

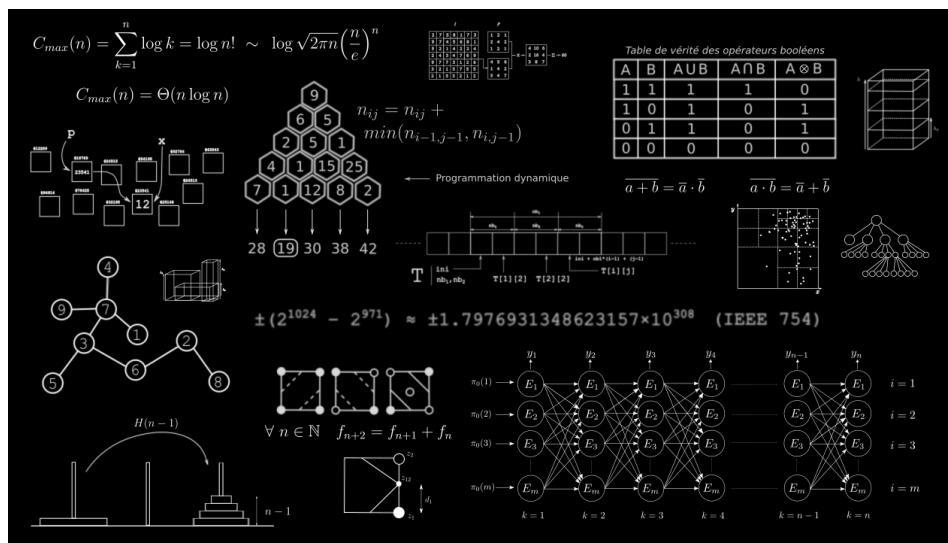
École Nationale des Sciences Géographiques

Algorithmique

Cycle Ingénieur 1^{ère} année



Année 2018-2019



Yann MÉNEROUX

Institut National de l'Information Géographique et Forestière

Direction de la Recherche et de l'Enseignement - Laboratoire LASTIG
73 Avenue de Paris, 94165 Saint-Mandé Cedex

Table des matières

Introduction	7
1 Le langage algorithmique ADL	11
1.1 Les bases du langage	13
1.1.1 Les identificateurs	13
1.1.2 Les types de données	13
1.1.3 La déclaration	15
1.1.4 L'affectation	16
1.1.5 Les expressions algébriques	18
1.1.6 Rappels d'algèbre booléenne	21
1.1.7 Les fonctions mathématiques	23
1.1.8 Synthèse	24
1.2 Les instructions de contrôle	25
1.2.1 Les disjonctions simples	25
1.2.2 Les disjonctions multiples	26
1.2.3 Les boucles à compteur	27
1.2.4 Les boucles tant que	29
1.2.5 Les boucles jusqu'à ce que	30
1.2.6 Les cas particuliers de boucle	31
1.2.7 Les débranchements de boucle	32
1.2.8 Les débranchements d'ordre n	33
1.3 La profondeur de code	34
1.3.1 Squelette d'un algorithme	34
1.3.2 Degré d'emboîtement	35
1.3.3 Optimisation du code	35
1.4 Les commentaires	37
1.5 Les modules	38
1.5.1 Définition	38
1.5.2 Débranchement de module	40
1.5.3 Gestion des erreurs	40
1.5.4 Paramètres formels et actuels	41
1.5.5 Entrées/sorties d'un module	42
1.5.6 Les types de modules	44
1.5.7 Les modules récursifs	46
1.6 Les chaînes de caractères	49
1.7 Exercices	60

2 Preuves d'un algorithme	67
2.1 Indécidabilité	68
2.1.1 Indécidabilité de la terminaison	69
2.1.2 Indécidabilité de la correction	71
2.2 Preuve de terminaison	71
2.2.1 La boucle à compteur	72
2.2.2 Autres types de boucles	73
2.2.3 Récursivité	74
2.2.4 Exercices	82
2.3 Preuve de correction	84
2.3.1 Programme simple	84
2.3.2 Programme avec boucles	84
2.3.3 Programme récursif	86
2.3.4 Exercices	87
3 Complexité algorithmique	89
3.1 Exemple d'introduction	90
3.2 Définitions	93
3.3 Unités	94
3.3.1 Taille des données	95
3.3.2 Coût d'exécution	96
3.4 Différentes complexités	97
3.5 Complexité multivariée	100
3.6 Notations de Landau	100
3.7 Classes de complexité	103
3.8 Théorème d'encadrement	106
3.9 Méthodologie	107
3.10 Cas des fonctions récursives	110
3.10.1 Récurrence linéaire simple	110
3.10.2 Récurrence linéaire multiple	113
3.10.3 Diviser pour régner	116
3.11 Pour aller plus loin...	129
3.12 Conclusions	129
3.13 Exercices	130
4 Structure de données	151
4.1 Tableaux	152
4.1.1 Définition	153
4.1.2 Tableau trié	155
4.1.3 Tri rapide	157
4.1.4 Tri de complexité optimale	161
4.1.5 Tri par dénombrement	162
4.1.6 Tri par paquets	163
4.2 Listes chaînées	165
4.2.1 Notion de pointeur	165
4.2.2 Liste simplement chaînée	167
4.2.3 Liste doublement chaînée	168
4.2.4 Liste chaînée circulaire	168
4.2.5 Fonctionnalités d'une liste chaînée	168
4.3 Files et piles	170

4.3.1	Définitions	170
4.3.2	Implémentation	170
4.4	Arbres	171
4.4.1	Définitions	171
4.4.2	Parcours d'un arbre	174
4.4.3	Arbres binaires de recherche	177
4.4.4	Exemples d'application	178
4.5	Tables de Hachage	185
4.5.1	Définition	185
4.5.2	Implémentation	189
4.6	Indexation spatiale	190
4.6.1	Exemple d'introduction	190
4.6.2	Quadtree	194
4.6.3	<i>Kd</i> -Tree	195
4.7	Exercices	197
5	La programmation dynamique	203
5.1	Mémoïsation	204
5.2	Théorie des Equations de Bellman	206
5.2.1	Formulation du problème	206
5.3	Principe général	210
5.3.1	Recherche exhaustive	210
5.3.2	Algorithme glouton	210
5.3.3	Programmation dynamique	211
5.3.4	Extension à la recherche du plus court chemin	213
5.4	Exemple d'application : produit matriciel	214
5.4.1	Recherche exhaustive	215
5.4.2	Résolution par programmation dynamique	215
5.4.3	Etude de la complexité	218
5.5	Exemple d'application 2 : calcul d'itinéraires	219
5.5.1	Recherche de toutes les longueurs optimales	219
5.5.2	Recherche d'un chemin le plus court	220
5.6	Exemple d'application 3 : estimation de temps de trajet	226
5.6.1	Formulation du problème	226
5.6.2	Complexité de l'approche naïve	231
5.6.3	Résolution par la programmation dynamique	232
5.6.4	Complexité de la solution	234
5.7	Exercices	235
A	Syntaxe ADL	247
B	Rappels de suites et séries	251
B.1	Suites et séries arithmétiques	251
B.2	Suites et séries géométriques	252
B.3	Suites arithmético-géométriques	252
B.4	Somme des puissances	253
B.5	Suites récurrentes d'ordre multiple	254
B.6	Série harmonique	256
B.7	Formule de Stirling	257
B.8	Nombres de Catalan	257

Index	259
Bibliographie	263

Introduction

Si l'histoire de la science informatique est relativement récente, et démarre dans les années 50 avec la naissance des premiers calculateurs électroniques, l'algorithmie¹, remonte en réalité à l'aube de la civilisation. Le premier abaque, que l'on pourrait qualifier d'ancêtre de la machine à calculer, a été créé par les babyloniens près de trois mille ans avant notre ère. Aux alentours du IIIe siècle avant J.C., Euclide donne son nom à un algorithme de calcul du plus grand diviseur commun de deux entiers naturels, bien connu de tous aujourd'hui. Le terme *algorithme* lui-même provient de la déformation du nom du mathématicien perse Al-Khwârizmî (780-850), que l'on considère souvent comme le père de l'algèbre. De fait, le développement de l'algèbre et de l'algorithmie sont intimement liés en cela qu'ils procèdent de la même logique : formaliser un problème à l'aide de variables littérales génératrices, et étudier la suite des opérations à mettre en oeuvre pour le résoudre. Tombé dans l'oubli après le XIIe siècle, le terme *algorismus*, version latinisée du nom Al-Khwârizmî, sera redécouvert sept siècles plus tard, période durant laquelle on assistera aux premiers travaux théoriques de grande ampleur, dont la naissance de l'algèbre booléenne, précurseur des circuits électroniques. En 1879, le mathématicien allemand Gottlob Frege développera le calcul propositionnel, avec le dessein de parvenir à une *lingua characterica*. Ce concept, introduit par Leibniz, visait à construire un langage universel, permettant de formaliser sans embellissement rhétorique, les pensées aussi bien mathématiques que physiques et philosophiques, devenant ainsi manipulables à l'aide de règles formelles bien définies. Il s'agit là du premier représentant des langages dits informatiques.

Le concept d'algorithmie, sera finalement théorisé par Alan Turing en 1936, avec la machine qui porte son nom, et qui à l'heure actuelle est encore (et plus que jamais) fréquemment invoquée dans de nombreux travaux d'informatique théorique. La seconde moitié du XX^e siècle, qui verra un développement exponentiel des calculateurs, constituera une formidable mise en application des principes développés jusque lors. L'on se gardera cependant de penser hâtivement que l'algorithmie a été rendue superflue par l'apparition des ordinateurs, et par le développement de logiciels d'édition de code toujours plus sophistiqués et *intelligents*. D'ailleurs, Michael Fellows avait écrit : "Computer science is not about machines, in the same way that astronomy is not about telescopes", indiquant par là que l'ordinateur n'est qu'un moyen d'observation de la science informatique.

Il existe de nombreuses définitions du mot algorithme. Dans ce cours, nous utiliserons l'acception suivante :

Un algorithme désigne la suite des opérations à mettre en oeuvre pour résoudre un problème.

Nous pourrions toutefois objecter que cette définition est à présent quelque peu désuète,

1. On emploiera aussi de manière équivalente le terme *algorithmique*

notamment depuis l'apparition des premiers processeurs parallèles, aujourd'hui embarqués dans toute machine digne de ce nom, sous forme de carte graphique. La notion d'algorithme ne désigne donc pas nécessairement une séquentialité d'opérations. Ces considérations dépassent cependant le cadre de ce cours, et nous nous référerons donc en général à la définition donnée ci-dessus.

L'algorithme, en tant qu'objet mathématique, sera donc au centre de nos préoccupations dans les pages qui suivent, et nous apprendrons à les concevoir, mais aussi à les comprendre et à les analyser. Cette phase de travail a été synthétisée par Toshio Kitagawa en 1986, qui introduisait le concept de *Brainware*, complétant ainsi au plus haut niveau le modèle à couches concentriques² et qu'il définit comme l'ensemble des concepts et des méthodes à mettre en oeuvre avant toute implantation en machine. Il ne s'agit donc pas d'une méthodologie, mais d'une philosophie, rappelant qu'il est inconcevable de commencer à écrire du code informatique sans procéder au préalable à une analyse méticuleuse sur papier.

Cette phase de brainware se détaille bien souvent en :

- Une phase de **modélisation du problème** (le développement d'un programme de gestion logistique d'une réseau ferroviaire, à titre d'exemple, nécessite au préalable une analyse aussi précise et exhaustive que possible, sous peine de finir avec un code buggé, inadapté et impossible à faire évoluer). Cette modélisation se fait à l'aide d'outils dédiés tels que la norme UML.
- Dans un second temps, on procède à l'écriture des briques algorithmiques.
- On analysera ensuite chacune d'entre elles, afin de s'assurer qu'elles terminent et qu'elles retournent le résultat attendu, quelque soient les valeurs numériques de ses entrées. Il ne s'agit évidemment pas de tester toutes les combinaisons possibles d'arguments, mais de mettre en oeuvre une démonstration mathématique rigoureuse. Cette phase est résumée sous le nom de **terminaison** et de **correction**.
- Dans le cas où l'application travaille avec une grande quantité de données (ce qui est de plus en plus fréquent de nos jours avec le phénomène du *big data*), l'analyse consistera également à calculer la **complexité de l'algorithme** conçu, ou autrement dit, à quantifier la manière dont le programme se comporte lorsque la charge de données à traiter augmente. Si la complexité obtenue s'avère être prohibitive, il sera alors nécessaire de reconcevoir certains algorithmes.

C'est uniquement lorsque toutes ces étapes ont été effectuées que l'on peut commencer l'implantation³ du programme en machine.

Ce cours ne traitera pas de la phase de modélisation du problème. Dans le chapitre 1, nous introduirons le langage algorithmique ADL, qui nous permettra d'écrire de manière simple, concise et rationnelle, l'ensemble de ce qui deviendra par la suite du code informatique. Des notions de preuves de correction et de complexité seront abordées respectivement dans les chapitres 2 et 3. Dans le chapitre 4, nous introduirons quelques structures permettant

2. Modèle traditionnellement utilisé en informatique, décrivant l'empilement des couches séparant le matériel de l'interface homme-machine

3. Dans ce cours, nous utiliserons improprement l'anglicisme *implémentation*.

de ranger les données de sorte à y accéder rapidement. Le recours à ces structures est bien souvent une condition *sine qua non* pour obtenir des programmes opérationnels et ce sera en particulier l'occasion d'aborder les notions d'indexation spatiale, indispensables au calcul efficace d'intersections entre objets géométriques. Enfin, nous conclurons en montrant comment il est possible, moyennant la sauvegarde d'informations judicieusement choisies, d'accélérer la résolution d'un grand nombre de problèmes.

Il est recommandé de suivre ce cours dans l'ordre des chapitres, chaque chapitre utilisant des notions présentées dans les chapitres précédents.

Chapitre 1

Le langage algorithmique ADL

Sommaire

1.1	Les bases du langage	13
1.1.1	Les identificateurs	13
1.1.2	Les types de données	13
1.1.3	La déclaration	15
1.1.4	L'affectation	16
1.1.5	Les expressions algébriques	18
1.1.6	Rappels d'algèbre booléenne	21
1.1.7	Les fonctions mathématiques	23
1.1.8	Synthèse	24
1.2	Les instructions de contrôle	25
1.2.1	Les disjonctions simples	25
1.2.2	Les disjonctions multiples	26
1.2.3	Les boucles à compteur	27
1.2.4	Les boucles tant que	29
1.2.5	Les boucles jusqu'à ce que	30
1.2.6	Les cas particuliers de boucle	31
1.2.7	Les débranchements de boucle	32
1.2.8	Les débranchements d'ordre n	33
1.3	La profondeur de code	34
1.3.1	Squelette d'un algorithme	34
1.3.2	Degré d'emboîtement	35
1.3.3	Optimisation du code	35
1.4	Les commentaires	37
1.5	Les modules	38
1.5.1	Définition	38
1.5.2	Débranchement de module	40
1.5.3	Gestion des erreurs	40
1.5.4	Paramètres formels et actuels	41
1.5.5	Entrées/sorties d'un module	42
1.5.6	Les types de modules	44
1.5.7	Les modules récursifs	46
1.6	Les chaînes de caractères	49
1.7	Exercices	60

Introduction

A l'aube de l'informatique, les données et les programmes étaient implantés dans la machine à l'aide d'un système de cartes perforées, dont le concept remonte en réalité à 1725. Nées dans l'esprit de l'inventeur du métier à tisser semi-automatique Basile Bouchon, elles seront par la suite utilisées en informatique, grâce à l'adaptation un siècle plus tard, du mathématicien britannique Charles Babbage. Ces cartes sont exprimées dans le seul langage compréhensible par un ordinateur, le langage binaire (un zone perforée dans la carte désignant un 0, son absence désignant un 1). Dans les années 50, les calculs scientifiques ont été considérablement accélérés par l'apparition des ces systèmes, bien que la conception des fiches d'observations sous formes de cartes, et le câblage de la matrice du système d'équations à résoudre, restaient des tâches longues et fastidieuses.

La seconde génération de programmeurs a eu la chance de se soustraire à cette étape, grâce à l'apparition des langages dits *assembleurs*, dont la lecture et la compréhension sont possibles pour un humain, mais dont la syntaxe reste toujours intimement liée au fonctionnement de la machine, et le programmeur ne pouvait faire abstraction de tâches annexes, telles que l'allocation de la mémoire. L'assembleur se charge alors de transcrire le programme en langage binaire en vue de son exécution par la machine.

De nos jours, on utilise des langages de troisième génération, beaucoup plus proches de l'humain, et le dispensant de la plupart des considérations bassement matérielles, lui permettant ainsi de se concentrer pleinement sur l'algorithme. Parmi ces langages, on établit généralement une hiérarchie, en qualifiant de *bas niveau* les langages qui restent néanmoins les plus proches de la machine tels que le C/C++. A l'opposée, les langages de *haut niveau*, comme le Java, sont plus proches du programmeur, en général plus faciles d'emploi mais moins flexibles et plus difficiles à optimiser. Le compilateur ou l'interpréteur se charge alors, via un programme assembleur, de convertir le code source en langage machine.

Il existe pléthore de langages informatiques, si bien qu'il est difficile de les dénombrer. De nouveaux se créent tous les jours, tandis que d'autres passent progressivement dans l'oubli. En 1994, Tim Robinson s'est attelé à recenser une collection de codes sources, écrits dans différents langages mais effectuant la même tâche algorithmique. Le projet contenait initialement 227 langages. Aujourd'hui, la plateforme (maintenue par différents successeurs), recense environ 1500 langages, dont la plupart sont réellement utilisés en pratique. Bien qu'intimidante pour les programmeurs débutants, cette grande variété de langage est en réalité une richesse incontestable, permettant ainsi de sélectionner le langage le plus adapté au type de tâche que l'on veut faire exécuter à la machine. C'est ainsi que la plupart des programmes de calculs scientifiques sont écrits en FORTRAN ou en Matlab, tandis que Pearl est traditionnellement associé au traitement de texte et Java aux applications destinées à être portées sur internet.

Cette grande variété pose néanmoins le problème de la communication entre programmeurs. Lorsque l'on souhaite décrire un algorithme, il est bien entendu impossible de l'écrire dans tous les langages existants à ce jour. En pratique, on a recours à du pseudo-code, en général fortement inspiré des langages les plus populaires comme le C, permettant de transcrire le code à l'aide d'un nombre minimal d'instructions communes à l'ensemble des langages. En 1977, François Bouillé, introduit le langage *Algorithm Description Language* (ADL), construit autour d'un cinquantaine de symboles et permettant de formuler de manière

simple et rationnelle¹ quelque algorithme qu'il soit.

En pratique, l'une des difficultés majeures pour les programmeurs débutants, est de saisir ce qu'il est autorisé d'écrire ou non dans un pseudo-code. Le papier ne compile pas le code produit, et ne renvoie donc pas d'erreur. Tous les raccourcis douteux sont ainsi autorisés dans la description de l'algorithme, et il en résulte la difficulté supplémentaire pour l'étudiant de savoir si son code a quelques chances de tourner en machine. Le langage ADL, par son formalisme rigoureux, couplée à sa simplicité, permet de décrire un code sans ambiguïté.

1.1 Les bases du langage

1.1.1 Les identificateurs

Tout programme informatique a pour vocation première de manipuler des données (bien souvent des quantités numériques, mais nous verrons que l'on peut également manipuler des chaînes de caractères), afin de produire un résultat. Ces quantités sont rangées en mémoire et les langages de programmation permettent d'y accéder via des noms de variables, que l'on appelle *identificateurs*.

Dans la norme du langage ADL, un identificateur commence nécessairement par une lettre, mais il peut contenir également des chiffres. VAR2 sera donc acceptable, contrairement à 2VAR. On proscrira également l'utilisation de caractères spéciaux (excepté le blanc souligné) ainsi que des espaces. Pour plus de lisibilité, on pourra remplacer l'identificateur MAVARIABLE par MA_VARIABLE.

Notons que les noms de variables sont sensibles à la casse, ainsi, l'identificateur *X* est différent de *x* et désigne donc potentiellement une donnée différente, comme c'est le cas dans de nombreux langages de programmation.

Remarquons également que dans la pratique, la plupart des normes de programmation recommandent de choisir des noms de variables en minuscule. En ADL, le choix est laissé libre, mais l'usage, probablement inspiré du FORTRAN, tend à dénoter les constantes en majuscules et les variables (tels que les identificateurs de boucle) en minuscule. Cette convention est bien entendu purement arbitraire et chacun est libre de la suivre ou non.

1.1.2 Les types de données

Nous avons vu dans la section précédente, que les identificateurs permettent de référencer des données. Intéressons-nous à présent au type de ces données. On distingue en général les données dites *scalaires* (constituées d'un seul élément) des données *structurées* (vecteurs, tableaux...). Notons dès à présent que les premières ne sont qu'un cas particulier des secondes et que la distinction n'a que vocation à être pédagogique et ne constitue pas une différence fondamentale dans l'utilisation concrète de la donnée.

1. Pour reprendre les termes de son inventeur

Les données scalaires

On parle aussi de données élémentaires. Elles sont constituées d'un unique élément, qui peut être :

- Un **entier**, positif ou négatif, sans restriction sur sa taille. En pratique, dans la plupart des langages de programmation, un entier (ou *integer*) est codé sur 4 octets (1 octet = 8 bits), et est donc contraint dans l'intervalle $[-2^{31}, 2^{31} - 1]$ (un bit est réservé pour le signe).
- Un nombre **décimal** (cette définition inclut ici également les nombres rationnels et irrationnels), sans restriction, ni sur sa taille, ni sur son nombre de décimales. Contrairement aux langages informatiques, ADL ne fait pas la distinction entre les nombres flottants en simple précision (*float*), codés 4 octets et les nombres en double précision (*double*) codés sur 8 octets.
- Un **booléen** (*bool*) permettant de symboliser les valeurs logiques VRAI et FAUX. En machine, 1 bit seulement est nécessaire à leur stockage en mémoire. En ADL, nous les noterons avec les symboles suivants :
 - VRAI : ↗
 - FAUX : ↘
- Une **chaîne de caractères** (*string*) de taille illimitée, notée entre double cotes (*e.g.* "ceci est un chaîne de caractères").

Par extension, lorsqu'un identificateur référence une donnée de type scalaire, on parle d'identificateur scalaire.

De ce qui précède, on comprend qu'ADL ne s'encombre pas des considérations sur les erreurs d'arrondis, indissociables de la machine utilisée et du langage de programmation vers lequel l'algorithme est porté. Cette étape est laissée à la charge du programmeur lors de la traduction. En ADL, l'espace mémoire allouée aux variables est considéré comme infini, et les calculs sont toujours exacts jusqu'à la dernière décimale.

Les données dimensionnées

On parle aussi de données structurées. Dans cette première partie de cours, nous ne consi-
dérerons que les tableaux. Nous verrons au chapitre 4 qu'il est possible de définir d'autres types de données dimensionnées. Nous ne traitons ici que des types présents de manière native dans tout langage de programmation, aussi bas niveau soit-il.

Un tableau est un ensemble d'éléments scalaires de **même type**, organisés suivant des colonnes et des lignes.

- Lorsqu'un tableau ne contient qu'une seule colonne (ou de manière équivalente qu'une seule ligne, la machine est indifférente à l'orientation), on parle de *vecteur*. On note V_i la i^{eme} composante d'un vecteur dont l'identificateur est V . Bien évidemment, i ne peut pas être supérieur à la taille du vecteur.

- Lorsqu'un tableau contient 2 dimensions, on parle de *matrice*, dont les éléments sont identifiés par M_{ij}
- Pour des tableaux de dimension supérieure, on utilise la terminologie de *tenseur*. Ainsi, les éléments d'un tenseur de dimension 3 dont l'identificateur est T , sont dénotés T_{ijk} .

Par extension, lorsqu'un identificateur référence une donnée dimensionnée, on parle d'identificateur dimensionné.

Remarque 1 : lorsqu'un tableau comprend un nombre élevé de dimensions (par exemple 5 ou 6), afin de clarifier la notation, on pourra avoir recours à des indices et des exposants : T_{ijk}^{lm} . Contrairement au domaine de la physique, la position d'une dimension est purement arbitraire, mais doit rester le même tout au long de l'algorithme. On veillera cependant à placer les indices de manière logique. Par exemple, pour un tenseur de dimension 3, dont chaque élément représente le nombre de véhicules circulant sur un réseau routier entre la ville i et la ville j à l'instant t , il paraît plus rationnel de noter T_{ij}^t plutôt que T_{it}^j .

Remarque 2 : contrairement à la plupart des langages informatiques, ADL fait le choix de commencer la numérotation des indices de tableau à 1 (et non à 0). Les trois éléments d'un vecteur V de taille 3 sont V_1 , V_2 et V_3 .

Remarque 3 : une donnée scalaire est un cas particulier de donnée structurée, en tant que tenseur de dimension 0 et contenant un unique élément.

Remarque 4 : lorsqu'il y a un risque de confusion, les indices pourront être séparés par des virgules (e.g. $M_{11,21}$ désigne l'élément située en 11^{eme} ligne et en 21^{eme} colonne de la matrice M , ce qui est différent de $M_{1,121}$ ou encore $M_{112,1}$).

Insistons à nouveau sur le fait qu'un tableau ne peut contenir des données de types différents. Par exemple, le vecteur : $V = [1, \text{♪}, 2.5, \text{"un mot"}, 3.14]$ est incorrect.

1.1.3 La déclaration

En informatique, la déclaration indique la partie du code où on annonce que l'on va utiliser un identificateur de variable. Cela correspond par exemple en mathématiques à la partie d'un énoncé où l'on écrit : "soit $x \in \mathbb{R}$..." Elle comprend deux informations principales : le nom de l'identificateur (ici x) et son type (\mathbb{R}). Lorsque l'on déclare une données structurée (e.g. $A \in \mathcal{M}_{np}(\mathbb{R})$), on spécifie également son nombre de dimensions (ici 2) et la taille de chaque dimension (n et p).

En ADL, **rien de tout ceci** n'est nécessaire ! La déclaration d'une variable est implicite lors de sa première affectation (voir section suivante). L'identificateur prend alors le type de la première donnée qu'on lui affecte. Par exemple, si l'on écrit : "soit $x = 3.14$ et $V = [1, 2, 3]$ ", l'identificateur X est automatiquement typé comme référençant une donnée scalaire numérique tandis que V est une donnée structurée sous forme de tenseur numérique de dimension 1. Nous verrons par la suite que la taille d'un tableau est extensible au

cours du déroulement de l'algorithme.

1.1.4 L'affectation

L'affectation consiste à associer une donnée à un identificateur. On la note :

IDENTIFICATEUR \leftarrow constante ou expression algébrique

Notons que la plupart des langages de programmation ont adopté le signe $=$ pour symboliser l'affectation, ce qui introduit bien souvent des confusions parmi les étudiants. Contrairement à l'égalité mathématique, l'affectation n'est pas symétrique. La partie gauche de l'instruction contient obligatoirement un identificateur, tandis que la partie droite est au choix une constante, un identificateur, ou bien encore une expression algébrique (comportant alors des constantes, des identificateurs et, comme nous le verrons plus tard, des fonctions).

Exemples :

- PI \leftarrow 3.14
- B \leftarrow PI
- C \leftarrow 5
- A \leftarrow (B \wedge 2 + C \wedge 2) \wedge (1/2)

Remarque : tous les identificateurs utilisés en partie droite de l'affectation doivent avoir été préalablement définis.

Les exemples ci-dessous sont incorrects :

- 3.14 \leftarrow PI
- A \wedge 2 \leftarrow B \wedge 2 + C \wedge 2

Lorsqu'un type de données a été affecté à un identificateur, rien n'interdit en théorie de lui réaffecter une donnée de type différent. Par exemple :

VARIABLE \leftarrow 2.718; VARIABLE \leftarrow †;

A chaque instant du programme, l'identificateur VARIABLE référence la dernière donnée qui lui a été affectée. Par souci de lisibilité du code, sauf cas très particulier, on évitera de changer le type référencé par un identificateur donné.

Exercice 1.1. Ecrire le morceau de code permettant d'échanger les valeurs contenues dans 2 identificateurs scalaires donnés N1 et N2.

Cas particulier des données structurées :

Dans un tableau, l'affectation se fait cellule par cellule.

Exemple :

```
V1 ← 37.2;
V2 ← 48.3;
V3 ← 14.7;
```

Lors de l'affectation d'une cellule V_j donnée d'un tableau, on s'assurera d'avoir au préalable défini V_i pour tout $1 \leq i \leq j - 1$, bien qu'il ne s'agisse pas là d'une obligation spécifiée par ADL.

Exercice 1.2. Soit un vecteur $V = [1, 2, 3]$ Que retourne l'algorithme suivant ?

```
V1 ← V1/(V1 + V2 + V3);
V2 ← V2/(V1 + V2 + V3);
V3 ← V3/(V1 + V2 + V3);
```

Dans le cas où la partie gauche de l'affectation est un identificateur de tableau (de dimension quelconque supérieure ou égale à 1), l'indice du tableau peut contenir une expression algébrique. Ainsi, les deux instructions suivantes sont correctes :

```
Vi+j ← Vi + Vj
MVi,k ← 62.0
```

Nous venons ainsi de définir notre premier type d'instruction exécutable. Il s'agira en fait de la seule instruction exécutable que nous utiliserons. Il se trouve donc qu'un programme, en tant que succession d'instructions exécutables, n'est rien d'autre qu'une suite d'affectations de variables. Dans la partie suivante, nous définirons un second type d'instruction, que l'on appellera *instruction de contrôle*, et qui permettra de fournir au code toute sa richesse et sa profondeur algorithmique.

Le lecteur attentif aura remarqué que lorsque deux instructions exécutables sont formulées sur la même ligne, elles sont obligatoirement séparées par un point-virgule.

Par abus de notation, nous autoriserons dans ce cours l'affectation directe d'un tableau. Par exemple, pour copier le contenu d'un vecteur V dans un second vecteur V_{bis} , nous noterons :

$$V_{bis} ← V$$

Dans ce cas, tous les éléments dans le tableau de droite sont simultanément affectés aux cellules d'indices correspondant dans le tableau positionné à gauche de l'affectation. Si ce tableau a précédemment été défini, on veillera à ce qu'il ait les mêmes dimensions que V .

1.1.5 Les expressions algébriques

Nous avons vu dans la section précédente comment référencer des données à l'aide d'identificateurs afin de les manipuler. Cette fonctionnalité n'aurait que peu d'intérêt s'il n'était pas possible d'effectuer des opérations (entre autres arithmétiques, mais pas uniquement). C'est justement ce que nous allons traiter dans ce paragraphe.

Pour rappel, une expression algébrique ne peut se trouver qu'en partie droite de l'affectation, sauf en indice d'un tableau, auquel cas elle peut également figurer en partie gauche. Elle contient généralement, des constantes (numériques, booléennes ou encore des chaînes de caractères), des identificateurs (auxquels on aura bien entendu préalablement affecté une valeur), des opérateurs et des fonctions (nous reviendrons plus en détail sur les fonctions à la fin de ce chapitre, lorsque nous parlerons des modules), le tout éventuellement parenthésé s'il est nécessaire de redéfinir la priorité des opérations.

Voici un exemple d'affectation avec une opération algébrique en partie droite :

$$A \leftarrow 2 * B + C - 4 * (2/NB + 7)$$

Dans cette expression, 2, 4 et 7 sont des constantes numériques tandis que B , C et NB sont des identificateurs préalablement définis. Notons que A peut avoir été défini précédemment mais il peut tout aussi bien s'agir de sa première affectation.

Le résultat de l'expression algébrique est alors calculé avant d'être stocké dans la variable A . Considérons un autre exemple :

$$T_{i+j,2*i-7} \leftarrow 17 * A + 2$$

Les deux expressions algébriques en indices du tableau sont calculées, puis le résultat de l'expression algébrique de la partie droite est affecté à la cellule correspondante de T .

Remarquons que les expressions algébriques ne sont pas nécessairement numériques et peuvent aussi contenir simultanément plusieurs types de données. Par exemple :

$$X \leftarrow B \cap (x = 37 + z)$$

avec : X et B des identificateurs booléens, x et z des identificateurs numériques. L'opérateur \cap (AND) prend comme opérandes deux booléens et retourne un booléen. L'opérateur $=$ prend en entrée deux variables de même type (numériques, chaînes de caractères ou booléens) et teste leur égalité (il retourne donc un booléen).

Si la manipulation des opérateurs algébriques ($+, -, *, ...$) ne pose a priori pas de problème, il convient ici de faire un petit rappel sur les opérateurs booléens.

Opérateurs à opérandes et résultat booléens

Il s'agit des opérateurs qui prennent en entrée un ou deux booléens pour donner un résultat de type également booléen. Nous récapitulons ci-dessous, les opérateurs de ce type, avec

leurs définitions associées.

Opérande 1	Opérande 2	AND	OR	XOR	NAND	NOR
a	b	$a \cap b$	$a \cup b$	$a \oplus b$	$a \boxtimes b$	$a \boxplus b$
\top	\top	\top	\top	\downarrow	\downarrow	\downarrow
\top	\downarrow	\downarrow	\top	\top	\top	\downarrow
\downarrow	\top	\downarrow	\top	\top	\top	\downarrow
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\top	\top

TABLE 1.1 – Opérateurs binaires à opérandes et résultat booléens

Opérande	NOT
a	\bar{a}
\top	\downarrow
\downarrow	\top

TABLE 1.2 – Opérateur unaire à opérande et résultat booléens

Exercice 1.3. Démontrer que, pour tout $n \in \mathbb{N}^*$ l'expression suivante est vérifiée si et seulement si un nombre impair de ses identificateurs sont vrais.

$$S_n = A_1 \oplus A_2 \oplus A_3 \oplus \dots \oplus A_n$$

Opérateurs de comparaison

Comme leur nom l'indique, ces opérateurs sont utilisés pour comparer deux données (nous nous restreindrons ici à des données de type numérique, mais ils peuvent aisément être généralisés à d'autres types).

- Egalité $=$: renvoie \top si et seulement si ses deux opérandes sont égales.
- Différent \neq : renvoie \top si et seulement si ses deux opérandes ne sont pas égales. Notons que $A \neq B$ est équivalent à $\overline{A = B}$.
- Supérieur $>$: renvoie \top si et seulement si sa première opérande est supérieure à la seconde.
- Supérieur ou égal \geq : renvoie \top si et seulement si sa première opérande est supérieure ou égale à la seconde.
- Inférieur $<$: renvoie \top si et seulement si sa première opérande est inférieure à la seconde.
- Inférieur ou égal \leq : renvoie \top si et seulement si sa première opérande est inférieure ou égale à la seconde.

Remarque : les opérateurs prenant une seule opérande sont qualifiés de *unaires*. Ceux qui prennent deux opérandes sont qualifiés de *binaires*.

Pour conclure cette partie, nous rappelons ci-dessous la liste des opérateurs disponibles en ADL, avec les types de données requis en entrée et les ordres de priorité.

Opérateur	Symbol	Type	Entrée(s)	Sortie	Priorité
Exponentiation	\wedge	binnaire	numériques	numérique	1
Multiplication	*	binnaire	numériques	numérique	2
Division	/	binnaire	numériques	numérique	2
Division entière	\div	binnaire	numériques	numérique	2
Concaténation	&	binnaire	alphanum.	alphanum.	2
Addition	+	binnaire	numériques	numérique	3
Soustraction	-	binnaire	numériques	numérique	3
Moins monadique	-	unaire	numérique	numérique	3
Supérieur	>	binnaire	numériques	booléenne	4
Supérieur ou égal	\geqslant	binnaire	numériques	booléenne	4
Egal	=	binnaire	-	-	4
Different	\neq	binnaire	-	-	4
Inférieur	<	binnaire	numériques	booléenne	4
Inférieur ou égal	\leqslant	binnaire	numériques	booléenne	4
Négation	\bar{a}	unaire	booléenne	booléenne	5
Conjonction	\cap	binnaire	booléennes	booléenne	6
Nand	\boxtimes	binnaire	booléennes	booléenne	6
Disjonction	\cup	binnaire	booléennes	booléenne	7
Nor	\boxdot	binnaire	booléennes	booléenne	7
Xor	\oplus	binnaire	booléennes	booléenne	7

TABLE 1.3 – Liste des opérateurs à une ou deux opérandes

Notons que dans le tableau ci-dessus, l'opération est effectuée d'autant plus tôt au sein de l'expression algébrique, que son code de priorité est faible.

Le symbole – dans les colonnes entrée(s) et sortie, signifie que le type est quelconque (numérique, alphanumérique ou booléen), mais qu'il doit être le même de part et d'autre de l'opérateur ainsi que dans la sortie.

Remarquons qu'il est possible de considérer l'affectation comme un ultime opérateur de priorité 8, prenant en entrée un identificateur et une expression algébrique et ne produisant aucun résultat en sortie.

Exercice 1.4. Indiquer la valeur de b après l'exécution des instructions suivantes :

$x \leftarrow 3; y \leftarrow 4; z \leftarrow 5; s \leftarrow \text{"voici une chaîne"};$
 $b \leftarrow (z \wedge 2 = x \wedge 2 + y \wedge 2) \cup (z \leqslant x + y) \cap (s \neq \text{"voici une chaîne"})$;

1.1.6 Rappels d'algèbre booléenne

Dans les propositions qui suivent, on note $\mathbb{B} = \{\downarrow, \uparrow\} = \{0, 1\}$, l'ensemble des valeurs booléennes (on parle aussi de valeurs logiques).

L'ensemble \mathbb{B} , muni des lois de compositions internes \cup et \cap (que l'on notera ici $+$ et \cdot), possède les propriétés suivantes :

Commutativités :

$$\forall a, b \in \mathbb{B}$$

$$a + b = b + a \quad a \cdot b = b \cdot a$$

Associativités :

$$\forall a, b, c \in \mathbb{B}$$

$$a + (b + c) = (a + b) + c = a + b + c \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot b \cdot c$$

Distributivités mutuelles :

$$\forall a, b, c \in \mathbb{B}$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \quad a + (b \cdot c) = (a + b) \cdot (a + c)$$

Notons que par convention, la loi \cdot est prioritaire sur la loi $+$.

Eléments neutres :

$$\forall a \in \mathbb{B}$$

$$a + 0 = a \quad a \cdot 1 = a$$

La valeur booléenne 1 est l'élément neutre de la loi \cdot tandis que 0 est celui de la loi $+$.

Idempotences :

$$\forall a \in \mathbb{B}$$

$$a + a + \dots + a = a \quad a \cdot a \cdot \dots \cdot a = a$$

Eléments absorbants :

$$\forall a \in \mathbb{B}$$

$$a + 1 = 1 \quad a \cdot 0 = 0$$

La valeur booléenne 0 est l'élément absorbant de la loi \cdot tandis que 1 est celui de la loi $+$.

Complémentaires :

$$\forall a \in \mathbb{B}$$

$$a + \bar{a} = 1 \quad a \cdot \bar{a} = 0 \quad \bar{\bar{a}} = a$$

La fonction complémentaire est involutive.

Lois de De Morgan :

$$\forall a, b \in \mathbb{B}$$

$$\overline{a + b} = \bar{a} \cdot \bar{b} \quad \overline{a \cdot b} = \bar{a} + \bar{b}$$

Les lois de De Morgan autorisent la pseudo-distributivité de la négation sur les lois de compositions, à condition de changer l'opérateur.

Remarque : d'un point de vue algébrique, \mathbb{B} est un ensemble ordonné à deux éléments dont les lois de composition internes peuvent être définies à l'aide des bornes suivantes :

$$a + b = \sup(a, b) \\ a \cdot b = \inf(a, b)$$

La structure $(\mathbb{B}, +, \cdot)$ est donc un treillis algébrique et on pourra également la noter en terme de relation d'ordre : (\mathbb{B}, \subseteq) avec :

$$a \subseteq b \Leftrightarrow a + b = b \Leftrightarrow a \cdot b = a$$

On appelle *littéral* un terme composé d'un unique identificateur logique, ou bien de sa négation. Ainsi, a , \bar{a} , b , \bar{b} ... sont des exemples de littéraux. En revanche $a + b$, $a\bar{b}$, $a + bc$... n'en sont pas. On appelle *clause disjonctive* une disjonction de littéraux, par exemple : $a + b$, $b + \bar{c} + d$... De même, une *clause conjonctive* est une conjonction de littéraux : ab , $\bar{a}\bar{b}c$... On dit qu'une formule logique est sous *Forme Normale Disjonctive* (FND) lorsqu'elle s'exprime par une disjonction de clauses conjonctives. De la même manière, une *Forme*

Normale Conjonctive (FNC) est une conjonction de clauses disjonctives. On peut facilement se convaincre que toute formule logique peut être transformée en FND ou en FNC à l'aide des lois de l'algèbre booléenne énoncées ci-avant. Par exemple :

$$a(\bar{b} + cd) = a\bar{b} + acd = a(\bar{b} + c)(\bar{b} + d)$$

Exercice 1.5. Montrer que l'opérateur \oplus (XOR) peut s'exprimer sous la forme normale conjonctive : $a \oplus b = (a + b) \cdot (\bar{a} + \bar{b})$.

Exercice 1.6. Simplifier les expressions ci-dessous.

$$U = (\bar{a} + b) \cdot (a + b + d) \cdot \bar{d}$$

$$V = (ab + c + d) \cdot ab$$

$$W = c \cdot (b + c) + (a + d) \cdot (\bar{a} + d) \cdot \bar{c}$$

$$X = abc + ab\bar{c} + a\bar{b}c$$

$$Y = abc + ab\bar{c} + a\bar{b}c + a\bar{b}\bar{c}$$

$$Z = \overline{c \cdot (a + \bar{a}b)}$$

Réciproquement, pour trouver l'expression logique correspondant à une table de vérité donnée, on pourra utiliser un tableau de Karnaugh.

1.1.7 Les fonctions mathématiques

Nous avons vu précédemment qu'une expression algébrique contenait des identificateurs (auxquels on avait au préalable affecté une valeur), des constantes et des opérateurs. Nous ajoutons ici un quatrième et dernier type de composant : les fonctions mathématiques.

Nous verrons plus loin qu'il s'agit en fait d'un cas particulier de *module*, et le programmeur aura tout loisir d'en définir autant qu'il souhaite. Cependant, tout langage de programmation met en général à disposition de l'utilisateur, une liste de modules mathématiques, appelée bibliothèque de fonctions. Cette liste, dont la longueur varie d'un langage de programmation à l'autre², comprend l'ensemble des fonctions indispensables à la plupart des programmes. En ADL, nous nous restreindrons aux 23 fonctions suivantes :

Fonctions de comparaison : MIN et MAX, prenant chacune en entrée 2 arguments

Fonction de conversion : FLOOR, renvoyant la partie entière d'un nombre décimal.

La fonctions racine carrée : SQRT, prenant en entrée un nombre positif ou nul.

La fonctions valeur absolue : ABS, prenant en entrée un nombre quelconque.

Fonctions exponentielle et logarithmique : EXP et LOG. Notons qu'il s'agit là du logarithme naturel, c'est-à-dire en base e .

2. A titre d'exemple, la librairie standard de Java contient une vingtaine de fonctions mathématiques. Dans ses premières versions FORTRAN en contenait déjà pas loin d'une centaine

Fonctions circulaires : SIN, COS et TAN, prenant en argument un nombre exprimé en radians

Fonctions circulaires réciproques : ASIN, ACOS et ATAN, prenant en argument un nombre exprimé en radians. Cette liste sera complétée par la fonction ATAN2, prenant en entrée deux nombres a et b et retournant l'argument du nombre complexe $a + bi$.

Fonctions hyperboliques : SINH, COSH et TANH

Fonctions hyperboliques réciproques : ASINH, ACOSH et ATANH

La fonction modulo : MOD, prenant en entrée deux nombres entiers naturels a et b et retournant le reste de la division de a par b . On la remplace parfois par l'opérateur % : MOD(a,b) = a % b.

La fonction aléatoire : RAND, ne prenant aucune entrée et retournant un nombre (pseudo) aléatoire tiré suivant une loi uniforme sur l'intervalle $[0, 1]$.

A cette liste classique et dans le but de simplifier le code produit dans ce cours, nous ajouterons la fonction SIZE, prenant en entrée un tenseur T de dimension $n \in \mathbb{N}$ quelconque, et produisant en sortie, un vecteur de taille n et dont chaque élément vaut la taille de la dimension correspondante dans T .

Par exemple, si T est un tenseur cubique de dimension $3 \times 200 \times 15$, l'appel de SIZE(T) retournera le vecteur :

$$V = [3, 200, 15]$$

On pourra alors aisément obtenir la dimension d'un tenseur quelconque par composition :

$$\text{nb_dimensions} \leftarrow \text{SIZE}(\text{SIZE}(T))$$

Dans le cas particulier où T est un vecteur (donc à une seule dimension), on obtiendra sa taille par :

$$\text{taille_vecteur} \leftarrow \text{SIZE}(T)$$

Ici, la variable de sortie taille_vecteur est un vecteur de taille 1, que l'on peut alors assimiler à un scalaire.

Notons que si T est un scalaire, alors SIZE(T) retourne la valeur 1.

Exercice 1.7. Sans utiliser la fonction MOD ni la division entière, écrire un programme calculant le reste et le quotient de la division euclidienne de deux entiers.

Nous admettons également en ADL l'utilisation de la constante PI sans définition préalable.

1.1.8 Synthèse

Nous avons donc vu dans cette première partie, comment effectuer un calcul et affecter le résultat dans une variable (scalaire ou dimensionnée). Nous donnons ci-dessous l'expres-

sion la plus générale possible d'une affectation (le symbole $< . >$ dénote une composante facultative de l'expression).

IDENTIFICATEUR $_{<ea_1,ea_2,\dots ea_n>}$ \leftarrow ea

où : $ea, ea_1, ea_2\dots$ désignent des expressions algébriques contenant des identificateurs, des constantes, des opérateurs et des fonctions. L'identificateur de la partie gauche de l'affectation peut être un scalaire ou un tableau (auquel cas ses indices seront également exprimés sous forme d'expression algébrique).

Remarque : l'instruction simple $A \leftarrow 2$ est également couverte par l'expression ci-dessous, 2 étant un cas particulier d'expression algébrique sans opérateur, sans fonction et sans identificateur.

1.2 Les instructions de contrôle

Dans cette seconde section, nous passons en revue les instructions qui permettent de modifier le flux d'exécution du code. Notons que contrairement aux instructions détaillées précédemment, ce nouveau type d'instruction ne modifie pas directement l'état de la machine, mais seulement l'ordre d'exécution des instructions élémentaires.

Il en existe trois types principaux :

- Les *disjonctions* permettent, en fonction de l'occurrence d'un certain évènement, de spécifier plusieurs lignes différentes d'exécution d'un programme. En fonction de la réalisation de l'évènement ou non, l'algorithme suit un cheminement différent dans l'architecture du code.
- Les *boucles* sont utilisées pour répéter un grand nombre de fois une série d'instructions (on parle d'itération pour définir une passe dans la boucle).
- Les *débranchements* permettent de quitter prématurément l'exécution d'une boucle d'instructions.

Ces trois types d'instructions de contrôle sont alors combinés pour créer un programme complexe.

1.2.1 Les disjonctions simples

On parle aussi de *branchement*. Une disjonction simple désigne une instruction du type :

Si (condition) alors faire quelque chose sinon faire autre chose

Elle comporte trois parties :

- Une condition, exprimée sous forme booléenne, et dont le résultat (vrai ou faux) indiquera le branchement à suivre dans le programme. Il s'agit d'une expression algébrique dont le résultat est un booléen.
- Une liste d'instructions à exécuter lorsque la condition est remplie (lie_1).
- Une liste alternative d'instructions à exécuter dans le cas contraire (lie_2).

En ADL, on la note :

condition ? lie₁ | lie₂ ;

Exemple : calcul de la valeur absolue d'un nombre (tiré au hasard entre -100 et 100)

$X \leftarrow 200 * \text{RAND}() - 100; X \geq 0 ? \text{RESULTAT} \leftarrow X | \text{RESULTAT} \leftarrow -X ;$

Remarque : lorsqu'il n'y a pas d'instructions à effectuer dans le cas où la condition n'est pas remplie, on peut noter :

condition ? lie | ;

Exemple : réordonnant de 2 variables (tirées au hasard).

$A \leftarrow \text{RAND}(); B \leftarrow \text{RAND}(); A < B ? C \leftarrow A; A \leftarrow B; B \leftarrow C | ;$

Remarque : si b est un identificateur de type booléen, l'instruction $b = \uparrow ?$ est superflue. Il est suffisant d'écrire : $b ?$ et la même remarque tient pour \downarrow et $\bar{b} ?$.

1.2.2 Les disjonctions multiples

On parle aussi de *branchement multiple*. Une disjonction multiple désigne une instruction du type :

Si une variable vaut a_1 alors faire quelque chose
 si elle vaut a_2 alors faire quelque chose d'autre
 ...
 si elle vaut a_n alors faire quelque chose d'autre
 sinon faire quelque chose d'autre

Notons que la variable en question peut être de type numérique ou chaîne de caractères. En toute rigueur, il peut aussi s'agir d'une variable booléenne, mais le débranchement multiple n'a alors que peu d'intérêt dans ce cas, puisque l'on n'aura que deux modalités.

En ADL, l'instruction de débranchement multiple s'écrit :

IDENTIFICATEUR ↗
 || ea₁ ⇒ lie₁
 || ea₂ ⇒ lie₂
 ...
 || lie_n ↘

C'est instruction n'est pas souvent utilisée en pratique. D'autre part, on peut aisément montrer qu'elle peut être traduite uniquement à l'aide de l'instruction de débranchement simple. Il s'agit donc d'une convention de notation, permettant ainsi de limiter le nombre de conditions dans le code, afin de le rendre plus lisible.

1.2.3 Les boucles à compteur

Une boucle à compteur permet de répéter une liste d'instructions un certain nombre de fois. Chaque passe dans la boucle est appelée une itération. Le nombre d'itérations est conditionné par trois entiers :

- La borne inférieure b_i indique la valeur initiale du compteur.
- La borne finale b_f indique la valeur finale du compteur. Lorsque le compteur dépasse cette valeur, le programme quitte la boucle.
- Le pas p désigne la quantité à ajouter au compteur à chaque fin d'itération.

En ADL, on utilise les notations suivantes :

{_{k:b_i}^{b_{f,p}} lie }_k

Dans cette expression, l'identificateur k contient la valeur du compteur de boucle. Il peut être appelé dans la liste d'instructions exécutables, y compris après la sortie de boucle (dans ce cas, il gardera la valeur qui lui a été affectée lors du dernier passage dans la boucle).

Notons que si b_i , b_f et p peuvent être des expressions algébriques, k est nécessairement un identificateur scalaire.

Exemple : calcul de la factorielle de n

Il s'agit là du "Hello World!" de la programmation itérative. On suppose que n est un identificateur scalaire contenant une valeur numérique quelconque.

$$N \leftarrow 1; \{_{i:2}^{n,1} N \leftarrow N * i\}_i$$

Dans le cas particulier où le pas d'incrémentation vaut 1, on pourra l'omettre dans la description de la boucle.

$$N \leftarrow 1; \left\{ \begin{array}{l} N \leftarrow N * i \\ \end{array} \right\}_i^n$$

Dans certains cas, la valeur de l'incrément peut être négative, dans ce cas, la borne finale sera en principe plus petite que la borne initiale et le programme quittera la boucle dès lors que le compteur sera plus petit que la borne finale.

Remarque importante : la borne finale est inclue dans l'itération. Ainsi, dans l'exemple de la factorielle, l'itération $i = n$ est effectuée, et le programme ne s'arrête qu'à $i = n + 1$.

Exemple : calcul de la somme des éléments d'indices pairs et impairs d'un tableau T de taille N . Les résultats sont retournés dans deux variables I et P .

$$I \leftarrow 0; P \leftarrow 0; \left\{ \begin{array}{l} I \leftarrow I + T_i \\ \end{array} \right\}_i^{N,2} \left\{ \begin{array}{l} P \leftarrow P + T_i \\ \end{array} \right\}_i^{N,2}$$

Trois solutions alternatives n'utilisant qu'une seule boucle :

$$\begin{aligned} & I \leftarrow 0; P \leftarrow 0; \left\{ \begin{array}{l} \text{MOD}(i, 2) = 0 ? \\ \quad P \leftarrow P + T_i \\ \quad I \leftarrow I + T_i \end{array} \right\}_i^N \\ & I \leftarrow 0; P \leftarrow 0; \left\{ \begin{array}{l} I \leftarrow I + T_i; P \leftarrow P + T_{i+1} \\ \end{array} \right\}_i^{N-1,2} \text{MOD}(N, 2) = 1 ? \quad I \leftarrow I + T_N \mid i \\ & I \leftarrow 0; P \leftarrow 0; \left\{ \begin{array}{l} I \leftarrow I + T_{2*i-1}; P \leftarrow P + T_{2*i} \\ \end{array} \right\}_i^{N \div 2} \text{MOD}(N, 2) = 1 ? \quad I \leftarrow I + T_N \mid i \end{aligned}$$

Nous laisserons le soin au lecteur de vérifier que les quatre algorithmes ci-dessus produisent des résultats équivalents.

