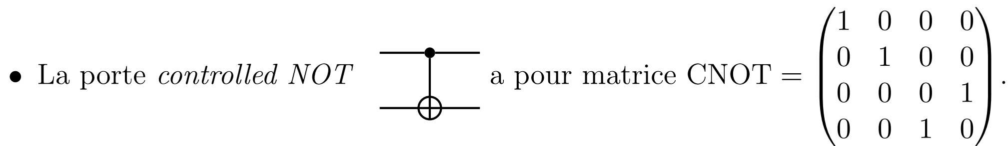
- La porte Y a pour matrice $Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$.



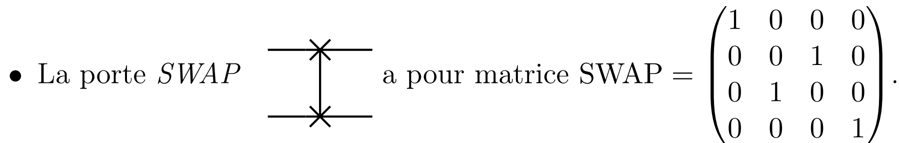
- La porte Z a pour matrice $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.



- 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 $\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$.



- La porte *SWAP* a pour matrice $\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

- La porte *Toffoli* a pour matrice $\text{CCNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$.

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 log-RAM augmenté de l'oracle f, le problème RECHERCHE UNIVOQUE est de complexité $\Omega(2^n)$; 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

9. On revient sur la question du nombre de qubits nécessaires en fin de section.

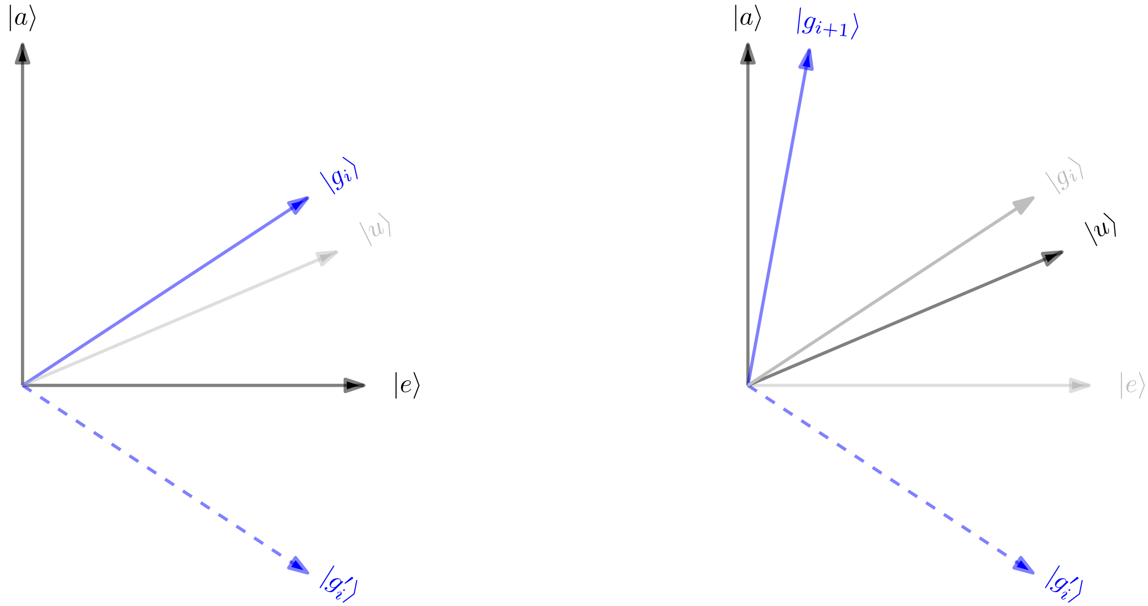


FIGURE 9.1 – Une itération de l'algorithme de Grover. Gauche : $|g_i'\rangle$ est obtenu en appliquant à $|g_i\rangle$ une réflexion d'axe $|e\rangle$. Droite : $|g_{i+1}\rangle$ est obtenu en appliquant à $|g_i'\rangle$ une réflexion d'axe $|u\rangle$.

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*.

9.6.1 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] $|e\rangle \stackrel{\text{def}}{=} |u\rangle - |a\rangle$.

> Les vecteurs $|u\rangle$, $|a\rangle$ et $|e\rangle$ sont contenus dans $\text{vect}_{\mathbb{R}}(|u\rangle, |a\rangle)$.
> Les vecteurs $|a\rangle$ et $|e\rangle$ sont orthogonaux.

L'algorithme de Grover calcule [11] une suite $|g_1\rangle, |g_2\rangle, \ldots$ de n-qubits dans le plan $\text{vect}(|u\rangle, |a\rangle)$. 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 $\text{vect}_{\mathbb{R}}(|u\rangle, |a\rangle)$,
- 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 $\text{vect}_{\mathbb{R}}(|u\rangle, |a\rangle)$.

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.

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 en Section 9.7.

9.6.2 La convergence

Commençons par la **convergence** de la suite $\{|g_i\rangle\}$ vers $|a\rangle$. Puisque les opérations géométriques se déroulent dans $\text{vect}_{\mathbb{R}}(|u\rangle, |a\rangle)$, 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

$$\cos\left(\frac{\pi}{2} - \theta\right) = |a\rangle \cdot |u\rangle = \frac{1}{2^{n/2}} > 0,$$

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,

$$i^* \leq \frac{1}{\theta} \leq \frac{1}{\sin\theta} = \frac{1}{\cos\left(\frac{\pi}{2} - \theta\right)} = 2^{n/2}.$$

9.6.3 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 $\text{vect}_{\mathbb{R}}(|u\rangle, |a\rangle)$. 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 $x = a$.

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.

12. Ce calcul peut mettre en jeu des qubits ancillaires, aspect que l'on ne discute pas ici pour simplifier. Cf Section 9.7.

13. Vérifier cela qubit par qubit grâce à l'identité $ac + bd = \frac{1}{2}((a+b)(c+d) + (a-b)(c-d))$.

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'accéleration 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 ; on a illustré cela sur le calcul du $\wedge$ logique, qui 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 réalier 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.

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 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 taille constante.

Notes personnelles

Annexes

Les chapitre qui suivent précisent les convention de présentation d'algorithmes et rappellent les principes de notation asymptotique, deux points qu'il est important de maîtriser. Suivent, à destination des plus curieux·ses, la preuve du théorème d'Akra-Bazzi (vu au Chapitre 4).

Annexe A

Comment présenter un algorithme dans ce cours

Ce cours étudie les algorithmes en tant que méthodes **d'organisation du calcul**. L'utilisation d'un algorithme sur un ordinateur [1] demande de le traduire dans un langage de programmation. L'**algorithmique**, en tant que discipline, réfléchit à l'organisation du calcul **indépendamment** de cette traduction. En montant ainsi en abstraction, l'algorithmique gagne en simplicité et en généralité. [2]

Dans ce cours, j'encourage à présenter les algorithmes de manière aussi synthétique que possible. Cela consiste à débarasser la présentation de l'algorithme en question des détails qui ne sont pas essentiels, tout en gardants ceux qui aident à la compréhension. Voici trois exemples valides de présentation d'un algorithme élémentaire pour calculer le minimum d'une liste de n entiers donnés en entrée :

```
def min(L[0..n-1]):
    best = L[0]
    for e in L[1:]:
        if e < best:
            best = e
    return best
```

```
fonction min(L)
    best = premier element de L
    parcourir la liste L
        si valeur courante <  best
            best := valeur courante
    renvoyer best
```

> 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.

Comparée à la 1ère solution, la deuxième omet de spécifier le sens de parcours du tableau. Ce détail n'est pas essentiel puisque l'algorithme est correct pour l'un ou l'autre sens. Comparée à la seconde solution, la troisième omet de nommer les variables, autre détail peu essentiel. (Si l'algorithme mettait en jeu plusieurs variables auxiliaires, une description textuelle pourrait cependant devenir plus laborieuse qu'un pseudocode du type de la deuxième présentation.)

1. Soulignons encore que si c'est le cas le plus courant, tout algorithme n'est pas nécessairement exécuté sur un ordinateur. C.f. par exemple les recettes de cuisine.
2. Corollaire immédiat : l'implantation d'un algorithme est un travail dont il ne faut sous-estimer ni l'ampleur, ni la valeur, ni l'intérêt.

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 palette des "routines" disponibles s'agrandit. Naturellement, lorqu'un algorithme $\mathcal{A}$ utilise un algorithme $\mathcal{A}'$, cet algorithme $\mathcal{A}'$ est rarement une instruction élémentaire (*c.f.* la discussion du Chapitre 2).

Les tableaux sont un moyen privilégié de représenter les entrées et sorties.

En effet, l'organisation en tableau reflète assez naturellement l'organisation de la mémoire, comme on l'explique 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 assez libre les plages d'indices utilisés par les tableaux[3] : $T[1..n]$ 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.