Il est possible d'imbriquer plusieurs boucles, par exemple pour balayer un tableau T multi-dimensionnel de taille $N \times M$:

$$S \leftarrow 0; \left\{ \begin{array}{l} \left\{ \begin{array}{l} S \leftarrow S + T_{ij} \\ \end{array} \right\}_j^M \\ \end{array} \right\}_i^N$$

Exemple : tri dans l'ordre croissant des éléments d'un vecteur V

$$N \leftarrow \text{SIZE}(V); \left\{ \begin{array}{l} \left\{ \begin{array}{l} V_i > V_j ? \\ \quad G \leftarrow V_i; V_i \leftarrow V_j; V_j \leftarrow V_i \end{array} \right\}_j^N \\ \end{array} \right\}_i^{N-1}$$

Exemple : recherche du minimum dans un vecteur V

$$N \leftarrow \text{SIZE}(V); \text{MIN} \leftarrow V_1; \left\{ \begin{array}{l} \left\{ \begin{array}{l} V_i < \text{MIN} ? \\ \quad \text{MIN} \leftarrow V_i \end{array} \right\}_i^N \\ \end{array} \right\}_i^N$$

Attention aux modifications de la borne finale et du compteur dans la boucle !

Les boucles suivantes ne terminent pas :

$$\left\{ \begin{array}{l} N \leftarrow N + 1 \\ \end{array} \right\}_i^N \quad \left\{ \begin{array}{l} i \leftarrow i - 1 \\ \end{array} \right\}_i^N \quad \left\{ \begin{array}{l} i \leftarrow i + 1 \\ \end{array} \right\}_i^{1, -1}$$

Ces opérations ne sont pas interdites en ADL, ni dans la plupart des langages informatiques, mais obscurcissent notablement le code, et doivent donc être utilisées à bon escient.

Exemple : un autre tri en combinant les exemples 3 et 4

$$N \leftarrow \text{SIZE}(V); \left\{ \begin{array}{l} \underset{i:1}{\overset{N-1}{\text{MIN}}} \leftarrow V_i; \text{ ID} \leftarrow i; \underset{j:i+1}{\overset{N}{V_j}} < \text{MIN} ? \quad \text{MIN} \leftarrow V_j; \text{ ID} \leftarrow \\ j \mid i \end{array} \right\}_j G \leftarrow V_{ID}; V_{ID} \leftarrow V_i; V_i \leftarrow G \right\}_i$$

Cet algorithme de tri est appelé *tri par sélection*, puisqu'à chaque étape on sélectionne l'élément à échanger. Nous aurons l'occasion de l'utiliser plusieurs fois dans ce cours.

Exercice 1.8. Indiquer le nombre d'itérations de la boucle suivante :

$$b \leftarrow 26812; p \leftarrow -1; \left\{ \underset{i:16}{\overset{b,p}{b}} \leftarrow b - 1; p \leftarrow p + 1 \right\}_i$$

Exercice 1.9. On considère la suite de Fibonacci $(\mathcal{F}_n)_{n \in \mathbb{N}}$, définie par :

$$\mathcal{F}_0 = 1 \quad \mathcal{F}_1 = 1 \quad \forall n \geq 2 \quad \mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n$$

Ecrire un algorithme rentrant, pour un entier N donné, la valeur de la suite de Fibonacci au rang N .

1.2.4 Les boucles tant que

Il s'agit d'une variation de la boucle à compteur, permettant de transcrire une séquence d'instructions du type :

Tant que (condition) alors faire quelque chose

En d'autres termes, on répète une liste d'instructions exécutables tant qu'une condition (formulée sous forme d'expression algébrique à valeur booléenne) est vérifiée. En ADL, on écrit ce type de boucle avec la syntaxe suivante :

$\left\{ \begin{array}{l} /eab \\ \text{lie} \end{array} \right\}$

Où : eab désigne une expression algébrique dont le résultat est un booléen.

Exemple : la méthode de Héron permet de calculer une approximation de la racine carré d'un nombre a grâce à la suite convergente :

$$\forall n \in \mathbb{N} \quad x_0 = 1 \quad x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right)$$

Le calcul de \sqrt{a} avec une précision de 16 décimales s'écrit alors :

```
R ← 1; TEMP ← 0; epsilon ← 10 ∧ (-16);
{/ABS(TEMP-R) ≥ epsilon    TEMP ← R; R ← (R+a/R)/2}
```

Exercice 1.10. La suite de Syracuse est définie, pour tout entier initial $N \in \mathbb{N}^*$, de la manière suivante :

$$u_0 = N \quad u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases} \quad (1.1)$$

La conjecture de Syracuse (qui reste toujours non-démontrée à ce jour) stipule que la suite converge vers la valeur 1.

Ecrire un algorithme renvoyant le nombre d'itérations avant convergence de la suite.

1.2.5 Les boucles jusqu'à ce que

A la différence de la boucle tant que, la boucle jusqu'à ce que n'est plus réitérée dès que la condition exprimée par l'*eab* est remplie.

$\{ lie \}^{/eab}$

où *lie* est une liste d'instructions exécutables et *eab* est une expression algébrique à valeur booléenne.

Exemple : les deux boucles suivantes sont équivalentes

$$n \leftarrow 0; \{ n \leftarrow n + 1 \}^{/n>3} \quad n \leftarrow 0; \{^{/n \leq 3} n \leftarrow n + 1 \}$$

D'autre part, la condition booléenne est testée à chaque fin de passage dans la boucle. Ainsi, après l'instruction suivante, *n* conservera la valeur 1 :

$$n \leftarrow 0; b \leftarrow \downarrow; \left\{ {}^{/b} n \leftarrow n + 1 \right\}$$

tandis que dans le cas d'une boucle jusqu'à ce que, n prendra la valeur 1 puisqu'on entre au moins une fois dans la boucle.

$$n \leftarrow 0; b \leftarrow \downarrow; \left\{ n \leftarrow n + 1 \right\} {}^{\bar{b}}$$

Exercice 1.11. A titre d'entraînement, montrer que chacun des trois types de boucles précédemment exposés, peut être transcrit à l'aide d'un (et un seul) autre type. Il y a donc au total $3 \times 2 = 6$ transcriptions algorithmiques à effectuer. On supposera ici que les bornes et le pas ne sont pas modifiés à l'intérieur des boucles. On supposera également que le pas est toujours non nul.

1.2.6 Les cas particuliers de boucle

Boucle infinie

$$\left\{ \text{lie} \right\}$$

Cette fonctionnalité ne présente aucun intérêt tant que nous n'avons pas traité des mécanismes permettant de sortir prématurément d'une boucle (nous y reviendrons dans la section suivante).

Boucle combinée

Il s'agit du cas le plus général de boucle, combinant le compteur, la condition "tant que" et la condition "jusqu'à ce que". La sortie de boucle est effectuée dès lors que l'une des conditions n'est plus remplie, ou bien que le compteur a dépassé sa borne finale.

$$\left\{ {}_{k:b_i}^{b_f, p/ea b_1} \text{lie} \right\}_k {}^{/ea b_2}$$

où, k est un identificateur scalaire entier, b_i , b_f et p sont des expressions algébriques à valeurs dans \mathbb{Z} tandis que $ea b_1$ et $ea b_2$ sont des expressions algébriques à valeurs dans \mathbb{B} .

Ce genre de boucle n'est que très rarement utilisé en pratique, d'autant plus que cette fonctionnalité n'est généralement pas permise par les langages de programmation.

Exercice 1.12. A l'aide d'une boucle combinée, réécrire l'algorithme de Syracuse, en ajoutant une clause permettant de terminer l'algorithme lorsque le nombre d'itérations dépasse une constante t donnée.

1.2.7 Les débranchements de boucle

Il existe trois types de débranchement de boucle : les sorties de boucle, les passages à l'itération suivante et les réitérations. Nous verrons que l'on peut également définir des débranchements sur les modules. Dans la partie traitant de l'optimisation du code algorithmique, nous apprendrons à utiliser ces débranchements pour réduire la profondeur du code (et donc le rendre plus lisible et/ou plus efficace).

Sortie de boucle

Une sortie de boucle est une instruction élémentaire permettant de sortir prématulement d'une boucle donnée, quelle que soit la valeur de son compteur et/ou de ses conditions.

On note ! l'instruction de débranchement. Elle doit être placée à l'intérieur de la boucle qu'elle contrôle.

$$\left\{ \begin{array}{l} \text{lie}_2 \quad ! \quad \text{lie}_3 \end{array} \right\}$$

Dans ce morceau de code, lie_1 est exécutée, puis on rencontre l'instruction de débranchement. On sort alors de la boucle sans exécuter lie_2 . L'instruction de débranchement n'a donc d'intérêt que si elle est encapsulée dans une disjonction.

Exemple : transformation d'une boucle combinée en une boucle infinie avec débranchement. On supposera ici que les bornes et le pas (non nul) ne sont pas modifiés à l'intérieur de la boucle.

$$\left\{ \begin{array}{l} \overset{b_f, p/eab_1}{\underset{k:b_i}{\text{lie}}} \Big\} / eab_2 \rightarrow \left\{ \begin{array}{l} \overline{eab_1} \ ? \ ! \mid \underset{i}{\dots} \quad (p > 0) \cap (k > b_f) \cup (p < 0) \cap (k < b_f) \ ? \ ! \\ \mid \underset{i}{\dots} \text{lie}; eab_2 \ ? \ ! \mid \underset{i}{\dots} \quad k \leftarrow k + 1 \end{array} \right\} \end{array} \right\}$$

Exemple : calcul de plus petite valeur de n tel que la somme des n^2 dépasse 10000 :

$$S \leftarrow 0; N \leftarrow 1; \left\{ S \leftarrow S + N * N; N \leftarrow N + 1; S > 10000 \ ? \ ! \mid \underset{i}{\dots} \right\}$$

Exemple : recherche d'un élément E dans un tableau

$$N \leftarrow \text{SIZE}(T); \text{PRESENT} \leftarrow \downarrow; \left\{ \underset{i:1}{\overset{N}{\text{ }} T_i = E \ ? \ \text{PRESENT} \leftarrow \uparrow; ! \mid \underset{i}{\dots} \right\}$$

Passage à l'itération suivante

Cette instruction permet de quitter prématulement l'exécution d'une itération de la boucle pour passer à la suivante. On la note :

$\{ \text{ lie}_1 \rightarrow \text{ lie}_2 \}$

A la rencontre de ce symbole, la suite d'instructions lie_2 n'est pas exécutée, et l'algorithme reboucle directement en lie_1 (à condition toutefois que les conditions de sortie de boucle n'aient pas été déjà remplies).

Exemple : recherche de l'indice de l'élément maximal dans un tableau

$$N \leftarrow \text{SIZE}(T); M \leftarrow T_1; I \leftarrow 1; \left\{ \begin{array}{l} \forall i:2^N, T_i < M ? \rightarrow | \& M \leftarrow T_i; I \leftarrow i \end{array} \right\}_i$$

Réitération

L'instruction de réitération (*reloop*), permet de revenir au début de l'exécution d'une itération donnée.

$\{ \text{ lie}_1 \leftarrow \text{ lie}_2 \}$

A la rencontre du symbole \leftarrow , le programme reboucle au début de lie_1 , à la même itération. Dans le cas d'une boucle à compteur, cela signifie que le compteur n'est pas incrémenté (ni décrémenté).

1.2.8 Les débranchements d'ordre n

Les débranchements présentés dans la section précédente peuvent être généralisés au cas de n boucles imbriquées. La profondeur du débranchement (on parlera également de *degré*) est indiquée par l'entier suivant immédiatement le symbole de débranchement.

Exemple :

$$\text{lie}_1 \left\{ \text{lie}_2 \left\{ \text{lie}_3 \left\{ \text{lie}_4 !3 \text{ lie}_5 \right\} \text{lie}_6 \right\} \text{lie}_7 \right\} \text{lie}_8$$

A la rencontre du symbole $!3$ (sortie de boucle de degré 3), on sort de la boucle interne et on reprend l'exécution en lie_8 .

$$\text{lie}_1 \left\{ \text{lie}_2 \left\{ \text{lie}_3 \left\{ \text{lie}_4 \rightarrow 2 \text{ lie}_5 \right\} \text{lie}_6 \right\} \text{lie}_7 \right\} \text{lie}_8$$

A la rencontre du symbole $\rightarrow 2$ (passage à l'itération suivante de degré 2), on sort de la boucle interne et on reprend l'exécution au début de l'itération suivante de la boucle parente, c'est-à-dire en lie_3 .

Remarque 1 : les symboles $!$, \rightarrow , et \leftarrow sont équivalents (respectivement) à $!1$, $\rightarrow 1$, et $\leftarrow 1$.

Remarque 2 : le degré de débranchement ne saurait être supérieur au nombre de boucles imbriquées. Par exemple, le code suivant est incorrect :

$$\left\{_{i:1}^M \left\{_{j:1}^N T_{ij} = 0 \ ? \ !3 \mid_i \right\}_j \right\}_i$$

Remarque 3 : en toute généralité, le degré de débranchement pourra également être un identificateur ou une expression algébrique à valeur dans \mathbb{N} . Il s'agira alors là de s'assurer que le degré obtenu lors de l'exécution n'excède pas le nombre de boucles imbriquées.

Exercice 1.13. Dans la plupart des langages de programmation, l'instruction de débranchement multiple n'existe pas. Traduire l'algorithme suivant sans utiliser de débranchement de degré supérieur à 1.

$$\left\{_{i:1}^{100} \left\{_{j:1}^{100} T_{ij} = 0 \ ? \ !2 \mid_j \text{ lie} \right\}_j \right\}_i$$

L'instruction goto

L'instruction "go to" (souvent abrégée en *goto*) existe dans de nombreux langages de programmation, et est très populaire, de par sa flexibilité et sa simplicité d'utilisation.

Il s'agit de référencer la ligne de programme vers laquelle on souhaite envoyer l'exécution.

Il existe de nombreuses raisons de proscrire l'utilisation de cette instruction. Edsger Dijkstra, célèbre informaticien néerlandais, avait d'ailleurs dès 1968 publié un article intitulé *Go To Statement Considered Harmful*, dans lequel il expose de manière rationnelle, l'influence néfaste d'une utilisation abusive du goto dans un code informatique. Il introduit en particulier son propos par : "For a number of years, I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce".

Suivant cette logique, et dans le but d'éviter la tentation d'écrire du code algorithmique "en spaghetti", ADL n'autorise pas l'utilisation de ce symbole (qui par ailleurs imposerait une numérotation des lignes de code).

Exercice 1.14. Vérifier que toutes les instructions exposées dans les sections précédentes peuvent être transcris à l'aide des trois instructions suivantes : affectation, conditions (*if then*) et go to (en ajoutant évidemment les opérateurs de calcul algébrique).

1.3 La profondeur de code

1.3.1 Squelette d'un algorithme

Le squelette d'un algorithme désigne la structure du code, c'est-à-dire l'ensemble de ses instructions de contrôle (disjonctions, boucles et débranchements). Il s'agit d'un outil important pour permettre une compréhension simple et rapide d'un algorithme sans rentrer dans le détail des instructions élémentaires.

Exemple : considérons le code de recherche de la position d'un élément e dans chaque ligne d'une matrice :

$$\left\{_{i:1}^{NL} J_i \leftarrow 0\right\}_i \left\{_{i:1}^{NL} \left\{_{j:1}^{NC} M_{ij} \neq e ? \rightarrow | J_i \leftarrow j; !_i\right\}_j\right\}_i$$

Remarquons que dans ce cas particulier et de manière équivalente, le symbole $!$ aurait pu être remplacé par \rightarrow^2 . Cela vient du fait qu'il n'y a aucune instruction à exécuter entre la fin de la boucle sur l'indice j et celle de la boucle sur l'indice i .

Le squelette de cet algorithme s'écrira alors :

$$\{\} \{\{ ? \rightarrow | !_i\}\}$$

1.3.2 Degré d'emboîtement

Il s'agit de la profondeur du code, c'est-à-dire du nombre maximal de boucles et de disjonctions imbriquées. Par exemple, dans l'algorithme précédent, on voit directement grâce au squelette que la profondeur du code est égale à 3 (la partie la plus profonde est imbriquée dans une double boucle et une condition). On notera alors :

$$d_E = 3$$

Exercice 1.15.

Q1. Ecrire un algorithme permettant de générer automatiquement une matrice aléatoire de dimension $N \times M$ dont les éléments sont des entiers tirés suivant une loi uniforme définie sur l'intervalle $[1, 100]$.

Q2. Compléter cet algorithme avec les instructions permettant de rechercher dans la matrice générée, la plus longue séquence de nombres entiers (une séquence étant définie comme une suite d'entiers consécutifs sur une ligne ou une colonne donnée). L'algorithme devra retourner la longueur de cette séquence. Notons que cette longueur ne peut excéder $\max(N, M, 100)$.

Q3. Ecrire le squelette de l'algorithme et indiquer son degré d'emboîtement.

1.3.3 Optimisation du code

Nous donnons ici une liste non-exhaustive de quelques techniques permettant de réduire le degré d'emboîtement d'un code algorithmique. Précisons toutefois que ces optimisations ne se traduiront pas nécessairement en un gain sur le temps d'exécution. Leur objectif premier est de simplifier le code.

Passage à l’itération suivante dans une boucle

$$\{ eab ? \text{ lie} | \zeta \} \rightarrow \{ \overline{eab} ? \sqsupset \mid \zeta \text{ lie} \}$$

En notant d le degré d’emboîtement de la partie *lie*, la solution exposée à droite possède l’avantage d’avoir un degré d’emboîtement total $d_E = d$ contre $d + 1$ pour la version originale (à gauche). Cette méthode peut être généralisée :

$$\{ eab ? \text{ lie}_1 \mid \text{lie}_2 \zeta \} \rightarrow \{ eab ? \text{ lie}_1 \sqsupset \mid \zeta \text{ lie}_2 \}$$

Pour maximiser l’optimisation, on choisira d’extraire de la boucle la suite d’instruction de degré maximal (parmi *lie*₁ et *lie*₂).

Sortie de boucle

Lorsqu’une condition permet de sortir prématurément d’une boucle, il est préférable de gérer ce cas particulier individuellement. Si on atteint la liste d’instructions exécutables *lie*, alors c’est qu’on se trouve dans la branche de négation du test.

$$\{ eab ? ! \mid \text{lie} \zeta \} \rightarrow \{ eab ? ! \mid \zeta \text{ lie} \}$$

Sortie de boucle interne

Une condition de sortie de boucle interne peut être remplacée par un passage à l’itération suivante de degré 2 dès lors qu’il n’y a pas d’instructions entre les deux fins de boucles :

$$\{ \text{lie}_1 \{ \text{lie}_2 ! \text{ lie}_3 \} \} \rightarrow \{ \text{lie}_1 \{ \text{lie}_2 \sqsupset^2 \text{ lie}_3 \} \}$$

Cette simplification peut être généralisée au cas de n boucles imbriquées terminant successivement sans instruction exécutable entre deux fin de boucles :

$$\{ \dots \{ \dots \{ \dots ! \dots \} \} \} \rightarrow \{ \dots \{ \dots \{ \dots \sqsupset^n \dots \} \} \}$$

Sortie de boucle en fin de module

Cette règle concerne les modules, que nous étudierons à la fin de ce chapitre. Si la fin du module est immédiatement consécutive à la fin d'une boucle, toute sortie de boucle dans le module est équivalente à une sortie de module :

$$\llbracket \text{lie}_1 \{ \text{lie}_2 ! \text{lie}_3 \} \rrbracket \rightarrow \llbracket \text{lie}_1 \{ \text{lie}_2 * \text{lie}_3 \} \rrbracket$$

Ajout/retrait de boucle

Lorsqu'on ajoute (resp. retire) une boucle dans un ensemble de boucles imbriquées, tous les débranchements multiples sont incrémentés (resp. décrémenté) d'une unité :

$$\left\{ \left\{ \dots \left\{ \begin{array}{c} !n \\ \nearrow n \\ \searrow n \end{array} \right\} \right\} \right\} \rightarrow \left\{ \left\{ \dots \left\{ \begin{array}{c} !n + 1 \\ \nearrow n + 1 \\ \searrow n + 1 \end{array} \right\} \right\} \right\}$$

Pour une liste plus complète de règles de simplifications, nous invitons le lecteur à se référer à (F. Bouillé, 2006).

1.4 Les commentaires

Nous sommes à présent en mesure de produire du code algorithmique beaucoup plus complexe. La documentation du code devient alors essentielle pour produire des programmes faciles à comprendre, à maintenir et à faire évoluer. Il sera donc important de commenter le code sur toutes les lignes comportant une difficulté technique particulière. Si le programme est simple et direct, mais qu'il comporte un nombre important de lignes (typiquement plus d'une trentaine), il faudra alors veiller à y introduire des commentaires de temps à autre afin d'en séquencer les étapes et d'en faciliter la compréhension. La rédaction d'un commentaire ne suit aucun code particulier, l'auteur est libre d'apporter les explications dans la mesure qui lui semble la plus appropriée.

La structure d'un commentaire se présente comme suit :

(C) ceci est un commentaire (C)

Nous restreindrons cette section à ce point particulier, dénotant ainsi l'importance des commentaires dans la conception d'un algorithme et dans la programmation en général.

1.5 Les modules

Un module est un ensemble de lignes de code permettant d'effectuer une tâche très précise, à laquelle on aura recours plusieurs fois dans l'algorithme. Programmer de manière "modulaire" signifie avoir recours de manière intensive à la structure de module, afin de créer un code facile à maintenir et à comprendre.

1.5.1 Définition

Supposons par exemple que l'on ait besoin de multiplier deux matrices A et B dans un code (on supposera que les matrices ont des dimensions adéquates). Le résultat sera stocké dans une troisième matrice C et le programme correspondant s'écrit de la manière suivante :

④ Récupération des tailles des matrices ④
 $SA \leftarrow \text{SIZE}(A); SB \leftarrow \text{SIZE}(B); NLA \leftarrow SA_1; NCA \leftarrow SA_2; NCB \leftarrow SB_2;$

④ Calcul du produit matriciel ④
 $\left\{_{i:1}^{\text{NLA}} \left\{_{j:1}^{\text{NCB}} C_{ij} \leftarrow 0; \left\{_{k:1}^{\text{NCA}} C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj} \right\}_k \right\}_j \right\}_i$

En guise d'entraînement, on donne le squelette et le degré d'emboîtement de l'algorithme :

$$\left\{ \left\{ \left\{ \right\} \right\} \right\} \quad d_E = 3$$

Supposons à présent que nous ayons besoin d'effectuer plusieurs produits matriciels dans un algorithme. Il paraît alors inutile de répéter plusieurs fois les lignes d'instructions pour effectuer une même tâche. D'autant que si l'on souhaite un jour changer d'algorithme (par exemple pour lui substituer un algorithme plus efficace afin de réduire le temps de calcul), il est avantageux de n'avoir à effectuer cette modification que dans une seule partie du code, et non à chaque appel d'un produit de matrices.

Nous allons donc regrouper ces lignes au sein d'une structure que l'on appelle module, et qui joue un rôle équivalent aux fonctions mathématiques.

Un module contient deux parties :

- l' **entête** du module : permettant de spécifier le nom du module (qui servira à l'appeler plus tard dans la suite du code) ainsi que ses éventuels paramètres d'entrée et de sortie.
- le **corps** du module : contenant la liste des instructions exécutables contenues dans ce module et faisant référence aux identificateurs définis dans l'entête.

D'un point de vue syntaxique, voici comment nous écrirons un nouveau module :

IDENTIFICATEUR_DE_MODULE($p_1 \downarrow, p_2 \downarrow, p_3 \uparrow, \dots, p_n \uparrow$) : [lie]

Cette notation comprend trois informations principales :

- Dans l'entête, l'identificateur de module (on parle aussi de nom de module), suit les mêmes spécifications que les identificateurs de variable abordés au début de ce cours. Ils permettent de référencer un module. Deux modules différents ne peuvent avoir le même nom. On veillera à choisir des noms de module pertinents, en rapport avec l'action effectuée par le module.
- Toujours dans l'entête, on donne la liste des paramètres formels manipulés dans le module. Il s'agit des variables qui par la suite seront insérées ou retournées par le module. Notons que chaque paramètre est associé à une flèche (\uparrow , \downarrow ou \ddagger), permettant de distinguer les paramètres en entrée et les paramètres en sortie du module.
 - \downarrow indique que le paramètre est passé en entrée du module.
 - \uparrow indique que le paramètre est une sortie (i.e. un résultat) du module.
 - \ddagger indique que le paramètre est passé en entrée dans le module, pour y être modifié et retourné en résultat.

Nous reviendrons plus en détail sur cette notion par la suite.

- Dans le corps de module, le symbole [] délimite la liste d'instructions exécutables contenues dans le module. Ces instructions font appel aux paramètres formels.

Notons dès à présent qu'un module peut ne contenir aucun paramètre en entrée, par exemple la fonction RAND, qui retourne un nombre aléatoire, ne nécessite pas de passage d'argument.

Ecrivons le produit matriciel sous forme de module :

MULTIPLY_MATRICES($A \downarrow, B \downarrow, C \uparrow$) : [

④ Récupération des tailles des matrices ④
 $SA \leftarrow \text{SIZE}(A); SB \leftarrow \text{SIZE}(B); NLA \leftarrow SA_1; NCA \leftarrow SA_2; NCB \leftarrow SB_2;$

④ Calcul du produit matriciel ④
 $\left\{_{i:1}^{\text{NLA}} \left\{_{j:1}^{\text{NCB}} C_{ij} \leftarrow 0; \left\{_{k:1}^{\text{NCA}} C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj} \right\}_k \right\}_j \right\}_i \right]$

Pour bien faire, il faudrait vérifier que les dimensions des matrices en entrée sont compatibles avec le produit matriciel. Pour cela, il nous faut d'abord introduire un nouveau type de débranchement : le débranchement de module.

1.5.2 Débranchement de module

Tout comme nous avons défini trois types de débranchement de boucles, il est possible de définir un type de débranchement sur les modules.

La sortie de module

Cette instruction permet de quitter prématurément un module avant la fin de l'exécution. On l'écrit à l'aide de la syntaxe suivante :

$\llbracket \text{ lie}_1 \ ! \ \text{lie}_2 \rrbracket$

Bien entendu, il ne peut se trouver qu'à l'intérieur d'un module. Voyons comment cela se passe en pratique : considérons le code ci-dessous :

INV($x \downarrow, y \uparrow$) : $\llbracket y \leftarrow -9999; x = 0 ? ! \mid ; y \leftarrow 1/x \rrbracket$

INV est une procédure qui prend en entrée un nombre, qui teste si ce nombre est différent de 0 et qui le cas échéant renvoie l'inverse de ce nombre, et renvoie le code d'erreur -9999 sinon (sortie prématurée du module). Considérons l'appel suivant :

lie₁; $x \leftarrow 0$; INV($x \downarrow, y \uparrow$); lie₂

A l'entrée du module INV, le paramètre x vaut 0. Le module se termine donc prématurément, et le programme continue avec la liste d'instruction lie₂, dans laquelle y entre avec la valeur -9999 .

La sortie de module est donc particulièrement adaptée à la gestion des exceptions, via des codes d'erreur.

1.5.3 Gestion des erreurs

Grâce au débranchement de module, nous pouvons à présent réécrire le code du module de multiplication matricielle, en prenant cette fois-ci en compte le cas où les matrices ont des dimensions inappropriées.

MULTIPLY_MATRICES($A \downarrow, B \downarrow, C \uparrow, err \uparrow$) : \llbracket

$err \leftarrow "ok";$

 © Récupération des tailles des matrices ©
 $SA \leftarrow \text{SIZE}(A); SB \leftarrow \text{SIZE}(B); NLA \leftarrow SA_1; NCA \leftarrow SA_2; NLB \leftarrow SB_1; NCB \leftarrow SB_2;$

 © Test de cohérence ©
 $NCA \neq NLB ? err \leftarrow "Problème dimensions des matrices"; ! \mid ;$

④ Calcul du produit matriciel ④ $\left\{ \begin{array}{l} \left\{ \begin{array}{l} NLA \\ i:1 \end{array} \right\} \left\{ \begin{array}{l} NCB \\ j:1 \end{array} \right\} C_{ij} \leftarrow 0; \left\{ \begin{array}{l} NCA \\ k:1 \end{array} \right\} C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj} \end{array} \right\}_k \right\}_j \right\}_i \right]$

L'identificateur *err* permet donc de s'assurer que le module a fonctionné correctement. En général, on utilise plutôt des codes d'erreur entiers, le code 0 signifiant que le module s'est déroulé correctement. L'algorithme est alors modifié en :

MULTIPLY_MATRICES($A \downarrow, B \downarrow, C \uparrow, err \uparrow$) : $\left[\right]$

err \leftarrow 0;

④ Récupération des tailles des matrices ④

$SA \leftarrow \text{SIZE}(A); SB \leftarrow \text{SIZE}(B); NLA \leftarrow SA_1; NCA \leftarrow SA_2; NLB \leftarrow SB_1; NCB \leftarrow SB_2;$

④ Test de cohérence ④

$NCA \neq NLB ? err \leftarrow 1; \quad$ ④ Problème dimension des matrices ④ ! | i

④ Calcul du produit matriciel ④ $\left\{ \begin{array}{l} \left\{ \begin{array}{l} NLA \\ i:1 \end{array} \right\} \left\{ \begin{array}{l} NCB \\ j:1 \end{array} \right\} C_{ij} \leftarrow 0; \left\{ \begin{array}{l} NCA \\ k:1 \end{array} \right\} C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj} \end{array} \right\}_k \right\}_j \right\}_i \right]$

Il est possible, en fonction de la complexité du programme, de définir autant de codes d'erreur que nécessaire.

Un test de cohérence aura donc toujours le format suivant :

test ? *erreur* \leftarrow *code*; ④ On détaille l'erreur rencontrée ④ ! | i

Dans ce formalisme, *test* est une expression algébrique booléenne, *erreur* est un identificateur retourné en sortie du module, tandis que *code* est une constante, un identificateur ou bien une expression algébrique à valeur entière.

1.5.4 Paramètres formels et actuels

Il s'agit là d'un concept important en informatique et que l'on retrouve dans la quasi-totalité des langages de programmation.

Considérons le module suivant (qui calcule la valeur absolue d'un nombre *X* et qui renvoie le résultat dans la variable *Y*).

ABS_VAL($X \downarrow, Y \uparrow$) : $\left[\right] \left[Y \leftarrow X; Y < 0 ? Y \leftarrow -Y \mid i \right] \right]$

Dans cette ligne d'instructions (que l'on appelle déclaration de fonction), les paramètres *X* et *Y* sont appelés *paramètres formels*, en cela qu'ils n'ont pas de valeurs définies.

En revanche, lorsque l'on effectue un appel à la fonction ABS_VAL dans un code, les paramètres deviennent alors des *paramètres actuels*³ :

$$\text{ABS_VAL}(X, Y)$$

De manière équivalente, lorsqu'en mathématiques on écrit :

$$\begin{aligned} f &: \mathbb{R} \rightarrow \mathbb{R}^+ \\ x &\mapsto |x| \end{aligned}$$

Dans cette déclaration, x est un paramètre formel puisqu'il n'a pas de valeur définie. A l'inverse, lorsque que l'on considère un x_0 particulier, celui-ci devient un paramètre actuel de l'expression $f(x_0)$. Par exemple : dans $f(2)$, la constante 2 est un paramètre actuel de f .

1.5.5 Entrées/sorties d'un module

Nous avons brièvement expliqué précédemment que les symboles \downarrow , \uparrow et \Downarrow , associés aux paramètres d'un module, permettaient de spécifier leur mode de passage. Nous allons ici détailler un peu plus cette affirmation.

La flèche descendante \downarrow est utilisée pour spécifier que l'argument est passé en entrée du module. En conséquence, la modification de sa valeur dans le module, n'a aucun impact en dehors du module. En informatique, on parle de *passage par valeur*.

Considérons le module ci-dessous

$$\text{MOD}(X \downarrow, Y \uparrow) : \left[\left[X < 0 ? X \leftarrow -X \mid ; Y \leftarrow X \right] \right]$$

Ce module prend en entrée une variable X et retourne sa valeur absolue dans la variable Y . Que se passe-t-il lors de l'exécution des instructions suivantes ?

$$X \leftarrow -12; \text{ MOD}(X, Y); \text{ PRINT}(X \downarrow)$$

Nous admettrons ici que la fonction PRINT prend en entrée une donnée et l'affiche dans la console.

L'exécution de ces lignes de code retournera -12 , puisque X est passé en entrée **uniquement** de la fonction MOD. Les modifications opérées sur la variable X au sein du module ne sont donc pas effectives en dehors de ce module. La variable X garde donc sa valeur de -12 . En revanche, le résultat est différent si l'on définit MOD de la manière suivante :

$$\text{MOD}(X \Downarrow, Y \uparrow) : \left[\left[X < 0 ? X \leftarrow -X \mid ; Y \leftarrow X \right] \right]$$

3. Notons qu'il s'agit d'une traduction directe de l'anglais *actual*, qui signifie "vrai". En ce sens un paramètre actuel désigne la véritable valeur qui est passée dans le module lors de son appel.

Dans ce cas, l'instruction ci-dessous retournera la valeur 12.

$$X \leftarrow -12; \text{ MOD}(X, Y); \text{ PRINT}(X)$$

L'identificateur X est à présent une variable d'entrée/sortie, il peut donc être modifié à l'intérieur du module et cela a une implication dans le programme appelant.

L'intérêt de cette fonctionnalité est de pouvoir directement transformer une valeur, la variable de sortie Y devient alors inutile et l'on peut écrire le module MOD de la manière condensée suivante :

$$\text{MOD}(X \Downarrow) : \llbracket X < 0 ? X \leftarrow -X \mid \iota \rrbracket$$

L'appel au module s'effectuera alors par l'instruction :

$$X \leftarrow -12; \text{ MOD}(X); \text{ PRINT}(X)$$

Le résultat retourné en console sera alors la nouvelle valeur affectée à l'identificateur X , soit la valeur 12.

Ce mode de passage des arguments est appelé *passage par référence*.

Exemple : écrivons un algorithme de transposition de matrice (nous supposerons ici que la matrice à transposer est carrée)

$$\text{TRANSPOSE}(A \Downarrow) : \llbracket S \leftarrow \text{SIZE}(A); NL \leftarrow S_1; NC \leftarrow S_2; \left\{ \begin{array}{l} \left\{ \begin{array}{l} i:1 \\ j:1 \end{array} \right\} G \leftarrow A_{ij}; A_{ij} \leftarrow \\ A_{ji}; A_{ji} \leftarrow G; \end{array} \right\}_j \right\}_i \rrbracket$$

Cette fonction transforme la matrice A en sa transposée. Ceci est complètement différent de la solution suivante, qui copie la transposée dans une nouvelle matrice, laissant ainsi A intacte. Cela nécessite en revanche de prévoir une variable de sortie pour le résultat de la fonction.

$$\text{TRANSPOSE}(A \Downarrow, TA \Uparrow) : \llbracket S \leftarrow \text{SIZE}(A); NL \leftarrow S_1; NC \leftarrow S_2; \left\{ \begin{array}{l} \left\{ \begin{array}{l} i:1 \\ j:1 \end{array} \right\} TA_{ij} \leftarrow \\ A_{ji} \end{array} \right\}_j \right\}_i \rrbracket$$

Nous verrons par la suite que lorsque le module retourne une unique valeur, on peut parler de *fonction*, et il sera possible de retourner le résultat via l'identificateur du module, sans passer par une variable de sortie dédiée comme TA ici.

Exercice 1.16. Montrer que tout module du type $F(X \Downarrow, Y \Uparrow)$ peut être exprimé sous la forme $G(X \Downarrow)$ et vis-versa.

Exercice 1.17. Sparse estimation

Ecrire un module prenant en entrée une vecteur de taille quelconque, et réduisant à zéro tous les éléments plus petits en valeur absolue qu'un paramètre ε , également fourni en argument. Le module retournera le nombre de composantes non nulles du vecteur après la régularisation. A titre d'entraînement, on indiquera également le squelette et le degré d'emboîtement du code.

Attention : nous conclurons cette partie en indiquant que si les paramètres formels d'un module sont nécessairement des identificateurs (scalaire ou dimensionné et de quelque type que ce soit), il n'en va pas de même pour les paramètres actuels qui peuvent être :

- des constantes
- des identificateurs auxquels on aura préalablement affecté une variable
- des expressions algébriques (à valeur dans l'ensemble spécifié par le paramètre formel correspondant).

Notons également qu'un paramètre en sortie (\uparrow ou \Downarrow) ne peut être qu'un identificateur, puisque le module doit savoir à qui affecter le résultat de son calcul.

Ainsi par exemple, si l'on dispose d'un module MOD prenant en entrée trois nombres et retournant un quatrième nombre :

$$\text{MOD}(15.7 \downarrow, X \downarrow, X + Y/T \downarrow, R \uparrow,)$$

L'appel ci-dessus est correct, à condition toutefois que les identificateurs X , Y et T aient été préalablement définis (c'est-à-dire qu'une valeur numérique leur a été affectée).

En revanche, l'appel :

$$\text{MOD}(15.7 \downarrow, X \downarrow, X + Y/T \downarrow, R1 + R2 \uparrow,)$$

est incorrect puisque la variable de sortie n'est pas un identificateur.

En somme, les arguments d'un module suivent les mêmes règles que l'affectation (si besoin, ne pas hésiter à consulter à nouveau la première section de ce cours) : les paramètres en entrée \downarrow (resp. sortie \uparrow) doivent pouvoir être placés en partie droite (resp. gauche) d'une affectation. Il en découle que les paramètres en entrée/sortie \Downarrow doivent pouvoir être indifféremment placés en partie droite ou gacuhe d'une affectation.

1.5.6 Les types de modules

Jusqu'à présent, nous n'avons traité qu'un seul type de module : les procédures.

Dans une **procédure**, les résultats sont retournés via des paramètres, spécifiées à l'aide des symboles \uparrow et \Downarrow .

A l'inverse, dans une **fonction**, le résultat est passé directement dans le nom du module. A l'intérieur du module, tout se passe comme si l'identificateur de module était aussi l'identificateur de variable destinée à recevoir le résultat du module. A l'extérieur du module, on considère que la fonction est typée, et elle peut être appelée au sein d'une expression algébrique (lorsqu'elle retourne un nombre, un booléen ou une chaîne de caractères).

A titre d'exemple, donnons les versions procédurales et fonctionnelles d'un module de calcul de la factorielle d'un nombre N :

Version procédurale :

$$\text{FACT}(N \downarrow, F2N \uparrow) : \left[\left[F2N \leftarrow 1; \left\{ \begin{array}{l} F2N \leftarrow F2N * i \\ \end{array} \right. \right\}_i^N \right] \right]$$

La procédure est alors appelée au sein d'un code par : $\text{FACT}(10, F2N)$ par exemple si l'on souhaite calculer $10!$. Le résultat est alors contenu dans la variable $F2N$.

Version fonctionnelle :

$$\text{FACT}(N \downarrow) : \left[\left[\text{FACT} \leftarrow 1; \left\{ \begin{array}{l} \text{FACT} \leftarrow \text{FACT} * i \\ \end{array} \right. \right\}_i^N \right] \right]$$

Dans cette seconde version, plus simple d'emploi lorsqu'une seule variable doit être à retournée, le résultat est stocké dans la variable FACT . Le calcul de $10!$ se fera alors par : $\text{FACT}(10)$.

L'intérêt de cette second formulation est bien entendu de pouvoir appeler la fonction FACT dans un calcul algébrique. La ligne d'instruction suivante sera alors correcte :

$$R \leftarrow 2 + 3 * \text{FACT}(10) + 7$$

Nous reconnaissons ici le formalisme des fonctions mathématiques présentées au 1.1.7, pour lesquelles nous avions précisé qu'il ne s'agissait que d'un cas particulier de module.

Exemple 1 : norme d'un vecteur

$$\text{NORM}(V \downarrow) : \left[\left[T \leftarrow \text{SIZE}(V); \text{NORM} \leftarrow 0; \left\{ \begin{array}{l} \text{NORM} \leftarrow \text{NORM} + V_i * V_i \\ \end{array} \right. \right\}_i^T \text{NORM} \leftarrow \sqrt{\text{NORM}} \right] \right]$$

Exemple 2 : normalisation d'un vecteur

$$\text{NORMALIZE}(V \uparrow) : \left[\left[T \leftarrow \text{SIZE}(V); N \leftarrow \text{NORM}(V); N = 0 ? ! | \left\{ \begin{array}{l} V_i \leftarrow V_i / N \\ \end{array} \right. \right\}_i^T \right] \right]$$

Exercice 1.18. Matrice creuse

On appelle matrice creuse, la représentation d'une matrice dont la plupart des composantes sont nulles. on ne considère ici que des matrices carrées.

Soit M une matrice de taille quelconque. La représentation creuse de M est une matrice C à trois colonnes, contenant sur chaque ligne un élément non nul de M . Les deux premières colonnes de C contiennent les indices de ligne et de colonne de l'élément dans M .

Q1. Montrer que la représentation creuse est avantageuse (en terme de mémoire occupée) pour une matrice M , si et seulement si le nombre d'éléments non-nuls de M est inférieur à une certaine quantité donnée, que l'on exprimera en fonction des dimensions de M .

Q2. Ecrire un module prenant en entrée la matrice M et retournant

- la représentation creuse C de M
- le gain de mémoire apportée par la transformation

Q3. Le module créé est-il une procédure ou une fonction ?

Q4. Construire son squelette et indiquer son degré d'emboîtement.

1.5.7 Les modules récursifs

Un module (fonction ou procédure) est dit *récursif*, dès lors que son corps de module comprend au moins un appel à lui-même.

 $\text{MOD_REC}(p_1 \downarrow, p_2 \downarrow, \dots, p_n \uparrow) : \llbracket \quad \text{lie}_1 \quad \text{MOD_REC}(a_1 \downarrow, a_2 \downarrow, \dots, a_n \uparrow) \quad \text{lie}_2 \rrbracket$

où les a_i sont des paramètres actuels correspondant aux paramètres formels p_i définis dans l'entête de module.

Exemple 1 : calcul de la factorielle

$$\text{FACTORIAL}(N \downarrow) : \llbracket \quad N \leqslant 1 ? \quad \text{FACTORIAL} \leftarrow 1; ! \mid ; \quad \text{FACTORIAL} \leftarrow N * \\ \text{FACTORIAL}(N - 1) \quad \rrbracket$$

Exemple 2 : exponentiation

$$\text{EXP}(X \downarrow, N \downarrow) : \llbracket N = 0 ? \quad \text{EXP} \leftarrow 1; ! \mid ; \quad \text{EXP} \leftarrow X * \text{EXP}(X, N - 1) \quad \rrbracket$$

Notons qu'un module récursif peut faire appel à lui-même plus d'une fois dans son corps de module.

Exemple 3 : exponentiation rapide

FAST_EXP($X \downarrow, N \downarrow$) : $\llbracket N = 0 ? \text{FAST_EXP} \leftarrow 1; ! \mid_i \text{MOD}(N, 2) = 0 ? \text{FAST_EXP} \leftarrow \text{FAST_EXP}(X, N/2) * \text{FAST_EXP}(X, N/2); \mid \text{FAST_EXP} \leftarrow X * \text{FAST_EXP}(X, N \div 2) * \text{FAST_EXP}(X, N \div 2); i \rrbracket$

Alternative (optimisée) :

FAST_EXP($X \downarrow, N \downarrow$) : $\llbracket X = 1 \cup N = 0 ? EXP \leftarrow 1; ! \mid_i \text{FAST_EXP} \leftarrow \text{FAST_EXP}(X, N \div 2); \text{FAST_EXP} \leftarrow \text{FAST_EXP} * \text{FAST_EXP}; \text{MOD}(N, 2) \neq 0 ? \text{FAST_EXP} \leftarrow X * \text{FAST_EXP} \mid_i \rrbracket$

Exemple 4 : suite de Fibonacci

FIBO($N \downarrow$) : $\llbracket N \leq 2 ? \text{FIBO} \leftarrow 1; ! \mid_i \text{FIBO} \leftarrow \text{FIBO}(N - 1) + \text{FIBO}(N - 2) \rrbracket$

Exercice 1.19. Multiplication égyptienne

Ecrire une fonction récursive permettant de calculer le produit de 2 nombres suivant les règles suivantes :

$$\text{Si } p \text{ est pair : } pq = \frac{p}{2} \times 2q \quad \text{sinon : } pq = q + \frac{p-1}{2} \times 2q$$

Les fonctions récursives sont particulièrement utilisées dans les stratégies de type *diviser pour régner*, où l'objectif est de partitionner un problème en sous problèmes, a priori plus simples à résoudre, avant de recombiner les solutions individuelles pour obtenir la réponse au problème initial. Nous reviendrons sur cette notion dans le chapitre sur la complexité algorithmique.

Exemple 5 : le tri fusion

Le tri fusion est une implémentation d'une stratégie de type diviser pour régner, pour trier un tableau de nombres. L'idée générale de l'algorithme consiste à diviser le tableau en 2 parties à peu près égales, puis à trier chaque sous-tableau séparément. Les deux tableaux sont alors réunis en un unique tableau en intercalant leurs éléments (en pratique, il est relativement simple, d'un point de vue algorithmique, de fusionner deux tableaux déjà triés).

Notons que l'algorithme est alors appelé de manière récursive sur chacun de sous-tableaux et ainsi de suite, jusqu'à ce que chaque tableau à trier ne contienne plus qu'un élément,

rendant ainsi le tri extrêmement simple.

Tout le travail algorithmique se situe donc au niveau de la recombinaison des sous-problèmes. Nous montrerons dans le chapitre sur la complexité, qu'en plus d'être relativement simple à implémenter, le tri fusion est beaucoup plus rapide que le tri naïf implémenté précédemment dans ce cours.

Commençons par écrire une fonction auxiliaire FUSION, prenant en entrée deux tableaux T_1 et T_2 (de tailles potentiellement différentes), dont on supposera qu'ils sont déjà triés (dans l'ordre croissant), et qui retourne un tableau fusionné et trié, contenant tous les éléments de T_1 et T_2 .

$$\text{FUSION}(T_1 \downarrow, T_2 \downarrow, T \uparrow) : \left[\begin{array}{l} N_1 \leftarrow \text{SIZE}(T_1); N_2 \leftarrow \text{SIZE}(T_2); I_1 \leftarrow 1; I_2 \leftarrow 1; \\ \left\{ \begin{array}{l} /I_1 \leq N_1 \cap I_2 \leq N_2 \\ I \leftarrow I_1 + I_2 - 1; T_{1I_1} \leq T_{2I_2} ? T_I \leftarrow T_{1I_1}; I_1 \leftarrow I_1 + 1 \mid T_I \leftarrow T_{2I_2}; I_2 \leftarrow I_2 + 1 \end{array} \right\} \quad \text{④ si on est là, c'est que l'une des deux listes est épuisée ④} \\ \left\{ \begin{array}{l} /I_1 \leq N_1 \\ I \leftarrow I_1 + I_2 - 1; T_I \leftarrow T_{1I_1} \end{array} \right\} \left\{ \begin{array}{l} /I_2 \leq N_2 \\ I \leftarrow I_1 + I_2 - 1; T_I \leftarrow T_{2I_2} \end{array} \right\} \end{array} \right]$$

Le module principal utilise alors le sous-module FUSION ainsi que le principe de récursivité pour résoudre le problème.

$$\text{TRIFUSION}(T \downarrow) : \left[\begin{array}{l} N \leftarrow \text{SIZE}(T); N \leq 1 ? \text{TRIFUSION} \leftarrow T; ! \mid_i \left\{ \begin{array}{l} _{i:1}^{N \div 2} T_{1i} \leftarrow T_i \\ \left\{ \begin{array}{l} _{i:N \div 2+1}^N T_{2i-N \div 2} \leftarrow T_i \end{array} \right\}_i \text{FUSION}(\text{TRIFUSION}(T1), \text{TRIFUSION}(T2), \text{TRIFUSION}); \end{array} \right\} \end{array} \right]$$

Exercice 1.20. Ecrire une version récursive du module auxiliaire FUSION.

Exercice 1.21. Recherche d'un élément dans une liste

Ecrire en récursif une fonction permettant de tester si un vecteur contient un élément e (donné en argument de la fonction). La fonction devra retourner une valeur booléenne égale à *true* si le vecteur contient l'élément et *false* sinon. Cet algorithme est-il plus rapide que l'algorithme de recherche linéaire ?

Exercice 1.22. Un peu de géométrie

Etant donné un ensemble de points (x_i, y_i) , écrire une fonction retournant un vecteur à trois éléments, contenant les indices du couple de points les plus proche l'un de l'autre, ainsi que la distance les séparant.

Transformer la fonction précédente pour l'exprimer sous forme récursive.

Récursivité croisée

On parle de récursivité croisée, lorsque deux fonctions s'appellent mutuellement. Soit les suites imbriquées $(u_n)_{n \in \mathbb{N}}$ et $(v_n)_{n \in \mathbb{N}}$, définies par : $u_0 = 0$, $v_0 = 1$ et :

$$u_{n+1} = \frac{u_n + v_n}{2} \quad v_{n+1} = \frac{u_n + 2v_n}{3}$$

En ADL, on programmera cette suite avec le code suivant :

```
U(N ↓) : [ N = 0 ? U ← 0; ! | ; U ← (U(N - 1) + V(N - 1))/2 ]
V(N ↓) : [ N = 0 ? V ← 1; ! | ; V ← (U(N - 1) + 2 * V(N - 1))/3 ]
```

Exercice 1.23. Ecrire deux fonctions récursives PAIR et IMPAIR, permettant de retourner la parité d'un nombre. Chaque fonction retournera un booléen. Par exemple, PAIR(3) devra retourner faux, tandis que IMPAIR(3) retournera vrai.

Remarque générale sur la récursivité : d'un point de vue théorique, la récursivité est très intéressante et permet d'exprimer de manière concise et élégante des algorithmes relativement complexes, ce qui explique sa popularité. D'un point de vue opérationnel, il faut noter que chaque appel récursif nécessite l'ajout d'une adresse mémoire dans la pile d'exécution de la machine, pile dont la taille est généralement limitée à quelques milliers d'adresses par la plupart des compilateurs/interpréteurs de langages informatiques (pour plus d'informations sur la notion de *pile de données*, nous renvoyons le lecteur à la section 4.3). L'expression récursive d'une procédure a alors le facheux inconvénient de produire un dépassement de la taille maximale autorisée pour cette pile d'exécution (c'est le fameux message d'erreur *stack overflow*). Par exemple, la sommation réursive $f(n) = f(n - 1) + 1$ ne pourra plus s'exécuter en machine pour n de l'ordre de quelques milliers, ce qui estridiculement petit au regard de la simplicité du calcul demandé. Notons qu'il est parfois possible de contourner ce problème avec la récursivité dite *terminale*.

1.6 Les chaînes de caractères

Jusqu'ici nous n'avons que très peu parlé des chaînes de caractères (que l'on appelle données alpha-numériques en ADL). De manière pragmatique, une telle chaîne peut être traitée comme un tableau d'entiers, chaque cellule étant occupée par le code ASCII de la lettre correspondante.

Par souci de simplification, nous manipulerons directement des chaînes de caractères, notamment à l'aide des fonctions suivantes :

- **LENGTH** : prenant en entrée une chaîne de caractères et retournant sa taille (en nombres de caractères).

- **SUBSTR** : prenant en entrée une chaîne de caractères S , un indice initial i et une longueur L , et retournant la sous-chaîne de S de longueur L et démarrant en position i . Notons que comme pour les tableaux, les indices débutent à 1. Exemple : pour une chaîne $S = \text{"voici un exemple"}$, la fonction $\text{SUBSTR}(S, 10, 7)$ retournera la chaîne "exemple".

Exemple 1 : écrire la fonction TRIM, qui prend en entrée une chaîne de caractères et qui retire tous les espaces blancs avant et après la chaîne.

$$\begin{aligned} \text{TRIMAVANT}(S \downarrow) : & \left[\begin{array}{l} N \leftarrow \text{LENGTH}(S); N = 0 ? \text{TRIMAVANT} \leftarrow S; ! | . \\ \left\{ \begin{array}{l} i:1 \\ N \end{array} \right. \begin{array}{l} \text{SUBSTR}(S, i, 1) \neq " " ? ! | . \\ \left. \begin{array}{l} \text{TRIMAVANT} \leftarrow \text{SUBSTR}(S, i, N - i + 1) \end{array} \right\}_i \end{array} \end{array} \right] \end{aligned}$$

En s'inspirant du modèle proposé ci-dessus, écrire la fonction TRIMAPRES. On obtient alors la fonction TRIM qui supprime les espaces en début et en fin de chaîne :

$$\text{TRIM}(S \downarrow) : \left[\begin{array}{l} \text{TRIM} \leftarrow \text{TRIMAVANT}(\text{TRIMAPRES}(S)) \end{array} \right]$$

A titre d'entraînement, on pourra formuler la fonction TRIMAVANT en récursif.