3. En particulier, dans ce cours, les indices d'un tableau peuvent ne pas commencer à 0.

Annexe B

Rappels sur les notations asymptotiques

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 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 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 indépendant de n. On dit qu'une fonction f est au plus **logarithmique** si $f = O(\log n)$: rappelons que comme $\log_a(n) = \frac{\log_b(n)}{\log_a b}$, la base du logarithme à l'intérieur du $O()$ importe peu dès lors que cette base est **bornée**. 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, toléré dans ce cours, consiste à oublier le terme « au plus ».[1]

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

On dit qu'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 (ne serait-ce que pour $f = g$).

> $f = \Omega(g)$ si et seulement si il existe un réel $M > 0$ tel que
>
> $$\forall n \geq 0, \quad \exists n' \geq n, \qquad f(n') \geq Mg(n').$$

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.

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

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

Autrement dit, $f = \Theta(g)$ s'il existe des constantes n_0, $m > 0$ et $M > 0$ tels que

$$\forall n \geq n_0, \qquad mg(n) \leq f(n) \leq Mg(n).$$

Soulignons que les constantes m et M peuvent être distinctes. Quand elles sont égales, on dit que f et g sont **équivalentes**.

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 C

Complément : quelques notions d'architecture

Avant de décrire notre modèle de calcul, examinons certains principes de fonctionnement des architectures matérielles dont il s'inspire. Cette rapide introduction à l'architecture des ordinateurs est à prendre comme une digression culturelle, donnée à seule fin de motiver les modélisations qui suivent.

C.1 Stockage de l'information

Le dispositif de mémorisation d'information d'un ordinateur comporte plusieurs niveaux :

- La construction d'une *cellule mémoire élémentaire*, qui permet de mémoriser un nombre 1-bit.
- L'assemblage de b cellules mémoire élémentaires en une *case mémoire* qui mémorise ainsi un nombre b-bits. Généralement $b = 8$ 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.

Les deux premiers niveaux ont peu d'influence sur la définition du modèle de calcul, si ce n'est dans le choix du paramètre b. Nous les laissons de côté et renvoyons les élèves intéressé·es à l'excellent ouvrage de Nisan et Schocken [NS21, §3.1].

La construction d'une unité mémoire met en jeu des mécanismes *d'adressage*. L'unité mémoire numérote les cases de 0 à $M - 1$, et dispose d'entrées et de sorties sur lesquelles on peut lire ou écrire des entiers (un entier b-bits étant transmis par b pattes). Pour accéder en lecture à une case, on transmet à l'unité mémoire le numéro de la case qui nous intéresse, et l'unité de mémoire alimente sa sortie avec la valeur mémorisée par cette case par un mécanisme de multiplexage. L'écriture se fait de manière similaire, par un mécanisme de démultiplexage. [1]

Plus précisément, et en première approximation, l'unité mémoire dispose d'une entrée (*bus d'adresse*) et d'une entrée/sortie (*bus de données*) pouvant recevoir ou émettre des mots binaires. Pour lire le contenu d'une case mémoire, on transmet sur le bus d'adresse la représentation binaire du numéro de la case qui nous intéresse et l'unité de mémoire alimente le bus de données avec le mot binaire mémorisé par cette case. Pour écrire dans une case mémoire, on transmet sur le

1. Notez que ce principe explique pourquoi un kilo-octet correspond non pas à 1000 octets, mais à $1024 = 2^{10}$ octets. En effet, la construction concrète d'une unité mémoire suppose de fixer un nombre de bits pour la transmission du No de la case. Il est dès lors naturel qu'une unité mémoire utilise pleinement ces possibilités d'adressage en choisissant pour valeur de M une puissance de 2.

bus d'adresse la représentation binaire du numéro de la case qui nous intéresse et sur le bus de données le mot à dans cette case. La taille des (mots binaires pouvant être écrits ou lus sur un) bus est fixée à la construction. Les concepteurs d'architecture informatique ayant horreur du gachis, remarquez que ce principe explique que les tailles de stockage mémoire soient exprimés en base 2 (1 kilo-octet $= 2^{10}$ octets).

Ces principes *très* simplistes ont été très largement raffinés et optimisés, mais ces évolutions n'ont globalement pas modifié l'organisation générale de la mémoire comme un tableau de cases auxquelles on peut accéder via leur indice.

C.2 Manipulation de l'information

La base du traitement de l'information par des systèmes physiques repose sur trois idées :

- Traiter de l'information équivaut à appliquer une ou plusieurs fonctions booléennes.
- Toute fonction booléenne peut être évaluée par une formule de logique propositionnelle.
- Toute formule de logique propositionnelle peut être évaluée par un circuit booléen.

Comme on sait construire des systèmes physiques qui réalisent des circuits booléens, ces systèmes permettent de traiter de l'information de manière générale. Détaillons chacune de ces étapes.

Fonctions booléennes. Sur un ordinateur, toute information est représentée par un mot binaire. Toute manipulation de l'information se ramène donc à transformer un mot binaire en un autre. Ainsi, tout *calcul* revient à appliquer une fonction $f : \{0,1\}^* \to \{0,1\}^*$. Pour des raisons (pratiques) qui apparaîtront dans les paragraphes suivants, il est nécessaire de décomposer un calcul en la composition de fonctions $\{0,1\}^\ell \to \{0,1\}^m$ où ℓ et m sont constants. Une telle fonction, qui associe à tout mot binaire de taille ℓ donnée, un mot binaire de taille m donnée, est appelée une **fonction booléenne**. Les fonctions booléennes sont essentiellement les *instructions élémentaires* à partir desquelles on compose tout calcul.

Prenons un exemple de fonction booléenne réalisant un calcul "intéressant". Supposons que l'on dispose en entrée de deux nombres $a, b \in \{0,1,2,3\}$ codés par leurs représentations binaires 2-bits a_1a_0 et b_1b_0. Définissons

$$c \stackrel{\text{def}}{=} a + b \mod 4 \qquad \text{et} \qquad d \stackrel{\text{def}}{=} \min(3, a+b).$$

Remarquons que c et d sont eux-aussi dans $\{0,1,2,3\}$ et notons les c_1c_0 et d_1d_0 leurs représentations binaires 2-bits. On peut vérifier que

$$(c_1, c_0) = f_1(a_1, a_0, b_1, b_0) \qquad \text{et} \qquad (d_1, d_0) = f_2(a_1, a_0, b_1, b_0).$$

où f_1 et f_2 sont les fonctions booléennes suivantes ($\ell = 4$ et $m = 2$) :

$$f_1 : \begin{cases} (0,0,0,0) & \mapsto & (0,0) \\ (0,0,0,1) & \mapsto & (0,1) \\ (0,0,1,0) & \mapsto & (1,0) \\ (0,0,1,1) & \mapsto & (1,1) \\ (0,1,0,0) & \mapsto & (0,1) \\ (0,1,0,1) & \mapsto & (1,0) \\ (0,1,1,0) & \mapsto & (1,1) \\ (0,1,1,1) & \mapsto & (0,0) \\ (1,0,0,0) & \mapsto & (1,0) \\ (1,0,0,1) & \mapsto & (1,1) \\ (1,0,1,0) & \mapsto & (0,0) \\ (1,0,1,1) & \mapsto & (0,1) \\ (1,1,0,0) & \mapsto & (1,1) \\ (1,1,0,1) & \mapsto & (0,0) \\ (1,1,1,0) & \mapsto & (0,1) \\ (1,1,1,1) & \mapsto & (1,0) \end{cases} \quad \text{et} \quad f_2 : \begin{cases} (0,0,0,0) & \mapsto & (0,0) \\ (0,0,0,1) & \mapsto & (0,1) \\ (0,0,1,0) & \mapsto & (1,0) \\ (0,0,1,1) & \mapsto & (1,1) \\ (0,1,0,0) & \mapsto & (0,1) \\ (0,1,0,1) & \mapsto & (1,0) \\ (0,1,1,0) & \mapsto & (1,1) \\ (0,1,1,1) & \mapsto & (1,1) \\ (1,0,0,0) & \mapsto & (1,0) \\ (1,0,0,1) & \mapsto & (1,1) \\ (1,0,1,0) & \mapsto & (1,1) \\ (1,0,1,1) & \mapsto & (1,1) \\ (1,1,0,0) & \mapsto & (1,1) \\ (1,1,0,1) & \mapsto & (1,1) \\ (1,1,1,0) & \mapsto & (1,1) \\ (1,1,1,1) & \mapsto & (1,1) \end{cases}$$

Une telle fonction est tout simplement obtenue en envoyant l'encodage des données d'entrées sur l'encodage des données de sortie.