Exemple 2 : indice de Jaccard entre deux listes de mots

On considère deux tableaux $T1$ et $T2$, de dimension 1 (mais de tailles potentiellement différentes), et de type alpha-numérique (chaque cellule contient une chaîne de caractères). L'indice de Jaccard permet de mesure la similarité entre deux ensembles A et B . Il est défini par :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

où $A \cap B$ et $A \cup B$ désignent respectivement l'intersection et l'union des deux ensembles A et B , tandis que $| . |$ désigne la cardinalité d'un ensemble.

Dans cet exercice, on supposera que chaque élément de A et B ne peut être présent qu'une seule fois (au plus) dans chacun des 2 ensembles.

$$\begin{aligned} \text{JACCARD}(T1 \downarrow, T2 \downarrow) : & \left[\begin{array}{l} N1 \leftarrow \text{SIZE}(T1); N2 \leftarrow \text{SIZE}(T2); \\ \text{intersection} \leftarrow 0; \left\{ \begin{array}{l} i:1 \\ j:1 \\ N1 \\ N2 \end{array} \right. \begin{array}{l} T1_i = T2_j ? \text{intersection} \leftarrow \text{intersection} + 1; ! | . \\ \left. \begin{array}{l} \left\{ \begin{array}{l} \text{union} \leftarrow N1 + N2 - \text{intersection}; \text{JACCARD} \leftarrow \text{intersection}/\text{union} \end{array} \right\}_i \end{array} \right\}_j \end{array} \end{array} \right] \end{aligned}$$

Exemple 3 : recherche de sous-chaîne

L'objectif de l'exercice est d'écrire une fonction SEARCH, prenant en entrée une chaîne de caractères C de longueur m ainsi qu'une sous-chaîne S de longueur $n \leq m$ et qui retourne la

position du premier caractère de S dans C (la fonction retournera -1 si S n'est pas trouvée).

La méthode naïve consiste à parcourir la chaîne externe et pour chaque caractères, tester si les caractères qui suivent correspondent à ceux de la sous-chaîne. Par exemple, avec $C = \text{"distribution"}$ et $S = \text{"tribus"}$, l'algorithme parcourt les premières lettres de C , jusqu'à rencontrer le caractère "t", qui correspond à la première lettre de S . On passe alors au caractère suivant, on teste à nouveau la correspondance entre S et C , et ce jusqu'à ce que l'une des deux chaînes soit épuisée ou que la correspondance soit rompue. Dans le premier cas, si c'est la chaîne S qui est épuisée, l'algorithme a trouvé une correspondance parfaite, il renvoie donc l'indice de la lettre "t" dans C , à savoir 4. Si c'est la chaîne C qui est épuisée, alors aucune correspondance n'a été trouvée, et la fonction retourne -1 .

$$\text{SEARCH}(S \downarrow, C \downarrow) : \left[\begin{array}{l} TS \leftarrow \text{SIZE}(S); TC \leftarrow \text{SIZE}(C); \text{SEARCH} \leftarrow -1; \left\{ \begin{array}{l} TC - i < \\ TS ? ! |_i \end{array} \right. \right. j \leftarrow 1; \left\{ \begin{array}{l} /S[j]=C[j] \\ j \leftarrow j + 1; j > TC ? \end{array} \right. \left. \left. \text{SEARCH} \leftarrow i; * |_i \right\} \right\}_i \end{array} \right]$$

Nous avons supposé ici que la chaîne à trouver C est nécessairement non-vide, sinon le problème est évidemment mal posé.

Squelette et degré d'emboîtement de l'algorithme :

$$\left\{ \begin{array}{l} ? ! |_i \left\{ \begin{array}{l} ? ! |_i \end{array} \right\} \end{array} \right\} \quad d_E = 3$$

L'inconvénient de cet algorithme est son coût de calcul. Dans le pire des cas, pour chacun des m caractères du texte à scanner, on effectue n comparaisons (une comparaison pour chaque lettre de la sous-chaîne), soit un coût total de $m \times n$ opérations. L'algorithme de Boyer-Moore permet de réduire ce coût en cherchant les correspondances à partir de la dernière lettre de la sous-chaîne, autorisant ainsi à ignorer un plus grand nombre de comparaisons à chaque échec de correspondance.

Exemple : convolution matricielle

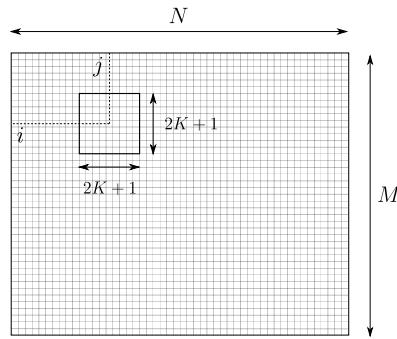
En traitement des images, on appelle convolution, l'opération qui consiste à choisir un masque⁴, sous forme d'une matrice de dimensions données (généralement beaucoup plus petites que celles de l'image à traiter), puis à faire glisser ce masque sur l'image en calculant en chacun de ses pixels le produit scalaire des valeurs des pixels de l'image par celles des pixels du masque.

Cette opération est utilisée entre autres pour lisser une image (on parle de flou gaussien), ou bien au contraire pour faire ressortir ses discontinuités (frontières, éléments saillants...). Elle est en particulier intensivement utilisée en intelligence artificielle appliquée à la reconnaissance d'images, lorsqu'il s'agit de construire des réseaux de neurones artificiels profonds, chaque couche procédant à une convolution de la couche de niveau précédent afin de rechercher dans l'image des motifs de plus en plus complexes. A titre d'exemple, l'un des derniers réseaux mis au point par Microsoft (He et al, 2015) ne comporte pas

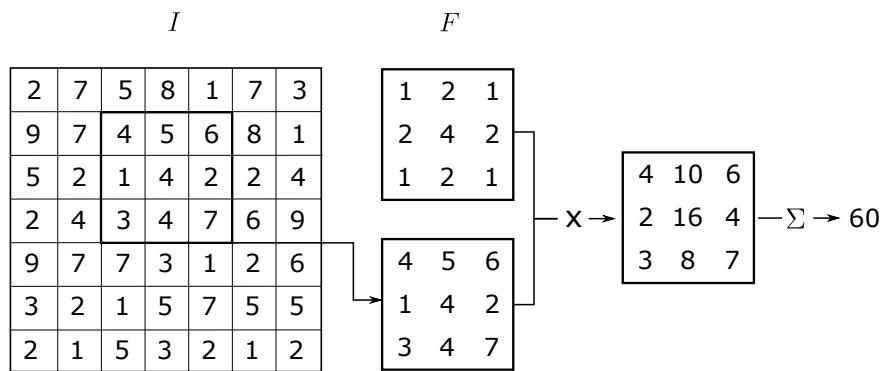
4. On utilise aussi les termes *noyau* ou *filtre*

moins de 34 couches, pour un total d'environ 7500 convolutions. Cela laisse pantois lorsque l'on sait que chaque convolution nécessite de l'ordre de 50 000 multiplications (pour une image de petite taille en entrée) et que pour espérer obtenir des résultats intéressants, l'entraînement du réseau requiert plusieurs centaines de milliers d'images. Au regard de ces quantités astronomiques, on comprend aisément l'importance de disposer d'un module efficace de convolution.

Considérons une image en niveaux de gris, modélisée par une matrice I de M lignes et N colonnes, dont chaque cellule I_{ij} contient un entier compris entre 0 et 255. Nous ne traiterons ici que le cas où le masque est une matrice carrée F de dimension impaire égale à $2K + 1$, ce qui représente l'écrasante majorité des cas d'applications en pratique.



La calcul de la convolution de l'image I par le masque F consiste alors à faire glisser F sur les cellules de I . En chaque position, on calcule le produit scalaire de F (dans sa version dépliée en une seule ligne) par la portion de I couverte par F . Prenons un exemple :



Dans l'exemple ci-dessus, la somme des produits terme-à terme de F par la zone de I couverte par F donne 60. Cette opération est alors réitérée par *balayage* sur I , si bien que la sortie du calcul est également une matrice (notons que la la convoluée $I * F$ est plus petite que l'image originale).

$$\begin{array}{c}
 I \\
 \hline
 \begin{array}{|c|c|c|c|c|c|c|c|} \hline 2 & 7 & 5 & 8 & 1 & 7 & 3 \\ \hline 9 & 7 & 4 & 5 & 6 & 8 & 1 \\ \hline 5 & 2 & 1 & 4 & 2 & 2 & 4 \\ \hline 2 & 4 & 3 & 4 & 7 & 6 & 9 \\ \hline 9 & 7 & 7 & 3 & 1 & 2 & 6 \\ \hline 3 & 2 & 1 & 5 & 7 & 5 & 5 \\ \hline 2 & 1 & 5 & 3 & 2 & 1 & 2 \\ \hline \end{array}
 \end{array}
 *
 \begin{array}{c}
 F \\
 \hline
 \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 2 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}
 \end{array}
 =
 \begin{array}{c}
 I * F \\
 \hline
 \begin{array}{|c|c|c|c|c|} \hline 49 & 47 & 41 & 49 & 42 \\ \hline 39 & 35 & 40 & 46 & 49 \\ \hline 44 & 47 & 36 & 38 & 45 \\ \hline 45 & 43 & 41 & 41 & 50 \\ \hline 39 & 35 & 39 & 36 & 36 \\ \hline \end{array}
 \end{array}$$

Dans l'exemple ci-dessus, le noyau F se déplace d'une cellule entre chaque produit scalaire. On dit qu'il s'agit d'un filtre de pas 1. Il est possible de généraliser et de créer des filtres de pas p . Remarquons que plus p est grand, plus la taille de la convoluée est petite.

Ecrire en ADL un algorithme, prenant en entrée une image I , un masque F (dont on vérifiera dans le code qu'il s'agit bien d'une matrice carrée de dimension impaire), ainsi qu'un pas entier p et qui retourne la matrice convoluée $I * F$.

$\text{CONV}(I \downarrow, F \downarrow, p \downarrow, \text{code} \uparrow) : \llbracket$

$\text{code} \leftarrow 0$ \textcircled{C} initialisation \textcircled{C}

\textcircled{C} Récupération des dimensions \textcircled{C}
 $TI \leftarrow \text{SIZE}(I)$; $TF \leftarrow \text{SIZE}(F)$; $M \leftarrow TI_1$; $N \leftarrow TI_2$; $K1 \leftarrow TF_1$; $K2 \leftarrow TF_2$;

\textcircled{C} Tests de validité des entrées \textcircled{C}

$K1 \neq K2$? $\text{code} \leftarrow 1$; $\textcircled{!} \mid \textcircled{C}$ le masque doit être une matrice carrée \textcircled{C}
 $K1 \% 2 = 0$? $\text{code} \leftarrow 2$; $\textcircled{!} \mid \textcircled{C}$ la taille du masque doit être impaire \textcircled{C} $K \leftarrow K1 \div 2$;
 $K1 > \text{MIN}(M, N)$? $\text{code} \leftarrow 3$; $\textcircled{!} \mid \textcircled{C}$ le masque doit être plus petit que l'image \textcircled{C}

\textcircled{C} Calcul des dimensions de la convoluée \textcircled{C}

$CI \leftarrow (M - K1) \div p + 1$; $CJ \leftarrow (N - K1) \div p + 1$; $\left\{ \begin{array}{l} CI \\ CJ \end{array} \right\}_{i:1} \left\{ \begin{array}{l} C_{ij} \\ i:j \end{array} \right\}_i \left\{ \begin{array}{l} C_{ij} \leftarrow 0 \\ i:j \end{array} \right\}_i$

\textcircled{C} Calcul de la convoluée \textcircled{C}

$\left\{ \begin{array}{l} CI \\ CJ \end{array} \right\}_{i:1} \left\{ \begin{array}{l} ci \leftarrow (i-1)*p+K+1 \\ cj \leftarrow (j-1)*p+K+1 \end{array} \right\}_{i:j} \left\{ \begin{array}{l} \text{centre de l'opération dans I} \\ \textcircled{C} \end{array} \right\}_{i:j} \left\{ \begin{array}{l} C_{ij} \leftarrow C_{ij} + I_{ci+k, cj+l} * F_{k+K+1, l+K+1} \\ k:-K \\ l:-K \end{array} \right\}_{i:j} \left\{ \begin{array}{l} \left\{ \begin{array}{l} C_{ij} \\ i:j \end{array} \right\}_i \\ \left\{ \begin{array}{l} C_{ij} \\ i:j \end{array} \right\}_j \end{array} \right\}_i$

\textcircled{C} Sortie \textcircled{C}

$\text{CONV} \leftarrow C \llbracket$

Squelette de l'algorithme : $? \mid \downarrow_i ? \mid \downarrow_i ? \mid \downarrow_i \{ \{ \} \{ \{ \{ \} \} \} \} \quad d_E = 4$

Exemple : calcul de courbes de niveau sur un MNT

Dans cet exercice, nous supposons avoir à disposition un modèle numérique de terrain (de résolution quelconque), sous forme d'une matrice de M lignes et N colonnes. Chaque cellule contient la valeur de l'altitude du terrain (exprimée en m). L'objectif de l'exercice est d'extraire les courbes de niveau associées à ce MNT.

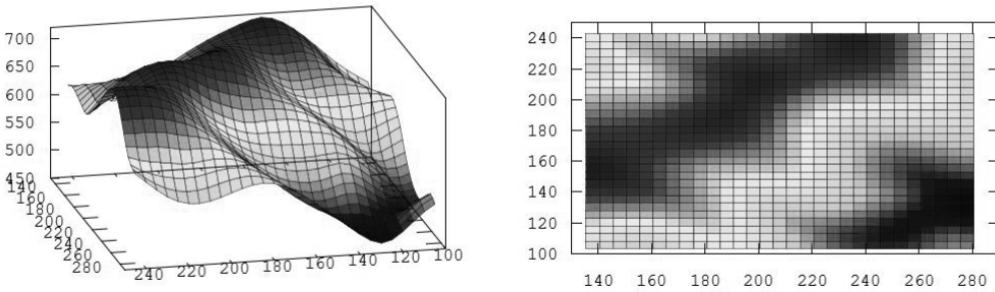
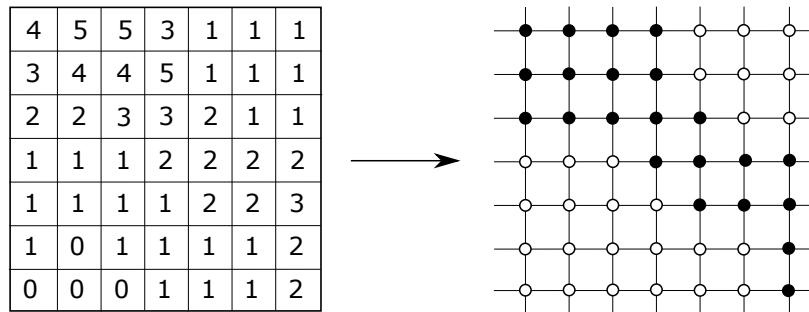


FIGURE 1.1 – Un modèle numérique de terrain sous forme raster

Pour résoudre ce problème, nous allons utiliser la technique dite des *marching squares* qui, comme son nom le laisse deviner, consiste à calculer les isolignes indépendamment dans chaque carré dont les sommets sont formés par 4 cellules voisines. Nous considérons ci-dessous la méthode pour calculer l'isoligne de cote z_0 donnée.

La première étape de la méthode consiste à construire le graphe dual du MNT, en plaçant un sommet sur chaque centre de cellule. Les sommets d'altitude inférieure à z_0 sont marqués en blanc, les autres sont marqués en noir. Par exemple, pour $z_0 = 1.5$, on obtient la construction duale ci-après.

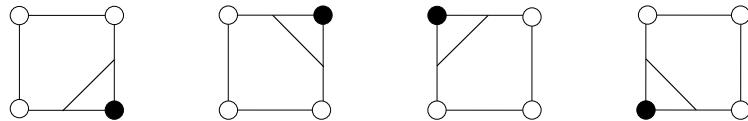


Dans chaque cellule, il y alors 3 cas de figure possibles :

Cas 1 : les 4 coins sont du même côté de la valeur seuil. Dans ce cas, aucune isoligne ne passe par la cellule.



Cas 2 : 3 des 4 coins exactement sont du même côté de la valeur seuil. Dans ce cas, l'isoligne passe dans le coin de la valeur qui est différente.



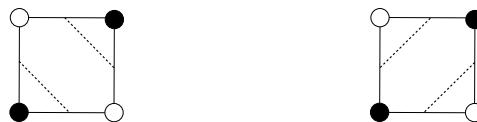
Notons que nous ne donnons ici que la représentation de 4 des 8 configurations du cas 2. Il ne faut pas oublier les 4 configurations symétriques que l'on peut obtenir en inversant les couleurs des marqueurs.

Cas 3 : enfin, dans le dernier cas de figure, 2 coins sont marqués d'une couleur tandis que les 2 autres coins sont marqués avec l'autre couleur. Cela donne alors lieu à 2 sous-cas différents.

Dans la première configuration, les marqueurs de même couleur sont situés sur les mêmes côtés du carré délimitant la cellule. L'isoligne coupe alors directement le carré en deux parties, de sorte à séparer les marqueurs blancs et noirs. L'orientation de l'isoligne est déterminée par la répartition des marqueurs.



La deuxième sous-configuration est quant à elle légèrement plus complexe à gérer puisqu'elle nécessite le passage de deux isolignes dans la cellule. Nous sommes sur un *point de col*. Cependant, pour une même configuration des marqueurs, 2 couples d'isolignes différents peuvent se produire.



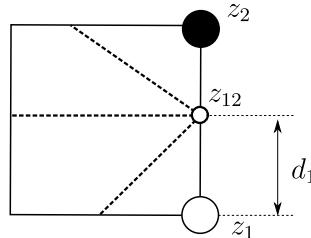
Il faut lever l'ambiguïté en calculant la valeur du point central de la cellule, que l'on obtient par moyenne arithmétique des altitudes des 4 coins. On affecte alors un marqueur blanc ou noir au nouveau point central (en fonction de sa position par rapport au seuil z_0) et on déduit la configuration valide. On parle de désambiguïsation.



Maintenant que nous savons par quels côtés de la cellule passent les segments d'isolignes en fonction de la configuration des marqueurs, il ne nous reste alors plus qu'à préciser comment calculer les positions des intersections.

Remarquons qu'une isoligne ne peut croiser que des côtés de cellule dont les marqueurs aux extrémités sont différents.

Notons alors z_1 et z_2 les altitudes correspondant aux extrémités de l'arête de la cellule. D'après la remarque précédente, nous avons nécessairement $z_1 \leq z_0 \leq z_2$ (moyennant si nécessaire un échange des indices des sommets).



L'objectif consiste alors à déterminer la distance d_1 de sorte à positionner le point de l'isoligne sur l'arête du carré. La solution retenue par l'algorithme des marching squares est de calculer la position de l'intersection z_{12} par interpolation linéaire pondérée par z_1 et z_2 :

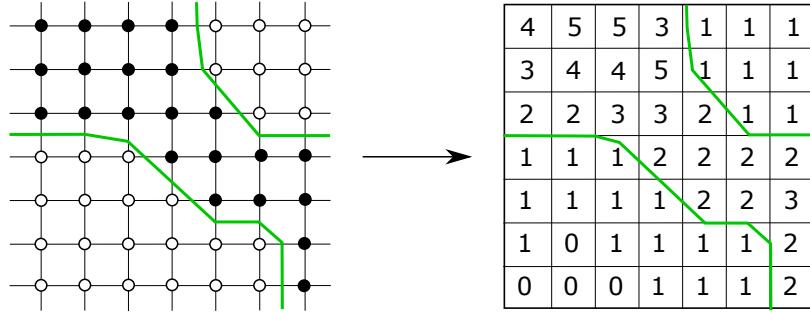
$$i_{12} = \frac{(z_2 - z_0)i_1 + (z_0 - z_1)i_2}{z_2 - z_1}$$

$$j_{12} = \frac{(z_2 - z_0)j_1 + (z_0 - z_1)j_2}{z_2 - z_1}$$

On pourra obtenir une expression plus générale de $\mathbf{p} = (i, j)$ en procédant avec des valeurs absolues, auquel cas l'expression est indépendante de l'étiquetage des altitudes z_1 et z_2 :

$$\mathbf{p}_{12} = \frac{|z_2 - z_0|\mathbf{p}_1 + |z_1 - z_0|\mathbf{p}_2}{|z_1 - z_2|}$$

L'ensemble des segments tracés dans les cellules est alors assemblé, et on obtient l'isoline de niveau z_0 (par exemple ci-dessous pour $z_0 = 1.5$ m).



Dans cet exercice, on se propose d'écrire une fonction ISOLINES, prenant en entrée un MNT sous forme matricielle, ainsi que deux bornes d'altitudes z_{min} et z_{max} (on supposera que $z_{max} \geq z_{min}$) et un pas p . Le module devra calculer les isolignes d'altitudes comprises entre les bornes indiquées, par incrément p . Ces isolignes seront stockées dans deux tables ISOX et ISOY, contenant deux colonne permettant de recenser les coordonnées initiales et finales des segments de courbes. Chaque table contiendra un segment de courbe de niveau par ligne. Enfin, nous différencierons les courbes associées aux différentes valeurs de seuil à l'aide d'un mécanisme de pointeurs.

Commençons par créer une fonction INTERP, qui prend en entrée une liste Z de 4 altitudes de sommets (rangés dans le sens inverse des aiguilles d'une montre, en partant du coin de coordonnées (i, j)) et qui calcule les coordonnées d'un point de l'isoligne sur une arrête $[p1; p2]$ du carré (où $p1$ et $p2$ sont les indices des sommets dans Z). Le nombre z désigne quant à lui l'altitude de la courbe de niveau. Les coordonnées du point interpolé sur l'arête du cube sont retournées dans deux nombres x et y . La procédure nécessite aussi de connaître les coordonnées (i, j) de la cellule à traiter.

INTERP($Z \downarrow, z \downarrow, p1 \downarrow, p2 \downarrow, i \downarrow, j \downarrow, x \downarrow, y \downarrow$) : $\llbracket x1 \leftarrow j; y1 \leftarrow i; x2 \leftarrow j; y2 \leftarrow i; p1 = 2 ? x1 \leftarrow x1 + 1 | \& p1 = 3 ? x1 \leftarrow x1 + 1; y1 \leftarrow y1 + 1 | \& p1 = 4 ? y1 \leftarrow y1 + 1 | \& p2 = 2 ? x2 \leftarrow x2 + 1 | \& p2 = 3 ? x2 \leftarrow x2 + 1; y2 \leftarrow y2 + 1 | \& p2 = 4 ? y2 \leftarrow y2 + 1 | \& \text{C} \text{ interpolation} \text{ C} w1 \leftarrow \text{ABS}(z - z_{p2}); w2 \leftarrow \text{ABS}(z - z_{p1}); x \leftarrow (w1 * x1 + w2 * x2) / (w1 + w2); y \leftarrow (w1 * y1 + w2 * y2) / (w1 + w2) \rrbracket$

Nous écrivons ensuite un module ADD, permettant d'ajouter un segment d'isoligne (défini par les coordonnées de ses deux extrémités) au tableau de résultat.

ADD($xini \downarrow, yini \downarrow, xfin \downarrow, yfin \downarrow, ISOX \downarrow, ISOY \downarrow$) : $\llbracket T \leftarrow \text{SIZE}(ISOX); n \leftarrow T_1; ISOX_{n,1} \leftarrow xini; ISOX_{n,2} \leftarrow xfin; ISOY_{n,1} \leftarrow yini; ISOY_{n,2} \leftarrow yfin; \rrbracket$

Ceci permet alors d'écrire plus simplement la fonction de calcul des segments d'une isoligne, de cote z donnée sur le MNT

ISOLINE(MNT \downarrow , $z \downarrow$, ISOX \downarrow , ISOY \downarrow) : \llbracket

④ Récupération des tailles des matrices en entrée ④
 $NI \leftarrow TISO_1$; $TMNT \leftarrow SIZE(MNT)$; $M \leftarrow TMNT_1$; $N \leftarrow TMNT_2$;

④ Boucle dans les cellules du MNT ④
 $\left\{_{i:1}^{M-1} \left\{_{j:1}^{N-1} Z_1 \leftarrow MNT_{ij}; Z_2 \leftarrow MNT_{i+1,j}; Z_3 \leftarrow MNT_{i+1,j+1}; Z_4 \leftarrow MNT_{i,j+1}; \right. \right\}_i \right\}_j$

④ Calcul des marqueurs ④ $B_1 \leftarrow Z_1 \leq z$; $B_2 \leftarrow Z_2 \leq z$; $B_3 \leftarrow Z_3 \leq z$; $B_4 \leftarrow Z_4 \leq z$;

④ Décompte des marqueurs ④
 $blanc \leftarrow 0$; $noir \leftarrow 0$; $\left\{_{i:1}^4 B_i ? \begin{cases} blanc \leftarrow blanc + 1 & | noir \leftarrow noir + 1 \end{cases} \right\}_i$

④ Etude par cas ④

$blanc = 0 \cup noir = 0$? ④ cas 1, tous les marqueurs sont de la même couleur : dans ce cas, on ne construit pas d'isoligne ④ $\rightrightarrows | i$

$blanc = 1 \cup noir = 1$? ④ cas 2, un marqueur est de couleur différente : construction de l'isoligne en coin ④

$blanc = 1 ?$
 $coin \leftarrow 0$; $\left\{_{i:1}^4 B_i ? \begin{cases} coin \leftarrow i; ! & | i \end{cases} \right\}_i$ ④ récupération de l'indice du coin ④
INTERP($Z, z, coin, (coin)\%4 + 1, xini, yini, i, j$); ④ interpolation du début du segment ④
INTERP($Z, z, coin, (coin - 2)\%4 + 1, xfin, yfin, i, j$); ④ interpolation de la fin du segment ④
ADD($xini, yini, xfin, yfin, ISOX, ISOY$); $\rightrightarrows | i$ ④ sauvegarde du segment d'isoligne ④

$| i$ ④ cas 3, il y a 2 sommets de chaque couleur ④

$B_1 \neq B_3$? ④ le segment d'isoligne coupe la cellule en passant par 2 arêtes opposées ④

$B_1 = B_2$? ④ la séparation est horizontale ④ $p11 \leftarrow 1$; $p12 \leftarrow 4$; $p21 \leftarrow 2$; $p22 \leftarrow 3$;
 $|$ ④ la séparation est verticale ④ $p11 \leftarrow 1$; $p12 \leftarrow 2$; $p21 \leftarrow 3$; $p22 \leftarrow 4$; i
INTERP($Z, z, p11, p12, xini, yini, i, j$); ④ interpolation sur l'arête gauche ④
INTERP($Z, z, p21, p22, xfin, yfin, i, j$); ④ interpolation sur l'arête droite ④
ADD($xini, yini, xfin, yfin, ISOX, ISOY$); \rightrightarrows ④ sauvegarde du segment d'isoligne ④

$|$ ④ on est sur un point de col : il faut calculer un point supplémentaire au milieu de la cellule et construire 2 segments d'isoligne ④
 $z5 \leftarrow (z1 + z2 + z3 + z4)/4$; $b5 \leftarrow z5 \leq z$;

$B_1 = b5$? ④ le point central et le point inférieur gauche sont de même couleur ④
 $p111 \leftarrow 1$; $p112 \leftarrow 4$; $p121 \leftarrow 3$; $p122 \leftarrow 4$; $p211 \leftarrow 1$; $p212 \leftarrow 2$; $p221 \leftarrow 2$; $p222 \leftarrow 3$;

$|$ ④ le point central et le point inférieur gauche sont de couleurs différentes ④
 $p111 \leftarrow 1$; $p112 \leftarrow 4$; $p121 \leftarrow 1$; $p122 \leftarrow 2$; $p211 \leftarrow 2$; $p212 \leftarrow 3$; $p221 \leftarrow 3$; $p222 \leftarrow 4$; i

(c) Premier segment d'isoligne (c)
INTERP($Z, z, p111, p112, xini, yini, i, j$); (c) interpolation sur l'arête gauche (c)
INTERP($Z, z, p121, p122, xfin, yfin, i, j$); (c) interpolation sur l'arête droite (c)
ADD($xini, yini, xfin, yfin, ISOX, ISOY$); $\lceil \rceil$ (c) sauvegarde du segment d'isoligne (c)

(c) Second segment d'isoligne (c)
INTERP($Z, z, p211, p212, xini, yini, i, j$); (c) interpolation sur l'arête gauche (c)
INTERP($Z, z, p221, p222, xfin, yfin, i, j$); (c) interpolation sur l'arête droite (c)
ADD($xini, yini, xfin, yfin, ISOX, ISOY$); $\lceil \rceil$ (c) sauvegarde du segment d'isoligne (c)

i (c) fin du test de comparaison de B_1 et B_2 (c)

$\} \}_{j_i}$ (c) fin de la double boucle sur les cellules du MNT (c)

$\]]$

Squelette de l'algorithme :

$$\left\{ \left\{ \left\{ \right\} ? \lceil \rceil \mid i \left\{ ? ! \mid i \right\} \mathbf{INTERP} \mathbf{INTERP} \mathbf{ADD} \lceil \rceil \mid i ? ? \mid i ? \mid i ? \mathbf{INTERP} \mathbf{INTERP} \mathbf{ADD} \lceil \rceil \mid ? \mid i \mathbf{INTERP} \mathbf{INTERP} \mathbf{ADD} \mathbf{INTERP} \mathbf{INTERP} \mathbf{ADD} \lceil \rceil \mid i \right\} \right\}$$

Degré d'emboîtement :

$$d_E = 4$$

Nous pouvons alors écrire la fonction complète :

ISOLINES(MNT \downarrow , zmin \downarrow , zmax \downarrow , p \downarrow , ISOX \uparrow , ISOY \uparrow) : $\llbracket c \leftarrow 1; z \leftarrow \text{zmin};$
 $\left\{ ^{z \leq \text{zmax}} \mathbf{ISOLINE}(\text{MNT}, z, \text{ISOX}, \text{ISOY}); P_{c,1} \leftarrow z; T \leftarrow \text{SIZE}(\text{ISOX}); P_{c,2} \leftarrow T_1;$
 $c > 1? P_{c,2} \leftarrow P_{c,2} - P_{c-1,2} \mid \lrcorner c \leftarrow c + 1; z \leftarrow z + p; \right\} \rrbracket$

Squelette de l'algorithme :

$$\left\{ \mathbf{ISOLINE} ? \mid i \right\}$$

Degré d'emboîtement :

$$d_E = 2$$

On obtient alors pour la fonction complète un degré d'emboîtement égal à 7 (ne pas oublier le module INTERP, de degré 1).

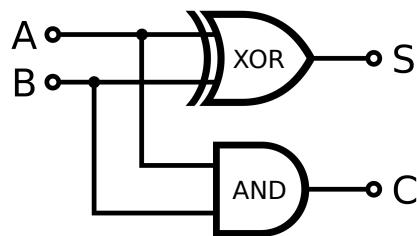
1.7 Exercices

Pour chacun des algorithmes ci-dessous, lorsqu'il est demandé d'écrire un algorithme, on donnera le code ADL, le squelette ainsi que le degré d'emboîtement.

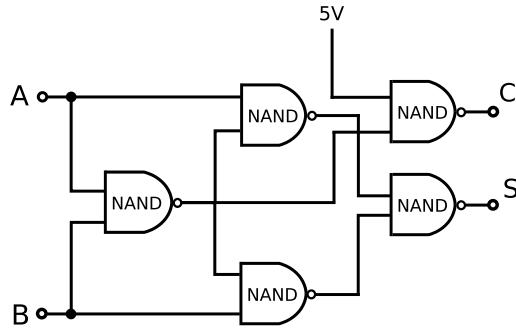
Exercice 1.24. Demi-additionneur logique *

Le processeur d'une machine contient un module spécialement dédié aux opérations arithmétiques, l'unité arithmétique et logique (ALU), a minima capable de réaliser des additions, des soustractions et des comparaisons, parfois même des multiplications en nombres flottants. Les opérations à effectuer y sont cablées matériellement, permettant ainsi d'atteindre une rapidité de calcul inégalable.

Dans cet exercice, nous nous intéressons à l'additionneur, composée de modules unitaires appelés *additionneurs 1 bit*, chargés de prendre en entrée 2 bits ainsi qu'éventuellement une retenue, de les sommer (en base binaire) et de propager la retenue sur le bit de poids immédiatement supérieur. L'additionneur du bit de poids le plus faible ne prend évidemment pas de retenue en entrée. Pour cette raison, il est appelé *demi-additionneur*. On donne ci-dessous le schéma logique d'un demi-additionneur, où A et B représentent les bits d'entrée, tandis que S et C dénotent respectivement le résultat de la somme binaire $A + B$ ainsi que la retenue éventuelle du calcul.



Pour n'avoir recours qu'à un unique type de circuit intégré, on peut construire un demi-additionneur à l'aide de portes logiques NAND, comme illustré sur le schéma ci-dessous. L'entrée 5V désigne une entrée reliée à l'alimentation positive du circuit, correspondant à un niveau logique invariablement haut.



Q1. Pour chacun des deux schémas ci-dessus, exprimer à l'aide d'une expression arithmétique de type booléen, les sorties S et C en fonction de A et B . Montrer que les deux circuits sont équivalents d'un point de vue logique.

Q2. La transformation ci-dessus est en réalité le résultat d'une propriété plus générale : tout circuit logique peut être exprimé uniquement à l'aide de portes logiques de type NON-ET. On parle d'*universalité de la porte NAND*. Démontrer cette propriété. Quid de la porte NOR ?

Exercice 1.25. *Miroir d'une chaîne de caractères* *

Ecrire une fonction qui prend en entrée une chaîne de caractères de longueur quelconque et qui retourne la chaîne renversée. Par exemple, pour l'entrée "programme", la fonction retournera "emmargorp".

Exercice 1.26. *Décompte de mots* *

Ecrire une fonction retournant le nombre de mots présents dans une chaîne de caractères.

Exercice 1.27. *Calcul de π par Monte-Carlo* *

Les méthodes dites de Monte-Carlo désignent l'ensemble des algorithmes permettant de trouver une approximation d'un résultat numérique à l'aide de tirages aléatoires. Malgré leur vitesse de convergence relativement faible, elles sont utilisées lorsque l'on ne connaît pas de méthode de résolution exacte au problème posé.

Dans cet exercice, on s'intéresse au calcul d'une approximation de π par un algorithme type Monte-Carlo. On considère un cercle de rayon r , centré en (x_c, y_c) et contenu dans une zone rectangulaire d'emprise $[x_{min}, y_{min}, x_{max}, y_{max}]$ (on suppose bien évidemment que les bornes sont choisies de telle sorte que l'intégralité du cercle soit inclue dans l'emprise de la zone). Le principe de la méthode consiste alors à tirer N points aléatoirement, de manière uniforme dans l'emprise, puis à compter le nombre n de points qui sont situés dans le cercle.

Le rapport $\frac{n}{N}$ tend alors à se rapprocher du rapport de la surface occupée par le cercle sur l'aire de l'emprise à mesure que le nombre N de points tirés est grand.

$$\frac{n}{N} \xrightarrow{N \rightarrow +\infty} \frac{\text{Aire(cercle)}}{\text{Aire(emprise)}}$$

Ecrire le programme permettant de déterminer une valeur approchée de π .

Exercice 1.28. *Analyse des fréquences dans une chaîne de caractères **

Ecrire une fonction prenant en entrée une chaîne de caractères de longueur quelconque, et retournant un tableau contenant la liste des mots présents dans le texte en première colonne (chaque mot devra être présent au plus une fois dans le tableau), tandis que la seconde colonne répertorie les fréquences (exprimées en pourcentage) d'occurrences des mots dans la chaîne passée en entrée.

Exercice 1.29. *Ligne de somme maximale dans une matrice **

Ecrire une fonction prenant en entrée un tableau à deux dimensions et retournant l'indice de la ligne dont la somme des éléments est maximale.

Exercice 1.30. *Logarithme itéré ***

On définit le logarithme itéré comme le nombre de fois qu'il faut appliquer le logarithme à une quantité avant d'obtenir un résultat inférieur ou égal à 1. On le note \log_b^* .

Par exemple, le logarithme itéré en base 2 de 64 vaut 4 puisque $\log_2(64) = 6$, $\log_2(8) = 3$, $\log_2(3) \approx 1.58$ et enfin $\log_2(1.58) \leq 1$.

Ecrire une fonction récursive prenant en entrée b et x , et retournant $\log_b^* x$.

Exercice 1.31. *Recherche dichotomique ***

Rechercher par dichotomie si un vecteur V de nombres triés dans l'ordre croissant, contient un élément e .

Exercice 1.32. *Opérations matricielles creuses ***

On donne deux matrices A et B exprimées sous forme matricielle creuse. Ecrire les algorithmes permettant de :

- Recherche un élément dans la matrice (on suppose que les indices sont triés dans l'ordre des lignes puis pour chaque ligne dans l'ordre des colonnes).
- Additionner 2 matrices creuses. La somme devra être retournée sous forme creuse.
- Multiplier 2 matrices creuses. Le produit devra être retourné sous forme creuse.

Exercice 1.33. *File de priorité ***

On donne une liste de noms associé à une liste de priorités (1, 2 ou 3) de mêmes cardinalités. Ecrire un algorithme efficace permettant de trier les noms en fonction de leur priorité (en partant de 1 jusqu'à 3).

Exercice 1.34. *Multiplication de grands entiers ★★*

On considère un entier sous forme d'un tableau dont chaque cellule représente une décimale. Ecrire un algorithme permettant d'effectuer la multiplication de deux entiers (de tailles potentiellement différentes).

Exercice 1.35. *Jeu de la vie ★★*

Le jeu de la vie est un automate cellulaire répondant aux règles suivantes :

Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît). Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Q1. Ecrire un algorithme permettant de générer N étapes d'évolution.

Q2. Réécrire l'algorithme en utilisant directement la fonction CONV présentée en page 53.

Exercice 1.36. *Problèmes de satisfiabilité ★★*

On appelle problème de *satisfiabilité logique* (abrégé problème SAT), le problème formulé par l'énoncé suivant :

Etant donnée une formule logique sous forme normale conjonctive (FNC), existe-t-il une affectation des variables permettant de vérifier la formule.

On rappelle qu'une FNC est une conjonction de clauses disjonctives de littéraux (*cf* page 22). Par exemple, les deux formules ci-dessous sont des instances du problème SAT :

$$F_1 = (w + x + y + z)(w + \bar{z})(y + z)$$

$$F_2 = (x + y)(x + \bar{y})(\bar{x} + y)(\bar{x} + \bar{y})$$

Notons que F_1 est satisfiable (il suffit de prendre : $(w, x, y, z) = (1, 0, 1, 0)$ par exemple). En revanche, F_2 n'est pas satisfiable (quelque soit l'affectation de (x, y) , exactement une des clauses est fausse).

Par convention, on pose que dans chaque clause les littéraux doivent être distincts (y compris à la négation près), ce qui se justifie facilement par les règles d'idempotence ($a + a = a$) et de complémentaire ($a + \bar{a} = 1$), qui montrent qu'il est toujours possible de se ramener à un problème respectant cette condition.

Notons enfin que le problème SAT n'impose pas de conditions sur le nombre de littéraux par clause. On considère donc le problème dérivé 3-SAT :

Etant donnée une FNC dont les clauses contiennent exactement 3 variables, existe-t-il une affectation permettant de vérifier la formule.

L'objectif du problème consiste à transformer toute instance du problème SAT en une instance de 3-SAT.

Q1. A l'aide des règles de calcul booléen, montrer les deux équivalences logiques suivantes :

$$\forall (x, a, b) \in \mathbb{B}^3 \quad x = (x + a + b)(x + \bar{a} + b)(x + a + \bar{b})(x + \bar{a} + \bar{b})$$

$$\forall (x, y, a) \in \mathbb{B}^3 \quad x + y = (x + y + a)(x + y + \bar{a})$$

Q2. Soit $(l_1, l_2, \dots, l_n) \in \mathbb{B}^n$ un ensemble de variables booléennes (avec $n > 3$), et on note :

$$F = \sum_{k=1}^n l_k = l_1 + l_2 + \dots + l_n$$

où le symbole Σ dénote une disjonction des n variables.

On introduit une variable supplémentaire $a \in \mathbb{B}$. Montrer que F est satisfiable si et seulement si G est satisfiable :

$$G = \left(\sum_{k=1}^{n-2} l_k + a \right) \left(l_{n-1} + l_n + \bar{a} \right)$$

Q3. A l'aide des réponses aux 2 questions précédentes, montrer que toute instance du problème SAT peut être transformée en une instance de 3-SAT.

Q4. Soit $(v, w, x, y, z) \in \mathbb{B}^5$. On donne le problème SAT suivant :

$$F = (x + y)z(\bar{x} + y + \bar{z})(v + w + x + \bar{y} + z)$$

A l'aide de la réponse apportée à la question 3, exprimer F sous la forme 3-SAT.

Q5. Pour un problème SAT de n variables et m clauses, donner un majorant du nombre de variables additionnelles créées par la procédure de transformation.

Q6. En admettant⁵ qu'il n'existe pas d'algorithme rapide permettant de résoudre le problème SAT, que peut-on en conclure pour le problème 3-SAT.

Remarque : on peut montrer à l'aide de la théorie des graphes, que le problème 2-SAT (i.e. le problème SAT réduit à des clauses de 2 littéraux) peut être efficacement résolu. Sous les hypothèses formulées à la question 6, il n'existe donc pas de transformation générique de toute FNC à une FNC ne comportant que des clauses de 2 littéraux.

Exercice 1.37. Recherche de la plus longue sous-chaîne ***

5. Hypothèse valide si on suppose également que $P \neq NP$.

Ecrire un algorithme prenant en entrée 2 chaînes de caractères A et B et qui recherche la plus longue sous-chaîne S présente à la fois dans A et B . L'algorithme retournera, en plus de S , la position du premier caractère de S dans A et B (c'est-à-dire que sa sortie sera un triplet composé d'une chaîne de caractères et de 2 entiers).

Exercice 1.38. *Algorithme mean-shift* ★★☆

On considère un ensemble de n points $p_i = (x_i, y_i) \in \mathbb{R}^2$ dont les coordonnées sont stockées dans 2 tableaux (vecteur-colonne) \mathbf{x} et \mathbf{y} , contenant chacun n lignes et 1 colonne. Ainsi, à titre d'exemple, la coordonnée y du i -eme point est stockée dans $\mathbf{y}[i]$.

On s'intéresse au problème de la recherche de la zone sur laquelle la densité de points est maximale. L'algorithme du *mean-shift* apporte une réponse partielle à ce problème, et permet de calculer la position d'un maximum local de cette densité.

L'algorithme est initialisé en tirant aléatoirement une position $c_0(cx_0, cy_0)$ dans l'emprise de la zone. On récupère alors l'ensemble P_0 des points contenus dans un cercle de rayon r centré en c_0 . Nous supposerons ici que la densité de points est suffisamment forte pour garantir que P_0 est non-vide. Le cercle est alors translaté sur un nouveau centre c_1 , positionné sur le centre de gravité des points de l'ensemble P_0 .

On recalcule alors l'ensemble P_1 des points situés dans le cercle, permettant d'estimer un nouveau centre de gravité c_2 , sur lequel on déplace le centre du cercle. A chaque itération j , le cercle c_j englobe l'ensemble de points P_j dont le centre de gravité définit le centre c_{j+1} d'un nouveau cercle et ainsi de suite, jusqu'à ce que l'une des deux conditions suivantes soit remplie :

- Le nombre d'itérations dépasse une valeur prédéfinie M
- Le déplacement de c_0 entre 2 itérations est inférieur à une valeur prédéfinie $\varepsilon > 0$

Ecrire en pseudo-code, la fonction *mean-shift* prenant en entrée 2 tableaux de flottants \mathbf{x} et \mathbf{y} , ainsi que les 3 paramètres r , M et ε , et retournant un tableau contenant les coordonnées du dernier centre calculé.

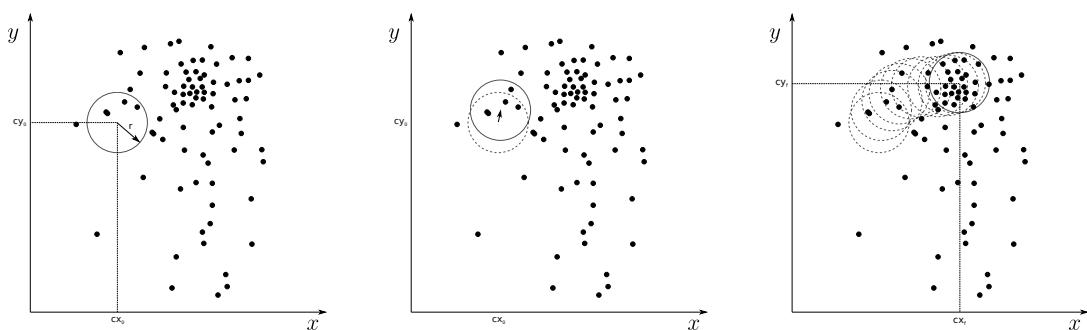


FIGURE 1.2 – Tirage d'un centre c_0 , calcul du centre de gravité c_1 des points contenus dans le cercle de centre c_0 et translation du cercle de c_0 vers c_1 . A droite : itération du processus jusqu'à convergence vers un centre stable

Indications : on commencera par calculer l'emprise $x_{min}, x_{max}, y_{min}, y_{max}$ de l'ensemble de points. Le tirage du premier centre s'effectuera à l'aide de la fonction RAND qui retourne un nombre aléatoire tiré uniformément dans $[0, 1]$.

Chapitre 2

Preuves d'un algorithme

Le test de programmes peut être une façon très efficace de montrer la présence de bugs mais est désespérément inadéquat pour prouver leur absence

Edsger Dijkstra

Sommaire

2.1	Indécidabilité	68
2.1.1	Indécidabilité de la terminaison	69
2.1.2	Indécidabilité de la correction	71
2.2	Preuve de terminaison	71
2.2.1	La boucle à compteur	72
2.2.2	Autres types de boucles	73
2.2.3	Récursivité	74
2.2.4	Exercices	82
2.3	Preuve de correction	84
2.3.1	Programme simple	84
2.3.2	Programme avec boucles	84
2.3.3	Programme récursif	86
2.3.4	Exercices	87

Introduction

Après la conception de chaque algorithme, le programmeur doit impérativement se poser les trois questions suivantes :

- **Terminaison** : le programme termine-t-il en un nombre fini d'opérations ?
- **Correction** : le résultat retourné est-il toujours celui qui est attendu ?
- **Complexité** : comment évoluent le temps de calcul et l'espace mémoire nécessaire en fonction de la taille du problème à traiter ?

Chacun des trois points ci-dessous correspond à un aspect différent de l'algorithme. Remarquons toutefois que terminaison et correction sont indissociables, et il est impossible en pratique de démontrer qu'un programme retourne le résultat attendu, sans démontrer au préalable que le résultat est retourné en un nombre fini d'opérations. En ce sens, il

est pertinent de considérer le problème de terminaison comme une vérification préalable à la preuve de correction. Nous traiterons donc les deux premières questions simultanément dans ce chapitre, reléguant la complexité algorithmique au chapitre suivant (dans lequel nous n'aborderons principalement que le volet temporel de la complexité, laissant ainsi le soin au lecteur de généraliser les techniques étudiées à l'analyse de l'espace mémoire).

Notons qu'il existe aujourd'hui de nombreux frameworks dédiés permettant de tester la terminaison et la correction des programmes écrits¹. Citons par exemple JUnit pour le langage Java, qui propose également un mécanisme de mesure du pourcentage de code couvert par les tests. S'il peut s'agir là d'un outil pratique pour l'évolution et le maintien du programme par une équipe de développeurs, il ne saurait en aucun cas remplacer le travail de preuve formelle qui doit être effectué individuellement par chaque membre de l'équipe en amont du versionnage du code sur la plateforme de travail. Le couvert du code testé n'est qu'un indicateur, et il est illusoire de penser qu'un score atteignant 100% garantisse un programme qui termine et qui donne le bon résultat dans tous les cas. Cela signifie simplement que le processus de test a enclenché l'exécution de la totalité des instructions... pour un nombre généralement très limité de configurations en entrée.

Dans ce chapitre, nous verrons dans un premier temps qu'il est impossible de concevoir un programme informatique qui prendrait en entrée un algorithme quelconque et qui déterminerait si ce dernier s'arrête après un nombre fini d'étapes. Ce résultat a été démontré par Alan Turing en 1936, et il s'agit là de l'une des manifestations les plus parlantes du résultat énoncé par le théorème d'incomplétude de Gödel. Nous verrons ensuite que ce constat d'indécidabilité s'étend également à la preuve de correction d'un programme. Nous apprendrons alors à manipuler les techniques communément utilisées pour établir ces preuves *à la main* pour un code informatique quelconque. De manière triviale, un algorithme qui ne possède pas d'instructions de contrôle sous forme de boucle (boucle *for*, *while*, *until* ou boucle infinie), terminera nécessairement. La preuve de terminaison ne concerne donc que les algorithmes munis de ces structures de boucle.

Le langage de pseudo-code utilisé ici sera le langage ADL, introduit dans le premier chapitre, auquel nous renvoyons le lecteur si besoin est.

2.1 Indécidabilité

Définition : *problème décidable*

En informatique théorique, un problème de décision est qualifié de *décidable*, lorsqu'il existe un algorithme permettant de résoudre le problème en un nombre fini d'étapes.

Remarque 1 : l'indécidabilité est donc la négation de la propriété de décidabilité.

Remarque 2 : dans ce cours, nous parlerons de décidabilité algorithmique, que l'on appelle aussi décidabilité au sens de Turing, ou encore *calculabilité*, par opposition à la décidabilité

1. Certains travaux de recherches récents visent également à produire des preuves de correction automatiques à l'aide d'une analyse formelle du code

logique, dite de Gödel. Notons toutefois que les deux concepts sont intimement liés.

2.1.1 Indécidabilité de la terminaison

Théorème : *indécidabilité de la terminaison*

Il n'existe pas d'algorithme universel prenant en entrée un autre algorithme et indiquant (en un nombre fini d'étapes), si ce dernier termine.

On parle généralement du problème de l'arrêt.

Preuve :

La preuve a été apportée par Alan Turing, suivant un raisonnement par l'absurde. Supposons qu'un tel algorithme existe. Nous l'appellerons TERMINE.

TERMINE prend en entrée une chaîne de caractères A représentant le code d'un programme donné, ainsi qu'une entrée d , et retourne (en un temps fini) la réponse à la question : le programme A termine-t-il pour la donnée d ? Construisons le module TEST :

$$\text{TEST}(A) : \llbracket \text{TERMINE}(A, A) ? \{ \} \mid_i \rrbracket$$

Notons A_T la transcription en chaîne de caractères du programme TEST et posons-nous la question de savoir si le programme TEST termine pour la donnée A_T , ce qui revient à essayer de déterminer si l'instruction **TEST**(A_T) termine.

- Si **TEST**(A_T) termine, alors la proposition **TERMINE**(A_T, A_T) est vraie, le programme entre dans une boucle infinie et l'instruction ne termine donc pas.
- Si **TEST**(A_T) ne termine pas, alors **TERMINE**(A_T, A_T) est fausse, le programme sort immédiatement après le test conditionnel et l'instruction termine donc.

Nous avons donc soulevé une contradiction. C'est donc que l'hypothèse selon laquelle on peut construire une fonction qui résout le problème de l'arrêt est fausse : il ne peut donc pas exister de programme de preuve de terminaison².

Nous donnons ici une seconde démonstration, formulée par Gregory Chaitin, et s'appuyant également sur un raisonnement par l'absurde. Cette alternative est légèrement plus complexe que la version proposée ci-dessus, mais est néanmoins très intéressante de par son niveau d'abstraction.

2. Notons toutefois qu'il existe à l'heure actuelle des programmes de preuve de terminaison sur le marché, mais dont les capacités sont restreintes à un nombre limité de classes d'algorithmes.

Supposons à nouveau avoir à disposition une fonction TERMINÉ décidant si un algorithme se termine après un nombre fini d'opérations.

Pour un langage informatique dont la syntaxe repose sur un ensemble (fini) \mathcal{S} de symboles, un algorithme de longueur k est un élément de \mathcal{S}^k et donc par suite, le nombre d'algorithmes de longueur au plus N que l'on peut écrire, est fini et vaut :

$$M(\mathcal{S}, N) = \sum_{k=0}^N |\mathcal{S}|^k = \frac{1 - |\mathcal{S}|^{N+1}}{1 - |\mathcal{S}|}$$

où $|\mathcal{S}| \geq 2$ désigne le cardinal de \mathcal{S} . Le cas dégénéré $|\mathcal{S}| \leq 1$ est sans intérêt puisque l'on ne peut construire de langage avec une alphabet d'entropie nulle).

La fonction TERMINÉ est alors appliquée à chacun des algorithmes (cette opération se fait également en un nombre fini d'étapes, puisque par définition, la fonction TERMINÉ s'arrête en un nombre fini d'étapes). Pour tous les algorithmes qui terminent, le résultat de la fonction est stockée dans une liste, que l'on parcourt pour en extraire le plus grand entier, que nous noterons n . La fonction TEST retourne alors $n + 1$.

D'un point de vue algorithmique, la fonction TEST s'écrit de la manière suivante (en supposant que REPERTORIE est la fonction générant tous les algorithmes de longueur inférieure à N à partir d'une liste \mathcal{S} de symboles).

TEST($\mathcal{S} \downarrow, N \downarrow$) : $\left[\left[\begin{array}{l} \text{MAX} \leftarrow 0; L \leftarrow \text{REPERTORIE}(N, \mathcal{S}); TL \leftarrow \text{SIZE}(L); \left\{ \begin{array}{l} \overset{TL}{b} \leftarrow \\ \text{TERMINÉ}(L_i); \bar{b} ? \rightarrow |_i r \leftarrow \text{EXECUTE}(L_i); b \leftarrow \text{ISINT}(r); \bar{b} ? \rightarrow |_i r > \text{MAX} \\ ? \text{MAX} \leftarrow r |_i \end{array} \right\}_i \text{TEST} \leftarrow \text{MAX} + 1 \end{array} \right] \right]$

Ici, nous supposons que ISINT est une fonction testant si une variable est entière, et EXECUTE permet de faire tourner un algorithme donné sous forme de chaîne de caractères.

Nous aboutissons à nouveau à une contradiction. Pour un entier N suffisamment grand, la longueur du programme TEST sous forme de chaîne de caractères est telle que TEST fait partie des programmes qu'il répertorie. Puisque le programme TEST ne fait principalement qu'appeler la fonction TERMINÉ (qui termine par hypothèse) sur un nombre fini d'algorithmes, il termine et la valeur $n + 1$ est donc inclue dans la liste des retours des algorithmes répertoriés. Sachant que la sortie de TEST est strictement plus grande que le maximum des retours des algorithmes répertoriés, on obtient un paradoxe, ce qui prouve que la fonction TERMINÉ ne peut exister³.

Il est possible à titre d'exercice, d'écrire l'algorithme REPERTORIE, qui liste l'ensemble des algorithmes écrits avec un nombre au plus égal à N de symboles de l'alphabet \mathcal{S} .

3. En toute rigueur, cette contradiction pourrait alternativement nous faire conclure que c'est la fonction EXECUTE qui ne peut exister. L'existence même de compilateurs et d'interpréteurs (ou de tout autre cas particulier de Machines de Turing Universelles) démontre la possibilité d'écrire EXECUTE.

Remarque : arrêtons-nous un instant sur la généralité du résultat obtenu. Les preuves construites par Turing et Chaitin montrent qu'il ne peut exister d'algorithme qui indique en un temps fini si un programme (*i.e.* une procédure avec des entrées fixées) termine. *A fortiori*, il ne peut donc pas exister de programme qui indique si une procédure termine quelles que soient les données qui lui sont passées en entrée.

2.1.2 Indécidabilité de la correction

La propriété d'indécidabilité peut être étendue au problème de la correction.

Définition : *correction d'un algorithme*

Pour un algorithme donné A , la preuve de correction désigne la démonstration qui permet d'établir que A retourne toujours le résultat attendu en un nombre fini d'opérations, quelles que soient les données qui lui sont passées en entrée.

Remarque : il est clair que l'on ne peut apporter de preuve de correction pour un programme qui ne termine pas.

Théorème de Rice :

Toute question *sémantique* non-triviale sur les programmes informatiques est indécidable.

Ici, *non-trivial* signifie que la réponse n'est pas une tautologie. Par exemple, la question : *le programme retourne-t-il une sortie de longueur supérieure ou égale à 0 ?* est triviale.

Le terme *sémantique*, par opposition à *syntaxique*, indique que la question porte sur le fonctionnement du programme (et donc en particulier sur ses sorties), et non sur le code.

Notons que ce résultat implique en pratique qu'il est impossible de créer un algorithme qui prédise à l'avance la quantité de mémoire requise par un programme (allocation optimale) ou qui détermine si un programme retourne le résultat attendu (par rapport à ses spécifications). D'un point de vue plus opérationnel, il est également impossible de déterminer automatiquement si un programme comporte un risque de faire planter la machine, s'il contient un virus informatique, s'il accède à un site internet, etc.

2.2 Preuve de terminaison

Dans cette partie, nous donnons les méthodes générales permettant de démontrer qu'un algorithme se termine en un nombre fini d'opérations.

Soit un algorithme f prenant ses valeurs dans un ensemble \mathcal{D} de cardinal éventuellement infini. Par exemple, lorsque l'entrée d'un algorithme est un entier : $\mathcal{D} = \mathbb{Z}$. Si l'algorithme prend en entrée un vecteur numérique de taille quelconque : $\mathcal{D} = \bigcup_{n \in \mathbb{N}} \mathbb{R}^n$, et ainsi de suite.

Définition 1 : on dit que f termine pour un argument $x \in \mathcal{D}$ si et seulement si l'appel de $f(x)$ se termine en un nombre fini d'opérations élémentaires (affectations, calculs algébriques...).

Remarque : chaque passage dans une boucle implique au minimum une affectation (celle du compteur) ou une comparaison (conditions), même lorsque le contenu de la boucle est vide. Nous admettrons qu'il en va de même pour les boucles infinies.

Définition 2 : on dit que f termine si et seulement si pour toute valeur d'entrée $x \in \mathcal{D}$, f se termine pour l'argument x .

Nous reviendrons sur cette notion au chapitre 3, mais on peut voir ici que prouver la terminaison d'un algorithme revient à montrer que sa complexité temporelle dans le pire des cas est finie, quelle que soit l'unité de coût utilisée.

Comme précisé précédemment, lorsque le code ne comporte pas de boucle, la preuve de terminaison est triviale et on pourra se dispenser de la détailler.

2.2.1 La boucle à compteur

En général, une boucle à compteur se termine au bout d'un nombre d'étapes :

$$N = \left\lceil \frac{f - i}{p} \right\rceil + 1$$

où : i , f et p désignent respectivement les valeurs initiale, finale et de pas de compteur.

La preuve de terminaison sur une boucle à compteur est donc bien souvent triviale excepté deux cas particuliers, qui peuvent la rendre sensiblement plus calculatoire :

- Un ou plusieurs des identificateurs f et p sont modifiés : dans ce cas, on devra les exprimer à l'aide de 2 suites numériques $(f_n)_{n \in \mathbb{N}}$ et $(p_n)_{n \in \mathbb{N}}$ dont on recherchera le terme de rang n . On exprimera alors le compteur sous forme d'une troisième suite :

$$u_n = i + \sum_{j=1}^n p_j$$

On cherchera alors à démontrer que, quelles que soient les données en entrée, pour un certain rang n_0 , $u_{n_0} \geq f_{n_0}$. L'entier n_0 correspond alors (à une unité près) au nombre de passages dans la boucle (cf exercice 8 du chapitre 1, page 29). Notons

que le compteur peut également être modifié dans la boucle.

Les quantités exprimées sous forme de suites (numériques, alphanumériques ou bien encore booléennes), sont appelées des **variants de boucle**.

- La boucle contient des débranchements (en particulier des retours à l’itération courante $\leftarrow \top$). Dans ce cas, on étudie séparément les données qui passent par le débranchement et on montre qu’après un nombre fini de débranchements, l’algorithme suit le cours normal de la boucle. En général, on a également recours à des variants de boucles booléens.

Notons que dans certains cas, une boucle peut contenir à la fois des paramètres variables et des débranchements. On évitera tant que possible d’écrire ce genre de code, afin de faciliter la preuve de terminaison.

2.2.2 Autres types de boucles

Pour les autres types de boucles (*while*, *until* et boucle infinie), le principe est le même. On cherche à exprimer un variant de boucle sous forme de suite (éventuellement à valeur dans l’ensemble des booléens ou dans l’ensemble des chaînes de caractères). On montre alors, qu’après un nombre fini d’itérations, les conditions de sortie de boucle sont réunies.

Exemple : calcul du PGCD de deux entiers par l’algorithme d’Euclide

$$\text{PGCD}(a \downarrow, b \downarrow) : \llbracket \left\{ \begin{array}{l} /^{b \neq 0} \\ g \leftarrow b; b \leftarrow \text{MOD}(a, b); a \leftarrow g \end{array} \right\} \text{PGCD} \leftarrow a \rrbracket$$

On pose $a_0 = a$ et $b_0 = b$. On recherche alors le nombre d’étapes n_0 tel que $b_{n_0} = 0$. Notons que a_n et b_n sont à valeurs dans \mathbb{N} .

A chaque passage dans la boucle, b_{n+1} prend la valeur de $a_n \bmod b_n$. De manière évidente, pour tout $i, j \in \mathbb{N} : 0 \leq i \bmod j < j$. Il en résulte que : $b_{n+1} < b_n \quad \forall n \in \mathbb{N}$.

La suite (b_n) est donc strictement décroissante et positive. Comme d’autre part elle est à valeur dans \mathbb{N} , il existe $n_0 \in \mathbb{N}$ tel que $b_{n_0} = 0$. L’algorithme se termine donc.

Exercice 2.1. Fonction aléatoire

On considère l’algorithme suivant, dans lequel MAX_NB désigne un paramètre de génération de système, dénotant la valeur maximale prise par un nombre⁴.

$$\left\{ \begin{array}{l} /^{a \neq 0} \cup ^{b \neq 0} \\ b > 0 ? \quad b \leftarrow b - 1 \mid \text{; } a \leftarrow a - 1; b \leftarrow \text{FLOOR}(\text{MAX_NB} * \text{RAND}()) \end{array} \right\}$$

4. Par exemple, en format double précision et dans la norme IEEE 754, de l’ordre de 10^{308} .

Démontrer que la boucle se termine.

2.2.3 Récursivité

Nous traitons ici le problème de terminaison pour des fonctions qui s'appellent elles-mêmes.

Même pour un algorithme récursif qui ne contient explicitement aucune boucle (de quelque type que ce soit), la terminaison n'est jamais garantie.

Dans ce qui suit, nous notons E un ensemble de cardinal potentiellement infini.

Définition : *relation d'ordre*

Une relation d'ordre (notée \preceq) sur E est une relation binaire, reflexive, anti-symétrique et transitive :

- $\forall x \in E \quad x \preceq x$
- $\forall x, y \in E \text{ si } x \preceq y \text{ et } y \preceq x \text{ alors } x = y$
- $\forall x, y, z \in E \text{ si } x \preceq y \text{ et } y \preceq z \text{ alors } x \preceq z$

Exemples :

- L'inégalité classique \leqslant sur l'ensemble des nombres réels est une relation d'ordre.
- L'ordre lexicographique sur les chaînes de caractères est une relation d'ordre.
- L'inclusion sur les parties d'un ensemble est une relation d'ordre

Remarque : lorsque E peut être muni d'une relation d'ordre \preceq , on dit que (E, \preceq) est un ensemble ordonné.

Définition : soit (E, \preceq) un ensemble ordonné. Deux éléments x et y de E sont dit incomparables si on a ni $x \preceq y$, ni $y \preceq x$.

Définition : une relation d'ordre est dite *totale* sur E si tous les couples d'éléments de E sont comparables. Sinon, on parle de relation d'ordre *partielle*.

Exemple : dans $E = \mathbb{N}^*$, on pose la relation d'ordre suivante :

$$m \preceq n \Leftrightarrow \exists k \in \mathbb{N} \text{ tel que } n = mk$$

est une relation d'ordre partielle sur \mathbb{N}^* . En particulier, les éléments 2 et 3 ne sont pas comparables. L'inclusion dans l'ensemble $\mathcal{P}(E)$ des parties de E en est un autre exemple.

Remarque : lorsque E est muni d'une relation d'ordre partielle \preceq , on dit que (E, \preceq) est un ensemble partiellement ordonné.

Définition : ordre lexicographique

Soient $(A_1, \preceq), (A_2, \preceq) \dots (A_n, \preceq)$ des ensembles ordonnés. L'ordre lexicographique sur le produit cartésien $A_1 \times \dots \times A_n$ est défini par :

$$(a_1, \dots, a_n) \preceq (b_1, \dots, b_n) \Leftrightarrow \exists j \in \{1 \dots n\}, a_1 = b_1, \dots, a_{j-1} = b_{j-1} \text{ et } a_j \preceq b_j$$

Ceci définit bien sûr une relation d'ordre sur $A_1 \times \dots \times A_n$. Si chaque ensemble est totalement ordonné, alors l'ordre lexicographique définit lui aussi un ordre total.

Remarque : notons que dans cette définition, les relations d'ordre sur chaque ensemble ne sont pas nécessairement les mêmes puisque les ensembles sont différents.

Définition : ordre strict

L'ordre strict associé à une relation d'ordre est défini par : $(x \prec y) \Leftrightarrow (x \preceq y) \text{ et } (x \neq y)$

Définition : élément minimal

Soit \preceq une relation d'ordre partielle ou totale sur un ensemble non vide E . Soit F un sous ensemble non-vide de E . Un élément $m \in E$ est un élément minimal de F si et seulement si les deux conditions suivantes sont réunies :

- $m \in F$
- $\forall x \in E \quad x \prec m \Rightarrow x \notin F \quad (\text{il n'existe pas d'élément } x \prec m \text{ dans } F)$

Définition : ensemble bien fondé

Soit \preceq une relation d'ordre partielle ou totale sur un ensemble non vide E . On dit que (E, \preceq) est bien fondé si et seulement si toute partie non-vide de E admet au moins un élément minimal. Un ensemble totalement ordonné et bien fondé est dit *bien ordonné*.

Théorème : dans un ensemble bien fondé, toute suite strictement décroissante est nécessairement finie : il n'existe pas de suite infinie (x_n) telle que $\forall n \in \mathbb{N} \quad x_{n+1} \prec x_n$.

Exemple : (\mathbb{N}, \leq) est bien fondé pour l'ordre canonique, tandis que (\mathbb{Z}, \leq) et (\mathbb{R}, \leq) ne le sont pas. On pourra trouver des exemples supplémentaires d'ordres totaux qui ne sont pas bien fondés dans l'ouvrage de B. Hauchecorne (2007) : $[0, 1]$ et $\{1/n \mid n \in \mathbb{N}^*\}$.

On peut démontrer que si les (A_i, \preceq) sont bien fondés, alors l'ensemble produit cartésien $A_1 \times A_2 \times \dots \times A_n$ muni de l'ordre lexicographique est bien fondé. En particulier : (\mathbb{N}^2, \preceq) où \preceq est l'ordre lexicographique est bien fondé. En revanche, l'ordre lexicographique n'est pas une relation bien fondée pour l'ensemble des chaînes de caractères : par exemple, la suite strictement décroissante $\{b, ab, aab, aaab, \dots\}$ est infinie.

Pour démontrer qu'un algorithme récursif termine, on définit une relation d'ordre sur un ensemble image des données par une application, puis on montre que chaque appel récursif est effectué sur une donnée d'ordre strictement inférieur à la donnée utilisée à l'itération

courante. Il suffit alors de vérifier que l'ensemble image muni de la relation d'ordre choisie est un ensemble bien fondé et que le cas de base est traité dans le code (cas terminal, permettant la sortie de l'algorithme récursif).

Pour démontrer qu'un algorithme récursif termine, on suivra les trois étapes suivantes (il s'agit en quelque sorte de la démarche inverse au raisonnement par récurrence) :

Etape 1 : Trouver une fonction $\varphi : \mathcal{D} \rightarrow E$ telle que E puisse être muni d'une relation d'ordre partielle ou totale \preceq de sorte à ce que (E, \preceq) forme un ensemble bien fondé (ce que l'on devra démontrer, sauf dans les cas triviaux).

Etape 2 : On détermine l'ensemble des cas de base $\mathcal{B} \subset \mathcal{D}$ tel que pour tout élément $b \in \mathcal{B}$, $\varphi(b)$ soit minimal dans $\varphi(\mathcal{D})$. On vérifie de même que l'algorithme termine pour l'ensemble des données correspondant à des cas de base.

Etape 3 : On vérifie enfin que l'itération calculant $f(x_n)$ termine et fait un nombre fini d'appels à $f(x_{n+1})$ avec : $\varphi(x_{n+1}) \prec \varphi(x_n)$.

Exercice 2.2. On considère un algorithme prenant en entrée deux entiers naturels : $\mathcal{D} = \mathbb{N}^2$. On choisit l'ensemble bien fondé \mathbb{N} muni de l'inégalité classique \leqslant et la fonction $\varphi : (a, b) \rightarrow \varphi(a, b) = \min(a, b)$. Trouver l'ensemble des cas de base de l'algorithme.

Remarque : en pratique, en plus de l'ensemble \mathcal{B} , il faut aussi considérer l'ensemble $\mathcal{L} \subset \mathcal{D}$ des cas limites pour lesquels la récursivité ne s'applique pas. Parfois \mathcal{B} et \mathcal{L} ne sont pas disjoints.

Exemple 1 : suite de Fibonacci

On rappelle l'expression du code permettant de calculer les termes de la suite de Fibonacci :

$$\text{FIBO}(N \downarrow) : \left[\left[N < 2 ? \text{ FIBO} \leftarrow 1; ! \mid ; \text{ FIBO} \leftarrow \text{FIBO}(N - 1) + \text{FIBO}(N - 2) \right] \right]$$

Dans cet algorithme, l'ensemble des arguments est $\mathcal{D} = \mathbb{N}$, on prendra donc la fonction φ égale à l'identité et l'ensemble bien fondé \mathbb{N} muni de l'inégalité classique \leqslant . L'ensemble des éléments minimaux de \mathbb{N} est le singleton $\{0\}$, dont l'image réciproque par φ est $\varphi^{-1}(\{0\}) = \{0\}$, ce qui nous donne l'ensemble des cas de base : $\mathcal{B} = \{0\}$.

Par ailleurs, la formule de récursivité ne s'applique que si $n - 2 \geqslant 0$, c'est-à-dire : $n \geqslant 2$. L'ensemble des cas limites est donc $\mathcal{L} = \{0, 1\}$. Dans ce cas particulier, on a $\mathcal{B} \subset \mathcal{L}$.

Exemple 2 : exponentiation

Reprendons l'exemple du calcul d'une puissance (entière) d'un nombre, étudié au chapitre précédent.

$$\text{EXP}(X \downarrow, N \downarrow) : \left[\left[N = 0 ? \text{ EXP} \leftarrow 1; ! \mid ; \text{ EXP} \leftarrow X * \text{EXP}(X, N - 1) \right] \right]$$

Démontrons que la fonction récursive EXP termine.

L'ensemble des arguments de l'algorithme est l'ensemble produit $\mathcal{D} = \mathbb{R} \times \mathbb{N}$. Montrons que la fonction termine pour tout élément x de \mathcal{D} .

On pose $\varphi : (a, n) \rightarrow n$ et on utilise l'ensemble bien fondé (\mathbb{N}, \leq) . L'ensemble des cas de base est : $\mathcal{B} = \mathbb{R} \times \{0\}$, qui est bien traité dans l'algorithme.

Enfin, pour toute donnée $x \in \mathcal{D}$, le calcul de $f(x)$ nécessite un nombre fini d'opérations ainsi qu'un appel à $f(y)$ avec $y = \{x, n - 1\}$, d'où $\varphi(y) = n - 1$ et donc : $\varphi(y) < \varphi(x)$.

L'algorithme EXP termine donc.

Exemple 3 : recherche dichotomique dans une liste

Considérons un vecteur V dans lequel on souhaite savoir s'il se trouve un élément donné e . Notons que le type des éléments de V n'a absolument aucune importance ici. On écrit alors la fonction de recherche par dichotomie suivante :

```
SEARCH( $V \downarrow, e \downarrow$ ) :  $\llbracket N \leftarrow \text{SIZE}(V); N = 0 ? \text{SEARCH} \leftarrow \downarrow; ! |_i N = 1$ 
?  $\text{SEARCH} \leftarrow V_1 = e; ! |_i \left\{ \begin{smallmatrix} N \div 2 \\ i:1 \end{smallmatrix} \right. T1_i \leftarrow V_i \right\}_i \left\{ \begin{smallmatrix} N \\ i:N \div 2 + 1 \end{smallmatrix} \right. T2_{i-N \div 2} \leftarrow V_i \right\}_i \text{SEARCH} \leftarrow$ 
SEARCH( $T1$ )  $\cup$  SEARCH( $T2$ )  $\rrbracket$ 
```

La fonction renvoie un booléen, égal à \uparrow si le vecteur contient au moins un élément égal à V et \downarrow sinon.

L'ensemble des données \mathcal{D} est l'ensemble des vecteurs (de taille quelconque). On utilise donc la fonction $\varphi : x \rightarrow \varphi(x) = |x|$ où $|x|$ désigne la taille du vecteur x . La fonction φ est donc à valeur dans \mathbb{N} que l'on ordonne avec l'inégalité classique, de sorte à former l'ensemble bien fondé (\mathbb{N}, \leq) .

L'image réciproque par φ de l'ensemble minimal $\{0\}$ est l'ensemble des vecteurs vides, constituant notre cas de base \mathcal{B} , qui est bien traité dans la fonction et qui termine. Le cas limite \mathcal{L} est quant à lui constitué des singletons, et il est également traité dans la fonction.

Chaque appel récursif nécessite un nombre fini d'opérations, ainsi que 2 appels à la fonction SEARCH, avec 2 sous-vecteurs, dont la taille vaut $\lfloor \frac{n}{2} \rfloor$ si n est pair ou alors est inférieure à $\lfloor \frac{n}{2} \rfloor + 1$ si n est impair.

On démontre facilement que dans le cas où n est pair :

$$\forall n \geq 2 \quad \lfloor \frac{n}{2} \rfloor < n$$

Tandis que dans le cas n impair, on a :

$$\forall n \geq 3 \quad \lfloor \frac{n}{2} \rfloor + 1 < n$$

Dans tous les cas, le calcul de $f(x)$ fait donc un nombre fini d'appels à des instructions $f(y)$ avec $\varphi(y) < \varphi(x)$. Nous avons donc démontré que la fonction SEARCH termine.

Exemple 4 : fonction de Ackermann

Jusqu'à présent toutes les fonctions récursives pouvaient en réalité facilement être programmées à l'aide d'une boucle, la récursivité n'étant utilisé que par élégance. En théorie, toute fonction récursive peut être écrite par un programme itératif, moyennant si besoin la sauvegarde de données dans un table, voire l'émulation complète de la pile d'exécution. Il existe cependant des fonctions pour lesquelles cette tâche est beaucoup plus difficile.

En 1926, Wilhelm Ackermann définit sur l'ensemble des couples d'entiers naturels, la fonction qui aujourd'hui porte son nom, construite via un double appel récursif emboîté, la rendant ainsi plus difficilement formulable en dehors d'un système récursif.

Cette fonction possède de nombreuses propriétés étonnantes, dont en particulier une croissance spectaculaire dès les premiers termes de la suite, et dont on peut montrer qu'elle est plus rapide que celle de toute fonction polynomiale ou même exponentielle. En particulier, le quatrième terme de la suite (dans l'ordre de son premier argument), dépasse de très loin le nombre d'atomes dans l'univers⁵ (alors que les termes précédents n'excèdent pas les quelques dizaines d'unités). Cela est d'autant plus surprenant que la définition de la fonction, outre ses appels récursifs multiples, ne contient que des ajouts et des retraits de $1!$

La fonction d'Ackermann est donc un parfait exemple de fonction calculable (l'objectif de cet exercice est de montrer qu'elle termine), mais dont la calculabilité est impossible en pratique passé les premiers termes.

ACK($M \downarrow, N \downarrow$) : $\llbracket M = 0 ? \text{ACK} \leftarrow N + 1; ! \mid _ N = 0 ? \text{ACK} \leftarrow \text{ACK}(M - 1, 1); !$
 $\mid _ \text{C} \circledcirc \text{ si on est là c'est que } M \text{ et } N \text{ sont tous deux non-nuls } \text{C} \circledcirc$
 $\text{ACK} \leftarrow \text{ACK}(M - 1, \text{ACK}(M, N - 1)) \rrbracket$

Démontrons que la fonction d'Ackermann termine.

Pour ce faire, on utilise l'ensemble bien fondé \mathbb{N}^2 muni de l'ordre lexicographique. La fonction φ est alors l'identité de $\mathbb{N}^2 \rightarrow \mathbb{N}^2$.

L'ensemble des éléments minimaux est le singleton $\{(0, 0)\}$, dont l'image réciproque par φ donne l'ensemble des cas de base : $\mathcal{B} = \{(0, 0)\}$.

L'ensemble des cas limites est l'ensemble des cas pour lesquels l'une des deux formules de récursivité ne fonctionne plus, c'est-à-dire : $m - 1 < 0$ et $n - 1 < 0$.

Il s'agit de l'ensemble des couples d'entiers dont l'un (au moins) est nul. On obtient alors : $\mathcal{L} = \{(m, 0), (0, n) \mid (m, n) \in \mathbb{N}^2\}$ et on remarque que $\mathcal{B} \subset \mathcal{L}$.

5. La fonction de Ackermann prise en (4,2) possède 19729 décimales. Il s'agit là bien entendu d'une valeur théorique, le calcul associé étant purement impossible dans un temps raisonnable.

L'ensemble $\mathcal{B} \cup \mathcal{L} = \mathcal{L}$ est bien traité dans la fonction (on montrera qu'il termine directement dans le cas général).

Chaque calcul de $f(m, n)$ fait appel à un nombre fini d'opérations, ainsi qu'à 3 appels de f , avec les arguments : $(m - 1, 1)$, $(m, n - 1)$ et $(m - 1, p)$, où p est un entier inconnu, dénotant le fait que l'on ne connaît pas a priori la valeur de la fonction imbriquée. Cependant, le choix de l'ordre lexicographique nous assure les trois inégalités strictes suivantes :

$$\begin{aligned}\varphi(m - 1, 1) &= (m - 1, 1) < \varphi(m, n) = (m, n) \\ \varphi(m, n - 1) &= (m, n - 1) < \varphi(m, n) = (m, n) \\ \varphi(m - 1, p) &= (m - 1, p) < \varphi(m, n) = (m, n) \quad \forall p \in \mathbb{N}\end{aligned}$$

La fonction de Ackermann termine.

Exemple 5 : les tours de Hanoï

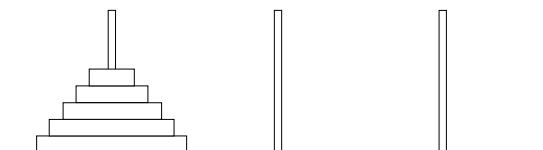
Il s'agit d'un problème classique consistant à déplacer une pile de disques de diamètres différents d'un support à l'autre.

On considère trois supports de même taille en forme de tiges verticales, et disposés de gauche à droite en face du joueur. A l'étape initiale, le support de gauche contient un certain nombre de disques (par la suite nous noterons ce nombre n), empilés en partant du plus large (à la base) jusqu'au plus étroit (au sommet). En général, les disques sont percés en leur centre d'un trou dont le diamètre est égal à celui des supports, de sorte à maintenir la pile en place. L'objectif du jeu est de déplacer (avec un nombre minimal de coups) la pile d'un support à un autre en respectant les règles suivantes :

Règle n°1 : A chaque mouvement, un et un seul disque est déplacé.

Règle n°2 : A chaque étape, un disque doit toujours être empilé sur la base d'un support, ou bien sur un disque dont le diamètre est supérieur.

Nous supposerons que les diamètres des disques sont strictement différents et que la taille de chacun des trois supports est suffisante pour contenir les n disques.



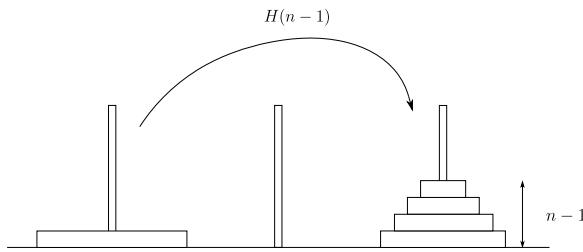
Nous modéliserons le problème en dénotant les supports de 1 à 3. Chaque mouvement sera quant à lui symbolisé par un entier, codé sur 2 chiffres, dont le premier désigne le support de départ et le second désigne le support d'arrivée. Par exemple, dans l'image précédente,

le code 13 désigne le transfert du disque du sommet de la pile 1, vers la base de la pile 3.

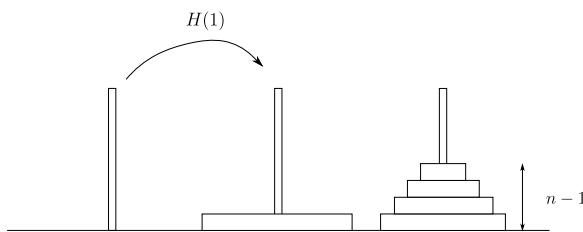
Résoudre le problème signifie donc trouver la suite de codes permettant de faire passer la pile de disques du premier support au second support (notons que le problème est symétrique avec le support d'arrivée, tant que celui-ci est différent du support de départ). La fonction à écrire prendra donc en entrée un nombre n de disques non nul, ainsi que le code du déplacement.

La stratégie de résolution consiste à analyser le problème de manière récursive. Supposons que l'on soit capable de déplacer $n - 1$ disques d'un support vers un autre (et notons $H(n - 1)$ la procédure à suivre). Il est alors possible de modéliser la marche à suivre $H(n)$ avec n disques, uniquement en fonction de $H(n - 1)$ et d'un mouvement élémentaire, que l'on pourra noter $H(1)$.

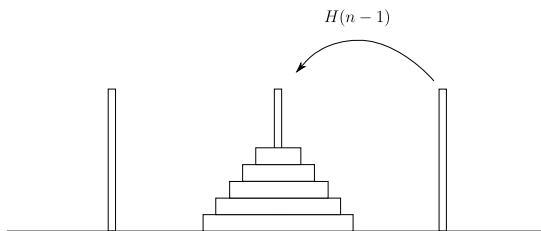
La première étape consiste à déplacer la pile des $n - 1$ disques supérieurs vers le support restant (celui qui n'est ni le support d'arrivée, ni celui de départ).



Dans un second temps, on déplace le disque restant de la première pile vers le support d'arrivée. Cette étape élémentaire correspond à la résolution triviale du problème des tours Hanoï dans le cas où nous ne disposons que d'un seul disque. Il s'agira là du cas de base de notre fonction récursive.



Enfin, on conclut la procédure en replaçant la pile de $n - 1$ disques sur le support d'arrivée, ce qui fait à nouveau appel à la manoeuvre $H(n - 1)$.



La procédure globale au rang n consiste donc à appliquer $H(n-1)$ avec le code 13 (support 1 vers support 3), suivi de $H(1)$ avec le code 12, et enfin $H(n-1)$ à nouveau, mais cette fois avec le code 32. Ces trois étapes permettent d'obtenir $H(n)$ avec le code 12.

Ecrivons la fonction récursive correspondante en ADL, en considérant le code de déplacement comme une chaîne de caractères de longueur fixe égale à 2. La fonction devra retourner une chaîne de caractères constituées de la séquence des codes des mouvements à effectuer. Deux mouvements consécutifs seront séparés par le caractère spécial "-". On utilisera l'opérateur & de concaténation des chaînes de caractères (cf tableau 1.3).

HANOI($n \downarrow$, code \downarrow) : $\llbracket d \leftarrow \text{SUBSTR}(\text{code}, 1, 1); \textcircled{C} \text{ départ } \textcircled{C} a \leftarrow \text{SUBSTR}(\text{code}, 2, 1); \textcircled{C} \text{ arrivée } \textcircled{C} \text{ sep} \leftarrow " - "; \textcircled{C} \text{ séparateur dans le résultat } \textcircled{C} n = 0 ? \text{ HANOI} \leftarrow ""; ! | _ n = 1 ? \text{ HANOI} \leftarrow \text{code}; ! | _ d = "1" ? a = "2" ? \text{temp} \leftarrow "3" | \text{temp} \leftarrow "2" | _ d = "2" ? a = "1" ? \text{temp} \leftarrow "3" | \text{temp} \leftarrow "1" | _ d = "3" ? a = "2" ? \text{temp} \leftarrow "1" | \text{temp} \leftarrow "2" | _ \text{code1} \leftarrow d \& \text{temp}; \text{code2} \leftarrow \text{temp} \& a; \textcircled{C} \text{ si } n \geq 2 \text{ alors } \textcircled{C} \text{ HANOI} \leftarrow \textbf{HANOI}(n-1, \text{code1}) \& \text{sep} \& \textbf{HANOI}(1, \text{code}) \& \text{sep} \& \textbf{HANOI}(n-1, \text{code2}) \rrbracket$

Squelette de l'algorithme :

? ! | _ ? ! | _ ? ? | _ | _ ? ? | _ | _ ? ? | _ | _ **HANOI HANOI HANOI**

Degré d'emboîtement :

$$d_E = 2$$

Notons qu'il est possible de simplifier substantiellement le code à l'aide de l'observation suivante : si d désigne l'indice du support de départ, et a celui du support d'arrivée, on peut calculer algèbriquement (*i.e.* sans instruction de contrôle) l'indice du support temporaire : $\text{temp} = 6 - d - a$ (moyennant une conversion de d et a sous forme numérique).

Montrons que la fonction HANOI termine.

La fonction prend ses arguments dans l'ensemble $\mathcal{D} = \mathbb{N} \times \{1, 2, 3\}^2$. Considérons l'ensemble bien fondé \mathbb{N} (muni de l'inégalité classique) et la fonction $\varphi : \mathcal{D} \rightarrow \mathbb{N}$ qui à une donnée x de \mathcal{D} associe $\varphi(x) = n$, le nombre de disques à traiter.

L'ensemble des cas de bases est $\mathcal{B} = \varphi^{-1}(\{0\}) = \{(0, d, a) \mid (d, a) \in \{1, 2, 3\}^2\}$.

La formule de récursivité ne s'applique plus dès lors que $n - 1 < 0$, i.e. si $n < 1$. L'ensemble des cas limites \mathcal{L} est donc égal à l'ensemble des cas de bases.

L'algorithme termine pour tous les éléments de $\mathcal{B} \cup \mathcal{L} = \mathcal{B}$ (et ressort une chaîne vide).

Enfin, le calcul de $H(x)$ nécessite un nombre fini d'opérations, ainsi que 3 appels (nombre fini également) à $H(y)$, où y est un argument dont la première valeur vaut $n - 1$ ou 1, c'est-à-dire que $\varphi(y) \leq n - 1 < n = \varphi(x)$.

Ceci montre donc que la fonction H termine.

Nous reviendrons sur ce point dans la partie traitant de la programmation dynamique, mais il apparaît d'ores et déjà que ce style de récursivité n'est guère efficace. En effet, dans le cas de la fonction de résolution du problème des tours de Hanoï, nous voyons qu'à chaque itération, la solution à l'itération précédente est calculée deux fois. Certes chacun de ces deux appels fait intervenir un code de déplacement différent, mais répond à la même logique et aurait donc pu être factorisé au sein d'un seul appel. Nous verrons dans la partie suivante que la complexité de cet algorithme est exponentielle et est donc égal au nombre d'étapes physiques à effectuer pour déplacer la pile. Nous verrons comment réduire significativement cette complexité.

2.2.4 Exercices

Exercice 2.3. Factorielle d'un entier *

Démontrer que la fonction factorielle (dans ses 2 versions, itérative et récursive) termine.

Exercice 2.4. Tri par sélection *

Démontrer que la fonction de tri par sélection, écrite au chapitre précédent, termine.

Exercice 2.5. Tri fusion *

Démontrer que la fonction de tri fusion, écrite au chapitre précédent, termine. On démontrera également que la version récursive de l'algorithme de fusion termine.

Exercice 2.6. Calcul des plus proches voisins **

Démontrer que la fonction de calcul du couple de points les plus proches dans un semis de points du plan (dans sa version récursive, ex 1.22), termine.

Exercice 2.7. Nombre de partitions d'un ensemble **

Ecrire une fonction récursive, calculant le nombre de parties d'un ensemble à n éléments. On écrira le squelette de l'algorithme et on indiquera son degré d'emboîtement. Prouver que le programme termine.

Exercice 2.8. *Plus grand diviseur commun* ★★

Ecrire une fonction récursive, calculant le pgcd de deux entiers naturels a et b . On écrira le squelette de l'algorithme et on indiquera son degré d'emboîtement. Prouver que le programme termine.

Exercice 2.9. *Fonction de Morris* ★★

On considère la fonction de Morris, prenant en argument 2 entiers naturels et définie par :

$$\mu(m, n) = \begin{cases} 1 & \text{si } m = 0 \\ \mu(m - 1, \mu(m, n)) & \text{sinon} \end{cases}$$

Cette fonction termine-t-elle ?

Exercice 2.10. *Fonction 91 de McCarthy* ★★★

On considère la fonction 91 de McCarthy, prenant en argument un entier naturel et définie par :

$$f(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f(f(n + 11)) & \text{sinon} \end{cases}$$

Démontrer que f termine. Quelle valeur retourne-t-elle ? On pourra s'inspirer du nom de la fonction.

Exercice 2.11. *Fonction de Takeuchi* ★★★

On considère la fonction de Takeuchi, prenant en argument 3 réels et définie par :

$$\tau(x, y, z) = \begin{cases} \tau(\tau(x - 1, y, z), \tau(y - 1, z, x), \tau(z - 1, x, y)) & \text{si } y > x \\ y & \text{sinon} \end{cases}$$

Démontrer que τ termine.

2.3 Preuve de correction

La preuve de correction consiste à démontrer que, pour toutes les entrée valides, le programme retourne la valeur attendue.

Notons que suivant une autre terminologie, la correction d'un programme désigne l'ensemble composé de la preuve de terminaison, ainsi que d'une preuve appelée correction partielle, consistant à démontrer que lorsque le programme s'arrête, il donne le résultat attendu. Dans ce cours, nous nous référerons à la première définition énoncée.

2.3.1 Programme simple

Par programme simple, nous entendons ici un programme sans boucle, ni récursivité.

La preuve s'effectue alors classiquement par calcul littéral (en prenant toutefois en compte les débranchements).

Exemple : valeur absolue

Démontrer que la fonction suivante calcule la valeur absolue du nombre qui lui est passé en entrée.

MODULE($a \downarrow$) : $\llbracket b \leftarrow (a + 1) * (a + 1); c \leftarrow (b - 2 * a); c \leftarrow c * c; c \leftarrow c - 2 * b + 4 * a + 1; \text{MODULE} \leftarrow \text{SQRT}(\text{SQRT}(c)) \rrbracket$

2.3.2 Programme avec boucles

Pour prouver qu'un algorithme à boucles termine, nous avons utilisé des suites numériques, que l'on appelait *variants de boucle*. Ici, le principe général est le même, et nous auront recours à des quantités appelées *invariants de boucle*, que nous utiliserons suivant une démarche similaire au raisonnement par récurrence.

Définition : invariant de boucle

On appelle invariant de boucle une propriété P qui satisfait les trois contraintes suivantes :

- **Initialisation :** la propriété est vraie à l'initialisation de l'algorithme : P_0 est vrai.
- **Transmission :** si la propriété est vraie à une itération donnée, alors on peut montrer qu'elle le reste à l'itération suivante : $\forall n \in \mathbb{N} \ P_n \Rightarrow P_{n+1}$
- **Terminaison :** lors de la terminaison de l'algorithme, la propriété P (éventuellement couplée à la condition de sortie de boucle) implique que la fonction retourne le résultat attendu.

Exemple 1 : exponentiation

Considérons l'algorithme suivant :

$$\text{EXPO}(a \downarrow, k \downarrow) : \left[\left[p \leftarrow 1; \left\{ \begin{array}{l} /k>0 \\ p \leftarrow p * a; k \leftarrow k - 1 \end{array} \right\} \text{EXPO} \leftarrow p \end{array} \right] \right]$$

La preuve de terminaison est immédiate, en considérant le variant de boucle k , strictement décroissant à chaque passage dans la boucle.

Pour la preuve de correction, nous utilisons l'invariant de boucle constitué par la propriété :

$$P_n : \text{après } n \text{ passages dans la boucle, la variable } p \text{ vaut } a^n$$

La propriété P_0 est vraie, puisque avant d'entrer dans la boucle, la variable p contient la valeur $1 = a^0$. D'autre part, supposons que P_n soit vérifiée pour $n \in \mathbb{N}$. Alors, lors de la $(n+1)$ -eme entrée dans la boucle $p = a^n$. Comme p est multiplié par a dans la boucle, il vaut a^{n+1} en sortie de boucle, et P_{n+1} est vérifiée. Enfin, on sort de la boucle lorsque k est nul, c'est-à-dire après k passages dans la boucle. P_k est donc vérifiée en sortie de module, ce qui démontre la correction de l'algorithme.

Exemple 2 : plus grand commun diviseur

On considère la transcription ADL de l'algorithme d'Euclide :

$$\text{PGCD}(a \downarrow, b \downarrow) : \left[\left[\left\{ \begin{array}{l} /b \neq 0 \\ g \leftarrow b; b \leftarrow \text{MOD}(a, b); a \leftarrow g \end{array} \right\} \text{PGCD} \leftarrow a \end{array} \right] \right]$$

Nous avons démontré dans la partie précédente que cet algorithme termine. Démontrons à présent qu'il retourne le bon résultat. Pour ce faire, on exhibe un invariant de boucle (en notant a_n et b_n la suite des valeurs prise respectivement par les identificateurs a et b après n passages dans la boucle) :

$$P_n : \text{le pgcd du couple } (a_n, b_n) \text{ est égal à celui du couple } (a_0, b_0)$$

De manière évidente, P_0 est vérifiée.

Supposons que pour un entier $n \in \mathbb{N}$ donné, la proposition P_n soit vraie, c'est-à-dire que $\pi(a_n, b_n) = \pi(a_0, b_0)$. En considérant le corps de la boucle, nous avons immédiatement que :

$$a_{n+1} = b_n \quad \text{et} \quad b_{n+1} = a_n - q_n b_n$$

où q_n est un entier numérique.

Alors : $\pi(a_{n+1}, b_{n+1}) = \pi(b_n, a_n - q_n b_n)$

Soit d un diviseur du couple (a_n, b_n) . Alors, comme d divise a_n et b_n , il divise également la différence $a_n - q_n b_n$, et il est donc également un diviseur du couple $(b_n, a_n - q_n b_n)$. Le même raisonnement tient dans le sens inverse et les couples (a_n, b_n) et (a_{n+1}, b_{n+1}) partagent donc le même ensemble de diviseurs, et donc nécessairement, ils ont même plus grand diviseur commun. Par conséquent, nous avons montré que :

$$\pi(a_{n+1}, b_{n+1}) = \pi(a_n, b_n) = \pi(a_0, b_0)$$

La transmission de la propriété P est bien vérifiée.

Enfin, la boucle termine lorsque b_n est nul, le programme retourne a_n qui est bien le PGCD du couple $(a_n, 0)$. Le résultat retourné est donc égal au résultat attendu, et ceci achève la preuve de correction.

Remarque : en toute rigueur, il faudrait aussi démontrer que les variables a_n et b_n restent positives lors de l'exécution du programme.

2.3.3 Programme récursif

Soit f une fonction définie par récurrence, prenant en entrée un argument $x \in \mathcal{D}$. Pour démontrer que f est correcte, il faut procéder par récurrence forte :

- Démontrer que f est correcte pour tous les éléments $x \in \mathcal{B} \cup \mathcal{L}$ (ensemble des éléments du cas de base et des cas limites).
- Supposer que f retourne le bon résultat pour tout x' tel que $\varphi(x') \prec \varphi(x)$ et démontrer qu'en conséquence, f est correcte pour x

Remarque : dans la plupart des cas, une récurrence simple ou double suffit. La récurrence forte énoncée ci-dessus permet de traiter le cas général.

Exemple : considérons la fonction récursive permettant de calculer la factorielle :

FACTORIAL($N \downarrow$) : $\left[\begin{array}{l} N = 0 ? \text{ FACTORIAL} \leftarrow 1; ! \\ | \downarrow \text{ FACTORIAL} \leftarrow N * \\ \text{FACTORIAL}(N - 1) \end{array} \right]$

L'ensemble des cas de base est le singleton : $\{0\}$, traité par la fonction : $f(0) = 1 = 0!$.

Supposons que f retourne le bon résultat pour un entier n donné. Alors : $f(n) = n!$. D'après la formule de récursivité : $f(n + 1)$ se calcule par $(n + 1)f(n) = (n + 1)n! = (n + 1)!$. La fonction est donc correcte pour tout entier n .

2.3.4 Exercices

Exercice 2.12. Recherche de l'élément maximal *

Démontrer que l'algorithme suivant retourne bien l'indice de la valeur maximale du tableau qui lui est passé en entré :

$$\text{ARGMAX}(T \downarrow) : \left[\begin{array}{l} N \leftarrow \text{SIZE}(T); M \leftarrow T_1; I \leftarrow 1; \\ \left\{ \begin{array}{l} \forall i:2^N, T_i < M \rightarrow |_i M \leftarrow T_i; I \leftarrow i \\ \text{ARGMAX} \leftarrow I \end{array} \right. \end{array} \right]$$

Exercice 2.13. Tri de liste **

Démontrer que l'algorithme de tri par sélection (exposé au chapitre 1) est correct.

Exercice 2.14. Décomposition d'un nombre **

Ecrire un algorithme de décomposition des décimales d'un nombre. Par exemple, l'entrée 986 devra retourner le tableau : $T = [9, 8, 6]$. On donnera le squelette et le degré d'emboîtement de l'algorithme, et on démontrera qu'il termine et qu'il retourne le résultat attendu.

Exercice 2.15. Coefficient binomial **

On donne l'algorithme calculant le coefficient binomial C_n^p :

$$\text{COEFF_BINOM}(N \downarrow, P \downarrow) : \left[\begin{array}{l} P \geq N \cup P = 0 ? \text{COEFF_BINOM} \leftarrow 1; ! \\ |_i \text{COEFF_BINOM} \leftarrow \text{COEFF_BINOM}(N - 1, P) + \text{COEFF_BINOM}(N - 1, P - 1) \end{array} \right]$$

Démontrer que la fonction termine et qu'elle retourne le bon résultat.

Exercice 2.16. Division euclidienne **

Ecrire l'algorithme permettant de trouver le quotient et le reste de la division euclidienne de deux entiers. Démontrer que la fonction termine et qu'elle retourne le bon résultat.

Exercice 2.17. Tri de liste récursif **

Démontrer que l'algorithme de tri fusion (exposé au chapitre 1) est correct.

Chapitre 3

Complexité algorithmique

Sommaire

3.1	Exemple d'introduction	90
3.2	Définitions	93
3.3	Unités	94
3.3.1	Taille des données	95
3.3.2	Coût d'exécution	96
3.4	Différentes complexités	97
3.5	Complexité multivariée	100
3.6	Notations de Landau	100
3.7	Classes de complexité	103
3.8	Théorème d'encadrement	106
3.9	Méthodologie	107
3.10	Cas des fonctions récursives	110
3.10.1	Récurrence linéaire simple	110
3.10.2	Récurrence linéaire multiple	113
3.10.3	Diviser pour régner	116
3.11	Pour aller plus loin...	129
3.12	Conclusions	129
3.13	Exercices	130

Introduction

Dans ce chapitre, nous nous intéressons à la troisième et dernière des questions fondamentales que l'on doit se poser après avoir conçu un nouvel algorithme : *est-il efficace ?*

Notons que la réponse à cette question n'est pas unique, et qu'il est nécessaire de préciser les critères sur lesquels on mesure l'efficacité d'un algorithme. S'agit-il de sa rapidité de calcul ? Ou bien de l'espace mémoire requis pour le faire tourner ? Souhaite-t-on connaître le comportement de l'algorithme dans le cas le plus défavorable, ou ses performances en moyenne sur le type de données qu'il sera amené à traiter en contexte opérationnel ?

Dans cette partie, il ne s'agira bien entendu pas de quantifier précisément le temps de calcul et l'espace mémoire nécessaires pour le déroulement d'un programme, mais de trouver de manière rigoureuse, des ordres de grandeurs asymptotiques, afin de répartir les algorithmes

en différentes classes d'efficacité.

3.1 Exemple d'introduction

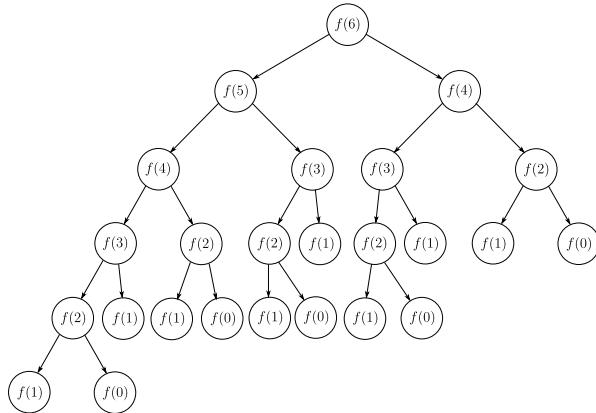
Dans cet exemple introductif, nous considérons la suite de Fibonacci $(f_n)_{n \in \mathbb{N}}$, définie par récurrence par :

$$f_0 = f_1 = 1 \quad \forall n \in \mathbb{N} \quad f_{n+2} = f_{n+1} + f_n$$

La solution naïve consiste à implémenter l'algorithme sous forme récursive, ce qui donne :

$$\text{FIBO}(N \downarrow) : \left[\begin{array}{l} N < 2 ? \text{ FIBO} \leftarrow 1; ! \\ \text{FIBO} \leftarrow \text{FIBO}(N - 1) + \text{FIBO}(N - 2) \end{array} \right]$$

Cet algorithme ne contient que des additions (hors appels récursifs). Nous nous intéressons alors au nombre d'additions à effectuer en fonction du nombre n passé en entrée de l'algorithme. Représentons l'arbre des appels à la fonction FIBO, pour $N = 6$ par exemple :



Dans ce graphe, chaque noeud implique une addition (permettant de réunir le résultat de chacun des deux appels récursifs du niveau inférieur). Chaque étage contient deux fois plus de noeuds que l'étage supérieur. Enfin, pour un entier n placé dans le noeud du sommet, la profondeur de l'arbre (c'est-à-dire son nombre d'étages) est égal à n (notons toutefois que les trois derniers étages ne sont pas complets dans le sens où ils ne contiennent pas 2 fois plus d'addition que l'étage supérieur). Le nombre d'étages complets de l'arbre est donc de $\lfloor n/2 \rfloor + 1$ (on peut s'en convaincre facilement en observant la branche la plus à droite de l'arbre, dans laquelle l'argument de l'appel récursif est décrémenté de 2 unités à chaque niveau).

En somme, le calcul du rang n de la suite de Fibonacci par l'algorithme énoncé ci-dessus, nécessite un nombre d'additions de l'ordre de :

$$T(n) = \sum_{i=0}^{\lfloor n/2 \rfloor} 2^i = 2^{\lfloor n/2 \rfloor + 1} - 1 \simeq 2 \times 1.414^n$$

Nous raffinerons cet ordre de grandeur plus loin dans ce cours.

Nous pouvons d'ores et déjà remarquer que cet algorithme est redondant, et que les mêmes termes apparaissent plusieurs fois dans la branche de calcul (à titre d'exemple, l'évaluation de f_2 est requise 5 fois pour le calcul du terme de rang 6). Il en résulte un coût élevé en terme de temps de calcul lorsque n devient grand. Le nombre d'opérations à effectuer est multiplié par 1.4 à chaque nouvel unité ajoutée à n . A titre d'exemple, le calcul du terme de rang 100 nécessite de l'ordre de 2 millions de milliards d'additions, soit avec un processeur classique dont la puissance de calcul équivaut à quelques GFLOPS¹, un temps de calcul proche de 2 semaines.

Nous verrons une manière générique de simplifier la branche de calcul dans le chapitre traitant de la programmation dynamique, permettant d'atteindre un nombre d'opérations purement linéaire en fonction de la taille de l'entrée.