Logique propositionnelle. Le calcul booléen définit des règles de calcul sur des variables pouvant prendre deux valeurs, traditionnellement décrites comme `vrai` et `faux`. (De telles variables sont appelées *booléennes*.) Le calcul booléen définit des opérations sur ces variables, les plus classiques étant l'opérateur `non` (noté $\neg$, prenant un argument), l'opérateur `et` (noté $\wedge$, prenant deux arguments) et l'opérateur `ou` (noté $\vee$, prenant deux arguments). À cela s'ajoute généralement l'opérateur `ou exclusif`, défini par $a \oplus b \overset{\text{def}}{=} (a \wedge \neg b) \vee (\neg a \wedge b)$. Ces règles de calcul sont données sous la forme de tables de vérité. [2]

Fixons un ensemble, par exemple $V \overset{\text{def}}{=} \{a_1, a_0, b_1, b_0\}$, dont les éléments sont appelés *variables*. L'ensemble Σ des *formules de logique propositionnelle* (de variables V) est défini, récursivement, comme le plus petit ensemble satisfaisant $V \subset \Sigma$, clôt par les opérateurs $\neg$, $\wedge$ et $\vee$. [3] Par exemple, $(a_0 \wedge \neg b_0) \vee (\neg a_0 \wedge b_0)$ est une formule de logique propositionnelle sur $\{a_0, b_0\}$.

Fixons une bijection $\{0,1\} \to \{$`vrai, faux`$\}$, par exemple [4] $0 \mapsto$ `faux` et $1 \mapsto$ `vrai`, et utilisons la pour transporter les règles du calcul booléen sur $\{0,1\}$. Ainsi, $0 \vee 0 = 0$ (puisque `faux ou faux = faux` en calcul booléen) et $0 \vee 1 = 1$ (puisque `faux ou vrai = vrai` en calcul booléen). Cette interprétation fait coïncider fonctions booléennes de sortie $m = 1$ bit et formules de logique propositionnelle :

> Pour toute fonction booléenne $f : \{0,1\}^\ell \to \{0,1\}$ il existe une formule de logique propositionnelle ϕ de variables $\{x_1, x_2, \ldots, x_\ell\}$ telle que $\phi = f(x_1, x_2, \ldots, x_\ell)$ pour tout $(x_1, x_2, \ldots, x_\ell) \in \{0,1\}^\ell$.

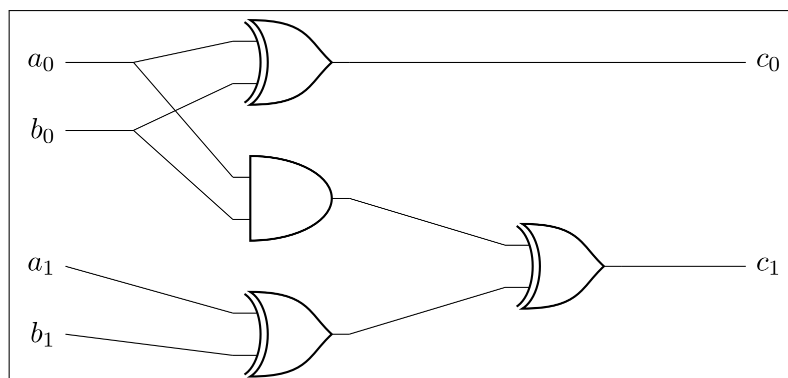
2. Cf `https://en.wikipedia.org/wiki/Boolean_algebra#Basic_operations` pour des exemples.
3. Autrement dit, pour tous $f_1, f_2 \in \Sigma$ les objets $\neg f_1$, $f_1 \wedge f_2$ et $f_1 \vee f_2$ sont dans Σ.
4. C'est la bijection utilisée de manière standard, mais l'autre fonctionnerait tout autant.

Plus généralement, une fonction booléenne de sortie m-bits correspond à m formules de logique propositionnelle. Par exemple, pour la fonction f_1 définie ci-dessus on a :

$$(c_1, c_0) = f_1(a_1, a_0, b_1, b_0) \qquad \Leftrightarrow \qquad \begin{cases} c_0 = a_0 \oplus b_0 \\ c_1 = (a_0 \wedge b_0) \oplus (a_1 \oplus b_1) \end{cases}$$

Circuits booléens. Il est facile de réaliser physiquement [5] un système calculant les fonctions booléennes élémentaires $\wedge$, $\vee$, $\neg$ et $\oplus$. On appelle de tels circuits élémentaires des **portes logiques** et on les schématise comme suit (dans l'ordre : `et, ou, non, ou exclusif`) :

La définition récursive des formules de logique propositionnelle permet de les reformuler comme des arbres d'évaluation, que l'on peut ensuite réaliser par des **circuits** à base de portes logiques. Par exemple, l'additionneur 2-bits (décrit par la fonction f_1) est réalisé par le circuit suivant :



C.3 Qu'est-ce qu'une instruction élémentaire ?

La partie d'un processeur qui réalise *effectivement* les opérations de calcul sur les données s'appelle l'*unité arithmétique et logique* (UAL). L'UAL est composée de circuits du type de l'additionneur que l'on vient de voir, et ces circuits correspondent à autant d'*instructions élémentaires* du processeur. Ces instructions élémentaires sont *choisies* à la conception du processeur, avec pour seule contrainte de devoir être chacune réalisable par un circuit booléen de taille constante, c'est-à-dire d'être chacune exprimable par une fonction booléenne. [6]

5. Par exemple, on peut construire un circuit électrique disposant d'autant de générateurs que l'on a de variables. On allume un générateur si et seulement si la variable correspondante prend la valeur `vrai`. On utilise ces générateurs pour commander des interrupteurs, que l'on dispose en série pour réaliser un `et`, en parallèle pour réaliser un `ou`. Ce n'est bien sûr pas le seul système physique (de la plomberie marcherait tout aussi bien). La technique permettant de réaliser physiquement des circuits booléens évolue avec le temps, et a par exemple connu une première rupture avec la découverte de l'effet supraconducteur. Cela ne change pas le fait que les fonctions calculables par les circuits sont les fonctions booléennes.

6. Par exemple, les *tensor cores* de la microarchitecture Volta des cartes graphiques du fabricant Nvidia disposent d'une instruction élémentaire qui prend en entrée trois matrices A, B, C, chacune composée de 4×4 nombres flottants (16 bits ou 32 bits selon la variante de l'instruction), et retourne la matrice $D \stackrel{\text{def}}{=} A \times B + C$, autre matrice 4×4 composée de nombres (16 bits ou 32 bits selon la variante de l'instruction). Cf le bas de la page `https://developer.nvidia.com/blog/cuda-9-features-revealed/`. Cette opération a été ajoutée au jeu d'instructions élémentaires *parce qu*'elle est utilisée intensivement dans les *algorithmes* d'entrainement de réseaux de neurones.

Tout calcul qui peut être encodé dans une fonction booléenne est admissible comme instruction élémentaire.

Soulignons encore une fois qu'une fonction booléenne envoie les mots binaires *de taille ℓ fixée* sur les mots binaires *de taille m fixée.*

C.4 Qu'est-ce qu'un programme ?

Terminons cette digression par quelques précisions sur la manière dont un processeur traite un programme.

Assembleur et compilation. Chaque processeur dispose d'un langage natif, appelé *assembleur* de ce processeur, qui reflète notamment les opérations réalisées par les sous-circuits qui le composent. Il y a donc a priori autant de langages assembleurs qu'il y a de modèles de processeurs. En pratique, il existe des familles de processeurs de langages compatibles à rebours ; 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). Les seuls programmes que peut exécuter un processeur sont ceux écrits dans son assembleur. Un programme écrit dans un autre langage doit nécessairement être traduit en assembleur, cette traduction pouvant se faire par un *interpréteur*[7] ou par un *compilateur*[8].

Langage machine. Le *langage machine* est une convention d'encodage des instructions assembleur en nombres. Ce sont ces nombres qui sont stockés en mémoire. Lorsque le processeur exécute un programme, il charge ces nombres depuis la mémoire les uns après les autres, les décode, et agit en conséquence. En un certain sens, on peut envisager le langage machine comme le langage sur lequel opère réellement le processeur, et l'assembleur comme une « traduction bijective » en un langage plus facilement lisible par un humain. En première approximation, chaque instruction assembleur correspond de manière non-ambigüe à un code du langage machine.

La conception du jeu d'instructions d'un langage machine offre, à nouveau, une grande liberté au concepteur d'un processeur. Les jeux d'instructions évoluent avec le temps, de manière à mieux s'adapter aux besoins des utilisateurs de processeurs. De même, il faut envisager tout langage de programmation comme une *construction* visant à permettre de décrire un programme de manière plus confortable que si on le spécifiait en langage machine. La notion de confort peut s'interpréter ici en différents sens :

- Un langage peut assurer une meilleure *portabilité*, c'est à dire pouvoir se compiler tel quel sur des architectures matérielles variées.
- Un langage peut permettre une plus grande *abstraction*, et fournir au programmeur des objets plus proches de son intuition ou de ses besoins.
- Un langage peut garantir de meilleures *performances*, en optimisant le code assembleur produit pour l'architecture matérielle sur laquelle il est compilé.

Ainsi, on peut envisager le traitement d'un problème algorithmique comme l'utilisation conjointe d'un algorithme, d'un langage et d'une architecture, adaptés les uns aux autres : l'implantation

7. À chaque exécution d'un programme python, l'interprèteur charge le programme, le traduit en assembleur, puis lance l'exécution de l'assembleur.

8. Une fois un programme C écrit, on le fait traduire par un compilateur pour obtenir un fichier objet, qui est ensuite lié à d'autres fichiers objets pour obtenir un exécutable.

```c
int main(){
  int a,b,c;

  a=1;
  b=3;
  c=12;

  a = a-b+c;

  return 0;
}
```

```
push    rbp
mov     rbp,rsp
mov     DWORD PTR [rbp-0xc],0x1
mov     DWORD PTR [rbp-0x8],0x3
mov     DWORD PTR [rbp-0x4],0xc
mov     eax,DWORD PTR [rbp-0xc]
sub     eax,DWORD PTR [rbp-0x8]
mov     edx,eax
mov     eax,DWORD PTR [rbp-0x4]
add     eax,edx
mov     DWORD PTR [rbp-0xc],eax
mov     eax,0x0
pop     rbp
ret
```

```
55 48 89 e5 c7 45 f4 01 00 00 00 c7 45 f8 03 00 00 00
c7 45 fc 0c 00 00 00 8b 45 f4 2b 45 f8 89 c2 8b 45 fc
01 d0 89 45 f4 b8 00 00 00 00 5d c3
```

FIGURE C.1 – Haut-gauche : un code en langage C. Haut-droit : une traduction en assembleur `x86` par le compilateur `gcc`. Bas : le codage de ce programme assembleur en langage machine (affiché ici en notation hexadécimale).

de l'algorithme dans le langage donne un programme dont la compilation sur l'architecture produit un code machine qui peut être exécuté pour résoudre le problème. [9]

9. À titre d'exemple, l'article [DTH20] cite que 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`.

Annexe D

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]

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...

D.1 Machine de Turing

La *machine automatique* a été introduite par Turing dans un article fondateur [Tur37], 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.)

Définition formelle

On appelle *alphabet* l'ensemble $\Gamma \stackrel{\text{def}}{=} \{0, 1, \llcorner\!\lrcorner\}$, où $\llcorner\!\lrcorner$ est appelé *symbole blanc*. Mathématiquement, une *machine de Turing* est une paire $M = (Q, \delta)$ 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 $\llcorner\!\lrcorner$ 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.)

3. Comme discuté en Section 1.2, 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.

4. Le processeur était conçu comme un homme dans le texte original de Turing, mais cela peut aussi être un système physique.

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 $w \in \{0,1\}^*$.

D.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 ». Corrolaire 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*.

D.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

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.

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 D.3.1. *La fonction H n'est pas calculable.*

Explicitons une conséquence pratique du Théorème D.3.1. 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].

D.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 $D^{-1}(\{1\}) = \{w \in \{0,1\}^* \colon D(w) = 1\}$ 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.

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* [Tur37], qui prend en entrée une paire 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 ?

Annexe E

Introduction aux structures de données et à leur analyse

E.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*.

E.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).

1. La liste complète des opérations supportées par une liste varie un peu selon les sources.

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.
- 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...

E.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.

E.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. Ainsi, si l'ajout du $n$ème élément prend souvent $O(1)$ opérations élémentaires de calcul, la complexité pire-cas est $O(n)$...

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]

2. cf `https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython`

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 F

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 F.0.1. *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.

F.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

$$f(x) = \begin{cases} h(x) & \text{pour } 1 \leq x \leq x_0 \\ \displaystyle\sum_{i=1}^{k} f\left(\alpha_i x + h_i(x)\right) + g(x) & \text{pour } x > x_0, \end{cases} \tag{F.1}$$

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 $1 \leq i \leq k$.

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.

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.

F.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 $x > x_0$,

$$c_5 x^p \left(1 + \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} \mathrm{d}u\right) \leq f(x) \leq c_6 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right)\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} \mathrm{d}u\right). \tag{F.2}$$

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 (F.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 (F.1) pour propager des encadrements de $\cup_{j'<j} I_{j'}$ à I_j :

Lemme F.2.1. *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 $\alpha_i x + h_i(x) < \inf I_j$. □

F.3 Fonctions à croissance polynomiale

Avant d'entreprendre la preuve de l'encadrement (F.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 F.3.1. *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. □

F.4 Initialisation de la récurrence

Pour assurer l'encadrement (F.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.

F.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 (F.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 F.3.1, 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

$$\begin{aligned} f(x) &\leq c_6 \sum_{i=1}^{k} {\alpha_i}^p x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right) \left(1 + \int_1^{x} \frac{g(u)}{u^{p+1}} \mathrm{d}u - c_3 g(x)\right) + g(x) \\ &= c_6 x^p \left(1 - \frac{1}{\log^{\epsilon/2} x}\right) \left(1 + \int_1^{x} \frac{g(u)}{u^{p+1}} \mathrm{d}u - c_3 g(x)\right) + g(x), \end{aligned}$$

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 (F.2).

F.6 Retour au Théorème F.0.1

Il reste à déduire le théorème d'Akra-Bazzi de l'encadrement (F.2). Considérons donc un algorithme $\mathcal{A}$ satisfaisant les hypothèses du Théorème F.0.1. 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 (F.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 (F.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 G

Complément : formalisation de la notion d'adversaire

Pour $n \in \mathbb{N}^*$ notons $[n] \stackrel{\text{def}}{=} \{1, 2, \ldots, n\}$ et pour un ensemble A notons $A^* \stackrel{\text{def}}{=} \bigcup_{k \in \mathbb{N}} A^k$ l'ensemble des séquences (i.e. suites finies) à valeurs dans A.

Un **adversaire** de taille n est une fonction R qui à toute séquence $S \in [n]^*$ associe une séquence $R(S) \in (\{0,1\}^*)^{|S|}$, c'est à dire une séquence de $|S|$ mots binaire.

Considérons un problème algorithmique en P dont l'entrée occupe n cases mémoire, c'est à dire prend la forme d'un tableau $T[1..n]$. Un adversaire R de taille n est **compatible** avec un tel problème algorithmique P si pour toute séquence $S \in [n]^*$, il existe une entrée $T[1..n]$ de P telle que

$$\forall i \in [|S|], \quad T[S_i] = R(S)_i.$$

Une telle entrée est dite **compatible** avec $R(S)$. Autrement dit, un adversaire compatible avec un problème algorithmique est cohérent avec (ou indistinguable d')une entrée de ce problème.

Un adversaire est **en-ligne** [1] si pour toutes séquences $S = [s_1, s_2, \ldots, s_k]$ et $S' = [s'_1, s'_2, \ldots, s'_{k'}]$, pour tout entier $\ell \leq \min(k, k')$, en notant $R(S) = [r_1, r_2, \ldots, r_k]$ et $R(S') = [r'_1, r'_2, \ldots, r'_k]$, on a

$$s_1 = s'_1, s_2 = s'_2, \ldots, s_\ell = s'_\ell \quad \Rightarrow \quad r_1 = r'_1, r_2 = r'_2, \ldots, r_\ell = r'_\ell.$$

L'expérience de pensée présentée en Section 6.2.2 se formalise de la manière suivante :

Soit P un problème algorithmique dont l'entrée occupe n cases mémoires. Si pour n assez grand, il existe un adversaire en-ligne R de taille n tel que pour toute séquence $S \in [n]^*$ de taille $|S| \leq f(n)$, il existe deux entrées e et e' de P qui sont compatibles avec $R(S)$ et telles que $P(e) \cap P(e') = \emptyset$, alors aucun algorithme ne peut résoudre P en moins de $f(n)$ accès mémoire.

1. Cela modélise les adversaires qui "n'ont pas besoin de connaître $s_{\ell+1}, s_{\ell+2}, \ldots$ pour choisir r_ℓ.

Annexe H

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 D.

H.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 H.1.1. *La machine de Turing et le modèle* log-*RAM 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. □

H.2 Machine de Turing non-déterministe

Spécialisons maintenant la définition d'une *machine de Turing* vue en Section D.1 au cas des problèmes de décision. On appelle *alphabet* l'ensemble $\Gamma \stackrel{\text{def}}{=} \{0, 1, \text{␣}\}$, où ␣ est appelé *symbole blanc*. Mathématiquement, une *machine de Turing* est une paire $M = (Q, \delta)$ 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.

H.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.

1. NP est l'acronyme de Non-deterministic Polynomial.

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.

Annexe I

Transformée de Fourier discrète, classique et quantique

Cette séance conclut ce cours en examinant le calcul de la transformée de Fourier discrète. Cet analogue discret de la transformée de Fourier est un opérateur fondamental en traitement du signal et son usage a été révolutionné par la mise au point de l'algorithme FFT (*Fast Fourier Transform*) par Cooley et Tukey. Il s'avère que la transformée de Fourier discrète est aussi au cœur de l'algorithme de factorisation quantique de Shor...

I.1 Problématique : retour sur le produit de polynômes

Revenons sur le problème algorithmique du calcul du produit $P \times Q$ de deux polynômes monovariés P, Q discuté en cours :

Multiplication de polynômes

Entrée : Deux tableaux $A[0..n]$ et $B[0..n]$ d'entiers

Sortie : Le tableau $C[0..2n]$ où $C[i] = \sum_{0 \leq j, i-j \leq n} A[j] * B[i-j]$

Nous savons déjà que l'algorithme naïf effectue $O(n^2)$ opérations scalaires, et que l'algorithme de Karatsuba en effectue $O(n^{\approx 1.58})$. Voyons les grandes lignes d'un troisième algorithme, dû à Schönhage et Strassen.