Ici, nous considérons une technique spécifique au cas du calcul des suites récurrentes d'ordre multiple, permettant d'obtenir un nombre d'opérations encore plus réduit. L'astuce consiste à se ramener à une équation récurrente d'ordre 1 à l'aide d'une multiplication matricielle. En effet, on a :

$$\begin{pmatrix} f_{n+2} \\ f_{n+1} \end{pmatrix} = A \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} \quad \text{en posant :} \quad A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

En considérant les cas de base, on obtient une expression explicite du terme de rang n :

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = A^n \begin{pmatrix} f_1 \\ f_0 \end{pmatrix}$$

Chaque multiplication de A avec elle-même nécessite 8 multiplications et 4 additions. Le calcul du terme de rang n de la suite de Fibonacci peut donc être réalisé avec de l'ordre de $8n$ multiplications et $4n$ additions. En supposant qu'une multiplication nécessite au maximum 256 additions², le temps de calcul requis pour le terme de rang 100 est à présent réduit à environ 200 000 opérations soit 0.1 ms.

Mais il est possible de faire encore beaucoup plus rapide, en ayant recours à l'algorithme exponentiation rapide (déjà abordé dans le chapitre 1). Ecrivons ci-dessous la version matricielle de cet algorithme :

1. Unité de mesure du nombre d'opérations à virgule flottante par seconde
 2. Ce qui constitue une bonne approximation pour des nombres flottants en double précision

MULT_MAT($A \downarrow, B \downarrow$) : $\llbracket \text{TA} \leftarrow \text{SIZE}(A); \text{TB} \leftarrow \text{SIZE}(B); \text{AM} \leftarrow \text{TA}_1; \text{AN} \leftarrow \text{TA}_2; \text{BM} \leftarrow \text{TB}_1; \text{BN} \leftarrow \text{TB}_2; \text{AN} \neq \text{BM} ? \quad \textcircled{C} \text{ les tailles des matrices en entrée ne sont pas cohérentes pour une multiplication} \quad \textcircled{C} \text{ } ! | i \left\{ \begin{array}{l} \text{AM} \\ \text{BN} \end{array} \right\}_{i:1} C_{ij} \leftarrow 0; \left\{ \begin{array}{l} \text{AN} \\ \text{C}_{ij} \end{array} \right\}_{k:1} C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj} \left\{ \begin{array}{l} \text{BN} \\ \text{AN} \end{array} \right\}_j \text{ MULT_MAT} \leftarrow C \rrbracket$

FAST_EXP($A \downarrow$) : $\llbracket \text{TA} \leftarrow \text{SIZE}(A); \text{AM} \leftarrow \text{TA}_1; N = 0 ? \left\{ \begin{array}{l} \text{AM} \\ \text{AM} \end{array} \right\}_{i:1} i = j ? C_{ij} \leftarrow 1 | C_{ij} \leftarrow 0 \left\{ \begin{array}{l} \text{AM} \\ \text{AM} \end{array} \right\}_j \text{ FAST_EXP} \leftarrow C; ! | i \text{ MOD}(N, 2) = 0 ? M \leftarrow \text{FAST_EXP}(A, N/2); \text{FAST_EXP} \leftarrow \text{MULT_MAT}(M, M) | M \leftarrow \text{FAST_EXP}(A, N \div 2); \text{FAST_EXP} \leftarrow \text{MULT_MAT}(A, \text{MULT_MAT}(M, M)) \left\{ \begin{array}{l} \text{AM} \\ \text{AM} \end{array} \right\}_i \text{ FAST_EXP} \leftarrow C \rrbracket$

Remarque : nous supposons ici que la matrice A est nécessairement carrée.

La multiplication matricielle opérant sur une matrice de taille constante (2×2), elle nécessite (comme énoncé précédemment) 8 multiplications et 4 additions, soit un équivalent de 2052 additions.

Le module d'exponentiation rapide nécessite quant à lui au plus $2 \log_2 n$ multiplications matricielles (chaque appel récursif est associé à une multiplication, et on peut démontrer facilement que la valeur de l'argument est au moins divisé par 2 toutes les 2 itérations). On obtient alors un coût total de l'ordre de $4000 \log_2 n$ additions pour évaluer le terme de rang n de la suite de Fibonacci avec cette version optimisée de l'algorithme. Le terme de rang $n = 100$ peut alors être obtenu en $13 \mu s$, c'est-à-dire 75 milliards de fois plus rapidement qu'avec la version naïve de l'algorithme.

Algorithme	Nombre d'additions	Temps de calcul ³
Fonction récursive	2×1.414^n	11 jours
Calcul matriciel simple	$2000n$	0.1 ms
Exponentiation rapide	$4000 \log_2 n$	$13 \mu s$

TABLE 3.1 – Comparaison des algorithmes de calcul de la suite de Fibonacci

Notons que si la différence entre les deux dernières versions ne paraît pas significative au premier regard, le gain de temps est sans appel pour n beaucoup plus grand. Par exemple, avec $n = 10^9$, l'algorithme linéaire tourne en une quinzaine de minutes, tandis que la version logarithmique reste aux alentours d'une trentaine de micro-secondes.

Nous avons donc ici trois algorithmes qui permettent de résoudre le même problème, mais dont l'un est *sub-linéaire*, un autre est *linéaire* tandis que le dernier est dit *supra-linéaire*. Nous verrons dans la dernière partie de ce chapitre les différences en terme de stratégie impliquées par l'utilisation de telle ou telle classe d'algorithme.

3. pour $n = 100$ sur un processeur de 2 GFLOPS

Au travers de cet exemple introductif, nous mesurons bien l'importance d'être capable de quantifier la charge de travail des algorithmes en fonction de la taille des problèmes qui leur sont passés en entrée. C'est ce que nous appelons l'étude de la complexité algorithmique.

3.2 Définitions

Dans cette partie, nous donnons une intuition (qui sera formalisée par la suite) de la définition de la complexité algorithmique.

L'analyse de la complexité algorithmique a pour objectif d'étudier le comportement des algorithmes en terme de ressources nécessaires, lorsque la taille du problème à traiter devient considérablement grande. Il s'agit donc de déterminer son comportement asymptotique, afin de répartir les algorithmes en différentes classes en fonction de leur efficacité.

En pratique, l'étude de la complexité vise à répondre à l'une des deux questions suivantes :

- Quelle est la vitesse d'exécution de l'algorithme ?
- Quelle est la quantité de mémoire nécessaire à l'exécution de l'algorithme ?

Ceci implique donc la définition de deux types de complexité :

La complexité temporelle, on parle aussi de *complexité en temps*, quantifie le nombre d'opérations élémentaires à effectuer pour obtenir le résultat souhaité.

La complexité spatiale, ou *complexité en mémoire*, quantifie le nombre de cases mémoire que la machine doit mettre à disposition du programme pour son exécution.

Considérons deux tableaux de nombres A et B de même taille, que l'on souhaite échanger (c'est-à-dire que l'on veut écrire un algorithme permettant de mettre tous les éléments de A dans B et inversement (cet algorithme ne présente aucun intérêt en pratique)).

ECHANGE($A \uparrow, B \uparrow$) : $\llbracket G \leftarrow A; A \leftarrow B; B \leftarrow G \rrbracket$

Dans cet algorithme, on crée un tableau de stockage temporaire G , permettant de réaliser la permutation. La complexité en temps (en considérant le nombre d'affectations), est n puisque l'algorithme réalise n affectations. La complexité en mémoire est n , puisque le programme doit avoir à disposition au minimum n cellules de tableau simultanément (en plus des $2n$ cellules contenant les données du problème). Considérons l'algorithme suivant qui réalise la même opération :

ECHANGE($A \uparrow, B \uparrow$) : $\llbracket T \leftarrow \text{SIZE}(A); N \leftarrow T_1; \left\{ \begin{array}{l} \underset{i:1}{\overset{N}{g \leftarrow A_i}}; \\ A_i \leftarrow B_i; \\ B_i \leftarrow g \end{array} \right\}_i \rrbracket$

La complexité en temps reste la même, mais nous avons réduit la complexité en mémoire à 1 puisque seule une variable de stockage temporaire est nécessaire.

Complexités temporelles et spatiales seront donc traitées comme des fonctions, prenant en entrée la taille du problème à résoudre par l'algorithme, et donnant en retour le nombre d'opérations (ou de cases mémoires) nécessaires. Il s'agit donc d'une fonction $C : \mathbb{N} \rightarrow \mathbb{N}$. Il nous reste cependant à préciser comment mesurer la taille d'une donnée passée en argument, et comment définir les unités de mesure de la performance de l'algorithme. C'est ce que nous allons introduire dans les deux prochaines sections.

Contentons-nous ici de remarquer que si l'objectif est de caractériser la fonction de complexité C , il est complètement illusoire de penser que nous serons systématiquement capables de dériver une expression analytique du temps de calcul en secondes (ou l'espace mémoire en Mo) nécessaire. Cela n'aurait d'ailleurs que peu d'intérêt en pratique, puisque la réponse est fortement dépendante de la machine et du langage d'implémentation utilisés. L'objectif sera simplement de caractériser le comportement asymptotique de l'algorithme. Sa croissance est-elle linéaire ? Polynomiale ? Exponentielle ? Cela peut sembler décevant au premier regard, mais comme nous le verrons par la suite, c'est en pratique largement suffisant.

Notons que la complexité en mémoire revêtait une importance capitale à l'aube des machines informatiques. En 1956 par exemple, 1 Mo de stockage ne coûtait pas moins de 10 000 dollars. Au début des années 90, il s'évaluait toujours aux alentours de 9 dollars. Une réduction de la complexité spatiale des algorithmes se traduisait alors généralement par un gain économique considérable. Cela n'est plus autant vrai aujourd'hui, et explique en partie la perte d'intérêt pour cette complexité, en faveur de la complexité temporelle. C'est aussi la logique que nous suivrons dans ce cours, en ne traitant que de ce second type de complexité, tout en soulignant que l'ensemble des concepts abordés seront en principe directement transposables au domaine de la complexité spatiale.

3.3 Unités

Dans cette section, nous formalisons un peu plus les unités utilisées pour mesurer la taille des données en entrée ainsi que le coût en sortie sur le nombre d'opérations élémentaires à effectuer. Notons qu'il n'y a pas de règle absolue, et que lorsque plusieurs systèmes d'unités différents peuvent être choisis, chacun est libre de choisir celui qui lui paraît le plus approprié, mais un mauvais choix (i.e. un choix peu représentatif du contenu réel des données et du programme) conduira à une interprétation erronée du comportement de l'algorithme. Conséquemment, si le choix n'est pas intuitif, il devra être clairement énoncé et justifié dans l'analyse.

Dans tout ce qui suit, \mathcal{A} désigne l'ensemble des algorithmes.

3.3.1 Taille des données

Soit un algorithme $f \in \mathcal{A}$ prenant ses entrées dans un ensemble \mathcal{D} . Définir la taille des données en entrée signifie plus formellement choisir une fonction de mesure de taille $|\cdot| : \mathcal{D} \rightarrow \mathbb{N}$, qui à une donnée x quelconque, associe sa taille $|x|$ sous forme d'un entier positif.

La taille des données en entrée (on parlera également de taille du problème à résoudre, en considérant x comme une instance du type de problèmes que f est capable de résoudre) doit être choisie de manière pertinente, en accord avec l'algorithme à analyser. Il n'y a généralement pas d'ambiguïté possible.

- Lorsque l'argument est un nombre entier (e.g. factorielle, suite de Fibonacci...), la valeur de cette entier pourra être prise comme unité de taille.
- S'il s'agit d'un réel, l'on pourra prendre la partie entière de ce nombre
- Si plusieurs nombres (entiers ou réels) sont passés en arguments (coefficient binomial, pgcd, exponentiation), il faudra définir un système en sommant les nombres significatifs (bien souvent un seul des nombres passés en entrée est représentatif de la taille du problème à traiter).
- Si l'argument est une liste (e.g. tri de liste, recherche d'éléments...), on prendra généralement le nombre d'éléments de la liste comme unité de taille.
- De la même manière, pour une chaînes de caractères (e.g. recherche de sous-chaine, image miroir...) , sa longueur paraît un bon candidat
- Enfin, si l'entrée est composée de plusieurs listes, il faudra définir un système en sommant les tailles des listes significatives

Notons que si cette liste de cas de figure recouvre la très grande majorité des cas en pratique, pour certains problèmes il peut arriver d'avoir à définir un système d'unité "maison".

Exemples :

Factorielle :

Il n'y a qu'un entier en entrée donc $|n| = n$ paraît être le seul candidat naturel.

Calcul du pgcd de 2 nombres :

Le plus grand des deux entiers : $|(a, b)| = \max(a, b)$ est un bon candidat.

Exponentiation rapide :

Dans le calcul de a^n , la taille de a n'influence pas le nombre d'itérations de l'algorithme. L'unité $|(a, n)| = n$ semble indiquée.

Tri d'une liste :

Le nombre d'éléments de la liste est un bon indicateur de la taille du problème à résoudre.

Tours de Hanoï :

De la même manière, on choisirait le nombre de disques de la pile à déplacer.

Le plus important est de se souvenir que la mesure de taille doit être un **entier naturel, représentatif** de la difficulté du problème et calculable **sans ambiguïté**.

Remarque : lorsque la donnée passée en entrée est un nombre entier, il est d'usage en théorie de la complexité de prendre comme unité de taille la longueur de la décomposition du nombre en système binaire.

3.3.2 Coût d'exécution

Le coût d'exécution est l'unité de la valeur d'arrivée de la complexité.

Tout comme pour l'unité de taille des données passées en argument, il n'existe pas de définition unique du coût, et le soin est laissé au programmeur de définir l'opération élémentaire qui lui paraît prépondérante dans l'exécution de l'algorithme.

L'opération choisie doit être celle qui prend le plus de temps et/ou qui est répétée le plus de fois. Là aussi, nous pouvons remarquer qu'en pratique il n'y a guère d'ambiguïté quant au choix de cette unité.

Exemples :

Factorielle :

Il n'y a que des multiplications dans l'algorithme, on l'utilise alors comme opération élémentaire.

Calcul du pgcd de 2 nombres :

Chaque itération comporte 1 opération d'arithmétique modulaire et 3 affectations. Le choix est laissé libre ici et le résultat n'en sera affecté que d'une constante multiplicative.

Exponentiation rapide :

La multiplication est prépondérante parmi la liste des opérations effectuées.

Tri d'une liste :

En général on choisit le nombre de comparaisons (on peut aussi s'intéresser au nombre d'affectations).

Tours de Hanoï :

Chaque déplacement de disque constitue une opération élémentaire.

De manière similaire au cas du choix de l'unité de taille des données, dès lors qu'un choix sort de l'ordinaire ou qu'aucun candidat ne se dégage naturellement, il sera important de le justifier dans l'analyse.

Concluons cette sections sur une remarque importante : la comparaison de deux algorithmes (résolvant le même problème) n'est possible que si les analyses de leurs complexités respectives, ont été menées avec les même unités de taille et de coût.

3.4 Différentes complexités

Dans les sections précédentes, nous avons défini l'étude de la complexité comme l'analyse du nombre d'opérations élémentaires à réaliser en fonction de la taille n du problème à traiter, ce que nous avons modélisé mathématiquement à l'aide d'une fonction $C : n \rightarrow C(n)$, dont on souhaite connaître le comportement asymptotique.

Cependant, cette définition est ambiguë, car rien ne permet d'affirmer que 2 données différentes de même taille n entraînent nécessairement le même nombre d'opérations. Cette affirmation est d'ailleurs généralement fausse. Par exemple, dans le cas d'un tri de liste rudimentaire, et en prenant le nombre d'affectations comme unité de coût, la complexité de l'algorithme sera bien évidemment plus faible pour une liste quasiment triée que pour une liste quelconque (dans l'absolu, si la liste est complètement triée, l'algorithme n'effectue aucune inversion, donc sa complexité est nulle, indépendamment de la taille de la liste). Or, pour un antécédent n donné, la fonction C ne peut associer qu'une seule image $C(n)$. Il est donc nécessaire de spécifier plusieurs types de complexités.

Dans ce qui suit, f désigne un algorithme, prenant ses entrées dans un ensemble de données \mathcal{D} et $\mathcal{D}_n (\subset \mathcal{D})$ désigne l'ensemble des données de taille n . Enfin, la fonction $C : \mathcal{D} \rightarrow \mathbb{N}$ désigne le nombre d'opérations à effectuer pour traiter une donnée $d \in \mathcal{D}$.

Complexité minimale : on parle aussi de *complexité dans le meilleur des cas*.

Il s'agit du nombre minimal d'opérations à effectuer pour traiter une donnée de taille n :

$$C_{min}(n) = \min \{C(d) \mid d \in \mathcal{D}_n\}$$

Cette complexité n'est que peu utilisée en pratique, puisqu'elle ne considère que ce qui se passe dans le meilleur des cas et n'a donc pas trop d'intérêt. Par exemple, comme indiqué précédemment, la complexité minimale d'un algorithme de tri de liste (tri par sélection) est nulle. Il existe cependant des cas où elle peut être intéressante. Par exemple si l'on souhaite trier un ensemble de listes, dont une grande partie sont déjà triées. Il peut alors être avantageux de choisir un algorithme dont la complexité dans le meilleur des cas n'est pas rédhibitoire (en somme, on veut que l'algorithme se comporte très bien sur les cas triviaux, ce qui n'est pas toujours garanti). L'étude de la complexité minimale permet de mettre en évidence (ou bien d'infirmer) les capacités d'un programme à traiter ces cas.

Complexité maximale : on parle aussi de *complexité dans le pire des cas*.

A l'inverse, il s'agit ici du nombre maximal d'opérations à effectuer pour traiter une donnée de taille n :

$$C_{max}(n) = \max \{C(d) \mid d \in \mathcal{D}_n\}$$

Cette complexité est beaucoup plus intéressante, et c'est celle que nous utiliserons le plus fréquemment dans ce cours. Elle fournit en quelques sortes une borne inférieure sur les performances de l'algorithme (on sait que quelle que soit la donnée de taille n passée en entrée, le programme n'effectuera pas plus de $C_{max}(n)$ opérations élémentaires). Pour reboucler sur le chapitre 2, remarquons que si pour tout entier n la valeur de $C_{max}(n)$ est finie, alors l'algorithme termine⁴.

Dans le cas de la complexité spatiale, la complexité dans le pire des cas permet de fournir une borne supérieure sur l'espace mémoire à allouer au programme avant de le lancer.

En général, lorsque l'on ne précise pas à quelle type de complexité on fait référence, il s'agit implicitement de la complexité dans le pire des cas.

Complexité moyenne :

La complexité moyenne est définie comme l'espérance conditionnelle du nombre d'opérations élémentaires à réaliser sachant que la donnée à traiter est de taille n .

$$C_{moy}(n) = \mathbb{E}\left[C(d) \mid d \in \mathcal{D}_n\right] = \sum_{i=1}^{|\mathcal{D}_n|} C(d_i) \pi(d_i)$$

où $|\mathcal{D}_n|$ représente le nombre de données de taille n contenues dans l'ensemble total des données \mathcal{D} et $\pi(d_i)$ représente la probabilité d'occurrence de la donnée $d_i \in \mathcal{D}_n$.

Lorsque toutes les données de taille n sont supposées avoir la même probabilité d'apparition (distribution uniforme sur \mathcal{D}_n), l'expression de la complexité moyenne se réduit en :

$$C_{moy}(n) = \frac{1}{|\mathcal{D}_n|} \sum_{i=1}^{|\mathcal{D}_n|} C(d_i)$$

Des trois types de complexité, il s'agit certainement de la plus intéressante et de la plus informative, mais aussi de par sa nature probabiliste, de la plus difficile à calculer en général. D'autre part, il devient alors nécessaire de définir une distribution de probabilité sur les données en entrée, ce qui n'est pas toujours évident. Par exemple, calculer la complexité moyenne d'un algorithme d'inversion matricielle n'a que peu de sens en pratique, puisque cela supposerait de poser une distribution uniforme⁵ sur les coefficients de la matrice à inverser, ce qui produirait un modèle peu réaliste et donc une complexité moyenne peu représentative des vraies capacités de l'algorithme. En particulier, certains algorithmes d'inversion sont efficaces sur des matrices creuses. Dès lors, comment modéliser de manière probabiliste et réaliste la tendance d'une matrice à être creuse ? En conséquence, le calcul de la complexité moyenne n'est souvent possible que pour des applications bien particulières, où la distribution des données passées en entrée est clairement identifiée.

4. A condition d'avoir choisi des unités de coût et de taille représentatives.

5. Définie sur l'ensemble des nombres représentables en machine par exemple

A défaut de pouvoir calculer simplement la complexité moyenne, nous auront recours à la complexité dans le pire des cas.

Exemple : tri par sélection

Considérons à nouveau l'algorithme de tri par sélection d'une liste de nombres.

$$\text{TRI}(L) : \left[\left[N \leftarrow \text{SIZE}(L); \left\{ \begin{array}{l} \min \leftarrow L_i; id \leftarrow i; \\ \left\{ \begin{array}{l} \forall j \in [i+1, N] : L_j < \min \Rightarrow \min \leftarrow L_j; id \leftarrow j \\ \end{array} \right. \end{array} \right\}_j \right. \right. \right. \\ \left. \left. \left| \begin{array}{l} i = id \Rightarrow G \leftarrow L_{id}; L_{id} \leftarrow L_i; L_i \leftarrow G \end{array} \right\}_i \right] \right]$$

Cherchons à établir l'expression analytique des 3 types de complexité.

Il nous faut d'abord définir une unité de mesure de taille des données : on prend la taille de la liste L . On utilise la comparaison comme opération élémentaire pour le calcul du coût.

L'algorithme possède deux boucles imbriquées sans débranchement, ni modification des valeurs de pas ou de borne. Chaque passage dans la boucle contient une comparaison. La complexité de l'algorithme est donc égal au nombre de passages dans la boucle, soit :

$$C_{min}(n) = C_{moy}(n) = C_{max}(n) = \frac{n(n-1)}{2}$$

Remplaçons maintenant l'unité de coût par le nombre d'échanges à effectuer pour trier la liste. Nous obtenons cette fois plusieurs valeurs différentes. Lorsque la liste est déjà triée, il n'y a aucun échange à effectuer et la complexité est nulle. A l'inverse, dans le pire des cas, la liste est triée dans l'ordre $[2, 3, 4, \dots, n-1, n, 1]$ et chaque itération nécessite un échange, soit une complexité dans le pire des cas égale à n .

Enfin, pour calculer la complexité en moyenne, nous avons besoin de définir en plus une distribution de probabilité sur les listes. On remarque ici que l'algorithme travaille sur des comparaisons et les valeurs absolues des éléments de la liste n'ont pas d'importance en pratique. Tout se passe donc comme si on traitait une liste de taille n contenant une permutation (aléatoire) des entiers $\Omega_n = \{1, 2, \dots, n\}$. On définit alors de manière naturelle la distribution de probabilité comme étant la loi uniforme sur l'ensemble des permutations de Ω_n . Chaque permutation a donc la même probabilité de survenir et, étant donnée qu'il y a $n!$ permutations possibles⁶, la probabilité $\pi(d_i)$ vaut $1/n!$, indépendamment de la nature de la permutation. On a alors :

$$C_{moy}(n) = \frac{1}{n!} \sum_1^{n!} C(d_i)$$

6. On fait ici l'hypothèse que chaque élément est unique dans la liste

Calculons $C(d_i)$ pour $d_i \in \mathcal{D}_n$.

On montre de manière évidente que le nombre moyen d'échanges à réaliser à l'itération i de la boucle externe est : $\frac{n-i}{n-i+1}$. En effet, à l'étape i , on parcourt la liste comprise entre les indices $i+1$ et n à la recherche d'un élément minimal. Il y a échange si et seulement si cet élément est plus petit que l'élément situé en position i . La probabilité que l'élément minimal du sous-tableau commençant en i soit situé entre les indices $i+1$ et n est donc égal au ratio du nombre de cellules entre $i+1$ et n sur le nombre de cellules entre i et n , d'où l'expression de l'espérance du nombre total d'échanges :

$$C(d_i) = \sum_{i=1}^n \frac{n-i}{n-i+1} = \sum_{k=1}^n \frac{k-1}{k} = n - \sum_{k=1}^n \frac{1}{k} \sim n - \log n$$

On obtient alors une complexité moyenne approximativement égale à $n - \log n$ c'est-à-dire linéaire dès lors que n est suffisamment grand.

3.5 Complexité multivariée

Dans les sections précédentes nous avons introduit la complexité comme une fonction définie sur l'espace des entiers naturels. Ceci implique qu'il est possible de caractériser la taille du problème à l'aide d'un unique entier n . Dans certains cas, ceci est impossible, et la complexité du problème est nécessairement multivariée. Par exemple, le problème de la recherche du plus court chemin dans un réseau fait intervenir 2 quantités partiellement indépendantes : le nombre n d'intersections et le nombre m de tronçons du réseau. Ainsi, on peut tout-à-fait imaginer un réseau avec très peu d'intersections et une grande connectivité, i.e. m très grand⁷. À l'inverse, et en particulier pour un réseau non connexe, le nombre d'intersections peut augmenter infiniment et indépendamment du nombre de tronçons.

Ainsi, l'algorithme de Dijkstra, qui calcule le plus court chemin entre 2 points d'un graphe, possède une complexité dans le pire des cas égale à $C(n, m) = m + n \log n$.

3.6 Notations de Landau

Jusqu'ici nous avons essayé d'établir les expressions analytiques exactes de la complexité. En pratique nous nous intéressons uniquement au comportement de la complexité lorsque le nombre n tend vers l'infini, l'objectif *in fine* étant de répartir les algorithmes en différents groupes de complexité afin de les caractériser et de les comparer entre eux (ceci impliquera de hiérarchiser les classes d'une manière ou d'une autre).

La complexité étant une fonction à valeurs entières positives, il a alors été nécessaire d'introduire des classes d'équivalence entre fonctions à valeurs positives. La réponse au problème a été apportée par le mathématicien allemand Edmund Landau, qui contribuera à la diffusion des notations qui portent son nom (en s'inspirant notamment des travaux de

7. Au plus, on pourra avoir $m = n^2$ dans le cas du graphe complet orienté K_n .

Paul Bachmann).

De nos jours, 6 des 9 symboles de la notation de Landau sont couramment utilisés, 2 ou 3 seront suffisants dans le cadre de l'analyse de la complexité algorithmique.

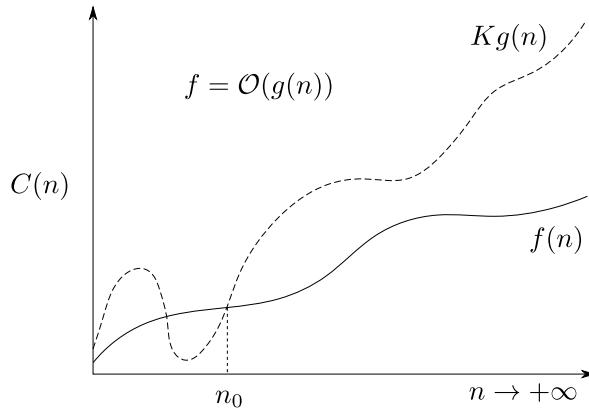
Définition : notation grand \mathcal{O}

Soient f et g deux fonctions de $\mathbb{N} \rightarrow \mathbb{R}^{+*}$.

On dit que f est *dominée par* g , et on écrit $f = \mathcal{O}(g)$ si et seulement si :

$$\exists K \in \mathbb{R}^{+*}, n_0 \in \mathbb{N} \mid n \geq n_0 \Rightarrow f(n) \leq Kg(n)$$

On dit aussi que f est de l'ordre de g .



Remarque 1 : l'écriture $f(n) = \mathcal{O}(g(n))$ est aussi possible.

Remarque 2 : notons que l'expression *est dominée par* peut prêter à confusion dans le sens où f n'est pas nécessairement inférieure à g . Par exemple $2n = \mathcal{O}(n)$.

Remarque 3 : $\mathcal{O}(g)$ peut être considérée comme une classe d'équivalence des fonctions, mais la relation $f = \mathcal{O}(g)$ n'implique pas (en général) que $g = \mathcal{O}(f)$. On pourra s'en convaincre en prenant par exemple les fonction n^2 et n^3 . Montrer que f est de l'ordre de g revient en effet à démontrer que le rapport f/g est borné à partir d'un certain rang.

Remarque 4 : en toute rigueur, la notation $f = \mathcal{O}(g)$ contredit la sémantique du signe égal et on devrait plutôt utiliser : $f \in \mathcal{O}(g)$. On peut également lever la contradiction en définissant $= \mathcal{O}$ comme un symbole à part entière traduisant la domination de f à l'égard de g .

Exercice 3.1. Arithmétique dans \mathcal{O}

Démontrer que, pour une fonction positive f donnée, $\mathcal{O}(f)$ est stable par addition et par multiplication par un scalaire positif, *i.e.* $(\mu, \lambda) \in \mathbb{R}^+, g_1, g_2 \in \mathcal{O}(f) \Rightarrow \lambda g_1 + \mu g_2 \in \mathcal{O}(f)$.

Proposer un contre-exemple permettant d'invalider l'implication pour la soustraction.

Nous avons démontré précédemment que l'algorithme de tri par sélection nécessitait (dans tous les cas) un total de $C(n) = n(n - 1)/2$ opérations. Cette notation nous permettra de rattacher C à la classe d'équivalence de la fonction de référence n^2 et nous noterons : $C(n) = \mathcal{O}(n^2)$. Nous verrons dans la section suivante une liste de fonctions de référence communément utilisées pour bâtir une hiérarchie dans les complexités algorithmiques.

Notons cependant que la notation \mathcal{O} n'est pas très précise. Dans le cas $C(n) = \mathcal{O}(n^2)$, nous aurions également pu écrire sans prendre de risque $C(n) = \mathcal{O}(n^3)$, où même $C(n) = \mathcal{O}(n!)$ ce qui correspond à des classes de complexité très différentes. Ecrire $C(n) = \mathcal{O}(g(n))$ signifie donc que C ne "croît pas plus vite" que g . Il s'agit donc d'une borne supérieure sur le résultat attendu. Voyons comment affiner un résultat avec la notation suivante.

Définition : notation grand Θ

Soient f et g deux fonctions de $\mathbb{N} \rightarrow \mathbb{R}^{+*}$.

On dit que f est *dominée et soumise* à g , et on écrit $f = \Theta(g)$ si et seulement si :

$$f = \mathcal{O}(g) \quad \text{et} \quad g = \mathcal{O}(f)$$

Ou, de manière équivalente, si et seulement si :

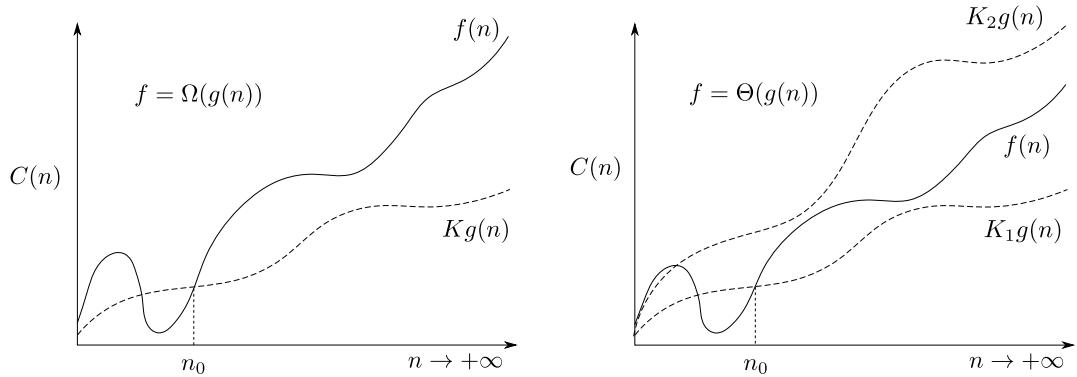
$$\exists K_1, K_2 \in \mathbb{R}^{+*}, n_0 \in \mathbb{N} \mid n \geq n_0 \Rightarrow K_1 g(n) \leq f(n) \leq K_2 g(n)$$

Il s'agit cette fois-ci d'une relation d'encadrement. Si $f(n) = n^2$, alors on peut écrire $f(n) = \mathcal{O}(n^3)$ mais plus $f(n) = \Theta(n^3)$.

Remarque 5 : on utilise aussi la notation Ω pour désigner la contraposée de \mathcal{O} : $f = \Omega(g)$ si et seulement si $g = \mathcal{O}(f)$. Avec cette nouvelle notation, la définition de Θ devient :

$$f = \Theta(g) \iff f = \mathcal{O}(g) \text{ et } f = \Omega(g)$$

La troisième et dernière des notations que nous introduirons pourra permettre de faciliter la manipulation des calculs intermédiaires (cf par exemple exercice 3.3).



Définition : équivalence

Soient f et g deux fonctions de $\mathbb{N} \rightarrow \mathbb{R}^{+*}$.

On dit que f est *équivalente* à g et on note $f \sim g$ si et seulement si :

$$\forall \varepsilon > 0 \quad \exists n_0 \in \mathbb{N} \quad | \quad n \geq n_0 \Rightarrow |f(n) - g(n)| \leq \varepsilon g(n)$$

Exercice 3.2. Démontrer que conformément à son nom, l'équivalence est une relation d'équivalence, c'est-à-dire que c'est une relation binaire réflexive, symétrique et transitive.

Exercice 3.3. Considérons trois fonctions f , g et h de \mathbb{N} dans \mathbb{R}^+ telles que $f \sim g$. Montrer que $f = \Theta(h)$ si et seulement si $g = \Theta(h)$.

3.7 Classes de complexité

Dans cette partie, nous listons (de manière non-exhaustive) un certain nombre de fonctions de référence permettant de comparer et de hiérarchiser les complexités des algorithmes.

Remarque 1 : d'une manière générale, la classe hyperexponentielle regroupe l'ensemble des algorithmes dont la complexité C est telle que le rapport $C(n)/a^n$ tende vers $+\infty$ pour toute valeur de $a \in \mathbb{R}^{+*}$

Remarque 2 : la base du logarithme n'est pas importante. En effet, on a : $\log_p n = \ln n / \ln p$, où $\ln p$ est une constante que l'on peut intégrer dans le \mathcal{O} .

Exercice 3.4. Démontrer qu'un algorithme de complexité factorielle appartient bien à la classe hyperexponentielle. Indication : on pourra utiliser la formule de Stirling.

Une classification plus grossière consiste à comparer le comportement de l'algorithme par rapport à un traitement qui serait proportionnel à la taille des données. Cette approche résulte en trois classes :

Noms	Classes	Exemples d'algorithmes
Constante	1	Accès à un élément dans une liste
Logarithmique	$\log n$	Recherche dichotomique
Linéaire	n	Maximum d'une liste, factorielle
Quasi-linéaire	$n \log n$	Tri fusion, tri rapide
Quadratique	n^2	Tri par sélection
Cubique	n^3	Multiplication matricielle
Polynomiale	$n^k \ k > 1$	Multiplication matricielle
Exponentielle	$a^n \ a > 1$	Tours de Hanoï
Hyperexponentielle	$n! \ n^n$	Problème du voyageur de commerce

TABLE 3.2 – Fonctions de références et classes de complexités associées usuelles

Noms	Classes
Sublinéaire	constante, logarithmique
Linéaire	linéaire, quasi-linéaire
Supralinéaire	polynomiale, (hyper)exponentielle

TABLE 3.3 – Comparaisons des classes de complexité avec un traitement linéaire

Remarque 3 : notons qu'un algorithme quasi-linéaire appartient stricto sensu à la classe supra-linéaire. En pratique, et pour des données de taille suffisante, on ne voit pas de différences significatives entre un algorithme en $n \log n$ et un comportement linéaire.

Remarque 4 : en théorie de la complexité pour les problèmes de décision, la distinction s'opère principalement entre les algorithmes au plus polynomiaux d'une part, et les algorithmes exponentiels (et hyperexponentiels) d'autre part. C'est en particulier l'objet du fameux problème P = NP (voir Garey *et al.*, 1991 par exemple).

A titre de comparaison, on donne ci-dessous un tableau des temps d'exécutions pour différentes complexités (en supposant que le coût évalué est le nombre d'opérations à virgule flottante et que le processeur réalise un milliard d'opérations par seconde).

Classe	$n = 5$	$n = 10$	$n = 100$	$n = 1000$
$\mathcal{O}(\log_2 n)$	2 ns	3 ns	7 ns	10 ns
$\mathcal{O}(n)$	5 ns	10 ns	100 ns	1 μ s
$\mathcal{O}(n \log_2 n)$	10 ns	20 ns	300 ns	7 μ s
$\mathcal{O}(n^2)$	25 ns	100 ns	10 μ s	1 ms
$\mathcal{O}(2^n)$	32 ns	100 μ s	$4 \cdot 10^{13}$ ans	$\sim 10^{300}$ ans
$\mathcal{O}(n!)$	120 ns	4 ms	$\sim 10^{150}$ ans	$\sim 10^{2500}$ ans

TABLE 3.4 – Comparaisons des classes de complexité avec un traitement linéaire

Remarque 5 : la comparaison n'est valide que lorsque n est suffisamment grand. Pour des données de petites tailles, il arrive bien souvent qu'un algorithme en temps polynomial par exemple, soit plus rapide qu'un algorithme quasi-linéaire (en particulier si le second nécessite quantité de pré-traitements sophistiqués, effectués en temps constant certes, mais

qui ne sont rentabilisés que pour des données volumineuses). On parle de *hidden constant factor*. Quel algorithme doit-on choisir si $f(n) = 10^{-9}n^{1.1}$ et $g(n) = 10^9n^{0.9}$ opérations élémentaires ? Combien de données faudrait-il avoir à traiter pour inverser la tendance ?

Remarque 6 : on se gardera également de faire des comparaisons générales entre algorithmes conçus pour des tâches différentes. Pour une multiplication de deux nombres à n décimales, un algorithme en $\mathcal{O}(n^k)$ avec $1 < k < 2$ est considéré comme étant très bon, bien que polynomial, tandis qu'un algorithme linéaire pour la recherche d'un élément maximal dans une liste triée est très médiocre.

Remarque 7 : lorsque l'entrée de l'algorithme est un nombre entier, il est parfois d'usage (en particulier en théorie de la complexité) de prendre comme unité de taille la longueur du nombre dans sa décomposition en bits. Avec cette convention, un temps de calcul linéaire devient exponentiel tandis qu'un temps de calcul logarithmique devient linéaire. Par exemple, le test de primalité d'un entier n par l'algorithme d'Agrawal-Kayal-Saxena s'effectue en un temps $\mathcal{O}((\log n)^{12})$, mais il est d'usage de dire que sa complexité est polynomiale. En effet, en représentation binaire, la longueur du nombre vaut $l = \log_2 n$, ce qui donne une complexité en fonction de l : $C(l) = \mathcal{O}(l^{12})$, soit une complexité polynomiale. Attention donc à bien considérer l'unité de mesure de la taille utilisée avant d'interpréter la complexité indiquée pour un algorithme.

Exemple : évaluation polynomiale

On considère ici un algorithme de calcul de la valeur prise par un polynôme en un point d'abscisse quelconque. On considère un premier algorithme naïf, prenant en entrée un tableau de n coefficients (pour symboliser un polynôme de degré $n - 1$) et un réel X :

$$\text{EVAL}(P, X) : \left[\left[N \leftarrow \text{SIZE}(P); s \leftarrow 0; \left\{ \sum_{i=1}^N s \leftarrow s + P_i * X \wedge (i - 1) \right\}_i \text{EVAL} \leftarrow s \right] \right]$$

Nous supposons ici que les coefficients sont rangés dans l'ordre des monômes de puissances croissantes.

Définition des unités : on mesure la taille du polynôme à son nombre de coefficients et on prend comme opération élémentaire la multiplication.

L'algorithme effectue i multiplications à chaque passage dans la boucle et il y a n itérations de boucle. La complexité de l'algorithme vaut donc : $C(n) = n(n+1)/2 = \mathcal{O}(n^2)$. Il s'agit donc d'une complexité quadratique.

Il est possible de réduire cette complexité en utilisant l'algorithme d'exponentiation rapide (introduit au chapitre 1). Ce dernier calcule une puissance en $2 \log_2 n$ opérations dans le pire des cas, d'où l'expression de la complexité totale :

$$C(n) = \sum_{i=1}^n 1 + 2 \log_2 i = n + 2 \sum_{i=1}^n \log_2 i = n + 2 \log_2 \left(\prod_{i=1}^n i \right) = n + 2 \log_2 n!$$

En utilisant la formule de Stirling, on montre aisément que le second terme de l'expression de $C(n)$ est équivalent à un $\mathcal{O}(n \log n)$. La complexité dans le pire des cas de l'algorithme

d'évaluation polynomiale optimisée avec la méthode d'exponentiation rapide est quasi-linéaire :

$$C_{\max}(n) = \mathcal{O}(n \log n)$$

Est-il possible de faire mieux ?

Considérons l'algorithme ci-dessous :

$$\text{EVAL}(P, X) : \left[\begin{array}{l} N \leftarrow \text{SIZE}(L); s \leftarrow P_N; \left\{ \begin{array}{l} s \leftarrow s * X; \\ s \leftarrow s + P_i \end{array} \right. \right\}_i^1, -1 \text{ EVAL} \leftarrow s \end{array} \right]$$

A l'aide des techniques vues au chapitre précédent, démontrer que la fonction EVAL est correcte (la preuve de terminaison est triviale).

Avec cette version améliorée, chaque itération nécessite exactement une multiplication. La complexité de la fonction (dans tous les cas) est cette fois linéaire :

$$C_{\min}(n) = C_{\text{moy}}(n) = C_{\max}(n) = \mathcal{O}(n)$$

3.8 Théorème d'encadrement

Dans certains cas, le calcul de la complexité moyenne est extrêmement ardu, alors qu'il est relativement simple d'identifier le cas le plus favorable et le plus défavorable.

Si dans le cadre d'une application particulière on cherche exclusivement à connaître la complexité moyenne d'un algorithme, il peut être intéressant tout de même de commencer par calculer les complexités minimale et maximale. En effet, s'il se trouve que ces deux complexités appartiennent à la même classe de complexité, on pourra alors tirer partie du théorème d'encadrement suivant.

Théorème : *encadrement de la complexité moyenne*

Soit f une fonction de $\mathbb{N} \rightarrow \mathbb{R}^{+*}$:

$$C_{\min} \in \Omega(f) \quad \text{et} \quad C_{\max} \in \mathcal{O}(f) \quad \Rightarrow \quad C_{\text{moy}} \in \Theta(f)$$

Preuve : d'après les hypothèses du théorème, il existe deux constantes réelles strictement positives K_1 et K_2 , et deux entiers n_0 et n_1 tels que :

$$\begin{aligned} \forall n \geq n_0 \quad & |C_{\min}(n)| \geq K_1 |f(n)| \\ \forall n \geq n_1 \quad & |C_{\max}(n)| \leq K_2 |f(n)| \end{aligned}$$

D'autre part, en tant que combinaison linéaire convexe de complexités sur les différents cas, on a de manière triviale que : $C_{\min}(n) \leq C_{\text{moy}}(n) \leq C_{\max}(n)$, d'où a fortiori, pour tout $n \geq n_2 = \max(n_0, n_1)$:

$$K_1|f(n)| \leq C_{moy}(n) \leq K_2|f(n)|$$

Et donc par définition : $C_{moy}(n) = \Theta(f(n))$.

3.9 Méthodologie

La stratégie d'évaluation de la complexité pour un programme non-récuratif (nous verrons le cas des programmes récursifs dans la section suivante) se compose comme suit :

- Choix d'une unité de taille pour caractériser la donnée en entrée
- Choix d'une unité de coût de complexité (i.e. choix d'une opération élémentaire)
- Recherche des cas dont ont peu démontrer qu'ils sont (respectivement) le plus favorable et le plus défavorable en terme de nombre d'opérations. Il suffit alors d'appliquer l'algorithme sur ces cas et de dénombrer les opérations effectuées en tenant compte des mises-à-jour de compteur de boucle et des débranchements. Si possible, on souhaite garder une expression analytique exacte du nombre d'opérations.
- Si les complexités minimale et maximale sont égales (ou au moins du même ordre de grandeur), on en tire une conclusion quant à la complexité moyenne, par le théorème d'encadrement. Sinon, on se munit d'une distribution de probabilité (généralement uniforme) sur l'ensemble (fini !) des données acceptées en entrée par l'algorithme. On recherche une expression analytique de la complexité de chaque donnée et on calcule la moyenne pondérée (par les probabilités d'apparition) de ces valeurs de complexités. Là aussi, on cherchera à conserver une expression analytique exacte le plus longtemps possible.
- Enfin, on cherche l'ordre (si possible avec un grand Θ) des complexités trouvées et on détermine les classes de complexité correspondantes (logarithmique, linéaire, polynomiale...).

Exemple : recherche dans un tableau

On se donne un tableau T à une dimension, dans lequel on souhaite rechercher un élément donné e . Ecrire une fonction (efficace) en ADL permettant de retourner l'indice de l'élément trouvé (ou bien -1 si l'élément n'est pas présent). Prouver que la fonction termine et qu'elle est correcte. Enfin, calculer la complexité de l'algorithme dans la meilleur des cas, dans le pire des cas et en moyenne.

L'algorithme s'écrit simplement avec une boucle et un débranchement de sortie dès lors qu'on a trouvé l'indice correspondant.

SRC($T \downarrow, e \downarrow$) : $\left[\left[N \leftarrow \text{SIZE}(T); \text{SRC} \leftarrow -1; \left\{ \begin{array}{l} \forall k:1^N \\ T_k = e ? \quad \text{SRC} \leftarrow k; \text{SRC} \leftarrow \text{SRC} \mid i \end{array} \right\}_k \right] \right]$

L'algorithme est composé d'une boucle sans mise-à-jour de son pas ni de sa borne finale. Elle contient un débranchement de type "sortie de boucle", et entraîne donc un nombre maximal de n itérations. L'algorithme termine donc nécessairement.

On utilise l'invariant de boucle suivant :

P : l'identificateur i contient soit la valeur -1 et e n'est pas présent dans T entre les positions 1 et k , soit l'indice de e dans T et dans ce cas la fonction se termine.

Avant l'entrée de boucle, P_0 est vérifiée puisque i est initialisé à -1 . Supposons que P_k soit vérifiée. Alors, soit i contient k et la fonction se termine en retournant la valeur correcte. Soit i vaut -1 . Dans ce second cas, on entre dans la boucle. Soit $T_k = e$ et alors i prend la valeur k donc P_{k+1} est vérifiée. Soit $T_k \neq e$ et i conserve la valeur -1 . Comme dans ce cas, d'après P_k , l'élément e n'était pas présent dans les k premières cellules du tableau et qu'il n'est pas non plus dans la $k + 1$ -eme, il n'est pas dans les $k + 1$ premières cellules et P_{k+1} est vérifiée.

Nous avons donc montré que P_k est vérifiée pour tout k . L'algorithme termine après un nombre k_0 (inconnu) d'itérations. Donc, au moment de la sortie P_{k_0} est vérifiée. Comme la fonction se termine, on est dans le second cas de la proposition et donc l'identificateur i contient l'indice de e dans T . L'algorithme est correct.

Pour l'étude de la complexité, nous prenons comme mesure de taille des données, la taille n du tableau et le test d'égalité comme opération élémentaire.

De manière évidente, le nombre de tests est compris entre 0 et n . Dans le meilleur des cas, l'élément i est situé en première position et la complexité est à temps constant $\Theta(1)$. Dans le pire des cas, il n'est pas dans le tableau et on a une complexité linéaire $\Theta(n)$. Les deux ordres sont différents, on ne peut donc pas en déduire la complexité moyenne par encadrement.

Posons une distribution de probabilité sur les données, en supposant que chaque composante du tableau prend ses valeurs de manière uniforme dans l'ensemble $\{1, 2, \dots, p\}$, et on supposera bien sûr que $1 \leq e \leq p$

La probabilité qu'un élément T_k soit égal à e vaut $1/p$. Le nombre de tests d'égalité à effectuer est égal à la position du premier élément égal à e . Cherchons alors la distribution de l'indice de cette position.

On s'intéresse alors à la probabilité $\pi(k)$ d'obtenir une configuration nécessitant k passages dans la boucle. Cette probabilité s'exprime différemment suivant que le tableau a été parcouru complètement ou non (si le tableau a été parcouru en entier, il faut comptabiliser le cas où e s'avère être le dernier élément, et celui où e n'est pas dans le tableau). La probabilité de faire k passages est aussi la probabilité que le premier élément égal à e soit situé en position k dans le tableau. C'est donc la probabilité que les $k - 1$ premiers éléments soient différents de e et que le k -eme soit égal à e , ce qui donne au bilan :

$$\pi(k) = \begin{cases} \frac{1}{p} \left(\frac{p-1}{p}\right)^{k-1} & \text{si } k < n \\ \left(\frac{p-1}{p}\right)^{n-1} & \text{sinon} \end{cases}$$

La complexité moyenne se calcule alors par (en posant $a = \frac{p-1}{p}$) :

$$\sum_{k=1}^n k\pi(k) = \frac{1}{p} \sum_{k=1}^{n-1} ka^{k-1} + na^{n-1}$$

Notons que $a \in \mathbb{R}$ donc, par linéarité de l'intégration, on peut écrire :

$$C_{moy}(n) = \frac{1}{p} \left(\sum_{k=1}^{n-1} a^k \right)' + na^{n-1} = (1-a) \left(\frac{a-a^n}{1-a} \right)' + na^{n-1} = p - \frac{(p-1)^n}{p^{n-1}}$$

Notons qu'une solution alternative consiste à développer le facteur $(1-a)$ puis à intégrer le terme résiduel na^{n-1} dans l'une des deux sommes obtenues, permettant ainsi par sommation télescopique d'exprimer le résultat purement sous forme d'une série géométrique.

On vérifie bien que, quand $p = 1$, on a $C_{moy}(n) = 1$.

En effet, dans ce cas il n'y a qu'un seul choix de tirage aléatoire. L'élément recherché est donc nécessairement en première position, ce qui implique un seul test de comparaison. Inversement, quand $p \rightarrow +\infty$, en exprimant $(p-1)^n$ avec ses deux premiers coefficients binomiaux, on obtient immédiatement que : $\lim_{p \rightarrow +\infty} C_{moy}(n) = n$

Lorsque le nombre de choix est infiniment grand, la probabilité de trouver l'élément dans la liste devient négligeable et le nombre d'opérations à effectuer tend vers la taille de la liste.

De manière très intéressante, on a également : $\lim_{n \rightarrow +\infty} C_{moy}(n) = p$. La complexité est bornée par la valeur de p et donc on peut écrire que : $C_{moy}(n) = \Theta(1)$, et de manière surprenante, l'algorithme est à temps constant. La complexité moyenne de l'algorithme est donc plus proche de sa complexité dans le meilleur que dans le pire des cas.

Notons que ce résultat surprenant ne tient plus si p augmente en proportion avec n et dans ce cas la complexité est linéaire. En effet, pour un tirage avec remise⁸ des éléments :

$$C_{moy}(n) = n - \frac{(n-1)^n}{n^{n-1}} = n - n \left(1 - \frac{1}{n}\right)^n \underset{n \rightarrow +\infty}{\sim} n(1 - e^{-1}) \approx 0.63n \in \Theta(n)$$

8. Pour un tirage sans remise, on obtient la solution triviale $C_{moy}(n) = n/2$.

3.10 Cas des fonctions récursives

Si l'algorithme dont on cherche à connaître la complexité est récursif, il est alors facile de trouver une relation de récurrence sur la complexité :

$$C(n) = a_1C(f_1(n)) + a_2C(f_2(n)) + \dots + a_pC(f_p(n)) + f(n)$$

avec : $n > f_1(n) \geq f_2(n) \geq \dots \geq f_p(n)$.

Il s'agit là bien entendu du cas le plus général, et la plupart des exemples rencontrés en pratique seront plus simples.

Exemples :

Factorielle : $C(n) = C(n - 1) + 1$

Suite de Fibonacci : $C(n) = C(n - 1) + C(n - 2) + 1$

Tri par fusion : $C(n) = 2C(n/2) + 1$ (liste de taille égale à une puissance de 2)

L'objectif de cette section est d'introduire les techniques et la méthodologie générale permettant de déterminer une expression explicite de la complexité, et par suite d'en donner un ordre de croissance, afin d'associer l'algorithme étudié à une classe de complexité.

3.10.1 Récurrence linéaire simple

Une récurrence linéaire simple (on dit aussi récurrence linéaire d'ordre 1) s'exprime sous la forme :

$$T(n) = aT(n - 1) + f(n)$$

où $a \in \mathbb{N}^*$ et $f : \mathbb{N} \rightarrow \mathbb{N}$ est une fonction dont on connaît l'expression explicite.

Cas : $a = 1$

Par sommation télescopique, on obtient l'expression de T sous la forme :

$$T(n) = T(1) + \sum_{k=1}^{n-1} f(k)$$

L'étude de la complexité revient alors à étudier la série associée à la suite $f(k)$. Il est alors bon de connaître quelques identités de sommes (voir annexe B).