I.1.1 Représentation d'un polynôme

Il peut sembler naturel de représenter un polynôme univarié par le tableau de ses coefficients, comme l'entrée du problème Multiplication de polynômes. Remarquons cependant qu'un polynôme univarié de degré n est déterminé par ses valeurs $y_0, y_1, \ldots, y_n$ en $n+1$ points $x_0, x_1, \ldots, x_n$ deux à deux distincts.

Pour vérifier que pour tout choix de $\{x_i\}$ et $\{y_i\}$, il existe un polynôme prenant ces valeurs, remarquons que le polynôme interpolateur de Lagrange convient :

$$P(X) \stackrel{\text{def}}{=} \sum_{j=0}^{n} \frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j}(X - x_k).s$$

L'unicité découle du théorème fondamental de l'algèbre.

On a dès lors deux manières naturelles de représenter un polynôme $P(X)$:

- par *coefficients*, via un tableau $[p_0, p_1, \ldots, p_n]$ tel que $P(X) = \sum_{i=0}^{n} p_i X^i$, ou
- par *valeurs*, via deux tableaux $[x_0, x_1, \ldots, x_m]$ et $[y_0, y_1, \ldots, y_m]$, choisis tels que $m \geq n$ les x_i soient deux à deux distincts et que pour tout $0 \leq i \leq n$ on ait $P(x_i) = y_i$.

Le choix d'autoriser $m \geq n$ dans la représentation par valeurs peut surprendre puisque $m = n$ suffit. Nous allons voir dans ce qui suit qu'il peut être pratique de prendre m plus grand.

I.1.2 Représentation et opérations

Supposons que P et Q sont deux polynômes de degré n, donnés par valeurs *et* que ces valeurs sont prises en les mêmes points $x_0, x_1, \ldots, x_m$. Examinons comment faire le calcul d'opérations usuelles telles que l'addition, la multiplication et l'évaluation.

Commençons par l'addition. Le degré de $P + Q$ est au plus n, aussi il suffit de connaître les valeurs prises par $P + Q$ sur les points $x_0, x_1, \ldots, x_m$. Il suffit pour cela, pour chaque i, de sommer les valeurs prises par P et par Q en x_i. Cela peut se faire en temps $O(m)$.

La même idée s'applique à la multiplication, avec la contrainte que $m \geq 2n$ puisque le polynôme $P \times Q$ est de degré $2n$. Si c'est le cas, il suffit, pour chaque i, de multiplier les valeurs prises par P et par Q en x_i. Cela peut se faire en temps $O(m)$.

Représenter les polynômes par valeurs a cependant un inconvénient sérieux : il n'est pas facile a priori d'*évaluer* le polynôme P donné par valeur en un point x qui ne fait pas partie des « points fixés ». Ainsi, évaluer le polynôme interpolateur de Lagrange requiert $\Omega(m^2)$ opérations. A contrario, une telle évaluation peut se faire par $O(n)$ opérations en représentation par coefficicents, par exemple via la méthode de Horner.

> L'évaluation d'un polynôme se fait en temps $O(n)$ lorsqu'il est représenté par coefficients. La multiplication de deux polynômes se fait en temps $O(n)$ lorsqu'ils sont représentés par (suffisamment de) valeurs (compatibles).

I.1.3 Évaluation multi-points d'un polynôme – idées

Supposons donnée la représentation par coefficients d'un polynôme P. Comment en calculer efficacement une représentation par valeurs ?

Idée 1 : choisir les x_i. Calculer une représentation par valeurs de P revient à l'évaluer en $m + 1 \geq n + 1$ valeurs $\{x_i\}_{0 \leq i \leq n}$. Remarquons que *le choix des x_i nous appartient.* La première idée est de chercher un ensemble de valeurs qui seront plus faciles à évaluer *conjointement.*

Par exemple, si notre polynôme P est pair, toute évaluation en une valeur α nous donne immédiatement la valeur de P en $-\alpha$. Il en va de même si P est un polynôme impair. Ainsi, choisir les x_i symétriques permettrait d'économiser la moitié des calculs. Ce gain n'est guère impressionnant : il est limité aux polynômes pairs ou impairs et ne divise le nombre d'évaluations à faire que par une constante. Nos deux idées suivantes vont améliorer ces deux points.

Idée 2 : décomposer en parties paire et impaire. L'idée 1 a souligné que la parité d'un polynôme peut s'avérer utile. Il se trouve que l'on peut décomposer *tout* polynôme P en la somme d'un polynôme pair et d'un polynôme impair : il suffit de séparer les monômes de P de degré pair de ceux de degré impair.

Détaillons cela. Soit $P(X)$ un polynôme de coefficients $[p_0, p_1, \ldots, p_n]$. Notons $P_0(X)$ le polynôme de coefficients $[p_0, 0, p_2, 0, p_4, 0, \ldots]$. De même, notons $P_1(X)$ le polynôme de coefficients $[0, p_1, 0, p_3, 0, p_5, 0, \ldots]$. Ainsi, si on connaît $P_0(\alpha)$ et $P_1(\alpha)$, on peut reconstruire

$$P(\alpha) = P_0(\alpha) + P_1(\alpha) \qquad \text{et} \qquad P(-\alpha) = P_0(\alpha) - P_1(\alpha).$$

Remarquons que $P_0(X)$ est en fait un polynôme de degré $n/2$ évalué en X^2. Pour mettre ceci en évidence, remplaçons P_0 et P_1 par

$$P_{\text{pair}}(X) \overset{\text{def}}{=} \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} p_{2i} X^i \qquad \text{et} \qquad P_{\text{impair}}(X) \overset{\text{def}}{=} \sum_{i=0}^{\lfloor \frac{n-1}{2} \rfloor} p_{2i+1} X^i,$$

On a alors $P(X) = P_{\text{pair}}(X^2) + X P_{\text{impair}}(X^2)$ et $P(-X) = P_{\text{pair}}(X^2) - X P_{\text{impair}}(X^2)$. Résumons ce que l'on a gagné :

> Les valeurs du polynôme P, de degré n, sur un ensemble V de valeurs se déduit en temps $O(|V|)$ des valeurs des polynômes P_{pair} et P_{impair}, chacun de degré $n/2$, sur l'ensemble de valeurs $V' \overset{\text{def}}{=} \{\alpha^2 \colon \alpha \in V\}$.

Cette transformation a remplacé *un* polynôme de degré n par *deux* polynômes de degré $n/2$. En choisissant $V \overset{\text{def}}{=} \{-\frac{n}{2}, -\frac{n}{2} + 1, \ldots, -1, 1, 2, \ldots, \frac{n}{2}\}$ on a de plus $|V'| = |V|/2$.

Idée 3 : répéter 1 et 2 récursivement. Les idées 1 et 2 suggèrent d'aborder la conversion d'une représentation par coefficients à une représentation par valeurs comme suit :

```
1 Construire un ensemble V dont le V' est petit
2 Décomposer P en parties paires et impaires
3 Évaluer les parties paires et impaires sur V'
4 Recomposer les valeurs de P sur V
```

Si les évaluations de la ligne 3 sont effectuées naïvement, le gain n'est pas significatif. En revanche, si on pouvait déléguer ces évaluations récursivement, on se trouverait à effectuer $k = 2$ appels récursifs de taille $\alpha n = \frac{1}{2} n$, pour un coût hors appels récursifs de $g(n) = O(n)$. D'après le théorème maître, la complexité serait alors de $O(n \log n)$.

Pourquoi ce conditionnel ?

Et bien... le problème résolu à la ligne 3 est différent du problème résolu globalement, puisque l'ensemble de valeurs V' est **fixé** par le choix de V à la ligne 1. Autrement dit, lors des appels récursifs on ne **peut plus** choisir les valeurs. Pour se débarasser du conditionnel, il faudrait donc que le choix de V effectué lors du **premier appel** garantisse non seulement que V' est de taille $n/2$, mais qu'ensuite $V'' \overset{\text{def}}{=} \{\alpha^2 \colon \alpha \in V'\} = \{\alpha^4 \colon \alpha \in V\}$ soit de taille $n/4$, etc.

Idée 4 : relaxer le problème. La construction d'un ensemble V qui rendrait possible la stratégie esquissée ci-dessus est impossible si on en reste à une évaluation de P en des points *réels*. En revanche, si on considère le problème sur $\mathbb{C}$ et que l'on choisit pour x_i des racines de l'unité, les idées précédentes peuvent s'instantier récursivement.

I.1.4 Évaluation multi-points d'un polynôme – synthèse

Synthétisons l'enchaînement de ces quatre idées.

Pour simplifier la discussion, nous évaluons notre polynôme P de degré n non pas en $n+1$ points, mais en m points où m est la puissance de 2 immédiatement supérieure ou égale à $n+1$. Autrement dit, $m \stackrel{\text{def}}{=} 2^{\lceil \log_2(n+1) \rceil}$. Remarquons que cela résout toujours notre problème (évaluer P en m points suffit à en donner une représentation par valeurs) et ne cause pas de perte d'efficacité asymptotique (car $m \leq 2n$).

Explicitons maintenant les conditions auxquelles un ensemble de valeurs $\{x_i\}_{0 \leq i < m}$ est adaptée à une mise en œuvre récursive des idées avancées jusqu'ici. On définit un sous-ensemble $V \subseteq \mathbb{C}$ comme **fondant** si l'une des conditions suivantes est satisfaite :

(i) V est un singleton, ou

(ii) l'ensemble $V' \stackrel{\text{def}}{=} \{\alpha^2 \colon \alpha \in V\}$ est de taille $|V|/2$ et V' est lui aussi fondant.

Cette définition est récursive. Elle conduit à l'algorithme suivant :

```
Construire un ensemble fondant V de taille m
Décomposer P en parties paires et impaires
Évaluer recursivement les parties paires et impaires sur V'
Recomposer les valeurs de P sur V
```

Comme annoncé, il évalue tout polynôme P de degré n sur $m \geq n+1$ valeurs distinctes en temps $O(n \log n)$.

I.2 Transformée de Fourier discrète

Resituons les idées de la section précédente dans le cadre plus général de la transformée de Fourier discrète (TFD, ou DFT en anglais).

I.2.1 Définition de la transformée de Fourier discrète

Soit $S = [s_0, s_1, \ldots, s_{n-1}]$ une séquence de n nombres complexes. La *transformée de Fourier discrète* (TFD) de S est la séquence $\hat{S} = [\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{n-1}]$ de n nombres complexes définis par

$$\hat{s}_k = \sum_{\ell=0}^{n-1} s_\ell e^{-i\frac{2\pi}{n}k\ell} \tag{I.1}$$

Inversement, les termes de la séquence S se déduisent de la séquence $\hat{S}$ par

$$s_\ell = \frac{1}{n} \sum_{k=0}^{n-1} \hat{s}_k e^{i\frac{2\pi}{n}k\ell} \tag{I.2}$$

La transformée de Fourier discrète et son inverse sont des opérateurs linéaires $\mathbb{C}^n \to \mathbb{C}^n$ qui satisfont de nombreuses propriétés classiques en théorie de Fourier.

I.2.2 TFD et conjugaison

Soit $S \in \mathbb{C}^n$ une séquence et notons $\bar{S} \stackrel{\text{def}}{=} [\overline{s_0}, \overline{s_1}^*, \ldots, \overline{s_{n-1}}^*]$ la séquence conjugée. Notons R la transformée de Fourier inverse de S, c'est à dire la séquence telle que $S = \hat{R}$. On peut vérifier que $\bar{R} = \frac{1}{n}\widehat{\bar{S}}$. Ainsi, tout algorithme permettant de calculer la TFD permet aussi, à quelques conjuguaisons près, de calculer la TFD inverse.

I.2.3 TFD et convolution

On définit le *produit* $\cdot$ de deux séquences $S = [s_0, s_1, \ldots, s_{n-1}]$ et $T = [t_0, t_1, \ldots, t_{n-1}]$ comme leur produit terme à terme,

$$S \cdot T \stackrel{\text{def}}{=} [s_0 t_0, s_1 t_1, \ldots, s_{n-1} t_{n-1}].$$

et leur *convolution* $*$ par

$$S * T \stackrel{\text{def}}{=} [u_0, u_1, \ldots, u_{2n-2}] \quad \text{où} \quad u_j \stackrel{\text{def}}{=} \sum_{k\colon 0 \leq k, j-k \leq n-1} s_k t_{j-k}.$$

L'opérateur de convolution a de très nombreuses applications, par exemple dans la réalisation de filtres en traitement d'image. Soulignons une propriété fondamentale de la convolution :

> La transformée de Fourier discrète d'une *convolution* $S*T$ égale (à quelques détails près) le produit des transformées de Fourier discrètes $\hat{S}$ et $\hat{T}$.

Les *quelques détails* expriment que du fait que l'on s'intéresse à des séquences **finies**, la transformée de Fourier discrète d'une convolution $S * T$ n'est pas exactement le produit des TFD $\hat{S}$ et $\hat{T}$. Pour énoncer ce résultat précisément, introduisons deux notations.

Notons $S|T$ la concaténation des séquences S et T, c'est-à-dire la séquence commençant par les termes de S (dans l'ordre) et continuant par les termes de T (dans l'ordre). Notons 0^k la séquence $[0, 0, \ldots, 0]$ de longueur k.

Théorème I.2.1. *Soient S et T sont deux séquences de n complexes et notons $W = (S * T)|0$. Alors, $\widehat{W} = \widehat{S|0^n} \cdot \widehat{T|0^n}$.*

On le voit, on retrouve essentiellement le résultat annoncé, à quelques soucis "d'alignement" près. Il existe différentes manières de traiter ces soucis d'alignement (convolution circulaire, travail sur des séquences indexées sur $\mathbb{Z}$ mais à support fini, etc.). Ces questions, importantes pour une mise en œuvre correcte de la TFD, sont largement secondaires du point de vue de l'algorithmique.

I.2.4 Lien avec la multiplication (rapide) de polynômes

Faisons le lien avec les questions discutées en Section I.1.

Soit $P(X) = \sum_{i=0}^{n} p_i X^i$ un polynôme de $\mathbb{C}[X]$ de degré n. Pour $m \geq n$ on note $P^{(m)} \stackrel{\text{def}}{=} [p_0, p_1, \ldots, p_{m-1}]$, avec la convention que $p_j = 0$ pour $n < j < m$. Notons $\omega_m \stackrel{\text{def}}{=} e^{i\frac{2\pi}{m}}$ la racine m-ième primitive de l'unité. Un examen de l'équation (I.1) révèle que

$$\widehat{P^{(m)}} = [P(1), P(\omega_m), P\left(\omega_m{}^2\right), \ldots, P\left(\omega_m{}^{m-1}\right)],$$

autrement dit, pour m supérieur au degré de P :

La TFD du vecteur de m coefficients du polynôme P est le vecteur des valeurs que P prend sur les racines $m$ièmes de l'unité.

La TFD et son inverse permettent donc de convertir une représentation par liste d'un polynôme en sa représentation par valeurs *sur les racines de l'unité*, et vice-versa.

Les grandes lignes de l'algorithme de Schönhage et Strassen consiste, étant donné deux polynômes P et Q, à...

a. préparer les listes $P^{(m)}$ et $Q^{(m)}$ pour un entier m strictement supérieur à la somme des degrés de P et Q,
b. calculer les TFD de $\widehat{P^{(m)}}$ et $\widehat{Q^{(m)}}$, les listes (étendues) de coefficients de P et Q,
c. calculer le produit de $\widehat{P^{(m)}}$ et $\widehat{Q^{(m)}}$,
d. calculer la TFD inverse de ce produit.

Les étapes (a) et (c) sont de complexité linéaire, aussi la complexité de cet algorithme est dominé par le coût des transformées de Fourier discrètes (directe et inverse).

Une autre manière d'envisager l'algorithme de Schönhage-Strassen est que le vecteur de m coefficients du polynôme $P \times Q$ est la convolution des vecteurs de m coefficients des polynômes P et Q, et qu'un calcul rapide de convolution se fait par le calcul de la TFD inverse du produit des TFD.

I.3 Algorithme de FFT classique

Présentons maintenant l'algorithme de Cooley-Tukey de calcul (rapide) de la TFD d'une séquence de longueur m lorsque m est une puissance de 2. On parle de *FFT Radix 2*. Du point de vue algorithmique, la FFT Radix 2 suffit à résoudre le cas général : si la longueur de notre séquence n'est pas une puissance de 2, il suffit de l'allonger par des 0, ce qui au plus double sa longueur et multiplie la complexité par une constante.[1] Pour m une puissance de deux, la TFD d'une séquence $S[0..m-1]$ de m nombres complexes peut se calculer comme suit :

```
1  def Radix2FFT(S[0..m-1])
2     si m est au plus 4 traiter directement
3     t = m/2
4     U[0..t-1] = [S[0], S[2], ..., S[m-2]]
5     V[0..t-1] = [S[1], S[3], ..., S[m-1]]
6     U'[0..t-1] = Radix2FFT(U[0..t-1])
7     V'[0..t-1] = Radix2FFT(V[0..t-1])
8     w = 1 ; x = racine m-ème de l'unité
9     Pour j = 0..t-1
10       R[j] = U'[j] + w*V'[j]
11       R[t+j] = U'[j] - w*V'[j]
12       w = w * x
13    Retourner R[0..m]
```

1. Cette réduction permet de se simplifier la vie quand on examine ces questions d'un point de vue théorique. En revanche, elle induit une perte d'efficacité injustifiable en pratique au vu les **immenses** applications de la TFD. Aussi, diverses variantes de l'algorithme FFT ont été développées pour traiter "directement" des séquences de taille quelconque. Cf l'annexe A.6 de l'ouvrage d'Erickson.