En règle général, si $f(k)$ est un $\Theta(k^p)$ alors, sa série associée est un $\Theta(k^{p+1})$.

Exemples :

Calcul de la factorielle : $T(n) = T(n - 1) + 1$ et $T(1) = 1 = \mathcal{O}(1)$ et $f(n) = \mathcal{O}(1)$. La sommation donne donc une complexité en $\mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$. On retrouve évidemment une complexité linéaire pour le calcul de la factorielle.

Tri par sélection sous forme récursive : on recherche le minimum d'un tableau et on le place dans la première cellule. On appelle alors la procédure de manière récursive sur le sous-tableau commençant en 2, et ce jusqu'à ce que le tableau soit complètement trié.

A chaque appel récursif, on doit effectuer une recherche du minimum, ce qui se fait en un temps $\mathcal{O}(n)$. D'autre part, la formule de récursivité appelle la procédure une fois pour un tableau de taille $n - 1$. On obtient alors : $T(n) = T(n - 1) + n$. La complexité s'exprime donc de manière explicite par : $T(n) = \mathcal{O}(1) + \sum_i \mathcal{O}(i) = \mathcal{O}(n^2)$ et on retrouve la complexité quadratique du tri par sélection.

Cas : $a \geq 2$

On cherche alors à se ramener au cas précédent en posant : $u_n = T(n)/a^n$

$$u_n = u_{n-1} + \frac{f(n)}{a^n}$$

La sommation s'exprime alors sous la forme : $u_n = u_1 + \sum_{k=1}^{n-1} \frac{f(k)}{a^k}$.

Résoudre le problème revient donc à étudier l'expression de la série de terme général $f(k)/a^k$ afin d'en déduire l'expression de $T(n) = u_n a^n$.

La complexité finale sera donc au minimum en $\mathcal{O}(a^n)$. On distingue en particulier trois cas principaux :

Si la quantité $f(n)$ croît beaucoup plus lentement que a^n , la somme converge et u_n tend vers une constante, autrement dit $u_n = \mathcal{O}(1)$. Ceci implique que : $T(n) = \mathcal{O}(a^n)$. Toute la complexité de l'algorithme est contenue dans la récursion (c'est-à-dire dans la structure de l'arbre des appels de la fonction).

Si $f(n) = \Theta(a^n)$, la série diverge linéairement et donc $u_n = \mathcal{O}(n)$ ce qui implique que $T(n) = na^n$. La charge de travail est équitablement répartie entre les appels récursifs (structure de l'arbre) et le travail à effectuer à chaque noeud (recomposition, décomposition).

Si $f(n) = \mathcal{O}(b^n)$ avec $b > a$ alors on a $T(n) = \mathcal{O}(f(n))$, toute la charge de travail est contenue dans les noeuds de l'arbre.

Nous retrouverons le même type de disjonction de cas plus loin lorsque nous étudierons le Master Theorem.

Exemples :

Tours de Hanoï : déplacer une pile de n disques revient à déplacer les $n - 1$ disques supérieurs sur une tige temporaire, puis à déplacer le disque inférieur sur la tige d'arrivée (ce qui se fait en temps constant), et enfin à déplacer à nouveau la pile des $n - 1$ disques sur la tige d'arrivée. L'équation de récurrence s'exprime alors :

$$T(n) = 2T(n - 1) + 1$$

On étudie la suite : $u_n = T(n)/2^n$, ce qui transforme l'équation de récurrence en : $u_n = u_{n-1} + 2^{-n}$ d'où une expression explicite :

$$\frac{T(n)}{2^n} = T(0) + \sum_{k=0}^{n-1} 2^{-k}$$

La somme converge (série géométrique de raison $1/2$ d'où :

$$\frac{T(n)}{2^n} = T(0) + 2^{-n} - 1 = \Theta(1)$$

On obtient alors la complexité de l'algorithme de résolution des tours de Hanoï :

$$T(n) = \Theta(2^n)$$

La complexité est exponentielle.

Suites annexes

Lorsque la fonction de décomposition/recomposition f est constante il est possible d'expliquer $T(n)$ simplement à l'aide d'une suite annexe bien choisie. C'est le cas par exemple pour les tours de Hanoï puisque chaque itération nécessite le déplacement d'un disque, indépendamment de la taille de la pile transmise dans l'appel récursif de niveau inférieur.

En effet, en posant $u_n = T(n) + 1$, on obtient la relation de récurrence géométrique : $u_{n+1} = 2u_n$. On obtient alors immédiatement : $T(n) = T(0)(2^n - 1) + 1 = \Theta(2^n)$ et on retrouve le même ordre de croissance.

3.10.2 Récurrence linéaire multiple

Une récurrence linéaire multiple (on dit aussi récurrence linéaire d'ordre $p > 1$) s'exprime sous la forme d'une suite $(u_n)_{n \in \mathbb{N}}$ définie par p valeurs initiales et une relation de récurrence de la forme :

$$u_{n+p} = a_{p-1}u_{n+p-1} + a_{p-2}u_{n+p-2} + \dots + a_1u_{n+1} + a_0u_n = \sum_{k=0}^{p-1} a_k u_{n+k}$$

Notons que certains coefficients a_i peuvent être nuls. Ainsi la relation de récurrence $u_{n+122} = u_n$ est bien une relation de récurrence linéaire, contrairement à $v_{3n} = v_n$.

L'exemple le plus classique de récurrence linéaire multiple est la suite de Fibonacci, que nous avons déjà eu l'occasion de rencontrer plusieurs fois dans ce cours :

$$f_0 = 1 \quad f_1 = 1 \quad f_{n+2} = f_{n+1} + f_n$$

Nous avons ici une relation linéaire d'ordre 2. Nous allons tenter de résoudre cette récurrence à la main, avant d'introduire un résultat plus général.

Nous avons montré précédemment que le programme terminait. Chaque itération entraîne 2 appels récursifs et une addition. En prenant l'addition comme unité de coût, la complexité du programme vérifie la relation de récurrence :

$$T(n) = T(n-1) + T(n-2) + 1$$

Grâce à la suite annexe $u_n = T(n) + 1$, on transforme la récurrence sous la forme :

$$u_n = u_{n-1} + u_{n-2}$$

et on est ramené à une récurrence linéaire d'ordre 2. Nous pouvons faire mieux, et réduire cette relation à une récurrence simple en se plongeant dans un espace vectoriel.

On note : $\mathbf{U}_n = (u_n, u_{n-1})^T$ et on obtient la récurrence :

$$\mathbf{U}_{n+1} = \mathbf{A}\mathbf{U}_n \quad \text{en posant :} \quad \mathbf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

En posant $\mathbf{U}_0 = (1, 1)^T$ le vecteur contenant les conditions initiales de la suite, on obtient une expression explicite du terme de rang n de la suite :

$$\mathbf{U}_n = \mathbf{A}^n \mathbf{U}_0$$

Il suffit de trouver une expression analytique de \mathbf{A}^n . Pour ce faire on diagonalise \mathbf{A} :

$$\mathbf{U}_n = \mathbf{P} \mathbf{D}^n \mathbf{P}^{-1} \mathbf{U}_0$$

où \mathbf{P} est une matrice de passage (dont nous ne rechercherons pas les termes) et \mathbf{D} est la matrice diagonale contenant les valeurs propres de \mathbf{A} . Calculons ces valeurs propres.

Comme nous avons une matrice de $\mathbb{R}^{2 \times 2}$ nous pouvons utiliser le fait que le produit des valeurs propre est égal au déterminant de \mathbf{A} tandis que la somme vaut la trace. Cela revient donc à trouver les racines du polynôme :

$$P(X) = X^2 - Tr(\mathbf{A})X + \det(\mathbf{A})$$

Le polynôme à résoudre est donc : $X^2 - X - 1$ (nous verrons par la suite qu'il correspond au polynôme caractéristique⁹ de la relation de récurrence). Les racines sont :

$$\lambda_1 = \frac{1 + \sqrt{5}}{2} \quad \lambda_2 = \frac{1 - \sqrt{5}}{2}$$

On reconnaît dans λ_1 (la plus grande des deux valeurs propres), le nombre d'or Φ .

Grâce à la relation explicite $\mathbf{U}_n = (\mathbf{P} \mathbf{D}^n \mathbf{P}^{-1}) \mathbf{U}_0$, et étant donné que la matrice de passage ne dépend pas de n , la complexité de l'algorithme de calcul de la suite de Fibonacci est :

$$T(n) = \Theta(\lambda_1^n) = \Theta(\Phi^n) \sim \Theta(1.618^n)$$

C'est une complexité exponentielle. Ceci constitue un raffinement de la complexité évaluée dans l'exemple d'introduction de ce chapitre.

Notons que cette méthode de réduction à une récurrence linéaire simple par calcul matriciel permet théoriquement toujours de calculer la complexité, sous réserve d'être capable de calculer les valeurs propres. Généralisons la méthode précédente.

Définition :

On appelle polynôme caractéristique de la suite $(u_n)_{n \in \mathbb{N}}$, le polynôme défini par :

9. Par construction c'est aussi le polynôme caractéristique de la matrice \mathbf{A} : $P_{\mathbf{A}}(X) = \det(\mathbf{A} - \mathbf{X}\mathbf{I})$.

$$P(X) = X^p - a_0 - a_1X - a_2X^2 - \dots - a_{p-1}X^{p-1}$$

Il s'agit d'un polynôme de degré p .

Théorème *espace des solutions d'une récurrence multiple*

Si P est un polynôme scindé de $\mathbb{R}[X]$, c'est-à-dire qu'on peut l'écrire sous la forme : $P(X) = \prod_{i=1}^m (X - r_i)$, avec $r_1, r_2, \dots, r_m \in \mathbb{R}$ les racines de P de degrés de multiplicité respectifs $\alpha_1, \alpha_2, \dots, \alpha_m \in \mathbb{N}^*$, alors, les solutions de la récurrence sont les suites :

$$R_n = \sum_{i=1}^m Q_i(n) r_i^n = \sum_{i=1}^m \left[\sum_{j=0}^{\alpha_i-1} \beta_{ij} n^j \right] r_i^n$$

avec : Q_i un polynôme de degré strictement inférieur à α_i , i.e. $\beta_{ij} = 0 \quad \forall j \geq \alpha_i$.

Exemple : suite récurrente linéaire d'ordre 2

Considérons une suite définie par la récurrence :

$$u_{n+2} = a_1 u_{n+1} + a_0 u_n$$

Le polynôme caractéristique de la suite s'écrit : $P(X) = X^2 - a_1X - a_0$. La résolution dépend alors des valeurs a_0 et a_1 .

Si le polynôme a deux racines distinctes r_1 et r_2 , la solution s'exprime sous la forme :

$$u_n = \lambda_1 r_1^n + \lambda_2 r_2^n$$

où λ_1 et λ_2 sont déterminés grâce aux conditions initiales de la suite (u_0 et u_1). Notons qu'en analyse de complexité, on écrira : $u_n = \mathcal{O}(r_1^n)$ ou, si λ_1 est nulle $u_n = \mathcal{O}(r_2^n)$.

Si le polynôme a une racine double r alors la solution s'exprime sous la forme :

$$u_n = (\lambda_1 n + \lambda_2) r^n$$

où à nouveau λ_1 et λ_2 sont déterminés grâce aux conditions initiales. On pourra écrire $u_n = \mathcal{O}(nr^n)$ ou, si λ_1 est nulle, $u_n = \mathcal{O}(r^n)$.

Exemple : considérons la suite récurrente d'ordre 2 ci-dessous

$$u_0 = 1 \quad u_1 = 1 \quad u_{n+2} = 3u_{n+1} + 4u_n$$

Résoudre le polynôme caractéristique donne deux racines dont la plus grande vaut 4. On en déduit alors l'ordre du terme général sans calculer les constantes : $u_n = \Theta(4^n)$.

Exercice 3.5. Que devient la complexité lorsqu'on retire le terme intermédiaire $3u_{n+1}$?

3.10.3 Diviser pour régner

"Diviser pour régner" est un paradigme de programmation consistant à s'appuyer sur le fait qu'il est bien souvent plus facile de décomposer un problème en deux (ou plus) sous-problèmes de plus petite taille, pour les résoudre individuellement avant de les recombiner. Il permet généralement d'écrire un code plus simple, mais surtout d'optimiser considérablement le temps de calcul. Bien entendu, ceci n'est valide que si le coût de décomposition/recomposition des problèmes est faible comparé au gain réalisé par la répartition du travail sur les différentes branches du calcul.

Nous avons déjà rencontré dans ce cours plusieurs exemples d'application de la stratégie *diviser pour régner*. Par exemple, le tri fusion sépare la liste à trier en deux sous-listes, trie chaque liste séparément, puis les recombine. Combiner deux listes déjà triées est simple et rapide, donc le coût de décomposition (découpage en deux listes) et recomposition (fusion de deux listes triées) est relativement faible. Nous verrons que dans le domaine des algorithmes de tri, les algorithmes de type diviser pour régner sont les plus rapides. L'algorithme d'exponentiation rapide est également un algorithme de type diviser pour régner. Calculer $a^{n/2} \times a^{n/2}$ est plus rapide que de calculer a^n d'autant plus que dans cet exemple, contrairement au tri fusion, les deux sous-problèmes sont identiques et donc un seul traitement est suffisant. Ce paradigme peut également être utilisé en géométrie algorithmique. Par exemple, il est plus facile de calculer l'enveloppe convexe de deux sous-semis de points et de calculer la fusion des enveloppes, plutôt que de calculer l'enveloppe itérativement en ajoutant les points les uns après les autres.

Un algorithme de type diviser pour régner n'est intéressant que si le principe est appliqué récursivement. C'est-à-dire que chaque sous-problème doit être également divisé en deux puis recombiné et ainsi de suite. En pratique, il est donc essentiel de pouvoir construire des fonctions récursives pour pouvoir avoir recours à ce principe.

Lorsqu'il est bien implémenté, le principe diviser pour régner permet bien souvent de descendre d'une classe de complexité par rapport à un algorithme naïf. C'est-à-dire que par exemple, un algorithme en temps linéaire s'exécutera en un temps logarithmique (dichotomie, exponentiation rapide), un algorithme quadratique tournera en un temps $\mathcal{O}(n^k)$ avec $1 < k < 2$ (multiplication de Karatsuba), ou même parfois en temps quasi-linéaire $\mathcal{O}(n \log n)$ (tri fusion, transformée de Fourier rapide, enveloppe convexe...), un algorithme cubique $\mathcal{O}(n^3)$ deviendra un $\mathcal{O}(n^k)$ avec $2 < k < 3$ (algorithme de Strassen pour la multiplication matricielle)... Dans certains cas, on peut même assister à un gain encore plus spectaculaire (il existe notamment une version *diviser pour régner* du calcul de la suite de

Fibonacci, permettant de résoudre le problème en temps quasi-linéaire, à comparer avec la complexité exponentielle de l'algorithme récursif de base).

Dans ce paradigme, la fonction de complexité obéit dans le cas général à une relation de récurrence de la forme :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

où $a \in \mathbb{N}$ désigne le nombre de sous-problèmes à traiter à chaque itération (on supposera ici que $a \geq 1$), tandis que $b \in \mathbb{N}^*$ correspond au facteur de réduction des sous-problèmes.

Exemples :

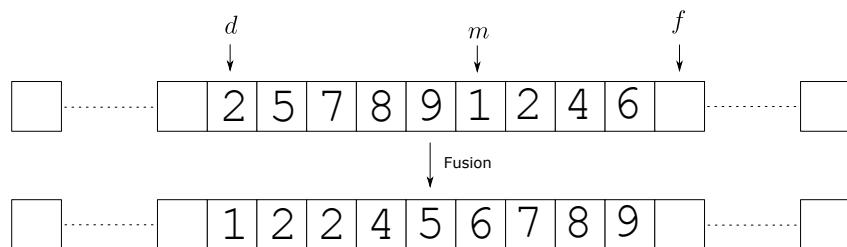
Tri fusion : $a = b = 2$, chacune des 2 sous-listes est 2 fois plus petite que la liste initiale.

Le nombre de sous-problèmes n'est pas nécessairement égal au facteur de réduction. Par exemple, pour l'algorithme d'exponentiation rapide, les 2 sous-problèmes sont identiques, donc en terme de complexité : $a = 1$ et $b = 2$.

A l'inverse, dans certains cas, le facteur de réduction est moins élevé que le nombre de sous-problèmes. On peut se faire une idée de cette situation en imaginant que les sous-problèmes se recouvrent partiellement. Ce sera le cas pour l'algorithme de multiplication matricielle rapide, que nous étudierons dans cette section, où on aura $a = 7$ et $b = 2$, c'est-à-dire 7 sous-problèmes 2 fois plus petits.

Commençons par étudier le cas pratique du tri fusion, déjà abordé dans le chapitre 1, mais nous utiliserons ici une architecture plus efficace.

Nous avons tout d'abord besoin d'une fonction qui permet de regrouper deux listes triées.



Cette fonction prend en réalité en entrée une unique liste, dont nous supposerons que la première et la seconde partie (délimitées par un indice de séparation m) sont triées individuellement. Nous avons également besoin de deux indices d et f permettant de localiser le tableau à fusionner dans le tableau global.

FUSION($T \uparrow, d \downarrow, f \downarrow, m \downarrow$) : $\llbracket i \leftarrow d; j \leftarrow m; \begin{cases} f \\ k:d \end{cases} j > f ? L_{k-d+1} \leftarrow T_i; i \leftarrow i+1; \triangleright | i \geq m ? L_{k-d+1} \leftarrow T_j; j \leftarrow j+1; \triangleright | i \leq T_i ? L_{k-d+1} \leftarrow T_i; i \leftarrow$

$$i + 1; \lceil \rightarrow | _ L_{k-d+1} \leftarrow T_j; j \leftarrow j + 1 | _ \}^f_k \{^f_{k:d} T_k \leftarrow L_{k-d+1} \}_k \rceil$$

On peut alors créer la fonction récursive tri-fusion qui s'écrit à présent de manière très simple, grâce au principe diviser pour régner.

$$\text{TRIFUSION}(T \uparrow, d \downarrow, f \downarrow) : \left[\left[d = f ? _ ! | _ m \leftarrow (d + f) \div 2; \text{TRIFUSION}(T, d, m - 1); \text{TRIFUSION}(T, m, f); \text{FUSION}(T, d, f, m) \right] \right]$$

Etudions la complexité de cette fonction, et comparons-là à la complexité quadratique (dans tous les cas) du tri fusion.

On utilise comme unité de coût le nombre de comparaisons effectuées et on s'intéresse à la complexité dans le pire des cas.

A chaque itération on appelle la fonction récursive deux fois, sur deux listes de taille $n/2$ (en supposant que n est pair) et l'algorithme de fusion nécessite au plus n comparaison (dans le cas où les deux sous-listes sont exactement imbriquées). L'équation de récurrence sur la complexité s'écrit alors :

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

On se retrouve bien dans le cas général avec $a = b = 2$ et $f(n) = \mathcal{O}(n)$, c'est-à-dire un arbre d'appels récursifs binaire avec chaque noeud de taille égale à la moitié de son noeud parent, et entraînant une recombinaison en temps linéaire.

La méthode de résolution générale consiste à supposer que la donnée à traiter est de taille égale à une puissance de b :

$$T(2^p) = 2T(2^{p-1}) + \mathcal{O}(2^p)$$

Comme pour le cas des récurrences linéaires, on étudie la suite $u_p = T(2^p)/a^p$:

$$u_p = u_{p-1} + \mathcal{O}(1)$$

On est alors ramené au cas linéaire d'ordre 1, ce qui se résout aisément et donne : $u_p = \mathcal{O}(p)$, c'est-à-dire $T(2^p) = \mathcal{O}(p \times 2^p)$. On exprime alors T en fonction de la taille originale n avec le changement de variable inverse $p = \log_2 n$:

$$T(n) = \mathcal{O}(n \log_2 n)$$

Enfin, il ne reste plus qu'à encadrer n entre deux puissances de a : $\exists p \in \mathbb{N}$ t.q. $2^p \leq n \leq 2^{p+1}$ pour conclure que la relation explicite trouvée ci-dessus est valable pour tout $n \in \mathbb{N}$.

La complexité dans le pire des cas du tri fusion est donc quasi-linéaire. Nous laissons le soin au lecteur de vérifier à titre d'exercice que la complexité dans le meilleur des cas est du même ordre (dans le meilleur des cas, la fonction de recombinaison n'opère que $f(n) = n/2$ comparaisons). Par théorème d'encadrement, on obtient alors la complexité moyenne de l'algorithme :

$$C_{moy}(n) = \Theta(n \log n)$$

La résolution proposée ci-dessus est fastidieuse. Nous introduisons donc un théorème très utile en pratique, que l'on appelle Master Theorem, ou encore Master Method, du fait qu'il permet de venir à bout de la grande majorité des récurrences de type diviser pour régner.

Théorème : *Master Theorem*

Etant donnés deux entiers $a, b \in \mathbb{N}^*$ ainsi qu'une fonction $T : \mathbb{N} \rightarrow \mathbb{R}^+$, on appelle équation de partition la relation de récurrence :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

où la quantité n/b est à comprendre au sens de la division entière, et où f est une fonction croissante de $\mathbb{N} \rightarrow \mathbb{R}^+$.

Le Master Theorem nous donne alors l'ordre de T :

- | | | |
|---|---------------|--------------------------------------|
| 1. Si pour un réel $\varepsilon > 0$ $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$ | \Rightarrow | $T(n) = \Theta(n^{\log_b a})$ |
| 2. Si $f(n) = \Theta(n^{\log_b a})$ | \Rightarrow | $T(n) = \Theta(n^{\log_b a} \log n)$ |
| 3. Si pour un réel $\varepsilon > 0$ $f(n) = \Omega(n^{\log_b a + \varepsilon})$ | \Rightarrow | $T(n) = \Theta(f(n))$ |

Dans le cas 3, on devra aussi vérifier la **condition de régularité**, c'est-à-dire que :

$$\exists k < 1 \text{ t.q. } kf(n) \geq af\left(\frac{n}{b}\right) \quad \forall n \in \mathbb{N}$$

En pratique, la condition de régularité est presque systématiquement vérifiée, puisque le coût de décomposition/recomposition pour une donnée de taille n est nécessairement plus élevé que la somme des coûts de décomposition/recomposition sur les sous-problèmes.

Preuve : nous ne donnons ici que la preuve pour le cas 1, les autres cas se démontrant par un procédé similaire. D'après l'équation de partition, nous pouvons schématiquement représenter le déroulement de l'algorithme comme un arbre de récursion, chaque niveau contenant a fois plus de noeuds que le niveau précédent et chaque noeud opérant sur une donnée b fois plus petite que les données traitées sur le niveau précédent (si besoin, nous renvoyons le lecteur à la section 4.4 traitant des arbres).

Par hypothèse, l'arbre des appels récursifs se termine dès lors qu'une donnée ne contient plus aucun élément. La profondeur de l'arbre est donc $h = \log_b n$. En effet, en divisant la

donnée initiale (de taille n) h fois par b , on obtient :

$$n \times \left(\frac{1}{b}\right)^h = n \times \left(\frac{1}{b}\right)^{\log_b n} = \frac{n}{b^{\log_b n}} = 1$$

S'ensuit alors l'expression explicite de la complexité :

$$C(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right) + \Theta(1)$$

Sous les hypothèses du cas 1, on peut trouver un réel strictement positif $\varepsilon \in \mathbb{R}^{+*}$ tel que $C(n)$ soit un Θ de la quantité suivante :

$$\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \varepsilon} = n^{\log_b a - \varepsilon} \sum_{i=0}^{\log_b n} a^i b^{-i \log_b a} b^{i\varepsilon}$$

En remarquant que : $b^{-i \log_b a} = (b^{-\log_b a})^{-i} = a^{-i}$ on reconnaît une série géométrique de raison b^ε :

$$n^{\log_b a - \varepsilon} \sum_{i=0}^{\log_b n} b^{i\varepsilon} = n^{\log_b a - \varepsilon} \left(\frac{1 - b^{\varepsilon(\log_b n + 1)}}{1 - b^\varepsilon}\right) \underset{n \rightarrow \infty}{\sim} n^{\log_b a - \varepsilon} \frac{n^\varepsilon}{1 - b^\varepsilon} = \frac{n^{\log_b a}}{1 - b^\varepsilon}$$

Bien qu'étant arbitrairement petit, le réel ε est fixé par hypothèse, et donc l'expression ci-dessus est un $\Theta(n^{\log_b a})$, ce qui conclut la preuve du Master Theorem pour le premier cas, et nous laissons le soin au lecteur d'établir les preuves pour les deux cas restants.

Donnons quelques exemples concret d'utilisation de ce théorème.

a. Tri fusion : on a $a = b = 2$ donc $n^{\log_b a} = n$. D'autre part, $f(n) = \Theta(n)$ (temps de décomposition/recomposition linéaire), on est donc dans le cas 2 du théorème et on obtient :

$$C(n) = \Theta(n \log n)$$

b. Exponentiation rapide : la relation de récurrence est donnée par l'équation : $C(n) = C\left(\frac{n}{2}\right) + 2$. En effet, dans le cas où l'exposant est impair, il faut effectuer 2 multiplications et 1 appel récursif de taille moitié. On est donc dans le cas $a = 1$, $b = 2$ et $f(n) = \Theta(1)$. On doit alors comparer $n^{\log_b a} = n^{\log_2 1} = 1$ à $\Theta(1)$ et on se trouve dans le cas 1, soit une complexité logarithmique :

$$C(n) = \Theta(\log n)$$

c. Tri dichotomique : la relation de récurrence est $C(n) = C\left(\frac{n}{2}\right) + \mathcal{O}(1)$ (chaque itération consiste à séparer le tableau en 2 et à identifier le sous-tableau à analyser (ce qui ne nécessite qu'une seule comparaison et se fait en temps constant). On a alors $n^{\log_b a} = 1$ et donc :

$$C(n) = \Theta(\log n)$$

d. Produit matriciel par l'algorithme de Strassen :

Considérons 2 matrices \mathbf{A} et \mathbf{B} dont on souhaite calculer le produit : $\mathbf{C} = \mathbf{A} \times \mathbf{B}$.

Nous avons écrit au chapitre 1 l'algorithme permettant de calculer de manière classique ce produit. Il nécessite n^3 multiplications élémentaires, soit une complexité cubique. L'algorithme de Strassen permet d'accélérer ce calcul en faisant l'observation suivante :

Si on écrit chaque matrice sous forme de 4 blocs de tailles à peu près égales (pour évaluer la complexité, on se place idéalement dans le cas où les tailles des matrices sont des puissances de 2) :

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix}$$

Chaque composante du produit s'écrit classiquement suivant les règles de multiplications par blocs :

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{2,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{2,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$

Ce calcul nécessite 8 multiplications et 4 additions sur des matrices de taille deux fois moindres. Si on implémentait le calcul du produit matriciel de manière récursive sur les blocs de matrice, on obtiendrait alors l'équation de partition :

$$C(n) = 8C\left(\frac{n}{2}\right) + \mathcal{O}(n^2)$$

Le terme quadratique provient du fait que les additions sont à effectuer autant de fois qu'il y a de termes dans les matrice, c'est-à-dire n^2 fois.

En utilisant le Master Theorem, on trouve que la complexité est inchangée et vaut : $\Theta(n^3)$.

L'innovation apportée par Volker Strassen consiste à faire remarquer que ces 8 produits peuvent être astucieusement réduits à 7 en utilisant les combinaisons suivantes :

$$\begin{aligned}\mathbf{M}_1 &= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_3 &= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_5 &= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\ \mathbf{M}_7 &= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})\end{aligned}$$

$$\begin{aligned}\mathbf{M}_2 &= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_4 &= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_6 &= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})\end{aligned}$$

Les blocs \mathbf{C}_{ij} sont alors donnés par :

$$\begin{aligned}\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 & \mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 & \mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6\end{aligned}$$

On vérifie aisément que ce calcul ne comporte plus que 7 multiplications (une par matrice \mathbf{M}_i) et quelques additions, ce qui réduit l'équation de récurrence sur la complexité à :

$$C(n) = 7C\left(\frac{n}{2}\right) + \Theta(n^2)$$

On compare alors $n^{\log_2 7} = n^{2.808}$ à $f(n) = \Theta(n^2)$. On peut trouver sans problème un $\varepsilon > 0$ tel que : $f(n) = \mathcal{O}(n^{2.808-\varepsilon})$ et donc la complexité finale s'écrit : $\Theta(n^{\log_2 7})$.

L'algorithme de Strassen permet donc de multiplier deux matrices en un temps $\mathcal{O}(n^{2.808})$.

Notons que l'algorithme de Strassen n'est plus à ce jour l'algorithme de produit matriciel doté de la plus faible complexité. En particulier, l'algorithme de Coppersmith-Winograd, bien que considéré comme inutile en pratique du fait de la constante prohibitive le rendant extrêmement lent sur les matrices de tailles usuelles, possède une complexité en $\mathcal{O}(2^{2.376})$. Une matrice contenant n^2 éléments potentiellement indépendants, il est certain qu'un algorithme ne pourra être qu'au mieux quadratique. Récemment, Raz (2002) a démontré que cette borne inférieure pouvait en réalité être réévaluée à un $\Omega(n^2 \log n)$. La question de savoir si elle pourra être atteinte reste encore ouverte à ce jour.

Remarque : le gain paraît ici relativement faible, mais appliqué sur des matrices de taille 100000, cet algorithme permet théoriquement de passer d'un temps de calcul de l'ordre de 15 minutes à 2 minutes 30, ce qui est loin d'être négligeable.

Nous verrons en exercice (exercice 3.27), une application similaire sur la multiplication de grands entiers (typiquement plusieurs millions de décimales).

e. Transformation de Fourier rapide :

Considérons l'espace de Hilbert $\mathcal{L}^2(\mathbb{R})$ des fonctions définies sur l'ensemble des réels et dont le carré est intégrable. D'un point de vue physique, cet espace correspond à l'ensemble des

signaux d'énergie finie, *i.e.* la totalité des fonctions que l'on peut avoir à traiter en pratique. Soit une fonction f de $\mathcal{L}^2(\mathbb{R})$: sa transformée de Fourier \hat{f} est une fonction de la variable réelle ω définie par :

$$\hat{f}(\omega) = \mathcal{F}[f] = \int_{-\infty}^{+\infty} f(x)e^{-j\omega x}dx$$

La transformée de Fourier $\hat{f} \in \mathbb{C}$ est une fonction complexe dont l'amplitude $|f(\omega)|$ indique la contribution des signaux de pulsation ω dans le signal original f . On peut montrer qu'il existe une application réciproque \mathcal{F}^{-1} de forme très similaire à la TF directe, à un signe et une constante près, permettant de calculer la synthèse d'un ensemble infini de signaux de pulsations différentes. En ce sens, la TF opère une décomposition réversible d'une fonction donnée pour la représenter dans un espace fréquentiel.

Les données représentables en machine étant nécessairement de taille finie, il existe une version discrète de la TF, adaptée aux signaux échantillonnés : la transformation de Fourier discrète (TFD) :

$$S_m = \sum_{k=0}^{n-1} s_k e^{-j2\pi \frac{km}{n}} \quad m \in \{0, 1, \dots, n-1\}$$

dans laquelle S_m désigne la m -eme composante de la transformée de Fourier d'un signal temporel s échantillonné en n points. Le nombre s_k est le k -eme échantillon de s . Pour plus de détails techniques, nous renvoyons le lecteur à l'excellent ouvrage de F. Cottet (2015).

Cette transformation est intensivement utilisée en analyse de signaux, en filtrage mais aussi comme nous le verrons plus loin, en accélération *computационnelle*, notamment grâce au développement d'une technique efficace de calcul de la TFD.

Calculons la complexité de la TFD. Nous prendrons comme unité de taille le nombre n de points d'échantillons du signal à transformer, et comme unité de coût d'exécution le nombre de multiplications complexes (en notant toutefois que ce choix général n'impactera pas les ordres de croissances trouvés, une multiplication complexe équivalant au plus 4 multiplications flottantes classiques).

Le signal transformé contient n échantillons. Pour calculer un échantillon, on doit sommer n termes, chacun requérant une multiplication par l'exponentielle complexe. La complexité de la TFD s'en déduit immédiatement :

$$C(n) = \Theta(n^2)$$

Cette complexité quadratique peut être pénalisante, en particulier dans les applications temps réels, qui sont bien souvent aussi embarquées et qui disposent donc de ressources matérielles très limitées. D'autre part, en considérant la taille typique d'un signal sous forme d'image (plusieurs millions de pixels pour une image typique de quelques Mo), la

moindre compression JPEG serait prohibitive en termes de temps de calcul.

L'algorithme de la transformation de Fourier rapide (souvent abrégé en FFT pour *Fast Fourier Transform*) a été publié en 1965 par James Cooley et John Tukey et repose sur le paradigme *diviser pour régner*. L'idée astucieuse de la méthode, consiste à diviser le signal en deux parties, la première contenant les échantillons pairs et la seconde les échantillons impairs, puis à opérer la transformation sur chaque partie individuellement, et enfin à utiliser les propriétés de symétrie de l'exponentielle complexe pour accélérer davantage le calcul (nous ne traiterons pas ici de cette optimisation).

En décomposant l'expression de la transformée en une somme d'échantillons pairs ($k = 2i$) et une somme d'échantillons impairs ($k = 2i + 1$), on obtient :

$$\begin{aligned} S_m &= \sum_{i=0}^{n/2-1} s_{2i} e^{-j2\pi \frac{2im}{n}} + \sum_{i=0}^{n/2-1} s_{2i+1} e^{-j2\pi \frac{(2i+1)im}{n}} \\ &= \sum_{i=0}^{n/2-1} s_{2i} e^{-j2\pi \frac{im}{n/2}} + e^{-j2\pi \frac{m}{n}} \sum_{i=0}^{n/2-1} s_{2i+1} e^{-j2\pi \frac{im}{n/2}} \end{aligned}$$

En notant $S_{0,(n/2)}$ et $S_{1,(n/2)}$ les transformées de Fourier calculées respectivement sur les échantillons pairs et impairs du signal s , on obtient l'expression condensée :

$$S = S_{0,(n/2)} + e^{-j2\pi \frac{m}{n}} S_{1,(n/2)}$$

Les termes $S_{i,(n/2)}$ sont bien évidemment eux aussi calculés par le même procédé, et ce de manière récursive, jusqu'à réduire le problème au calcul de la TFD d'un signal composé d'un unique échantillon, résoluble en temps constant. Notons que similairement au tri fusion, le schéma de partitionnement suppose que le nombre d'échantillons est initialement une puissance de 2. Cette hypothèse n'est pas réductrice, moyennant un ajout de zéros non-significatifs dans le signal (on parle de *zero-padding*).

Calculons la complexité de la FFT. Chaque décomposition nécessite n multiplications. L'équation de partition est alors :

$$C(n) = 2C\left(\frac{n}{2}\right) + n$$

ce qui correspond au cas 2 du Master Theorem, avec $a = 2$, $b = 1$ et f linéaire, soit $n^{\log_b a} = n = \Theta(f(n))$. L'algorithme de la FFT a donc une complexité quasi-linéaire :

$$C(n) = \Theta(n \log n)$$

Un exemple d'application de la FFT est donné ci-dessous, un second pourra être trouvé dans l'exercice 3.27.

Dans le chapitre 1 (page 51), nous avons introduit l'opérateur de convolution, et nous avons indiqué qu'il était fréquemment utilisé en traitement d'images, mais que son coût de calcul était relativement élevé, ce qui pouvait s'avérer problématique lorsqu'un grand nombre de convolutions étaient à opérer en cascade, comme c'est le cas par exemple dans les réseaux de neurones artificiels. Nous laisserons le soin au lecteur de vérifier à titre d'exercice que la complexité d'un calcul de convolution d'une image de taille $n \times n$ par un masque de même taille est de l'ordre de n^4 opérations. Cependant, la transformée de Fourier possède une propriété très intéressante pour le calcul des convolutions : la multiplication \times et la convolution $*$ sont analogues dans les espaces temporel et fréquentiel. Autrement dit : la TF d'une convolution est égale à la multiplication des TF, et inversement, la TF d'une multiplication équivaut à la convolution des TF.

$$\mathcal{F}[f * g] = \mathcal{F}[f]\mathcal{F}[g] \quad \text{et :} \quad \mathcal{F}[fg] = \mathcal{F}[f] * \mathcal{F}[g]$$

Plutôt que de calculer directement la convolution de deux images f et g , nous pouvons envisager de : calculer séparément la TF de f et g , multiplier les transformées obtenues (la multiplication est bien moins coûteuse que la convolution), puis récupérer la TF inverse du signal produit. Mathématiquement :

$$f * g = \mathcal{F}^{-1}[\mathcal{F}[f * g]] = \mathcal{F}^{-1}[\mathcal{F}[f] \times \mathcal{F}[g]]$$

Evaluons la complexité de cette nouvelle approche : il faut réaliser 2 transformées de Fourier directes sur des signaux de n^2 éléments chacun (soit $2n^2 \log n^2$ opérations), un produit classique (n^2 opérations) et enfin une TF inverse, dont nous avons mentionné qu'elle avait la même expression analytique, et donc la même complexité que la TF directe, (soit à nouveau $n^2 \log n^2$ opérations). En sommant ces trois contributions, on obtient :

$$C(n) = 2n^2 \log n^2 + n^2 + n^2 \log n^2 = 3n^2 \log n^2 + n^2 = \Theta(n^2 \log n^2) = \Theta(n^2 \log n)$$

La complexité de l'approche est donc quasi-quadratique, à comparer avec la complexité en puissance 4 de l'approche directe, soit un gain quadratique en temps de calcul. Notons que cette optimisation n'est possible que grâce à l'utilisation de la FFT. Un calcul similaire avec la TFD standard montre que le temps de passage dans l'espace fréquentiel n'est plus

négligeable par rapport à celui du calcul direct de la convolution.

On pourra trouver de nombreux autres exemples d'applications (*e.g.* en statistiques avec l'estimation de densités de probabilités par la méthode des noyaux, en ingénierie pour le calcul indirect de la fonction d'autocorrélation à partir de la densité spectrale de puissance ou encore en géologie pour l'analyse variographique d'un semis régulier d'observations...).

f. Calcul de la médiane :

Dans cette section, nous explorons deux méthodes de type diviser pour régner, permettant d'accélérer le calcul de la médiane d'un ensemble de nombres, et dont l'analyse rigoureuse de la complexité met en échec le Master Theorem. Ce sera donc l'occasion d'introduire un second théorème plus général : le théorème d'Akra-Bazzi (1998).

La médiane empirique d'un ensemble de n nombres réels est défini comme étant :

- le nombre de rang $(n + 1)/2$ dans la liste triée si n est impair : $L_{\frac{n+1}{2}}$
- la moyenne arithmétique des nombres de rangs $n/2$ et du $n/2 + 1$ dans la liste triée si n est impair : $\frac{1}{2}(L_{\frac{n}{2}} + L_{\frac{n}{2}+1})$

Moyennant cette définition, l'existence et l'unicité de la médiane empirique est garantie pour tout ensemble non-vide de nombres (dans la suite on supposera donc que $n \geq 1$).

L'approche naïve du calcul de la médiane, consiste à trier la liste, puis à effectuer les opérations indiquées dans la définition ci-dessus.

Dans un premier temps, nous allons supposer qu'il n'y a pas de doublons dans la liste des nombres. En pratique, cette hypothèse n'est pas restrictive, car il est toujours possible de définir une quantité ε , par exemple le plus petit réel (en valeur absolue) représentable en machine, puis de poser : $T_i \leftarrow T_i + i * \varepsilon$. Cette modification n'affecte pas le rang des nombres de l'ensemble mais garanti l'absence de doublon. Par exemple, pour $\varepsilon = 0.001$ (en pratique on peut choisir un nombre beaucoup plus petit, mais pour la clarté de la notation, nous choisissons 0.001), la liste de nombres $T = [2, 3, 2, 4]$ devient : $T' = [2.001, 3.002, 2.003, 4.004]$, dont la médiane est 2.5025, qui est alors arrondi à 2.5.

Suivant le paradigme diviser pour régner, on divise le tableau de nombres T en deux sous-parties A et B de tailles à peu près égales (à une unité près), et on suppose connaître la médiane de chacune de deux sous-parties (par appel récursif de l'algorithme). Nous noterons a et b ces deux médianes. Peut-on calculer la médiane du tableau T à partir de la connaissance de a et b ? Comme bien souvent, la réponse est oui moyennant quelques ajustements et une recherche complémentaire, dont on l'espère, le coût sera amorti par la structure récursive de la résolution.

Nous supposerons sans perte de généralité que $a \leq b$. Notons a^- (resp. b^+) les nombres de T immédiatement inférieur à a (resp. supérieur à b). Il est alors facile de démontrer que la médiane de T (que nous noterons m) est comprise dans l'intervalle $[a^-, b^+]$. En effet, par l'absurde, supposons que $m > b^+$ (la démonstration dans le cas $m < a^-$ est identique).

Etant donné que b est la médiane de B et que $b < b^+ < m$, on sait qu'il existe $n/4 + 1$ nombres inférieurs (strictement) à m dans B . D'autre part $a \leq b$, et donc il y a au moins $n/4$ autres nombres inférieurs à m dans A . Au total, le tableau T contient donc au minimum $2n/4 + 1 = n/2 + 1$ nombres inférieurs (strictement) à m . Ceci contredit le fait que m est la médiane de T . En conclusion : $m \in [a^-, b^+]$. En toute rigueur, il faut séparer les cas où n est pair et impair pour traiter la division par 2.

L'algorithme découle alors directement de la propriété démontrée ci-dessus. Dans un premier temps, on cherche les nombres a^- et b^+ à partir de la connaissance de a et b (cette étape se fait en $2n$ opérations). On dénombre ensuite les quantités n^- et n^+ , désignant respectivement le nombre d'éléments inférieurs à a^- et supérieurs à b^+ (à nouveau, cette phase nécessite un nombre linéaire d'opérations). Le nombre d'éléments éliminable (sans changer la médiane) est donc $M = \min(n^-, n^+)$. On retire alors M éléments parmi les nombres inférieurs à a^- , puis M autres parmi les nombres supérieurs à b^+ (ce qui prend également un temps linéaire). Nous savons que la médiane m appartient à l'intervalle $[a^-, b^+]$, et nous avons éliminé un nombre égal d'éléments de part et d'autre. La médiane m est donc égal à la médiane sur le sous-tableau restant. On conclut le travail en appelant (récursevement) le calcul de la médiane sur ce sous-tableau.

Le calcul de la médiane sur un tableau de taille n , nécessite 2 appels récursifs sur des tableaux de taille $n/2$, puis un travail de décomposition/recomposition linéaire $f(n) = \Theta(n)$, et enfin un troisième appel récursif sur le sous-tableau restant. On montre aisément que dans le pire des cas, ce dernier sous-tableau contient également de l'ordre de $n/2$ éléments. L'équation de partition s'exprime alors :

$$C(n) = 3C\left(\frac{n}{2}\right) + \Theta(n)$$

Via le Master Theorem, on trouve une complexité polynomiale : $C(n) = \Theta(n^{1.58})$.

Il s'agit d'une amélioration par rapport à l'algorithme naïf lorsque le tri de liste est effectué par un algorithme à complexité quadratique (par exemple le tri par sélection). En revanche, si le tri de liste est effectué par fusion (complexité dans le pire des cas en $n \log n$), l'algorithme présenté ci-dessus est moins efficace. Nous pouvons toutefois montrer que sa complexité en moyenne est arbitrairement proche d'être linéaire, ce qui représente un avantage indéniable sur l'utilisation du tri fusion.

En moyenne, lorsque la liste est suffisamment grande, les médianes a et b auront tendance à être proches l'une de l'autre, et l'intervalle $[a^-, b^+]$ en sera d'autant plus réduit, ce qui par suite entraîne la réduction de la charge de travail du troisième appel récursif.

On montre en probabilités, que l'écart entre a et b est proportionnel à \sqrt{n} en moyenne, d'où :

$$C(n) = 2C\left(\frac{n}{2}\right) + C(\sqrt{n}) + \Theta(n)$$

Or, pour toute constante $k > 1$ il existe un rang $n_0 \in \mathbb{N}$ tel que pour tout n supérieur à n_0 : $\sqrt{n} \leq \frac{n}{k}$. On peut donc majorer C par un suite T telle que :

$$T(n) = 2T\left(\frac{n}{2}\right) + T\left(\frac{n}{k}\right) + \Theta(n)$$

Le théorème d'Akra-Bazzi nous dit alors qu'il existe ρ tel que :

$$\frac{2}{2^\rho} + \frac{1}{k^\rho} = 1$$

$$T(n) = \Theta\left(n^\rho \left(1 + \int_1^n \frac{x}{x^{\rho+1}} dx\right)\right)$$

En prenant k arbitrairement grand, on peut obtenir un exposant ρ arbitrairement proche de 1 : $\rho = 1 + \varepsilon$. On obtient alors une complexité arbitrairement proche d'être linéaire : $C(n) = \mathcal{O}(n^{1+\varepsilon})$. On trouve un résultat similaire dans l'algorithme de Toom-Cook (ex 3.27). D'autre part, on remarquera que l'algorithme de la médiane des médianes, également fondé sur une approche diviser pour régner, permet d'obtenir une complexité linéaire, moyennant un découpage plus sophistiqué du problème.

Stratégie générale

Nous donnons ci-dessous la méthodologie globale permettant de transformer un algorithme itératif suivant le paradigme diviser pour régner.

Étape 1 : on découpe le problème en 2 (ou plus) parties et on suppose connaître la solution sur chacun des sous-problèmes.

Étape 2 : on étudie si la solution du problème principal peut être exprimée comme une combinaison de la solution sur chaque sous partie, plus éventuellement d'un terme d'ajustement faisant intervenir les deux parties (en général, si cet ajustement est trop coûteux en temps de calcul, c'est que le découpage choisi à l'étape 1 n'est pas judicieux ou alors que l'approche *diviser pour régner* n'est pas adaptée au problème considéré).

Étape 3 : on étudie le cas de base de la récurrence, i.e. on montre que l'on peut donner facilement la solution d'un sous-problème élémentaire

Étape 4 : on évalue le coût $f(n)$ de décomposition/recomposition à l'étage n .

Étape 5 : on utilise le Master Theorem (les valeurs de a et b dépendent de l'étape 1), pour vérifier que l'approche possède une complexité plus faible que la solution itérative.

3.11 Pour aller plus loin...

Dans certains domaines d'application, il arrive que les concepts de complexités minimale, moyenne et maximale soient insuffisants. Par exemple, dans le cas de l'analyse du comportement des structures de données, il est fréquent de rencontrer une opération qui s'exécute dans la grande majorité des cas en un temps très bref, mais qui parfois nécessite l'agrandissement, ou la réorganisation de ladite structure.

Un exemple simple est donné par l'insertion d'éléments dans un tableau de taille fixée égale à n . Chaque insertion se fait en un temps $\mathcal{O}(1)$, excepté lorsque le tableau est rempli, auquel cas il est nécessaire de migrer ses éléments vers un second tableau de taille supérieure. Cette opération nécessite un temps proportionnel au nombre d'éléments déjà présents dans le tableau, soit $\mathcal{O}(n)$. Si le nouveau tableau est pris de taille $n + 1$, alors il est clair que la procédure d'ajout est extrêmement inefficace puisque chaque ajout nécessite une migration de l'ensemble des éléments. On vérifie mathématiquement, qu'une règle suffisante pour garantir une complexité moyenne en $\mathcal{O}(1)$, consiste à prendre un nouveau tableau de taille $2n$, impliquant ainsi qu'une nouvelle migration survient d'autant plus tardivement qu'elle est coûteuse en temps de calcul. L'analyse de la *complexité amortie* permet de donner un cadre théorique à ce type de considérations. Nous verrons des exemples concrets de ces structures de données dans le chapitre suivant.

Un second cas problématique apparaît lorsqu'on considère des algorithmes dont la complexité maximale théorique est relativement mauvaise, mais dont le comportement en pratique est plutôt bon. C'est le cas par exemple de l'algorithme du simplexe pour l'optimisation linéaire, qui possède une complexité maximale exponentielle, mais pour lequel cette borne supérieure est rencontrée sur des exemples extrêmement atypiques et peu réalistes en pratique. D'autre part, la notion de complexité moyenne n'est pas satisfaisante non plus pour deux raisons : d'une part elle est difficile à évaluer en théorie, et d'autre part, il est bien souvent illusoire de pouvoir donner une distribution de probabilité des instances rencontrées en pratique. Comme souligné par Edelman (1992) : *What is a mistake is to psychologically link a random matrix with the intuitive notion of a "typical" matrix or the vague concept of "any old matrix". In contrast, we argue that "random matrices" are very special matrices*, jouant ainsi sur la polysémie du terme anglais *random* qui signifie à la fois *aléatoire*, mais aussi *quelconque* et *arbitraire*. L'analyse de la *complexité lisse*, introduite au début des années 2000, consiste à évaluer la valeur maximale de la complexité moyennée sur des perturbations des données. Lorsque les entrées qui pénalisent l'algorithme sont extrêmement peu fréquentes, la complexité lisse est un meilleur représentant du comportement pratique de l'algorithme. Par exemple, la méthode du simplexe, de complexité maximale exponentielle, possède une complexité lisse polynomiale, beaucoup plus révélatrice de ses capacités réelles. On pourra trouver plus d'informations dans Spielman et Shang-Hua (2009).

3.12 Conclusions

Nous conclurons cette partie sur une remarque générale.

Si la complexité fait généralement office d'indicateur pour comparer plusieurs algorithmes effectuant la même tâche, elle peut aussi être très utile pour organiser plus efficacement le

programme global.

Lorsque l'on doit trier une liste d'intensités de pixels dans une image de taille 5000 x 5000, les temps de calculs observés se rapprochent des résultats asymptotiques calculés théoriquement.

La stratégie générale à suivre consiste à découper tant que possible les données destinées à être traitées par des algorithmes en temps exponentiel (on inclut ici également dans cette définition les complexités polynomiales, et d'une manière générale toute complexité supralinéaire). A l'inverse, lorsque l'algorithme à appliquer s'exécute en un temps sublinéaire, on aura tout intérêt à essayer de traiter des batchs de données les plus gros possibles.

3.13 Exercices

Exercice 3.6. Algorithme quasi-polynomial *

Un algorithme de complexité f est dit *quasi-polynomial* si pour tous $\alpha, \beta \in]1; +\infty[$ on a : $f(n) \in \Omega(n^\alpha) \cap \mathcal{O}(\beta^n)$. Montrer que $f(n) = n^{\log n}$ est quasi-polynomiale.

Exercice 3.7. Test de primalité *

Ecrire un algorithme prenant en entrée un nombre entier n positif et retournant le booléen *vrai* si n est premier et *faux* sinon. On indiquera la squelette et le degré d'emboîtement du code. Evaluer la complexité de l'algorithme.

Exercice 3.8. Evaluation d'une suite récurrente d'ordre 3 *

Reprendre l'exemple d'introduction et appliquer la même méthode pour écrire un code ADL efficace du calcul des termes de la suite définie par :

$$a_0 = a_2 = 1 \quad a_1 = 2 \quad \forall n \in \mathbb{N} \quad a_{n+3} = 2a_{n+2} + 5a_{n+1} - 6a_n$$

On indiquera sa complexité algorithmique ainsi que le gain par rapport à la méthode récursive naïve.

Exercice 3.9. Master Theorem *

Démontrer qu'une complexité obéissant à l'équation de récurrence suivante est en $\Theta(n^2 \log n)$:

$$C(n) = 9T(n/3) + n^2 + 2n + 1$$

Qu'advient-il si on parvient à supprimer l'un des sous-problèmes ?

Exercice 3.10. *Application du Master Theorem* *

Indiquer l'ordre de complexité contrôlé par chacune des équations de partition suivantes :

- | | |
|--------------------------------------|---------------------------------|
| 1. $C(n) = 2C(n/2) + 1$ | 2. $C(n) = 2^n C(n/2) + n^n$ |
| 3. $C(n) = 3C(n/3) + n$ | 4. $C(n) = 3C(n/2) + n^2$ |
| 5. $C(n) = 2C(n/2) + n \log n$ | 6. $C(n) = 9C(n/3) + n$ |
| 7. $C(n) = 7C(n/2) + n$ | 8. $C(n) = 5C(n/3) + n$ |
| 9. $C(n) = 0.5C(n/2) + n \log n$ | 10. $C(n) = 27C(n/3) + 3n^3$ |
| 11. $C(n) = 6C(n/3) + n^2 \log n$ | 12. $C(n) = 8C(n/2) - n \log n$ |
| 13. $C(n) = 2C(n/2) + 10^n + n!$ | 14. $C(n) = 2C(n/4) + n^{0.51}$ |
| 15. $C(n) = \sqrt{2}C(n/2) + \log n$ | 16. $C(n) = 3^n C(n/3) + n^3$ |

Exercice 3.11. *Tours de Hanoï* *

On revient sur le problème des Tours de Hanoï, décrit dans le chapitre 2.

D'après l'exposé original du problème par le mathématicien Edouard Lucas, la tour sacrée du Brahmâ est composée de 64 disques, disposés par Dieu au commencement et que des ouvriers s'évertuent à déplacer d'un support à l'autre (dans cette version, les supports sont des aiguilles de diamant). Selon cette légende, la fin du Monde arrivera lorsque tous les disques auront été déplacés.

En admettant que les ouvriers déplacent un disque par seconde, calculer l'espérance de vie du Monde décrit par Lucas.

Exercice 3.12. *Décompte de mots* *

On se donne une chaîne de caractères, dans laquelle on souhaite connaître le nombre d'occurrences d'un mot m .

Q1. Ecrire un algorithme prenant en entrée la chaîne s ainsi que m et renvoyant le nombre de fois où m est trouvé dans s . On proposera une solution itérative et une solution du type diviser pour régner.

Q2. Calculer la complexité de chacune des deux solutions et conclure quant à l'intérêt de la stratégie *diviser pour régner* dans ce cadre d'application.

Exercice 3.13. *Complexité du calcul du plus grand diviseur commun* *

Ecrire une fonction récursive renvoyant le plus grand diviseur commun de 2 entiers $a \leq b$ et donner sous forme de majoration \mathcal{O} , sa complexité. On prendra $n = a$ comme unité de taille du problème à traiter et la division modulaire comme coût d'exécution. Indication : on pourra commencer par montrer que pour $a > b$: $a \bmod b \leq \frac{a}{2}$.

Exercice 3.14. Boucles imbriquées ★

Indiquer l'ordre de complexité (en n) du programme (on pourra s'aider de l'annexe B) :

NESTED _ LOOPS($n \downarrow, y \uparrow$) : $\llbracket s \leftarrow 0; b \leftarrow n * n * n; \left\{ \begin{smallmatrix} 0 \\ i:1 \\ j:i*i*i,-1 \end{smallmatrix} \right. y \leftarrow y + 2 \right\}_j \left\} _i \rrbracket$

Exercice 3.15. *Encadrement des loyers* *

On considère une liste L de nombres triés dans l'ordre croissant. On pourra imaginer par exemple, que L représente la liste des loyers (en €) d'une sélection d'appartements rentrée par le moteur de recherche d'une agence immobilière. Un locataire potentiel souhaite visiter un appartement : il se fixe comme critère un loyer maximal b (soit un tiers de ses revenus mensuels) ainsi qu'un loyer minimal a (s'il considère par exemple que les loyers trop faibles sont susceptibles de correspondre à des arnaques).

Q1. Ecrire un algorithme naïf, prenant en entrée la liste L de nombres triés dans l'ordre croissant ainsi que les critères a et b , et renvoyant l'indice d'un loyer compris entre a et b . S'il n'existe pas de tel loyer dans la liste, la fonction devra renvoyer la valeur 0. Quelle est la complexité de l'algorithme dans le pire des cas ?

Q2. En utilisant le principe *diviser pour régner*, proposer un second algorithme et évaluer sa complexité pour montrer qu'il est plus efficace.

Exercice 3.16. Somme en ligne ★★

On considère un tableau T de n nombres réels (positifs ou négatifs). On souhaite calculer la somme maximale qui peut être obtenue avec une séquence d'éléments consécutifs. Par exemple, considérant le tableau de données :

$$T = [5, 1, -3, \mathbf{7}, \mathbf{-2}, \mathbf{4}, \mathbf{8}, -4, 3, 2, -1, 5, 1, 3]$$

la réponse au problème est $17 = T_4 + T_5 + T_6 + T_7$, obtenue avec le couple $(4, 7)$.

Q1. Ecrire un algorithme naïf, prenant en entrée le tableau T et retournant les indices i et j ($i \leq j$), ainsi que la valeur S^* maximisant la quantité : $S = \sum_{k=i}^j T_k$, pour $i, j \in \{1, 2, \dots, n\}$.

Evaluer la complexité de l'algorithme.

Q2., Pour accélérer la résolution du problème, on se propose de calculer en amont l'intégrale du tableau T , i.e. un tableau I , tel que, pour tout indice $t \in \{1, 2, \dots, n\}$, l'élément I_t vaut la somme des éléments de T situés à gauche de la cellule t . Formellement :

$$I_t = \sum_{k=1}^t T_k$$

Ecrire une fonction **INTEGRATE** permettant de calculer l'intégrale d'un tableau T .Modifier l'algorithme de la question précédente à l'aide de la fonction **INTEGRATE** et réévaluer sa complexité.

Q3. En utilisant le principe *diviser pour régner*, proposer un troisième algorithme et évaluer sa complexité pour montrer qu'il est plus efficace que les deux méthodes proposées dans les questions précédentes.

Exercice 3.17. Eliminer pour régner **

Un jeu télévisé voit s'affronter n candidats sur une île déserte. A la fin de chaque phase de jeu, les candidats doivent voter pour éliminer celui qu'ils jugent être le plus mauvais d'entre eux. Le nombre de votes différents est donc au plus égal au nombre de candidats. Si un candidat remporte strictement plus de la moitié des voix, il est éliminé.

Q1. Démontrer qu'au maximum, 1 candidat est éliminé à l'issue du vote.

On représente le vote des candidats par une liste d'éléments $[c_1, c_2, \dots, c_n]$, où $c_i \in \{1, 2, \dots, n\}$ représente le vote du i -eme candidat.

Q2. Ecrire une fonction **IS_ELIMINATED**, prenant en entrée la liste des votes et un candidat x quelconque, et retournant (à l'aide d'un booléen) si x est éliminé à l'issue de la phase de votes.

Q3. A l'aide de la fonction écrite à la question précédente, écrire un algorithme itératif permettant (sans avoir recours à un tableau annexe) de déterminer l'identifiant (entre 1 et n) du candidat éliminé. Si aucun candidat n'est éliminé, l'algorithme devra retourner la valeur 0. Calculer la complexité en temps de l'algorithme, en fonction du nombre n de candidats.

Fort de leur succès commercial, les producteurs de l'émission décident de créer une version en ligne du jeu, où les candidats s'affrontent sur une île virtuelle. Le système de vote reste le même, mais le nombre de candidats est cette fois potentiellement très élevé. On s'interroge alors sur une manière de résoudre la phase de vote plus efficacement.

Q4. En exploitant le paradigme *diviser pour régner*, écrire un algorithme permettant de déterminer si un candidat est éliminé à l'issue de la phase de vote et, le cas échéant, de retourner son identifiant. Calculer la complexité de l'algorithme.

Exercice 3.18. Suite de Fibonacci **

Dans cet exercice, nous proposons un algorithme de type diviser pour régner pour calculer les termes de la suite de Fibonacci :

FIBO($n \downarrow$) : $\llbracket n = 0 \cup n = 1 ? \text{ FIBO} \leftarrow 1; ! \mid \ddot{\wedge}$

```


$$\text{MOD}(N, 2) = 1 ? \quad M \leftarrow \text{FLOOR}(N/2); \text{FIBO} \leftarrow \text{FIBO}(M) * (\text{FIBO}(M+1) + \text{FIBO}(M-1)) \mid a \leftarrow \text{FIBO}(n/2); b \leftarrow \text{FIBO}(n/2 - 1); \text{FIBO} \leftarrow a * a + b * b; \dots \quad ]]$$


```

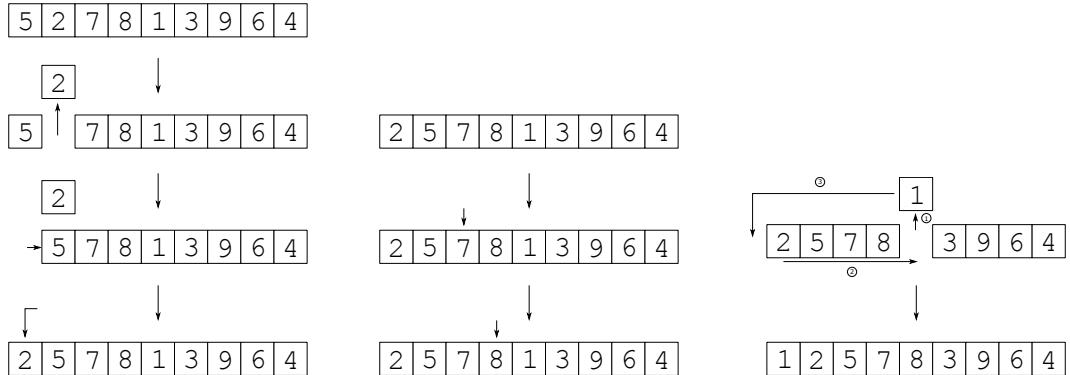
Apporter les preuves de terminaison et de correction du programme et évaluer sa complexité dans le pire des cas. On prendra comme unité de coût le nombre d'additions.

Comparer le résultat obtenu avec la complexité du calcul des termes de la suite obtenus par les méthodes précédemment développées (réécriture linéaire d'ordre 2, produit matriciel, produit matriciel avec exponentiation rapide). Conclusion ?

Exercice 3.19. *Tri par insertion* **

Nous étudions ici un nouveau type de tri que l'on appelle *tri par insertion*. Il s'agit de procéder d'une manière similaire au classement des cartes dans une main au bridge par exemple.

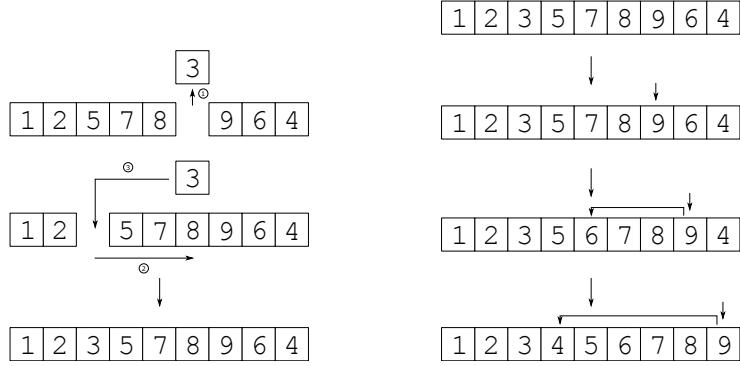
Le tri s'effectue en considérant les éléments un par un, en partant du deuxième élément (image de gauche ci-dessous).



On recherche alors la position de l'élément 2 dans la sous liste située à gauche de cet élément. Cette sous liste n'est composée que d'un unique élément (5), le 2 est donc renvoyé complètement en tête de liste. On passe alors ensuite au troisième élément (figure du centre ci-dessus), qui vaut 7 et qui est donc à la bonne place dans la partie gauche de la liste. On passe au quatrième (8) qui est également à la bonne place, puis au suivant (figure de droite) que l'on doit cette fois basculer en tête de liste puisqu'il est plus petit que tous les éléments de la partie gauche de la liste.

On passe alors à l'élément suivant (3) qui doit être inséré entre le 2 et le 5, et ainsi de suite jusqu'à ce que la suite soit complètement triée, comme illustré sur la figure ci-dessous.

Q1. Ecrire un module permettant d'appliquer le tri par insertion sur une liste. On indiquera le squelette et le degré d'emboîtement du code.



Q2. Evaluer les complexités (minimale, maximale et moyenne) du code écrit, en considérant le nombre de comparaison comme unité de coût.

Q3. Conclure quant à l'efficacité de cet algorithme par rapport aux tris par sélection et par fusion, étudiés dans ce chapitre.

Q4. Reprendre les questions Q2 et Q3 en considérant cette fois l'affectation comme unité de coût.

Exercice 3.20. *Tri à bulles* ******

Le tri à bulle consiste à parcourir tous les couples d'éléments consécutifs, et à les échanger lorsque l'élément de gauche est plus grand que l'élément de droite (dans le cas d'un tri dans l'ordre croissant). On réitère l'opération, jusqu'à ce que le tableau soit trié.

Q1. Ecrire un algorithme permettant d'implémenter le tri à bulles. On prêtera attention à la condition d'arrêt en observant qu'après la i -eme étape, les i derniers éléments sont à leur place finale. On pourra également utiliser un identificateur booléen, pour arrêter prématièrement le tri si aucune interversion n'a été opérée sur la liste.

Q2. Démontrer que le code produit termine et est correct.

Q3. Reprendre les questions 2 et 3 de l'exercice précédent.

Exercice 3.21. *Inversion matricielle* ******

Soit $A \in \mathbb{R}^{n \times n}$ une matrice carrée de taille n . Lorsqu'elle existe, on note A^{-1} la matrice inverse de A , i.e. l'unique matrice telle que $AA^{-1} = A^{-1}A = I_n$. On suppose que n est pair et on donne l'identité suivante (qui pourra aisément être vérifiée à la main) :

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

avec A, B, C et D les matrices block d'ordre $\frac{n}{2}$.

Dans cet exercice, on suppose que $n = 2p$ avec $p \in \mathbb{N}$. Par ailleurs, on rappelle que l'algorithme d'inversion par la méthode du pivot de Gauss, possède une complexité en $\Theta(n^3)$.

Q1. Soient $A, B \in \mathbb{R}^{n \times n}$. Donner la complexité en temps $M(n)$ du calcul de AB par l'algorithme de multiplication matricielle classique.

Q2. Montrer que :

$$\begin{bmatrix} I_n & A \\ I_n & B \end{bmatrix}^{-1} = \begin{bmatrix} I_n & -A & AB \\ & I_n & -B \\ & & I_n \end{bmatrix}$$

Q3. A l'aide de l'identité démontrée à la question précédente, justifier que l'inversion est a priori au moins aussi difficile que la multiplication matricielle.

Q4. En utilisant le principe *diviser pour régner*, et à l'aide de l'identité donnée dans l'en-tête de cet exercice, montrer qu'il est possible de concevoir un algorithme d'inversion matricielle de complexité $C(n) = \Theta(M(n))$. On ne demande pas d'écrire le pseudo-code correspondant.

La compilation des réponses apportées aux questions 3 et 4 permet de montrer que les opérations d'inversion et de multiplication matricielles possèdent la même complexité.

Q5. En déduire que l'inversion matricielle peut-être effectuée avec une complexité en temps (strictement) meilleure que celle du pivot de Gauss.

Indication : on pourra utiliser l'algorithme de Strassen présenté en page 121.

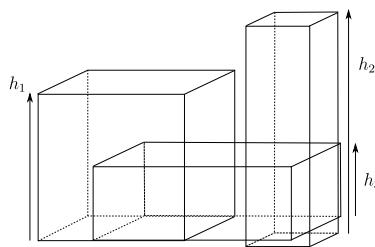
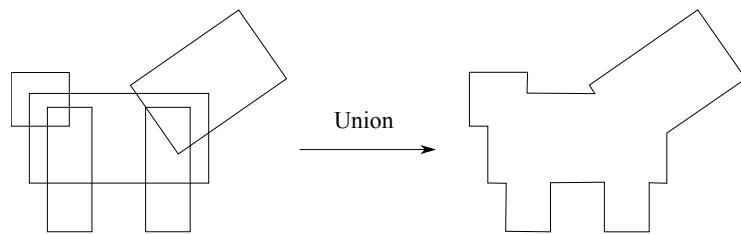
Exercice 3.22. Calcul de la surface de plancher constructible **

Dans cet exercice, on cherche à résoudre un problème pratique de géomatique.

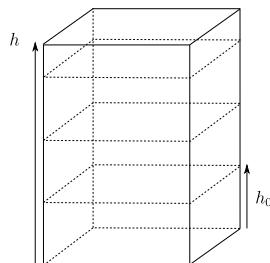
On dispose d'un programme générateur de bâtiments, prenant en entrée un ensemble de règles d'urbanisme, un ensemble de parcelles, un ensemble de bâtiments et un indice de parcelle p , et retournant le "plus grand" bâtiment constructible sur la parcelle p , compte-tenu de son environnement. De manière plus détaillée, l'algorithme détermine ce bâtiment en optimisant un agencement de solides individuels, chaque solide étant défini par un polygone de base (rectangulaire pour la clarté des figures ci-dessous) et une hauteur.

Le bâtiment généré est alors l'union des solides élémentaires.

Etant donnée une hauteur standard d'étage h_0 , on définit alors la surface de plancher constructible d'un bâtiment comme le nombre d'étages multiplié par la surface de la base



du bâtiment. Par exemple, dans la figure ci-après, cette surface vaut 3 fois l'aire de la base du parallélépipède. Notons qu'un étage doit faire exactement une hauteur h_0 , ainsi l'étage résiduel n'est pas compté dans le calcul.



On modélise chaque solide élémentaire comme une liste de valeurs contenant les coordonnées des sommets de la base, et on place en fin de liste, la hauteur du bâtiment. D'autre part, on suppose avoir à disposition une fonction INTERSECTION, prenant en entrée deux polygones (sous forme de listes de flottants) et retournant leur intersection¹⁰, également sous forme de liste. Nous supposerons avoir également à disposition une fonction UNION (pour calculer la réunion de deux polygones) et DIFF (pour calculer l'ensemble constitué de tous les points du plan appartenant au premier polygone mais pas au second).

La hauteur d'un étage standard h_0 est une constante du problème

10. Notons que les bases ne sont pas nécessairement convexes, et l'intersection peut être un multi-polygone, ce qu'il faudrait gérer dans le format des listes, mais dont nous ferons abstraction ici.

Q1. Ecrire un algorithme itératif (ajoutant les solides un à un dans le calcul) et permettant de calculer la surface de plancher constructible du bâtiment total. On apportera les preuves de terminaison et de correction de l'algorithme.

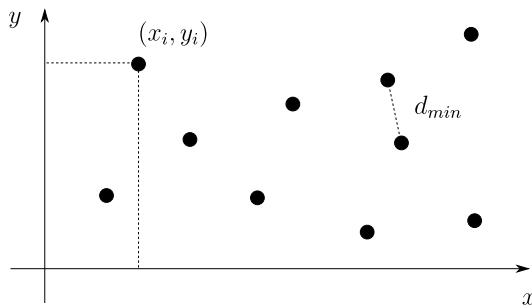
Q2. Trouver la complexité du programme, en prenant comme unité de taille des données le nombre n de solides élémentaires, et comme unité de coût le nombre d'appels à la fonction d'intersection.

Exercice 3.23. *Distance minimale entre points ****

Nous poursuivons ici l'exercice 1.22.

On considère un semis de n points sous forme d'un tableau T à n lignes et 2 colonnes. Chaque point est représenté sous forme d'un couple de coordonnées (x, y) , où x est stocké dans la première colonne et y dans la deuxième colonne.

L'objectif du problème est de calculer la distance la plus courte entre les couples de points.

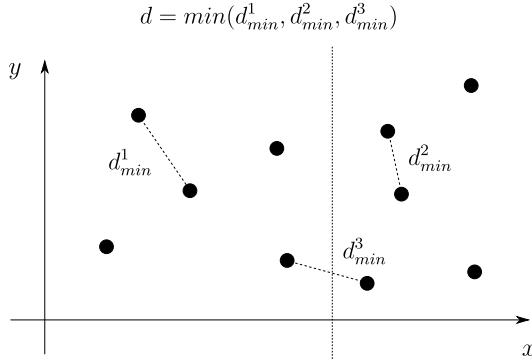


Q1. Ecrire une fonction itérative MIN_DIST, prenant en entrée le tableau T et renvoyant les indices du couple de points les plus proches l'un de l'autre, ainsi que la distance les séparant. Evaluer la complexité de l'algorithme.

Q2. Pour des semis contenant plusieurs millions de points, on a besoin d'une fonction plus efficace. Nous allons donc tenter une approche *diviser pour régner*. Dans un premier temps, on prétraite les données, en créant deux copies du tableau T : Tx (resp. Ty) dans lequel les points sont entrés dans l'ordre croissant de leur abscisse (resp. ordonnée). Quel(s) algorithme(s) de tri est-il judicieux d'utiliser ici ? Dans les questions suivantes, nous supposerons avoir à disposition un module TRI permettant de créer Tx et Ty à partir de T .

Q3. Pour mener une approche de type diviser pour régner, nous devons à chaque itération séparer le jeu de points en deux parties à peu près égales en coupant au niveau d'une abscisse médiane (cf figure ci-dessous). Notons que le choix de couper suivant les abscisses plutôt que les ordonnées est complètement arbitraire.

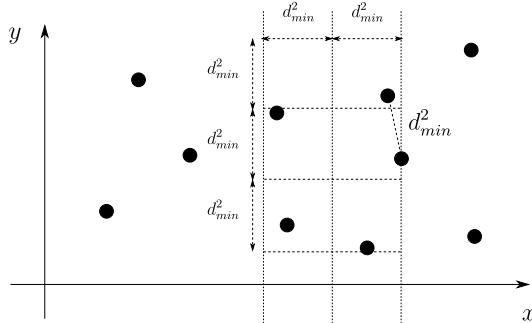
Grâce au pré-tri Tx , la séparation est rapide. Ecrire une fonction prenant en entrée le tableau T ainsi qu'une abscisse médiane x_m et renvoyant deux tableaux $T1$ et $T2$, correspondant aux deux parties créées.



Q4. La démarche générale de l'algorithme va alors être de calculer le couple de voisins les plus proches (ainsi que la distance associée) sur chacun de deux sous-problèmes \mathcal{P}_1 et \mathcal{P}_2 . Ceci donnera deux distances minimales d_{min}^1 et d_{min}^2 . On évalue alors $d = \min(d_{min}^1, d_{min}^2)$.

Si un couple de points est séparé d'une distance inférieure à d alors, cela signifie que l'un des points est situé dans le jeu de points de gauche, tandis que l'autre est situé dans le jeu de droite. Nous devons donc trouver la distance minimale d_{min}^3 de l'ensemble des couples (p_1, p_2) avec $p_1 \in \mathcal{P}_1$ et $p_2 \in \mathcal{P}_2$. Calculer la complexité d'une telle opération.

Q5. Pour accélérer le calcul de d_{min}^3 , on se place uniquement dans une bande de largeur d et centrée sur l'abscisse du découpage.



Ecrire une fonction prenant en entrée le tableau T ainsi que l'abscisse du découpage x_m et la distance d , et qui retourne la liste des points de T_y situés dans la bande. La sortie de la fonction devra conserver l'ordre de tri de T_y .

Q6. Montrer qu'une tranche de hauteur d de cette bande contient au maximum un nombre q de points, que l'on devra déterminer. En déduire que si deux points de la bande sont plus proches l'un de l'autre que la distance d , alors ils sont séparés de q cellules ou moins dans le tableau retourné à la question précédente. En utilisant cette observation, écrire un module permettant de rechercher de manière efficace si la bande contient un couple de points séparés par une distance inférieure à d .

Q7. En déduire un algorithme récursif, utilisant les modules écrits dans les questions précédentes et utilisant le principe diviser pour régner pour résoudre le problème.

Q8. Evaluer la complexité de l'algorithme et la comparer à l'approche de la question 1.

Exercice 3.24. *Optimalité de Pareto ****

On considère un ensemble d'objets évalués suivant deux critères x et $y \in \mathbb{R}^+$, que l'on suppose d'autant plus favorables qu'ils sont proches de zéro. Par exemple, chaque point représente un même produit commercial vendu par différentes enseignes, x est son prix et y est son taux de défaillance dans l'année qui suit l'achat.

Les offres du marché sont généralement telles que les produits les moins chers sont aussi ceux qui sont de moins bonne qualité, et donc qui auront le plus haut taux de défaillance. Il est donc bien souvent impossible de trouver un produit qui batte tous ses concurrents sur les deux critères retenus. Pour aider l'acheteur, on se propose d'extraire l'ensemble des points optimaux au sens de Pareto parmi le jeu de données.

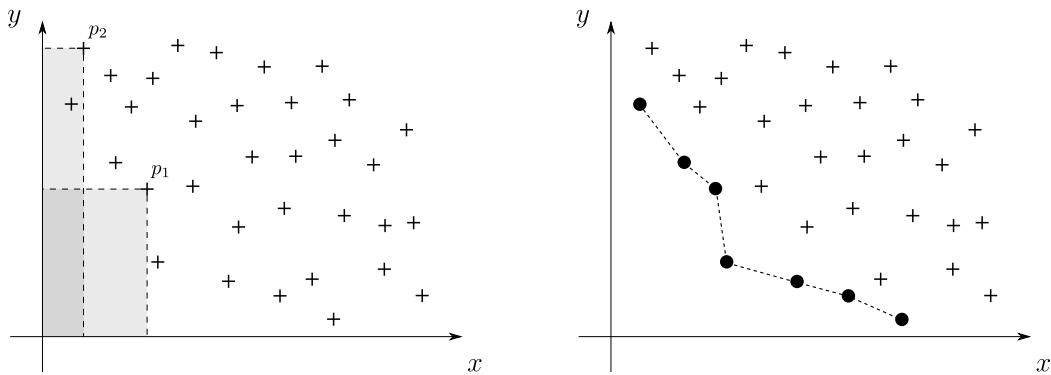


FIGURE 3.1 – Ensemble de points de $\mathbb{R}^+ \times \mathbb{R}^+$ (à gauche) et front de Pareto (à droite).

Définition 1. On dit qu'un point p_1 est uniformément meilleur qu'un point p_2 (et on note $p_1 \preceq p_2$) si et seulement si p_1 est au moins aussi bon que p_2 sur tous les critères :

$$p_1 \preceq p_2 \iff x_1 \leqslant x_2 \text{ et } y_1 \leqslant y_2$$

Définition 2. On dit qu'un point p est optimal au sens de Pareto (ou Pareto-optimal) si et seulement si il n'existe pas de point q tel que $q \preceq p$. Par exemple sur la figure 1, p_1 est Pareto-optimal, contrairement à p_2 qui ne l'est pas puisque le rectangle dont il est le coin supérieur droit contient un autre point du jeu de données.

Définition 3. On appelle front de Pareto l'ensemble des points optimaux au sens de Pareto.

Dans ce problème, les données seront modélisées par 3 vecteurs X , Y et I (de même taille) listant respectivement les critères x et y des objets, et leurs identifiants (par exemple le

nom des objets).

Q1. Ecrire une fonction naïve PARETO, prenant en entrée 3 vecteurs X, Y, I et retournant un vecteur contenant l'ensemble des identifiants des points Pareto-optimaux. Indiquer la complexité de l'algorithme (on prendra comme unité de coût le nombre d'opérations élémentaires : comparaisons, additions...).

Dans les questions suivantes, nous chercherons à montrer qu'il est possible de concevoir un algorithme plus rapide en utilisant le principe *diviser pour régner*.

Considérons la partition suivante : étant donnée y_{moy} la moyenne arithmétique des coordonnées y des points, P_{inf} (resp. P_{sup}) désigne l'ensemble des points dont la coordonnée y est inférieure (resp. supérieure) à y_{moy} . De plus, la partie P_{sup} est elle-même divisée en P_{gauche} (resp. P_{droite}) l'ensemble des points dont la coordonnée x est inférieure (resp. supérieure) à la coordonnée x minimale de P_{inf} (que l'on note x_{min} sur la fig. 2).

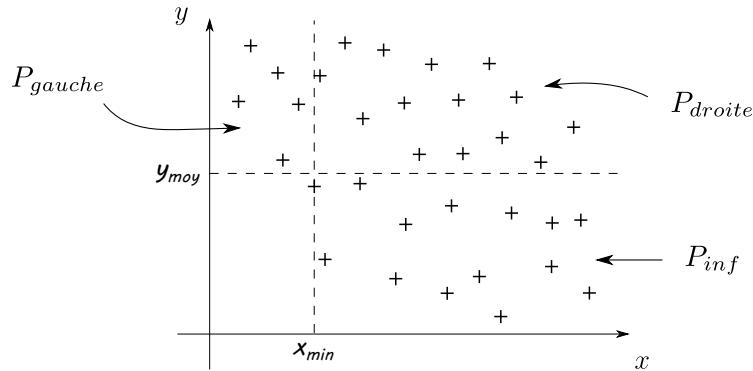


FIGURE 3.2 – Division de l'ensemble des points en 3 parties

Q2. Justifier que P_{droite} ne contient aucun point Pareto-optimal. En déduire une fonction récursive PARETO_REC, calculant l'ensemble des points Pareto-optimaux, en travaillant uniquement sur P_{inf} et P_{gauche} . Pour faciliter la manipulation des cas de bases, on pourra utiliser le symbole \emptyset pour désigner une liste vide.

Indication : on pourra commencer par écrire deux fonctions MIN et MOY calculant respectivement le minimum et la moyenne d'un vecteur de nombres.

Q3. On suppose que la distribution des points est homogène. En observant que sous cette condition, l'une des parties de la division est presque vide, justifier soigneusement que l'équation de partition est de la forme :

$$C(n) = C\left(\frac{n}{2}\right) + \Theta(n)$$

En déduire l'ordre de la complexité de cette deuxième approche.

Q4. Supposons maintenant quelconque la répartition des points. Montrer que dans le pire des cas, la complexité de l'algorithme est en $\Theta(n^2)$.

Q5. Afin d'optimiser les performances de l'algorithme dans le pire des cas, on propose ici une troisième approche, en redéfinissant le seuil y_{moy} comme étant la médiane (et non plus la moyenne) des coordonnées y de l'ensemble de points. Ecrire une fonction MED, prenant en entrée une liste de nombres et retournant sa médiane. On supposera avoir à disposition une fonction de tri de liste TRI (qu'on ne demande pas de réécrire). On précisera toutefois l'algorithme de tri utilisé ainsi que sa complexité (dans le pire des cas).

Démontrer que la complexité dans le pire des cas est à présent meilleure que quadratique.

Q6. A l'aide du théorème d'encadrement, en déduire la complexité moyenne de cette troisième approche.

Exercice 3.25. Comparaison de préférences ***

On dispose d'un jeu de traces GPS issues d'une population animale quelconque (sans importance pour la question qui nous occupe) à partir duquel on a extrait n points d'intérêt P_1, P_2, \dots, P_n , caractérisant les positions géographiques fréquemment visitées par les animaux observés. On associe chaque trajectoire du jeu de données à une liste des n points d'intérêt, triés dans l'ordre décroissant de la fréquence des visites. Par exemple, pour un total de 6 points d'intérêt, une trajectoire associée à la liste $(4, 2, 1, 3, 5, 6)$ signifie que le point P_4 est le plus visité par la trace, suivi du point P_2 , puis P_1, P_3 , etc.

Pour pouvoir répartir les animaux en plusieurs groupes, on a besoin de se doter d'une métrique pour évaluer la distance entre les listes ordonnées de points d'intérêt. On utilise la définition suivante :

Définition. Etant données deux listes L et M de points d'intérêt classés par ordre de préférence, la distance $d(L, M)$ entre les listes L et M est définie par le nombre de couples d'entiers $(i, j) \in \{1, 2, \dots, n\}^2$ avec $i < j$, tels que les ordres de préférence relative des points P_i et P_j dans les listes L et M sont différents.

Exemple : prenons l'exemple des listes $L = (4, 3, 1, 2)$ et $M = (3, 1, 4, 2)$. Alors $d(L, M) = 2$ puisqu'on peut dénombrer deux couples (i, j) , avec $i < j$, tels que les ordres de préférences de P_i et P_j soient inversés dans les listes L et M , nomément $(1, 4)$ et $(3, 4)$. On peut représenter le résultat sous forme matricielle, où 0 (resp. 1) indique l'absence (resp. la présence) d'inversion des préférences entre les deux listes.

i/j	1	2	3	4
1	-	0	0	1
2	-	-	0	0
3	-	-	-	1
4	-	-	-	-

Remarquons dans un premier temps qu'il est toujours possible sans perte de généralité, de réindexer les noms de points d'intérêt, de sorte à ce que la première liste soit triée dans

l'ordre des indices : $L = (1, 2, \dots, n)$. Dans l'exemple ci-dessus, on est réduit à calculer la distance entre $L = (1, 2, 3, 4)$ et $M = (2, 3, 1, 4)$. La liste L étant fixée, la distance $d(L, M)$ devient alors une fonction de M uniquement (que l'on notera $d_0(M)$), et son évaluation est réduite à la recherche des inversions dans M par rapport à l'ordre canonique de \mathbb{N} . On dénombre 2 inversions : $(1, 2)$ et $(1, 3)$, et on retrouve donc bien : $d_0(M) = 2 = d(L, M)$.

Q1. On considère la liste $M = (2, 4, 1, 3, 6, 5)$. Calculer $d_0(M)$.

Q2. Ecrire un algorithme prenant en entrée une liste M (de taille n quelconque), et renvoyant $d_0(M)$. Evaluer la complexité de l'algorithme.

Dans les questions suivantes, on cherche à optimiser la fonction de calcul de distance à l'aide du principe *diviser pour régner*. Considérons une liste M (de taille n quelconque). On peut séparer M en deux parties M_g et M_d (de tailles à peu près égales en fonction de la parité de n). Par exemple, pour $M = (2, 4, 1, 3, 6, 5)$, on a $M_g = (2, 4, 1)$ et $M_d = (3, 6, 5)$. On peut alors compter séparément les inversions de M_g , les inversions de M_d et les inversions faisant intervenir un élément dans chacune des deux parties M_g et M_d . Pour accélérer la recherche des inversions entre ces deux sous-parties, on pourra trier M_g et M_d à l'aide du tri rapide.

Q3. Illustrer le principe ci-dessus sur l'exemple $M = (2, 4, 1, 3, 6, 5)$. En particulier, on montrera que la recherche des inversions entre les deux sous-parties triées M_g et M_d peut se faire en un temps linéaire.

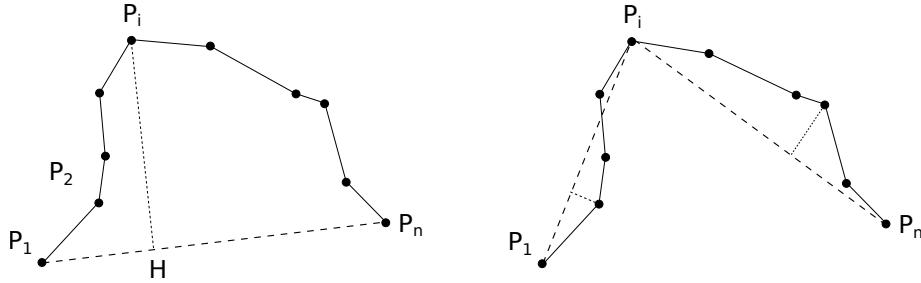
Q4. Montrer que cette solution est plus rapide que l'approche de la question 2.

Exercice 3.26. Simplification de trajectoires ***

Pour cet exercice, on pourra s'inspirer de la démarche utilisée pour l'analyse de complexité du tri rapide, présentée dans la section 4.1.3.

On considère une trajectoire de points GPS, modélisée par deux listes X et Y de n nombres flottants, telles que le couple (X_i, Y_i) désigne les coordonnées planes du i -ème point P_i de la trajectoire.

L'algorithme de Douglas-Peucker permet de simplifier la trajectoire de la manière suivante : on cherche l'indice i (entre 1 et n) du point P_i de la trajectoire dont la distance euclidienne avec le segment P_1P_n est maximale (figure ci-dessous à gauche). Si cette distance P_iH est supérieure à un seuil ε préfixé, on découpe la polylinéenne P_1P_n en deux portions P_1P_i et P_iP_n (figure ci-dessous à droite). On réitère alors l'algorithme récursivement sur chacune des deux sous-portions, jusqu'à ce que la distance maximale entre une polylinéenne et le segment joignant ses extrémités, soit inférieure à la distance seuil ε . À l'issue du processus récursif, on retourne la polylinéenne (simplifiée) : $[P_1, P_{i_1}, P_{i_2}, \dots, P_{i_l}, P_n]$, où les nombres i_1, \dots, i_l correspondent aux indices des points de cassure P_i , rangés dans l'ordre de la polylinéenne originale.



Toutes les questions peuvent être considérées comme indépendantes (c'est-à-dire que l'on pourra si nécessaire appeler le module faisant l'objet d'une question non traitée dans des questions subséquentes).

Q1. Ecrire une fonction POINT_TO_CART prenant en entrée 4 variables $X1, Y1, X2$ et $Y2$, et retournant dans un tableau $[a, b, c]$ les coefficients de l'équation cartésienne $ax + by + c = 0$ décrivant la droite passant par les points $(X1, Y1)$ et $(X2, Y2)$.

Q2. On rappelle que la distance minimale d'un point $p_0 = (x_0, y_0)$ à une droite \mathcal{D} d'équation cartésienne $ax + by + c = 0$ s'exprime par :

$$d(p_0, \mathcal{D}) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Ecrire une fonction DIST_TO_LINE prenant en entrée un tableau $[a, b, c]$ de coefficients d'une droite \mathcal{D} , ainsi que les coordonnées X et Y d'un point P , et retournant la distance minimale entre P et \mathcal{D} .

Q3. Dans cette question, on considère un point P et un segment $[AB]$. Remarquons que la distance entre P et $[AB]$ est égale à la distance entre P et la droite portée par $[AB]$ si et seulement si le triangle PAB possède l'un de ses 2 angles \widehat{PAB} ou \widehat{PBA} obtus. Sinon, cette distance vaut le minimum des distances PA et PB .

Ecrire une fonction DIST_TO_SEG prenant en entrée 4 paramètres $X1, Y1, X2$ et $Y2$ décrivant les extrémités d'un segment \mathcal{S} , ainsi que 2 paramètres $X0, Y0$ décrivant les coordonnées d'un point P arbitraire du plan, et retournant la distance de P à \mathcal{S} .

Q4. Ecrire une fonction GET_MAX_DIST prenant en entrée 2 tableaux X et Y décrivant une polylinéaire, et retournant un vecteur $[i, d]$, contenant l'indice i du point de la polylinéaire situé à la distance maximale du segment joignant les extrémités de la polylinéaire (avec d cette distance maximale associée).

Q5. Montrer que la complexité de la procédure GET_MAX_DIST est linéaire. On pourra prendre comme unité de coût une opération algébrique de bas niveau (comparaison, addition...), que l'on spécifiera.

Q6. Ecrire une fonction récursive DP, prenant en entrée 2 tableaux X et Y décrivant une polylinéaire, ainsi qu'un seuil ε , et retournant la polylinéaire simplifiée par l'algorithme

de Douglas-Peucker (le format de sortie de l'algorithme sera choisi de manière à faciliter l'expression de la récursion).

Q7. Justifier que la complexité de l'algorithme DP obéit à une équation de récurrence du type :

$$C(n) = C(i) + C(n - i + 1) + \Theta(n)$$

Q8. Dans un cas idéal, i vaut de l'ordre de $n/2$ à chaque étape de la récursion. Etablir l'équation de partition sur C et en déduire la complexité de l'algorithme de Douglas-Peucker dans ce cas favorable.

Q9. En s'inspirant de la question 8), établir la complexité de l'algorithme dans le cas le plus défavorable.

Q10. On étudie à présent la complexité moyenne de l'algorithme, en supposant qu'à chaque division du problème en deux sous-parties, la position du point de coupure est distribuée aléatoirement et uniformément sur l'ensemble des sommets (hors-extrémités) de la polylinéaire. Justifier soigneusement que l'équation de partition s'écrit à présent :

$$C(n) = \frac{2}{n-2} \sum_{i=1}^{n-1} C(i) + \Theta(n)$$

Q11. Démontrer que : $(n-2)C(n) = (n-1)C(n-1) + \Theta(n)$

Indication : on pourra commencer par exprimer $(n-2)C(n) - (n-3)C(n-1)$.

Q12. On pose $U_n = \frac{C(n+1)}{n}$ ($n \in \mathbb{N}^*$). Démontrer que l'équation de récurrence sur U_n s'exprime par :

$$U_n = U_{n-1} + \Theta\left(\frac{1}{n}\right)$$

Q13. En déduire l'ordre de croissance de U_n , puis la complexité moyenne de l'algorithme de Douglas-Peucker.

Q14. Discuter physiquement de l'allure des polylignes pour les cas respectivement les plus favorables 7) et défavorables 8). On pourra proposer un exemple schématique de tracé pour chacun de ces 2 cas de figure.

Exercice 3.27. *Calcul des premières décimales de π ★★*

Dans cet exercice, on s'intéresse au calcul des n premières décimales du nombre π .

Dans le chapitre 1, nous proposions de résoudre le problème par la méthode de Monte-Carlo, en tirant aléatoirement des points dans un carré et en comptabilisant ceux qui tombent dans le cercle inscrit dans le carré. On obtenait alors une approximation de la valeur de π en calculant le ratio des points situés dans le cercle sur le nombre total de points échantillonnés (à un facteur près dépendant des dimensions du carré). Le théorème central limite nous assure que la quantité mesurée converge vers le nombre π , et on peut démontrer que le taux de convergence est en $1/\sqrt{N}$, où N est le nombre de points tirés.

Q1. En déduire une approximation de la complexité du calcul des premières décimales de π avec cette méthode, en prenant le tirage aléatoire d'un point comme unité de complexité. Montrer que cette méthode est impraticable, dès lors que l'on souhaite plus que 6 ou 7 décimales.

Nous allons donc avoir recours à un algorithme déterministe, en utilisant la méthode de Salamin et Brent, consistant à évaluer les termes de deux suites récursives croisées.

$$a_0 = 1 \quad b_0 = 2^{-2} \quad a_{n+1} = \frac{a_n + b_n}{2} \quad b_{n+1} = \sqrt{a_n b_n}$$

Lorsque les deux suites ont atteint leur valeur de convergence (à ε près, où ε est un réel fonction de la précision à laquelle on souhaite calculer π), on peut alors évaluer une troisième suite, dont la limite théorique est égale à la valeur recherchée :

$$U_m = \frac{4a_m^2}{1 - 2 \sum_{n=1}^m 2^n (a_n^2 - b_n^2)} \xrightarrow[m \rightarrow \infty]{} \pi$$

Les concepteurs de l'algorithme indiquent que sa complexité (en prenant comme unité de taille le nombre n de décimales à calculer) est un $\Theta(C_m(n) \log n)$, où C_m est la complexité de la multiplication de 2 nombres à n décimales.

Q2. Ecrire un algorithme permettant de multiplier 2 nombres de n chiffres, à l'aide de la méthode classique enseignée à l'école primaire. On supposera que les nombres sont des entiers (ce qui moyennant un changement de position de la virgule, revient au même que de travailler sur des nombres décimaux).

Q3. Evaluer la complexité $C_m(n)$ de l'algorithme. En déduire la complexité $C(n)$ du calcul des n premières décimales de π . Donner une estimation du nombre de décimales calculables en 24 heures sur une machine doté d'un processeur effectuant 1 milliard d'opérations à la seconde.

On souhaite améliorer les performances de l'algorithme. Pour ce faire, on va remplacer notre programme de multiplication par l'algorithme de Karatsuba, dont on donne le principe ci-dessous.

Etant donnés deux nombres à n décimales. L'idée de Karatsuba consiste à remarquer que :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + \left((ac + bd - (a - b)(c - d)) \times 10^k + bd \right)$$

où l'expression à droite nécessite 3 multiplications sur des nombres 2 fois plus petits que ceux qui interviennent dans le produit à gauche du signe égal.

Q4. En supposant que n est une puissance de 2 (moyennant éventuellement un ajout de zéros non-significatifs) et en utilisant l'observation ci-dessus, écrire un algorithme récursif de type *diviser pour régner* permettant de calculer le produit de deux entiers à n chiffres. On apportera les preuves de terminaison et de correction de l'algorithme.

Q5. Evaluer la complexité de l'algorithme et montrer qu'elle est inférieure à celle de la multiplication naïve. En déduire la complexité du programme de calcul des n premières décimales de π . Combien de décimales peut-on à présent espérer calculer en 24 heures ?

Q6. L'algorithme de Toom-Cook, que nous n'étudierons pas ici, fonctionne sur le même principe que celui de Karatsuba, et permet de découper une multiplication à n chiffres en 5 multiplications contenant chacune 3 fois moins de chiffres. Evaluer la complexité de cet algorithme, et réévaluer l'estimation du nombre de décimales calculables en 24 heures.

Q7. Les record actuels du plus grand nombre de décimales calculées par une machine, sont réalisés à l'aide d'un procédé de multiplication par transformée de Fourier rapide (FFT). Cet algorithme permet de réaliser le produit de 2 nombres à n chiffres en un temps $\Theta(n \log n \log \log n)$. Reprendre la question précédente avec ce nouvel algorithme.

Q8. En pratique, il faut prendre en compte les opérations de division et de racine carrée, plus longues à exécuter que la multiplication. La méthode classiquement utilisée est celle de Newton, permettant :

- de calculer l'inverse de deux nombres à n décimales, à l'aide d'un procédé itératif où chaque passe nécessite un nombre constant de multiplications à n chiffres
- de calculer la racine carrée d'un nombre à n décimales, à l'aide d'un procédé itératif où chaque passe nécessite un nombre constant d'inversions à n chiffres

On sait d'autre part qu'avec la méthode de Newton, le nombre de décimales correctes double à chaque itération.

Réévaluer la complexité de l'algorithme du calcul de π .

Exercice 3.28. Suite de Fibonacci

On considère la suite de Fibonacci $(f_n)_{n \in \mathbb{N}}$, définie par :

$$f_0 = 1, \quad f_1 = 1 \quad \text{et :} \quad \forall n \in \mathbb{N} \quad f_{n+2} = f_{n+1} + f_n$$

Q1. Ecrire une fonction récursive FIBO, prenant en entrée un entier N et retournant la valeur du terme de rang N de la suite de Fibonacci.

Q2. Justifier soigneusement que l'équation de récurrence régissant la complexité en temps de l'algorithme écrit à la question Q1 est :

$$C(n) = C(n - 1) + C(n - 2) + 1$$

On pensera à indiquer les unités (de taille du problème et de coût d'exécution) utilisées.

Q3. On pose $T(n) = C(n) + 1$. Réexprimer l'équation de récurrence en fonction de T .

Q4. On rappelle qu'une complexité exponentielle est de la forme $\Theta(a^n)$, avec $a > 1$.

Montrer que, pour toute suite $(\psi_n)_{n \in \mathbb{N}}$ obéissant à une équation de récurrence du type $\psi(n) = 2\psi(n - 2)$, on a la relation $\psi(n) = \mathcal{O}(T(n))$.

En utilisant un changement de variable judicieux, montrer que ψ est de croissance exponentielle, et en déduire que la complexité de l'algorithme écrit à la question Q1 est au moins exponentielle.

Q5. En utilisant une démarche similaire à celle de la question précédente, montrer que si $(\varphi_n)_{n \in \mathbb{N}}$ est une suite obéissant à une équation de récurrence du type $\varphi_n(n) = 2\varphi_n(n - 1)$ alors on a la relation $T(n) = \mathcal{O}(\varphi_n(n))$. En déduire que la complexité de l'algorithme écrit à la question Q1 est au plus exponentielle.

Q6. On pose la conjecture que $T(n) = \Phi^n$, avec $\Phi \in \mathbb{R}^*$, une constante inconnue non-nulle. En utilisant l'équation de récurrence sur T , déterminer la valeur de Φ .

Q7. En utilisant un raisonnement par récurrence sur $p \in \mathbb{N}^*$, montrer que :

$$\forall n \in \mathbb{N}^* \quad f_{n+p} = f_n f_p + f_{n-1} f_{p-1}$$

Q8. En utilisant la relation démontrée à la question Q7, écrire une fonction récursive FIBO_DPR, de type *diviser pour régner*, prenant en entrée un entier N et retournant la valeur de la suite de Fibonacci au rang N . On pensera à distinguer les cas en fonction de la parité de n , ainsi qu'à minimiser autant que possible le nombre d'appels récursifs (en particulier dans le cas n impair).

Q9. Ecrire l'équation de partition régissant la complexité de l'algorithme récursif écrit à la question **Q8**. En utilisant le *Master Theorem*, résoudre cette équation et indiquer à quelle classe de complexité appartient FIBO_DPR.

Q10. En utilisant un tableau pour stocker les valeurs calculées, peut-on écrire un algorithme (non-récursif) FIBO_TAB calculant la valeur de la suite de Fibonacci au rang n , en un temps $\Theta(n)$?

Q11 (bonus). On se place à présent dans l'espace \mathbb{R}^2 , et on considère la suite $(F_n)_{n>0}$ définie par :

$$\forall n \in \mathbb{N}^* \quad F_n = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$$

Montrer que sous cette formulation, on peut calculer F_{2n} et F_{2n+1} uniquement en fonction de F_n , autrement dit, qu'il existe deux fonctions \mathcal{S} et \mathcal{T} :

$$\begin{aligned} \mathcal{S} : \quad \mathbb{R}^2 &\mapsto \mathbb{R}^2 \\ F_n &\mapsto F_{2n} = \mathcal{S}(F_n) \end{aligned}$$

$$\begin{aligned} \mathcal{T} : \quad \mathbb{R}^2 &\mapsto \mathbb{R}^2 \\ F_n &\mapsto F_{2n+1} = \mathcal{T}(F_n) \end{aligned}$$

En utilisant les définitions \mathcal{S} et \mathcal{T} , écrire une fonction récursive FIBO_R2, de type *diviser pour régner*, prenant en entrée un entier $N \geq 1$ et retournant (sous forme d'un tableau à deux éléments) la valeur de la suite F au rang N . Evaluer la complexité de la fonction et conclure quant à l'efficacité de cette approche.

Chapitre 4

Structure de données

Sommaire

4.1	Tableaux	152
4.1.1	Définition	153
4.1.2	Tableau trié	155
4.1.3	Tri rapide	157
4.1.4	Tri de complexité optimale	161
4.1.5	Tri par dénombrement	162
4.1.6	Tri par paquets	163
4.2	Listes chaînées	165
4.2.1	Notion de pointeur	165
4.2.2	Liste simplement chaînée	167
4.2.3	Liste doublement chaînée	168
4.2.4	Liste chaînée circulaire	168
4.2.5	Fonctionnalités d'une liste chaînée	168
4.3	Files et piles	170
4.3.1	Définitions	170
4.3.2	Implémentation	170
4.4	Arbres	171
4.4.1	Définitions	171
4.4.2	Parcours d'un arbre	174
4.4.3	Arbres binaires de recherche	177
4.4.4	Exemples d'application	178
4.5	Tables de Hachage	185
4.5.1	Définition	185
4.5.2	Implémentation	189
4.6	Indexation spatiale	190
4.6.1	Exemple d'introduction	190
4.6.2	Quadtree	194
4.6.3	Kd-Tree	195
4.7	Exercices	197

Introduction

Jusqu'à présent nous avons vu comment formaliser un algorithme à l'aide d'un langage de pseudo-code, comment démontrer qu'il retourne bien le résultat attendu et donner une

évaluation grossière du temps de calcul (ou de l'espace mémoire) nécessaire à son exécution en fonction de la taille du problème à traiter. Dans les algorithmes étudiés, la gestion d'une collection de données (nombres, booléens ou chaînes de caractères) se faisait systématiquement via un tableau (que l'on appelle aussi parfois improprement liste). L'avantage d'une telle gestion des données est de permettre un accès simple à toute donnée, sous réserve bien entendu de connaître l'indice de sa position dans le tableau. Ainsi, chaque élément peut être récupéré en un temps constant, indépendamment de la taille de la liste. En contre-partie, lors de son implémentation en machine, ce type de structure de données nécessite d'allouer une quantité précise de cases mémoires contiguës. Cela peut s'avérer problématique lorsque l'on ne connaît pas a priori, le nombre d'éléments que nous aurons besoin de stocker dans le tableau (par exemple si la quantité de données est difficilement évaluable à l'aide d'une expression analytique avant l'exécution, ou bien si elle est non-déterministe comme c'est le cas dans les algorithmes de type Monte-Carlo). D'autre part, dans de nombreuses situations pratiques, la structure de données est utilisée pour sauvegarder des éléments durant l'exécution du programme, tandis qu'une deuxième phase se charge de passer en revue les données qui ont été mises de côté pendant la première phase. Dans ce cas de figure, la possibilité d'accéder directement à un élément par son indice n'est pas indispensable, et il serait alors judicieux d'avoir recours à d'autres types de structures de données dynamiquement extensibles et autorisant un parcours séquentiel (on dit aussi linéaire) des éléments enregistrés.

Il existe de nombreuses structures de données, chacune étant adaptée à un type de tâche particulier. Structures de données et algorithmes sont intimement liés, comme le souligne le titre d'un ouvrage de référence écrit par le célèbre informaticien suisse Niklaus Wirth : *Algorithms + Data Structures = Programs*. En particulier, chaque structure est définie à l'aide d'une batterie d'algorithmes permettant de remplir un cahier des charges propre aux fonctionnalités de la structure. D'autre part, en contexte opérationnel, l'on ne peut espérer produire un algorithme rapide et efficace sans utiliser les structures adéquates.