I.4 Algorithme quantique de transformée de Fourier discrète

Il s'avère que la TFD peut se reformuler en un opérateur unitaire $\mathbb{C}^n \to \mathbb{C}^n$, et est donc calculable dans le modèle quantique. De plus, l'algorithme FFT se traduit naturellement en un algorithme quantique. Voyons cela...

I.4.1 La TFQ, version unitaire de la TFD

La définition de la transformée de Fourier discrète peut être ajustée de manière à définir un opérateur unitaire $\mathbb{C}^n \to \mathbb{C}^n$:

Proposition I.4.1. *Notons $\omega \stackrel{def}{=} e^{i\frac{2\pi}{n}}$ la racine primitive n-ème de l'unité. L'application $\widehat{\ }: \mathbb{C}^n \to \mathbb{C}^n$ définie par*

$$(q_0, q_1, \ldots, q_n) \mapsto (\widehat{q_0}, \widehat{q_1}, \ldots, \widehat{q_{n-1}}) \quad \text{où} \quad \widehat{q_k} \stackrel{def}{=} \frac{1}{\sqrt{n}} \sum_{\ell=0}^{n-1} q_\ell \omega^{-k\ell}$$

est un opérateur unitaire. Son inverse est donné par $q_\ell = \dfrac{1}{\sqrt{n}} \displaystyle\sum_{k=0}^{n-1} \widehat{q_k} \omega^{k\ell}$.

On peut constater qu'outre l'usage de la notation ω, la formulation de la Proposition I.4.1 diffère de celle des équations (I.1) et (I.2) en la « répartition » du terme de normalisation entre l'opérateur $\widehat{\ }$ et son inverse ($\frac{1}{\sqrt{n}}$ et $\frac{1}{\sqrt{n}}$ ici, 1 et $\frac{1}{n}$ en (I.1) et (I.2)). La preuve est reportée à la Section I.4.4. Pour le reste cette Section I.4, on utilise la notation $\widehat{\ }$ pour la TFQ (et non plus la TFD) et [2] $\omega \stackrel{\text{def}}{=} e^{i\frac{2\pi}{2^m}}$ pour la racine primitive 2^m-ème de l'unité.

Pour $n = 2^m$, l'opérateur $\widehat{\ }$ défini à la Proposition I.4.1 envoie un m-qubit sur un m-qubit. Cet opérateur est appelé *transformée de Fourier quantique* (TFQ).

I.4.2 Factorisation tensorielle de la TFQ

Afin de donner un algorithme quantique efficace de calcul de la TFQ, commençons par établir une factorisation tensorielle de cet opérateur. Cela nous demande d'établir quelques notations.

Pour tout mot $w \in \{0,1\}^m$, on numérote ses lettres par $w = w_1 w_2 \ldots w_m$ et on note $v(w)$ la valeur du nombre binaire qu'il code. Ainsi, pour $m = 3$ on a $v(000) = 0$, $v(001) = 1$, $v(010) = 2$, $v(011) = 3$, ... Plus généralement, $v(w) = \sum_{k=1}^{m} 2^{m-k} w_k$. On étend cette fonction $v()$ aux écritures binaires fractionnaires [3]. Ainsi, $v(10.1) = 2 + \frac{1}{2}$ et $v(0.011) = \frac{1}{4} + \frac{1}{8}$.

On rappelle que $\{|w\rangle : w \in \{0,1\}^m\}$ est l'ensemble des vecteurs de la base standard de $\mathbb{C}^{2^m}$, ordonnés par ordre croissant des valeurs $v(w)$, et que $|w\rangle = |w_1\rangle \otimes |w_2\rangle \otimes \ldots \otimes |w_m\rangle$. La transformée de Fourier quantique d'un vecteur $|w\rangle$ avec $w \in \{0,1\}^m$ se factorise comme suit :

Proposition I.4.2. *Pour tout $w \in \{0,1\}^m$ on a*

$$\widehat{|w\rangle} = \left(|0\rangle + e^{-2i\pi v(.w_m)}|1\rangle\right) \otimes \left(|0\rangle + e^{-2i\pi v(.w_{m-1}w_m)}|1\rangle\right) \otimes \ldots \otimes \left(|0\rangle + e^{-2i\pi v(.w_1 w_2 \ldots w_m)}|1\rangle\right).$$

La preuve de cette décomposition, essentiellement calculatoire, est reportée en Section I.4.4.

---

2. ω dépend de m, mais on laisse cette dépendance implicite pour alléger les notations.
3. Tout comme pour la notation décimale, les chiffres à gauche du point ont un poids 2^ℓ avec $\ell \geq 0$, et ceux à droite du point ont un poids 2^ℓ avec $\ell < 0$.

I.4.3 L'algorithme quantique de calcul de la TFQ

Comme on l'a vu au chapitre précédent, pour définir un (circuit réalisant un) opérateur unitaire, il suffit de le définir sur la base standard. On peut donc utiliser la factorisation donnée par la Proposition I.4.2 pour construire un circuit quantique calculant la QFT. Ce circuit est composé d'opérateurs H et R_ℓ, de matrices respectives

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \quad \text{et} \quad R_\ell = \begin{pmatrix}1 & 0\\ 0 & e^{-i\frac{2\pi}{2^\ell}}\end{pmatrix}.$$

On a déjà vu l'opérateur H, dit de Hadamard, au Chapitre 9. L'opérateur R_ℓ est appelé opérateur *de changement de phase*. On va aussi se servir des versions *contrôlées* des opérateurs R_ℓ.

D'après la Proposition I.4.2, Le premier qubit de $\widehat{|w\rangle}$ est $|0\rangle + e^{2i\pi v(.w_m)}|1\rangle$, c'est à dire $|0\rangle + |1\rangle$ si $w_m = 0$ et $|0\rangle - |1\rangle$ si $w_m = 1$. Autrement dit, le premier qubit de $\widehat{|w\rangle}$ s'obtient en appliquant une porte H au dernier qubit de $|w\rangle$.

Toujours d'après la Proposition I.4.2, le second qubit de $\widehat{|w\rangle}$ est

$$|0\rangle + e^{2i\pi v(.w_{m-1}w_m)}|1\rangle = |0\rangle + e^{2i\pi v(0.0w_m)} \cdot e^{2i\pi v(0.w_{m-1})}|1\rangle.$$

On peut l'obtenir en appliquant une porte H au $(m-1)$ème qubit de $|w\rangle$, puis en appliquant au résultat une porte R_2 *controlée* par le $(m-1)$ème qubit de $|w\rangle$.

Et ainsi de suite, on obtient le circuit de la Figure I.1. Il calcule la TFQ, à ceci près que les qubits de sortie sont inversés : le dernier qubit devrait être le premier, l'avant-dernier le deuxième, etc. Cela peut se corriger par l'ajout de m portes SWAP vues au Chapitre 9, que l'on omet ici pour des questions de place. Dans l'ensemble, on obtient le résultat suivant :

Théorème I.4.3. *Pour $n = 2^m$, le circuit de la Figure I.1 calcule TFQ de tout vecteur unitaire de $\mathbb{C}^n$ en $\Theta(\log^2 n)$ opérations quantiques élémentaires.*

I.4.4 Détail des preuves

Terminons ce chapitre par les preuves des Propositions I.4.1 et I.4.2.

Proposition. Notons $\omega \stackrel{\text{def}}{=} e^{i\frac{2\pi}{n}}$ la racine primitive n-ème de l'unité. L'application $\hat{}: \mathbb{C}^n \to \mathbb{C}^n$ définie par

$$(q_0, q_1, \ldots, q_n) \mapsto (\widehat{q_0}, \widehat{q_1}, \ldots, \widehat{q_{n-1}}) \quad \text{où} \quad \widehat{q_k} \stackrel{\text{def}}{=} \frac{1}{\sqrt{n}}\sum_{\ell=0}^{n-1} q_\ell \omega^{-k\ell}$$

est un opérateur unitaire. Son inverse est donné par $q_\ell = \dfrac{1}{\sqrt{n}}\displaystyle\sum_{k=0}^{n-1}\widehat{q_k}\omega^{k\ell}$.

Démonstration. Pour $0 \le k \le n-1$ notons

$$\chi_k \stackrel{\text{def}}{=} \frac{1}{\sqrt{n}}\begin{pmatrix}1\\ \omega^{-k}\\ \omega^{-2k}\\ \vdots\\ \omega^{-(n-1)k}\end{pmatrix},$$

de sorte que la matrice de l'opérateur $\widehat{\ }$ dans la base standard est $\begin{pmatrix}\chi_0 & \chi_1 & \dots & \chi_{n-1}\end{pmatrix}$. D'une part on a

$$\|\chi_k\|_2{}^2 = \langle \chi_k, \chi_k \rangle = \frac{1}{n} \sum_{\ell=0}^{n-1} \omega^{-\ell k} \cdot \overline{\omega^{-\ell k}} = \frac{1}{n} \sum_{\ell=0}^{n-1} \omega^{-\ell k} \cdot \omega^{\ell k} = 1.$$

D'autre part on a pour $0 \leq j \neq k \leq n-1$,

$$\begin{aligned}\langle \chi_j, \chi_k \rangle = \frac{1}{n} \sum_{\ell=0}^{n-1} \omega^{-\ell j} \cdot \overline{\omega^{-\ell k}} = \frac{1}{n} \sum_{\ell=0}^{n-1} \omega^{\ell(k-j)} &= \frac{1}{n} \sum_{\ell=0}^{n-1} \left(\omega^{k-j}\right)^{\ell} \\ &= \frac{1}{n} \cdot \frac{1-\left(\omega^{k-j}\right)^n}{1-\omega^{k-j}} = \frac{1}{n} \cdot \frac{1-1^{k-j}}{1-\omega^{k-j}} = 0.\end{aligned}$$

Les χ_k forment donc une base orthonormée, et l'opérateur $\widehat{\ }$ est bien unitaire. □

Proposition. Pour tout $w \in \{0,1\}^m$ on a

$$\widehat{|w\rangle} = \left(|0\rangle + e^{-2i\pi v(.w_m)}\,|1\rangle\right) \otimes \left(|0\rangle + e^{-2i\pi v(.w_{m-1}w_m)}\,|1\rangle\right) \otimes \ldots \otimes \left(|0\rangle + e^{-2i\pi v(.w_1 w_2 \ldots w_m)}\,|1\rangle\right).$$

Démonstration. La preuve est purement calculatoire. On commence par réécrire la définition de la TFQ avec les notations introduites :

$$\widehat{|w\rangle} = 2^{-m/2} \sum_{u \in \{0,1\}^m} \omega^{-v(w)v(u)}\,|u\rangle\,.$$

On décompose ensuite la sommation sur les 2^m mots u en un produit de sommations sur chacune des valeurs possibles pour les lettres,

$$\widehat{|w\rangle} = 2^{-m/2} \sum_{u_1=0}^{1} \sum_{u_1=0}^{1} \cdots \sum_{u_m=0}^{1} \omega^{-v(w)v(u_1 u_2 \ldots u_m)}\,|u\rangle\,,$$

et on substitue $v(u) = \sum_{k=1}^{m} 2^{m-k} u_k$:

$$\widehat{|w\rangle} = 2^{-m/2} \sum_{u_1=0}^{1} \sum_{u_1=0}^{1} \cdots \sum_{u_m=0}^{1} \omega^{-v(w)\cdot \sum_{k=1}^{m} 2^{m-k} u_k}\,|u\rangle\,.$$

On remplace l'exponentielle de la somme par le produit des exponentielles,

$$\widehat{|w\rangle} = 2^{-m/2} \sum_{u_1=0}^{1} \sum_{u_1=0}^{1} \cdots \sum_{u_m=0}^{1} \prod_{k=1}^{m} \omega^{-v(w)2^{m-k}u_k}\,|u\rangle\,,$$

puis on utilise le fait que tout vecteur $|u\rangle$ de la base standard est séparable

$$\widehat{|w\rangle} = 2^{-m/2} \sum_{u_1=0}^{1} \sum_{u_1=0}^{1} \cdots \sum_{u_m=0}^{1} \prod_{k=1}^{m} \omega^{-v(w)2^{m-k}u_k} \bigotimes_{k=1}^{m} |u_k\rangle\,.$$

La multilinéarité du produit tensoriel permet de distribuer les facteurs du produit ($\prod$) dans le produit tensoriel ($\otimes$) :

$$\widehat{|w\rangle} = 2^{-m/2} \sum_{u_1=0}^{1} \sum_{u_1=0}^{1} \cdots \sum_{u_m=0}^{1} \bigotimes_{k=1}^{m} \left(\omega^{-v(w)2^{m-k}u_k}\,|u_k\rangle\right).$$

À ce stade, on peut utiliser la bilinéarité du produit tensoriel pour regrouper les termes de mêmes $u_1, u_2, \ldots, u_{m-1}$ par paires,

$$\widehat{|w\rangle} = \left(2^{-m/2} \sum_{u_1=0}^{1} \sum_{u_1=0}^{1} \cdots \sum_{u_{m-1}=0}^{1} \bigotimes_{k=1}^{m-1} \left(\omega^{-v(w)2^{m-k}u_k} |u_k\rangle \right) \right) \otimes \left(\sum_{u_m=0}^{1} \omega^{-v(w)2^0 u_m} |u_m\rangle \right),$$

puis, regrouper les termes de mêmes $u_1, u_2, \ldots, u_{m-2}$ par paires, etc.

$$\widehat{|w\rangle} = 2^{-m/2} \bigotimes_{k=1}^{m} \left(\sum_{u_k=0}^{1} \omega^{-v(w)2^{m-k}u_k} |u_k\rangle \right).$$

Il suffit alors de développer la somme pour trouver $\widehat{|w\rangle} = 2^{-m/2} \bigotimes_{k=1}^{m} \left(|0\rangle + \omega^{-v(w)2^{m-k}} |1\rangle \right)$. Substituons alors ω :

$$\begin{aligned} \widehat{|w\rangle} &= 2^{-m/2} \bigotimes_{k=1}^{m} \left(|0\rangle + \omega^{-v(w)2^{m-k}} |1\rangle \right) \\ &= 2^{-m/2} \bigotimes_{k=1}^{m} \left(|0\rangle + \left(e^{i\frac{2\pi}{2^m}} \right)^{-v(w)2^{m-k}} |1\rangle \right) \\ \widehat{|w\rangle} &= 2^{-m/2} \bigotimes_{k=1}^{m} \left(|0\rangle + \left(e^{2i\pi} \right)^{-v(w)2^{-k}} |1\rangle \right) \end{aligned}$$

Le nombre $v(w)2^{-k}$ est le nombre fractionnaire d'écriture binaire

$$w_1 w_2 \ldots w_{m-k} \,.\, w_{m-k+1} w_{m-k+2} \ldots w_m.$$

On a par conséquent

$$\left(e^{2i\pi} \right)^{-v(w)2^{-k}} = \left(e^{2i\pi} \right)^{-v(w_1 w_2 \ldots w_{m-k} \,.\, w_{m-k+1} w_{m-k+2} \ldots w_m)} = \left(e^{2i\pi} \right)^{-v(0.\, w_{m-k+1} w_{m-k+2} \ldots w_m)},$$

puisque $\left(e^{2i\pi} \right)^{-\ell} = 1$ pour tout entier ℓ. On obtient donc

$$\widehat{|w\rangle} = \left(|0\rangle + e^{-2i\pi v(.w_m)} |1\rangle \right) \otimes \left(|0\rangle + e^{-2i\pi v(.w_{m-1}w_m)} |1\rangle \right) \otimes \ldots \otimes \left(|0\rangle + e^{-2i\pi v(.w_1 w_2 \ldots w_m)} |1\rangle \right),$$

comme annoncé. $\square$

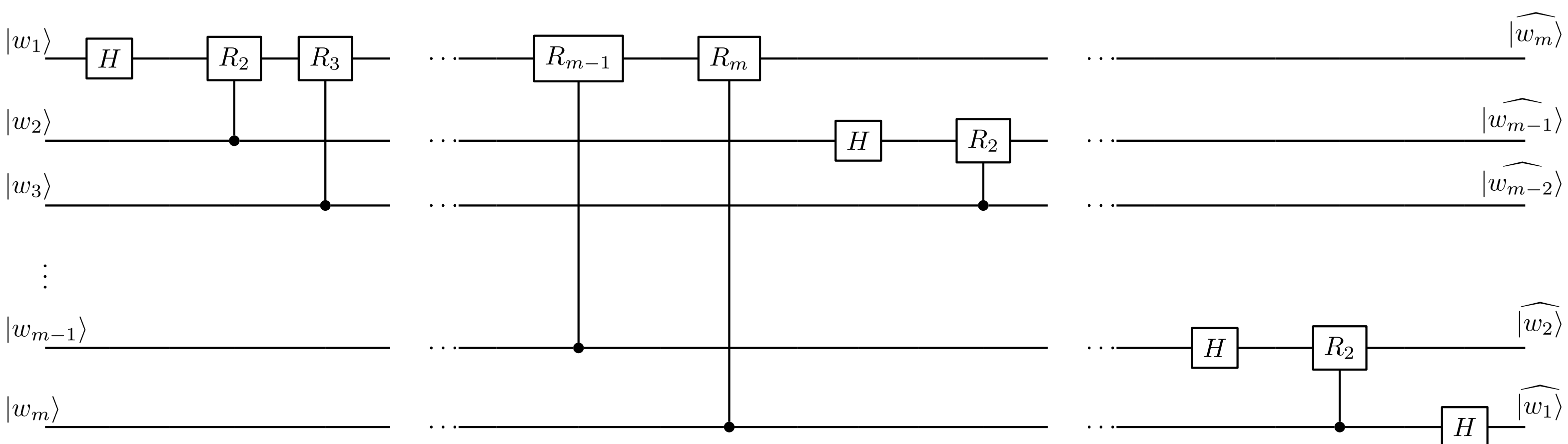


FIGURE I.1 – Circuit quantique calculant la TFQ, au renversement des qubits de sortie près.