Après quelques rappels sur les tableaux, qui nous en permettront aussi de formaliser et classer les algorithmes de tri, nous introduirons tour à tour des structures de données dynamiques de type séquentiel : les listes chaînées, les piles et les files. Notons que ce nouveau type de structure nécessitera auparavant d'aborder la notion de pointeur, outil puissant de tout langage de programmation de bas niveau. Dans un second temps, nous étudierons des types de données plus évolués, tels que les arbres (utilisés à la fois pour démontrer des résultats théoriques et pour implémenter de nouvelles structures de données) et les tables de hachage (utilisées de manière intensive dans tout programme moderne un tant soit peu complexe). Ceci constituera une introduction au mécanisme d'indexation, indispensable au domaine des bases de données. En guise de conclusion de chapitre, nous étendrons ce concept au domaine de la géomatique, avec l'indexation spatiale, qui nous permettra d'effectuer des opérations de nature géométrique (recherches de voisins, intersections, unions...) en un temps record.

4.1 Tableaux

Il s'agit de la structure de données la plus simple possible, présente dans tout langage de programmation digne de ce nom (généralement sous l'appellation *array*).

Jusqu'à présent, nous avons manipulé les tableaux en ADL comme s'il s'agissait de simples identificateurs indicés par des séquence d'entiers. En réalité, la manipulation des tableaux par la machine est légèrement plus complexe.

4.1.1 Définition

Dans une représentation linéaire de la mémoire, créer un tableau signifie réserver un espace mémoire constitué d'un ensemble de cases contiguës destinées à recevoir les éléments du tableau. Nous admettrons que par défaut, les cellules contiennent la valeur *null* tant qu'une première valeur ne leur a pas été affectée. On parle d'allocation de mémoire.

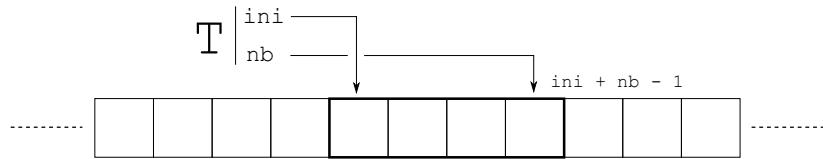


FIGURE 4.1 – Déclaration d'un tableau de longueur *nb* débutant en position *ini*

Comme indiqué sur la figure ci-dessus, la déclaration du tableau nécessite de connaître le nombre de cases à allouer. L'instruction de déclaration devra donc nécessairement connaître le paramètre *nb*. L'indice *ini* désignant la position du premier élément du tableau dans la mémoire est quant à lui affecté automatiquement par la machine lors de la création du tableau. Ces deux paramètres sont stockés dans l'identificateur *T* de sorte que tout appel d'un élément du tableau *T_i* peut être traité en récupérant l'élément situé dans la bande de mémoire en position :

$$m(T, i) = ini + i - 1$$

La fonction *m* est une fonction associée au système, qui à un identificateur dimensionné *T* et un indice *i*, renvoie la position (dans la bande de mémoire) de l'élément *T_i*. Notons qu'en toute rigueur, cette fonction n'utilise pas le paramètre *nb*, qui n'a donc en théorie pas besoin d'être stocké dans l'identificateur *T*. Cette remarque ne tiendra plus dès lors que nous manipulerons des identificateurs de dimension 2 ou plus (matrices, tenseurs). Notons également que dans la plupart des langages de programmation, les indices de tableau commencent à 0, permettant ainsi d'ignorer le terme -1 dans l'expression de la fonction d'accès aux éléments d'un tableau.

Par exemple, lorsque l'on souhaite affecter une valeur à la deuxième cellule du tableau, l'instruction $T_2 \leftarrow 7$ est exécutée par la machine en appelant la fonction $m(T, 2)$ qui retourne la valeur $ini+1$, indiquant ainsi l'adresse mémoire de la cellule destinée à contenir la valeur 7.

Le même procédé s'applique lorsque l'on souhaite récupérer le contenu d'une cellule d'un tableau. Ainsi, l'instruction $n \leftarrow T_3$ s'exécute en allant récupérer la valeur enregistrée dans

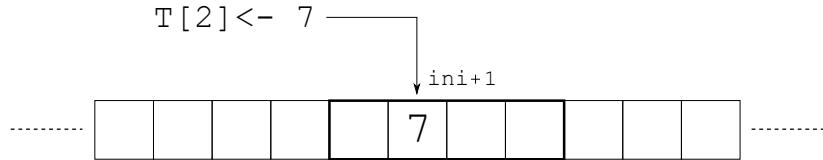


FIGURE 4.2 – Modification de la valeur d'une cellule dans un tableau

la cellule d'indice $m(T, 3) = ini + 2$.

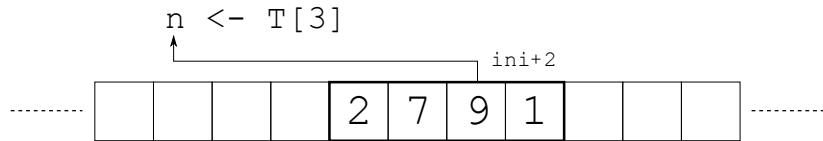


FIGURE 4.3 – Récupération de la valeur d'une cellule dans un tableau

Les choses se compliquent lorsque l'on souhaite créer un tableau multidimensionnel. Il faut alors spécifier, en plus de la position initiale dans la mémoire, la taille à allouer sur chacune des dimensions. Les éléments sont alors enregistrés ligne par ligne dans la mémoire. La longueur de la bande de mémoire à allouer est égale au produit des tailles du tableau suivant chacune des dimensions.

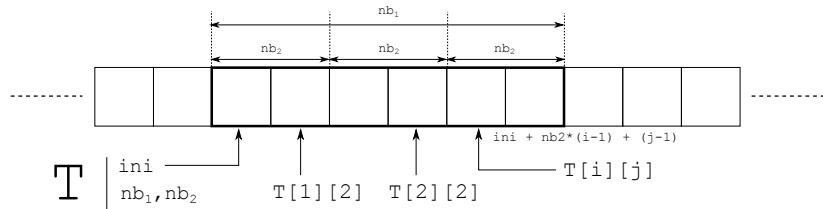


FIGURE 4.4 – Récupération de la valeur d'une cellule dans un tableau multi-dimensionnel

Pour une matrice, l'accès à un élément requiert à présent de connaître l'indice ini ainsi que la taille suivant la première dimension nb_1 .

Dans le cas le plus général, pour un tableau T de dimensions $n_1 \times n_2 \times \dots \times n_p$, débutant à l'indice ini , la fonction d'accès à la mémoire s'exprime à présent :

$$m(T, [i_1, i_2, \dots, i_p]) = ini + i_p - 1 + \sum_{k=1}^{p-1} (i_k - 1) \prod_{l=k+1}^p n_l$$

On observe donc que lorsque le nombre de dimensions du tableau est élevé, même si l'accès à un élément se fait en un temps constant (c'est-à-dire en un temps indépendant du

nombre d'éléments dans le tableau), l'appel de la fonction m nécessite un nombre considérable de multiplications et d'additions sur les indices. Plus spécifiquement, le nombre de multiplications à effectuer évolue comme le carré du nombre de dimensions. Si dans un code informatique on doit avoir recours plusieurs fois à un élément d'un tenseur de grande dimension, il sera préférable de référencer cet élément par un identificateur scalaire.

Par exemple, une instruction du type (pour N grand) :

$$\left\{_{i:1}^N T_{jklm} \leftarrow T_{jklm} + L_i \right\}_i$$

sera avantageusement remplacée par une simplification du type :

$$x \leftarrow T_{jklm}; \left\{_{i:1}^N x \leftarrow x + L_i \right\}_i T_{jklm} \leftarrow x;$$

Notons qu'en pratique, le type des éléments d'un tableau conditionne la place requise pour l'allocation. Un tableau de booléens nécessite des cellules d'un bit, tandis que les caractères nécessitent 1 octet et les flottants 4 octets (8 octets pour les flottants en double précision). La déclaration d'un tableau est donc généralement composée à la fois de ses dimensions et de son type. Le pseudo-code étant non-typé, nous ferons abstraction de cette subtilité ici.

4.1.2 Tableau trié

L'accélération d'un traitement passe bien souvent par une meilleure structuration des données en entrée. C'est le cas par exemple dans une recherche du couple de points les plus proches respectivement l'un de l'autre, qui peut être fait en un temps quasi-linéaire sous réserve d'avoir trié les points suivant leurs abscisses et leurs ordonnées (prises séparément), à comparer à la complexité quadratique de l'algorithme naïf¹. Le tri de tableau est donc une première optimisation possible.

Nous avons déjà étudié dans les chapitres précédent, deux types de tris naïfs, par insertion et sélection, ainsi que l'algorithme de tri-fusion. Dans cette section, nous formalisons le concept de tri de liste et nous introduisons quelques propriétés souhaitables sur les tris (malheureusement pas nécessairement respectées par toutes les approches). Ce sera l'occasion de développer le tri rapide, qui constitue à ce jour l'un des algorithmes les plus efficaces en contexte opérationnel pour trier des tableaux de grande taille.

1. Voir exercice 22 du chapitre 3

Définition : algorithme de tri

Soit E un ensemble muni d'une relation d'ordre total \preceq et T un tableau d'éléments de E :

$$T = [x_1, x_2, \dots, x_n] \quad x_i \in E$$

Un algorithme de tri est une procédure prenant en entrée le tableau T , et le réordonnant suivant une permutation σ :

$$T = [x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)}]$$

de sorte à avoir : $x_{\sigma(1)} \preceq x_{\sigma(2)} \preceq x_{\sigma(3)} \preceq \dots \preceq x_{\sigma(n)}$

On distingue alors deux types de tri, suivant que les opérations sont effectuées dans le tableau passé en entrée lui-même, ou bien nécessitent la création d'un tableau temporaire. Pour des tableaux de très grandes tailles, le premier type de tri sera bien souvent préféré.

Définition : tri en place

Un algorithme de tri de listes est dit *en place* si et seulement si sa complexité spatiale dans le pire des cas est indépendante de la taille du tableau, c'est-à-dire plus formellement :

$$C_{spatiale}(n) = \mathcal{O}(1)$$

Exemple : le tri par sélection est un tri en place puisque toutes les opérations sont réalisées par des inversions directement dans le tableau. Le tri par fusion, qui lui nécessite la création d'un tableau temporaire pour fusionner deux tableaux déjà triés, n'est pas un tri en place.

Exercice 4.1. Le tri à bulles (voir exercice 20 du chapitre 3) est-il un tri en place ? Qu'en est-il du tri par insertion ?

Une autre propriété souhaitable sur les tris dans certains cas de figure bien particuliers est la stabilité. Par hypothèse, l'ensemble E parmi lesquels sont tirés les éléments du tableau est muni d'une relation d'ordre total. Cela signifie que deux éléments sont toujours comparables, mais ne garantit pas qu'ils sont différents. Lorsqu'il existe des doublons dans les données, plusieurs résultats de tri peuvent survenir. Considérons le tableau suivant :
 $T = [7, 2, 8, 3, 1, 2, 5]$

L'élément 2 est présent deux fois. Pour distinguer les deux copies l'une de l'autre, nous noterons : $T = [7, 2, 8, 3, 1, \bar{2}, 5]$. Deux situations différentes peuvent alors se produire à

l'issue du tri :

Ou bien l'ordre des éléments avant le tri est respecté lorsqu'il y a un cas d'égalité : $T = [1, 2, \bar{2}, 3, 5, 7, 8]$.

Ou alors, les éléments sont inversés par rapport au tableau d'origine : $T = [1, \bar{2}, 2, 3, 5, 7, 8]$.

On dit qu'un tri est *stable* lorsqu'il garantit la conservation de l'ordre pour les éléments égaux. Si on peut trouver un exemple de tableau tel qu'on se trouve dans le cas 2, alors le tri est dit *instable*.

Définition : stabilité

Un algorithme de tri de listes est dit *stable* si et seulement si pour tout tableau $T = [x_1, x_2, \dots, x_n]$ passé en entrée, la permutation σ trouvée par l'algorithme respecte la condition suivante :

$$\forall (i, j) \in \{1, n\}^2 \quad i < j \quad \text{si } x_i = x_j \quad \text{alors} \quad \sigma(i) < \sigma(j)$$

Autrement dit, la permutation est une fonction croissante des indices sur chaque sous-ensemble d'éléments égaux.

Exemple : démontrons que le tri par sélection n'est pas stable. Considérons le tableau $[2, \bar{2}, 1]$. La première passe de l'algorithme consiste à parcourir le tableau entier pour en extraire l'élément minium et l'échanger avec le premier élément du tableau. On obtient alors le tableau $[1, \bar{2}, 2]$. Lors de la seconde, passe, le minimum trouvé dans le tableau ($\bar{2}$) est à la bonne position, et aucun échange n'est effectué. L'algorithme se termine donc en laissant les éléments 2, et $\bar{2}$ dans un ordre inverse à celui du tableau d'entrée. Le tri n'est donc pas stable.

Exercice 4.2. Démontrer que le le tri par insertion, le tri à bulles et le tri-fusion sont stables.

4.1.3 Tri rapide

Dans cette section nous introduisons un nouveau type de tri, le tri-rapide (*quicksort* en anglais), reposant sur le principe *diviser pour régner*, abordé au chapitre précédent. Notons que malgré une complexité quadratique dans le pire des cas, sa complexité moyenne quasi-linéaire le rend très efficace sur la plupart des cas rencontrés en pratique.

Le principe de l'algorithme consiste à choisir un élément que l'on appelle pivot, puis à créer deux sous-tableaux A et B , l'un contenant toutes les valeurs inférieures au pivot tandis que l'autre contient toutes les autres valeurs. L'algorithme est alors appliqué récursivement à chacun des deux sous-tableaux. Le tableau trié final s'obtient en concaténant les deux tableaux avec le pivot : $[A, x, B]$.

Il existe plusieurs manières de choisir le pivot. Ici, nous prendrons le premier élément du tableau T_1 . Cette stratégie permet d'obtenir un algorithme très simple.

Dans un premier temps, on écrit le module permettant de décomposer le problème en deux sous-problèmes à chaque niveau de récursivité. Etant donné un tableau T et deux indices de début et de fin i et j , le module travaille sur la portion $[i..j]$ du tableau, positionne tous les éléments inférieurs au pivot en début de portion, puis insère le pivot, et enfin complète avec les éléments restants. Nous allons écrire ce module sous forme d'une fonction, de sorte à pouvoir retourner l'indice du pivot dans le tableau arrangé.

PARTITION($T \uparrow\downarrow, i \downarrow, j \downarrow$) : $\left[\begin{array}{l} x \leftarrow T_i; \textcircled{C} \text{ choix du pivot } \textcircled{C} \text{ } l \leftarrow i + 1; \\ \left\{ \begin{array}{l} \left. \begin{array}{l} T_k \leq x ? \ g \leftarrow T_k; \ T_k \leftarrow T_l; \ T_l \leftarrow g; \textcircled{C} \text{ échange de } T_k \text{ et } T_l \textcircled{C} \text{ } l \leftarrow l + 1; \mid i \end{array} \right\}_k \\ g \leftarrow T_i; \ T_i \leftarrow T_{l-1}; \ T_{l-1} \leftarrow g; \text{ PARTITION } \leftarrow l - 1 \end{array} \right] \end{array} \right]$

La fonction de décomposition en sous-problèmes étant écrite, il ne reste plus qu'à écrire la récurrence :

TRI($T \uparrow\downarrow, i \downarrow, j \downarrow$) : $\left[\begin{array}{l} i = j ? \textcircled{C} \text{ le tableau contient un seul élément } \textcircled{C} ! \mid i \textcircled{C} \text{ découpage en sous-problèmes } \textcircled{C} \text{ pivot } \leftarrow \text{PARTITION}(T, i, j); \textcircled{C} \text{ résolution des sous-problèmes } \textcircled{C} \text{ TRI}(T, i, \text{pivot} - 1); \text{ TRI}(T, \text{pivot} + 1, j); \end{array} \right]$

Dans cette procédure, la recomposition est implicite puisque les opérations sont réalisées dans le tableau.

On peut alors empaqueter la fonction dans une entête pour rendre les appels plus simples :

QUICKSORT($T \uparrow\downarrow$) : $\left[\begin{array}{l} \text{TRI}(T, 1, \text{SIZE}(T)) \end{array} \right]$

Exercice 4.3. On peut montrer que l'algorithme est plus efficace en moyenne quand la position du pivot est choisie aléatoirement. Modifier l'algorithme en conséquence.

Analyse du tri rapide

De manière évidente, le tri rapide n'est pas stable. Il suffit de considérer l'entrée à trois éléments : $[x, \bar{x}, \bar{\bar{x}}]$ avec $x = \bar{x} = \bar{\bar{x}}$. L'application du tri rapide sur ce tableau va retourner, après un étage d'appel récursif : $[\bar{x}, \bar{\bar{x}}, x]$ (tous les éléments du tableau sont inférieur ou égaux au pivot, ils seront donc placés dans le premier sous-tableau).

Remarquons que le tri n'est pas plus stable avec un choix aléatoire de la position du pivot.

En revanche, chaque itération ne nécessite qu'une case mémoire supplémentaire (la variable de stockage temporaire g). La complexité spatiale de l'algorithme ne dépend donc pas de la taille de l'entrée. Le tri rapide est un tri en place.

En supposant que le pivot est la k -eme plus petite valeur du tableau, l'équation de partition à un rang n s'exprime par :

$$C(n) = C(k - 1) + C(n - k) + \Theta(n)$$

En effet, chaque décomposition nécessite un nombre de comparaisons égal à la taille du tableau, et on a donc $f(n) = \Theta(n)$.

La performance de l'algorithme en terme de complexité dépend donc du rang du pivot dans la liste, et nous nous trouvons dans une situation similaire à celle de l'algorithme de Douglas-Peucker, étudié dans l'exercice 3.26. Notons que k n'est généralement pas constant sur tous les étages de l'arbre de récursion.

Dans le cas optimal, on prend la médiane comme pivot du tableau. On a alors $k = n/2$ et l'équation de partition devient :

$$C(n) = 2C\left(\frac{n}{2}\right) + \Theta(n)$$

Par application du Master Theorem, on trouve que la complexité de l'algorithme est quasi-linéaire : $C(n) = \Theta(n \log n)$. Ceci nécessite cependant d'être capable de trouver la médiane du tableau, ce qui en principe ne peut se faire qu'en triant le tableau auparavant. Il est possible de récupérer la médiane d'une liste de nombres sans trier cette liste et avec une complexité linéaire². Le surcoût lié à la recherche de la médiane serait ainsi absorbé dans la fonction de décomposition/recomposition f et on est théoriquement capable d'obtenir un algorithme qui soit garanti en $\Theta(n \log n)$. Malheureusement, un tel algorithme est extrêmement inefficace en pratique, et on préfère retenir l'une des deux solutions suivantes :

- Toujours utiliser le premier élément comme pivot
- Tirer la position du pivot aléatoirement

Si on utilise systématiquement le premier élément comme pivot, alors le pire des cas survient quand la liste est déjà triée. En effet, dans ce cas, le pivot est aussi le plus petit élément. Il ne partitionne donc pas le tableau puisque tous les autres éléments sont du même côté du pivot. L'équation de partition se transforme alors en

$$C(n) = C(n - 1) + \Theta(n)$$

ce qui ne répond plus au principe *diviser pour régner*.

En utilisant les techniques exposées dans le chapitre précédent, on trouve une complexité dans le pire des cas : $C_{max}(n) = \Theta(n^2)$.

2. Voir méthode de la médiane des médianes

Analysons alors ce que devient la complexité lorsque le pivot est choisi aléatoirement.

De manière triviale, le choix aléatoire du pivot contient la situation où on tire systématiquement un élément extrémal (i.e. un élément qui ne partitionne pas le tableau). La complexité dans le pire des cas de la solution de tirage aléatoire est donc également quadratique. L'avantage est que cette complexité ne survient généralement pas sur des tableaux déjà triés. Cette version de l'algorithme possède donc l'avantage d'être tout aussi efficace sur un tableau trié que sur une donnée quelconque (en moyenne).

Qu'en est-il de la complexité moyenne du tri rapide avec tirage aléatoire du pivot ?

L'équation de partition sur la complexité moyenne s'exprime alors en fonction de la probabilité $\mathbb{P}(k = i) = 1/n$ que le pivot soit situé au rang i , et comporte un terme de décomposition égal à $n - 1$ (il faut comparer le pivot à tous les autres éléments du tableau) :

$$C_{moy}(T) = \sum_{i=1}^n [C(i-1) + C(n-i)] \mathbb{P}(k = i) + (n-1)$$

$$C(n) = \frac{2}{n} \sum_{i=0}^{n-1} C(i) + (n-1) \quad \text{d'où :} \quad nC(n) - (n-1)C(n-1) = 2C(n-1) + (2n-2)$$

$$\text{Et par suite : } nC(n) = (n+1)C(n-1) + (2n-2) \Rightarrow \frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2n-2}{n(n+1)}$$

En posant (pour $n \in \mathbb{N}^*$) : $w_n = \frac{C(n-1)}{n}$ on simplifie la récurrence : $w_{n+1} = w_n + \Theta(\frac{1}{n})$.

La résolution est alors immédiate : $w_n = \Theta\left(\sum \frac{1}{i}\right) = \Theta(\log n)$

Finalement, on obtient une complexité moyenne quasi-linéaire :

$$C_{moy}(n) = \Theta(n \log n)$$

Cela porte à 5 le nombre d'algorithmes de tri que nous avons passés en revue : 2 algorithmes intuitifs dotés d'une complexité prohibitive sur des tableaux de grande taille (tri par sélection et insertion), 1 algorithme tout aussi coûteux en temps mais original et intéressant d'un point de vue pédagogique (tri à bulles) et enfin 2 algorithmes tirant partie du principe *diviser pour régner* et permettant d'obtenir des complexités moyennes quasi-linéaires.

Notons qu'il existe des exemples de tris non-récurrsifs³ permettant d'atteindre cette complexité en $\mathcal{O}(n \log n)$, comme par exemple le tri à peigne qui consiste en un raffinement du tri à bulles.

Les algorithmes les plus efficaces produits à ce jour, combinent souvent plusieurs des solutions ci-dessus. Par exemple, le tri *introsort* (ou *introspective sort*) contrôle en temps

3. Tri n'utilisant pas le paradigme *diviser pour régner*

Algorithme	C_{min}	C_{moy}	C_{max}	En place	Stable	DpR ²
Tri par sélection	n^2	n^2	n^2	x		
Tri par insertion	n^2	n^2	n^2	x	x	
Tri à bulles	n^2	n^2	n^2	x	x	
Tri fusion	$n \log n$	$n \log n$	$n \log n$		x	x
Tri rapide	$n \log n$	$n \log n$	n^2	x		x

TABLE 4.1 – Algorithmes de tri les plus fréquemment utilisés

réel la profondeur de récursion du tri rapide. Si cette dernière s'avère trop élevée en cours d'exécution (typiquement plus d'un certain nombre de fois $\log n$), alors cela signifie que la composition de la liste à trier est telle que le choix du pivot n'est pas performant. Dans ce cas, on corrige la situation *à la volée* en triant chaque sous-problème à l'aide d'un algorithme dont la complexité dans le pire des cas est quasi-linéaire (e.g. tri fusion). Enfin, lorsque les listes à trier ne contiennent plus que quelques éléments, on achève le travail avec un tri simple comme le tri par sélection ou par insertion, qui restent efficaces sur des listes de très petite taille. Les différents algorithmes travaillent ainsi en coopération, la force de l'un compensant les faiblesses des autres.

Il existe un dernier type de tri efficace communément utilisé, mais dont nous auront l'occasion de reparler lorsque nous aborderons les structures de piles de priorité.

4.1.4 Tri de complexité optimale

On considère un algorithme de tri de tableaux procédant par comparaison (nous verrons dans la suite que dans certains cas de figure bien particuliers il est possible de trier une liste sans effectuer une seule comparaison !).

La question que l'on se pose toujours face à un problème donné est de connaître la complexité minimale en dessous de laquelle aucun algorithme ne pourra espérer venir à bout du problème. Dans le cas d'un tri, la réponse est apportée par le théorème suivant :

Théorème : *borne inférieure sur la complexité d'un tri par comparaison*

La complexité dans le pire des cas de tout algorithme de tri de tableaux par comparaisons est au moins quasi-linéaire :

$$C(n) = \Omega(n \log n)$$

Preuve : il est possible de représenter le déroulement de l'algorithme de manière abstraite via un arbre binaire dont la racine correspond au départ de l'exécution, chaque noeud dénotant la disjonction de cas qui peut survenir à l'issue d'une comparaison. Enfin, les feuilles de l'arbre, c'est-à-dire les noeuds terminaux (voir sections suivantes de ce chapitre)

représentent l'issue de l'algorithme.

Etant donné qu'il y a $n!$ permutations possibles pour un ensemble de n éléments, l'arbre d'exécution de l'algorithme doit donc comprendre $n!$ noeud terminaux.

Pour un arbre binaire parfait, le nombre de noeuds à un niveau p donné est égal au double du nombre de noeuds du niveau précédent. Il en découle que si l'arbre possède h niveaux, alors le nombre de noeuds terminaux vaut 2^h . Pour couvrir toutes les permutations possibles d'une liste de n éléments, la hauteur h de l'arbre doit donc vérifier :

$$2^h \geq n! \quad \Rightarrow \quad h \geq \log_2 n!$$

En utilisant la formule de Stirling, on obtient asymptotiquement l'inégalité suivante :

$$h \geq \log_2 \sqrt{2\pi n} \left(\frac{n}{e}\right)^n = \log_2 \sqrt{2\pi n} + n \log_2 n - n \log_2 e$$

En remarquant que h est aussi le nombre de comparaisons effectuées par l'algorithme, on obtient la borne inférieure suivante sur toute complexité de tri de liste par comparaisons :

$$C(n) = \Omega(n \log n)$$

Ceci montre donc qu'un tri par comparaison ne pourra être plus rapide que quasi-linéaire.

Implication pratique : si l'on parvient à montrer qu'un tri de liste par comparaisons est doté d'une complexité $C(n) = \mathcal{O}(n \log n)$ alors on obtient par encadrement : $C(n) = \Theta(n \log n)$.

4.1.5 Tri par dénombrement

La borne inférieure calculée à la section précédente ne s'applique que pour les tris qui procèdent par comparaisons. Nous exposons ici un cas particulier de tri, efficace quand les données appartiennent à une ensemble d'entiers bien défini $E = \{1, 2, \dots, p\}$.

Si la quantité p est fixée indépendamment de la taille des listes à traiter, alors on peut montrer qu'un tri par dénombrement possède une complexité linéaire.

Exemple d'application : une université souhaite classer une liste d'étudiants en fonction des notes obtenues au dernier partiel. On suppose que ces notes sont des entiers compris entre 0 et 20. Ecrire l'algorithme permettant de classer les étudiants par ordre croissant de la note obtenue.

Réponse : le concept du tri par dénombrement consiste à balayer 2 fois la liste : une première fois pour créer un histogramme des valeurs rencontrées, puis une seconde fois pour répartir les étudiants dans un tableau en fonction de leur position dans l'histogramme.

Ce tri n'est efficace que si les deux conditions suivantes sont réunies :

- La taille de l'histogramme est indépendante du nombre d'éléments à trier. C'est le cas ici, le nombre de classes (20) n'augmente pas quand on ajoute des étudiants à la liste.
- Les éléments sont des entiers (on peut relaxer cette condition, et simplement requérir qu'il soit fini, ou bien infini dénombrable, sinon la notion d'histogramme n'a pas de sens). C'est le cas ici avec des notes entières.

L'algorithme s'exprime alors (pour une liste L à deux colonnes : étudiants/note) :

$$\text{TRI_DENOMB}(L \uparrow\downarrow) : \left[\left[\left\{ \begin{array}{l} {}^{20}_{i:1} C_i \leftarrow 0 \\ N \leftarrow \text{SIZE}(L); \end{array} \right\}_i \left\{ \begin{array}{l} {}^N_{i:1} \left\{ \begin{array}{l} {}^{20}_{j:1} L_{i,2} = j ? \\ C_j \leftarrow C_j + 1; \end{array} \right\}_j \left\{ \begin{array}{l} {}^{20}_{j:2} C_j \leftarrow C_j + C_{j-1} \\ j \leftarrow L_{i,2}; \end{array} \right\}_j \\ \text{TEMP}_{C_j,1} \leftarrow L_{i,1}; \text{TEMP}_{C_j,2} \leftarrow L_{i,2}; C_j \leftarrow C_j + 1 \end{array} \right\}_i L \leftarrow \text{TEMP} \right] \right]$$

L'algorithme ne contient que des boucles à compteur, sans modification du pas ni de la borne finale et sans débranchement du type réitération. Il termine donc. La preuve de correction est triviale et on laisse le soin au lecteur de l'établir.

Nous prendrons ici comme unité de coût le nombre de tests d'égalité. Il y a une égalité dans la double boucle mais seule la boucle externe dépend de n . La complexité est donc linéaire, et nous avons trié la liste sans recourir à la moindre comparaison.

Enfin, notons que ce tri est stable mais qu'il n'est pas effectué en place.

Exercice 4.4. On considère une image en noir et blanc sous forme de matrice M . Chaque pixel M_{ij} contient un entier dans l'ensemble $\{0, 1, \dots, 255\}$. Le tri par dénombrement est-il adapté ici ? Justifier. Ecrire un programme retournant en un temps linéaire la médiane des valeurs des pixels de l'image.

4.1.6 Tri par paquets

Nous étudions ici un second tri permettant d'atteindre une complexité moyenne linéaire.

Nous partons cette fois de l'hypothèse que les données à trier sont des éléments de \mathbb{R} , tirés aléatoirement et uniformément sur un intervalle $[a, b]$, dont les bornes sont connues.

Le principe du tri consiste à segmenter l'intervalle en n parties (où n est la taille de la liste à trier). Les données sur chaque intervalle sont triées à l'aide d'un algorithme de tri quelconque (par exemple un tri sélection) puis assemblées pour former la liste triée finale.

Supposons, sans perte de généralité, que les données (que l'on notera X_1, X_2, \dots, X_n) appartiennent à l'intervalle réel $[0, 1]$: $X_i \sim \mathcal{U}([0, 1])$. Soit $Y_k \in \mathbb{N}$ la variable aléatoire qui indique le nombre d'éléments X_i contenus dans l'intervalle $[\frac{k-1}{n}; \frac{k}{n}]$, d'où l'expression de la complexité moyenne totale :

$$C_{moy}(n) = \mathbb{E} \left[\sum_{k=1}^n C_u(Y_k) \right] = \sum_{k=1}^n \mathbb{E}[C_u(Y_k)]$$

où C_u est la complexité *unitaire*, c'est-à-dire la complexité de l'algorithme utilisé pour trier chaque intervalle. Dans la suite, nous choisissons un algorithme peu efficace, par exemple le tri par sélection, ou le tri à bulles, et par conséquent : $C_u(n) = n^2$. La complexité totale se réécrit alors :

$$C_{moy}(n) = \sum_{k=1}^n \mathbb{E}[Y_k^2] = n\mathbb{E}[Y^2]$$

où la seconde égalité résulte du fait que les variables Y_k^2 sont identiquement distribuées.

D'autre part, une règle classique de statistique nous permet d'écrire :

$$\mathbb{E}[Y^2] = \text{Var}[Y] + \mathbb{E}[Y]^2$$

Par définition, Y est distribuée suivant une loi binomiale à n tirages, avec une probabilité de succès $p = 1/n$. On obtient aisément que son espérance vaut $np = 1$ (on a en moyenne un point par intervalle), avec une variance associée égale à $np(1-p)$, d'où les simplifications :

$$C_{moy}(n) = n \left(n \frac{1}{n} \left(1 - \frac{1}{n} \right) + 1 \right) = 2n - 1 = \Theta(n)$$

Ceci conclut la preuve, et nous avons montré que le tri par paquets a donc bien une complexité moyenne linéaire. En particulier, on prêtera attention au fait que ce résultat est indépendant de l'algorithme utilisé pour trier chaque intervalle élémentaire, sous réserve que sa complexité soit meilleure que quadratique. On peut généraliser pour des complexités polynomiales de degré n ($n > 2$) et même pour une complexité exponentielle. Par exemple, en prenant $C_u(n) = \exp(n)$: il est bien connu que la fonction génératrice des moments de la loi binomiale est donnée par : $M_Y(t) = \mathbb{E}[e^{tY}] = (1 - p + pe^t)^n$, d'où :

$$\mathbb{E}[e^Y] = M_Y(1) = (1 - p + pe)^n = \left(1 + \frac{e-1}{n} \right)^n \xrightarrow[n \rightarrow +\infty]{} \exp(e-1) \approx 5.57$$

On obtient à nouveau une complexité moyenne linéaire : $C_{moy}(n) \underset{+\infty}{\sim} 5.57n = \Theta(n)$

Arrêtons-nous un instant sur la généralité du résultat obtenu : nous avons supposé ici que les données étaient distribuées suivant une loi uniforme, et il pourrait être objecté que cette hypothèse est très restrictive dans le cadre d'applications réelles. Néanmoins, on sait

que toute distribution continue de nombres x_i dont on connaît la fonction de répartition, peut être passée dans une fonction strictement croissante f telle que la distribution des $y_i = f(x_i)$ est uniforme. Il est alors possible d'appliquer le tri par paquets sur les données y_i pour en déduire un tri des données originales x_i (la monotonie de f implique la conservation de l'ordre, et donc par suite, la conservation du tri des données entre l'espace uniformisé et l'espace de départ). La technique employé ci-dessus peut donc être généralisée pour trier, par exemple, une liste de nombres réels distribués suivant une loi normale, de moyenne et de variance connues.

Les enseignements à retenir de cet exemple sont les suivants.

- Il est parfois possible d'optimiser un algorithme supra-linéaire en divisant le travail en plusieurs tâches réduites (ce que nous avions souligné dans la conclusion du chapitre 3), en particulier lorsque le travail de recomposition est simple et rapide.
- Il est souvent intéressant de connaître la distribution statistique des données à traiter. Il n'en résulte généralement pas d'amélioration sur les complexités minimale et maximale (les cas extrêmes sont toujours réalisables). En revanche, il est possible d'utiliser cette connaissance pour réduire le temps moyen de traitement. Nous verrons un exemple similaire au paragraphe 4.5.

Exercice 4.5. Ecrire l'algorithme de tri par paquets en ADL. La procédure prendra en argument le tableau T ainsi que les bornes a et b de l'intervalle d'échantillonnage des données.

4.2 Listes chaînées

Contrairement au cas d'une structure de tableau, une liste chaînée n'offre qu'un accès séquentiel à ses éléments, c'est-à-dire que pour récupérer un élément i donné de la liste, il faut d'abord parcourir tous les éléments de 1 à $i - 1$. En contrepartie, la taille d'une liste chaînée est dynamiquement modifiable et il est très facile d'y insérer ou d'en retirer des éléments, y compris en milieu de liste.

Nous passons ici en revue les différents types de listes chaînées utilisés en pratique, mais pour ce faire, nous devons tout d'abord introduire un nouveau type de données : les pointeurs.

4.2.1 Notion de pointeur

Un pointeur est un identificateur, destiné à contenir l'adresse physique d'une autre donnée. Prenons un exemple simple, en représentant l'espace mémoire d'une machine comme un ensemble de cases. Chaque case est munie d'une adresse permettant de savoir où aller chercher la case dans l'espace mémoire. Pour différencier les adresses des valeurs contenues dans les cellules, nous les ferons précédé d'un symbole @ dans les schémas.

Lorsque l'on écrit l'instruction : $x \leftarrow 12$, l'ordinateur enregistre la valeur 12 dans une case mémoire, dont l'adresse est @23541. Cette adresse est quant à elle enregistrée dans l'identificateur, si bien qu'à chaque fois que l'on appelle x , la machine récupère cette adresse, ce

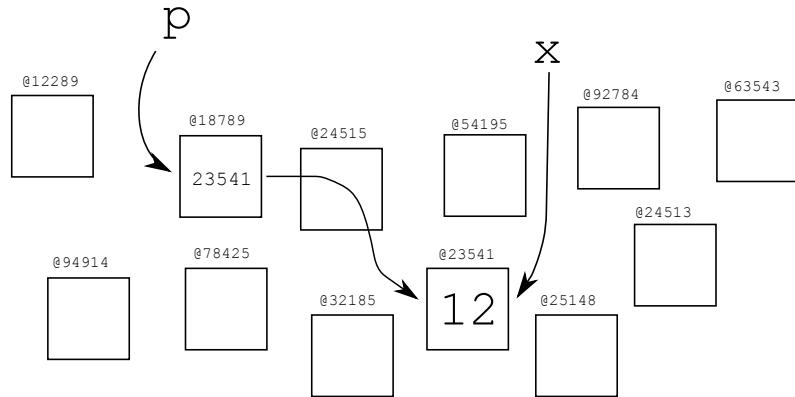


FIGURE 4.5 – Mécanisme de pointeur pour référencer une donnée

qui lui permet d'aller lire la données inscrite dans la cellule correspondante.

En pratique, on peut souhaiter avoir un deuxième identificateur qui référence la même donnée. Si on écrit $p \leftarrow x$, alors la machine va allouer une seconde case mémoire, y inscrire une copie de la valeur contenue par x (c'est-à-dire 12), puis sauvegarder l'adresse de cette case dans l'identificateur p . Ce n'est pas ce que nous souhaitons, puisque x et p désignent ici deux données différentes (mais toutes deux égales à 12). Si on modifie l'une, on ne modifie pas l'autre. On ne peut donc pas dire que p référence la donnée contenue par x .

Par exemple, après le code suivant :

$$x \leftarrow 12; p \leftarrow x; x \leftarrow 0;$$

l'identificateur p contient toujours la valeur 12, tandis que x a été modifié à 0.

Pour référencer une donnée à l'aide d'un second identificateur, nous utiliserons en ADL l'affectation par référence :

$$p \rightarrow x$$

ce qui se lit : l'identificateur p référence x .

On généralise cette expression pour obtenir une syntaxe similaire à l'affectation classique :

Syntaxe : *affectation par référence*

$$\text{ID1}_{ea_1, ea_2, \dots ea_m} \rightarrow \text{ID2}_{ea_1, ea_2, \dots ea_n}$$

Notons que cette fois, les deux côtés de l'opérateur de référence doivent être des identificateurs, éventuellement dimensionnés, contrairement à l'affectation classique, où la partie droite pouvait dans le cas le plus général être une expression arithmétique.

En admettant que nous disposons d'une fonction PRINT permettant d'afficher le contenu d'un identificateur dans la console, que devrait afficher le code suivant ?

```
x ← 12; p → x; y ← p; x ← 0; PRINT(y);
```

Correction : de la même manière que précédemment, la console affiche 12. En effet, lors de l'affectation de la variable *y*, le contenu de la case mémoire référencée par le pointeur *p* (c'est-à-dire la valeur 12) est copié dans une autre case mémoire, référencée par l'identificateur *y*. La modification est effectuée sur *x* (et donc par voie de fait sur la valeur de la case référencée par *p*) mais pas sur *y*. La valeur de *y* est donc toujours 12.

En revanche, chacune des deux instruction suivantes retourne la valeur 0 dans la console :

```
x ← 12; p → x; x ← 0; y → p; PRINT(y);
```

```
x ← 12; p → x; x ← 0; PRINT(p);
```

Dans la terminologie des langages de programmation, on dit que l'affectation classique \leftarrow est une *copie* tandis que l'affectation \rightarrow est une *affectation par référence*. En règle générale (c'est le cas de Java et du C++), les types primitifs (entiers, flottants, booléens...) sont affectés par copie tandis que les types structurés (tels que ceux que nous étudions dans ce chapitre et qui généralement sont définis par l'utilisateur) sont affectés par référence. Dans ce cours, nous spécifierons à chaque fois de quel type d'affectation nous parlons.

4.2.2 Liste simplement chaînée

Une liste chaînée utilise avantageusement le mécanisme de pointeur. Chaque élément du tableau contient ainsi 2 informations :

- La valeur de l'élément
- Un pointeur référençant l'adresse physique de l'élément suivant

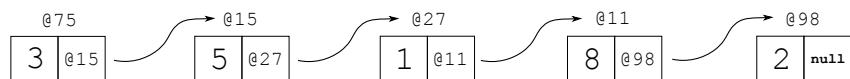


FIGURE 4.6 – Liste simplement chaînée contenant 5 éléments

La liste chaînée ci-dessus est donc équivalente au tableau $[3, 5, 1, 8, 2]$, seulement, il est impossible de récupérer l'élément 1 par exemple, sans parcourir les éléments 3 et 5.

On dit que la liste est simplement chaînée puisque le référencement est effectué uniquement avec l'élément suivant.

Pour identifier le dernier élément de la liste, le dernier pointeur est fixé à la valeur conventionnelle *null*.

4.2.3 Liste doublement chaînée

Dans une liste doublement chaînée, chaque élément contient trois informations : la valeur de l'élément, un pointeur vers l'élément suivant et un second pointeur vers l'élément précédent.

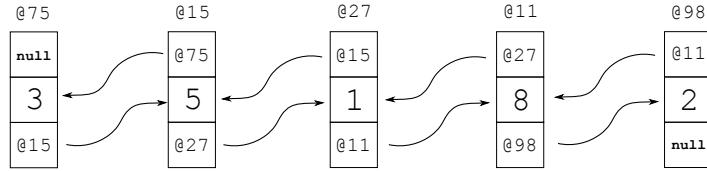


FIGURE 4.7 – Liste doublement chaînée contenant 5 éléments

4.2.4 Liste chaînée circulaire

Une liste circulaire est une liste chaînée dont le pointeur du dernier élément référence l'adresse du premier élément.

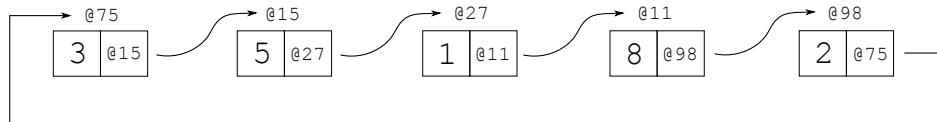


FIGURE 4.8 – Liste circulaire contenant 5 éléments

4.2.5 Fonctionnalités d'une liste chaînée

En tant que type abstrait de données (*Abstract Data Type*), une liste chaînée se doit d'implémenter certains nombres de fonctionnalités typiques, telles que l'accès à un élément donné, l'insertion/suppression d'un élément, le décompte des données contenues dans la liste...

Chaque élément L d'une liste contiendra deux composantes :

- La valeur de l'élément, que nous noterons $L.\text{val}$
- Un pointeur vers l'élément suivant, que nous noterons $L.\text{next}$

Pour créer une liste chaînée, il suffit donc simplement d'affecter sa valeur et son pointeur au premier élément de la liste :

CREATE() : $\llbracket L.\text{val} \leftarrow 0; \text{ } \textcircled{C} \text{ valeur par défaut } \text{ } \textcircled{C} L.\text{next} \rightarrow \text{null}; \text{ CREATE} \leftarrow L \rrbracket$

Disposant d'une liste chaînée L , nous pouvons à présent tester si la liste est vide (c'est-à-dire plus précisément si elle contient ou non un élément suivant).

ISEMPTY($L \downarrow$) : $\llbracket \text{ISEMPTY} \leftarrow L.\text{next} \neq \text{null} \rrbracket$

Notons que la fonction ISEMPTY, comme toutes les fonctions qui vont suivre dans cette section, est spécifique à l'objet de type liste chainée. Il est alors possible d'utiliser la notation fonctorielle pour attacher ce module à l'objet, de sorte à retirer L de la liste des arguments (il s'agit d'un argument implicite). Ainsi, les deux instructions suivantes deviendront équivalentes :

$b \leftarrow \text{ISEMPTY}(L);$ et $b \leftarrow L' \text{ISEMPTY}();$

Avec la notation de droite, nous parlerons de *méthode*, et non plus de module.

On peut maintenant créer une fonction permettant de compter le nombre d'éléments contenus dans une liste :

SIZE($L \downarrow$) : $\llbracket \text{SIZE} \leftarrow 1; \left\{ \begin{array}{l} / \overline{\text{ISEMPTY(L)}} \\ L \rightarrow L.\text{next}; \text{SIZE} \leftarrow \text{SIZE} + 1 \end{array} \right\} \rrbracket$

Notons qu'après application de cette fonction, le curseur de chaîne est toujours positionné sur le premier élément puisque la liste est passé par valeur \downarrow dans la procédure. Les modifications opérées sur la liste ne sont donc pas effectives à l'extérieur du module.

On peut ensuite écrire les fonctions permettant de récupérer le premier et le dernier élément de la liste :

HEAD($L \downarrow$) : $\llbracket \text{HEAD} \leftarrow L.\text{val} \rrbracket$

TAIL($L \downarrow$) : $\llbracket \left\{ \begin{array}{l} / \overline{\text{ISEMPTY(L)}} \\ L \rightarrow L.\text{next} \end{array} \right\} \text{TAIL} \leftarrow L.\text{val} \rrbracket$

De la même manière, il nous faut deux fonctions permettant d'insérer un élément en début et en fin de liste :

INSERT_HEAD($L \uparrow, x \downarrow$) : $\llbracket L2 \leftarrow \text{CREATE}(); L2.\text{val} \leftarrow x; L2.\text{next} \rightarrow L; \rrbracket$

INSERT_TAIL($L \uparrow, x \downarrow$) : $\llbracket L2 \leftarrow \text{CREATE}(); L2.\text{val} \leftarrow x; \left\{ \begin{array}{l} / \overline{\text{ISEMPTY(L)}} \\ L \rightarrow L.\text{next} \end{array} \right\} L.\text{next} \rightarrow L2 \rrbracket$

On peut également insérer un élément à un indice i donné :

INSERT($L \downarrow, x \downarrow, i \downarrow$) : $\llbracket k \leftarrow 1; L2 \leftarrow \text{CREATE}(); L2.\text{val} \leftarrow x; \begin{cases} / \overline{\text{ISEMPTY}(L)} \cup k < i \\ L \rightarrow L.\text{next}; k \leftarrow k + 1 \end{cases} \} \text{ISEMPTY}(L) ? ! | \& L2 \leftarrow \text{CREATE}(); L2.\text{val} \leftarrow x; L2.\text{next} \rightarrow L.\text{next}; L.\text{next} \rightarrow L2 \rrbracket$

4.3 Files et piles

Dans cette sections, nous introduisons simultanément les files et les piles de données. Ces deux structures peuvent être aisément implémentées à l'aide de liste chaînées.

4.3.1 Définitions

Définition : file et pile

Une file (resp. pile) est une structure de données contenant un ensemble d'éléments et munie de fonctions permettant essentiellement d'effectuer les trois opérations suivantes :

- retourner la taille de la file (i.e. son nombre d'éléments)
- ajouter un élément en tête de liste
- retirer l'élément de queue (resp. tête) de liste

Ces deux structures sont donc différencier par la méthode d'accès aux éléments. On désigne également une file par l'acronyme FIFO (*First In - First Out*). C'est le principe d'une file d'attente par exemple. Pour une pile, on parle de LIFO (*Last In - First Out*), et on peut imaginer l'ordre d'entrées-sorties des gens dans un ascenseur (à une seule porte!) comme une pile. Le problème des tours de Hanoï par exemple, se modélise très bien avec trois piles de données.

4.3.2 Implémentation

Une pile se schématisé donc de la manière suivante :

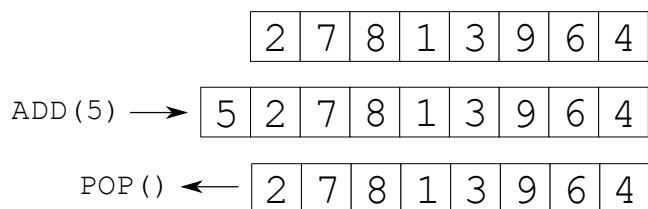


FIGURE 4.9 – Pile de données contenant 8 éléments

L'instruction INSERT est directement implémentée à partir du module INSERT_HEAD des listes chaînées. De la même manière, l'instruction POP correspond simplement à une itération du parcours d'une liste chaînée :

$$\text{POP}(L \Downarrow) : \llbracket \text{POP} \leftarrow L.\text{next}; L \rightarrow L.\text{next} \rrbracket$$

Nous laissons le soin au lecteur d'écrire la fonction POP pour le cas d'une file :

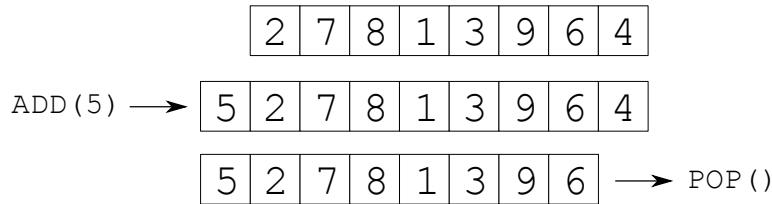


FIGURE 4.10 – File de données contenant 8 éléments

Nous verrons à la fin de ce chapitre une manière de modifier dynamiquement les priorités des éléments dans une file à l'aide d'un tas binaire.

4.4 Arbres

4.4.1 Définitions

Un arbre est une structure permettant de stocker les données d'une manière organisée de sorte à pouvoir y accéder plus rapidement. Dans le domaine de la théorie des graphes, on parle d'un graphe non-orienté, acyclique et connexe. En conséquence, on a trivialement la propriété d'unicité des chemins entre couples de sommets donnés. Pour faciliter la compréhension de la structure, on représente l'arbre à partir d'un noeud spécifique, que l'on appelle *racine*⁴, chaque connexion est alors placée graphiquement en dessous de son noeud parent.

On appelle *niveau* ou *profondeur* d'un noeud, le nombre de noeuds le séparant de la racine de l'arbre. Par définition, le chemin séparant ce noeud à la racine est unique, ce qui nous permet de définir cette quantité sans ambiguïté. Graphiquement, on essaye si possible de faire figurer les noeuds de même profondeur au même niveau.

Remarque : pour un graphe acyclique donné, la profondeur de l'arbre dépend du choix du noeud racine. Par exemple, sur la figure précédente, la profondeur de l'arbre du milieu vaut 4 tandis qu'elle vaut 5 dans la configuration de droite.

Les successeurs d'un noeud donné sont appelés ses *fils*. Le noeud précédent d'un ensemble de noeuds est appelé noeud *parent*. Un noeud qui ne possède pas de successeur est

4. Notons que pour une même structure d'arbre, tout noeud peut être choisi comme un noeud racine

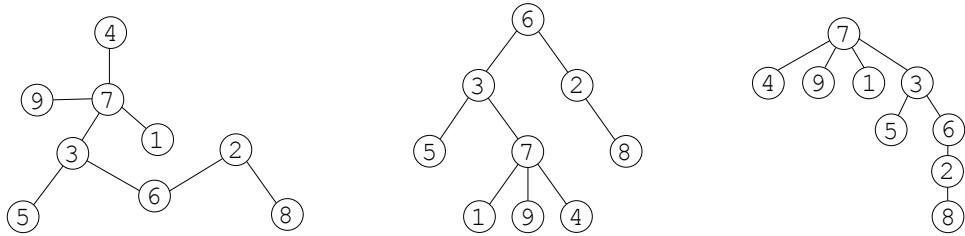


FIGURE 4.11 – Deux représentations arborescentes d'un graphe connexe acyclique (à gauche) en prenant comme racine les noeuds 6 (au milieu) et 7 (à droite).

appelé une *feuille* de l'arbre.

Pour deux noeuds n_1 et n_2 : on dit que n_2 est un descendant de n_1 si et seulement si le chemin reliant la racine à n_2 passe par le lien n_1 .

Définition : *arbre binaire*

On appelle arbre binaire, un arbre dont chaque noeud possède au plus deux fils.

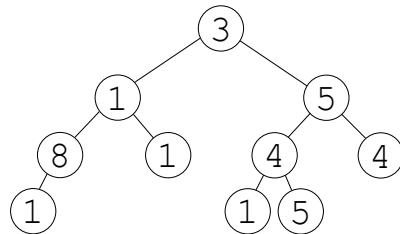


FIGURE 4.12 – Arbre binaire

Définition : *arbre binaire complet*

On appelle arbre binaire complet, un arbre binaire dont toutes les feuilles ont la même profondeur.

Définition : *profondeur d'un arbre*

On appelle profondeur d'un arbre, la profondeur maximale des feuilles de l'arbre

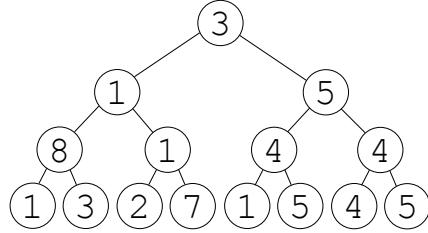


FIGURE 4.13 – Arbre binaire complet

D'un point de vue informatique, nous implémentons un arbre binaire de manière similaire au cas de la liste chaînée. Chaque élément N possède quatre champs :

- $N.\text{val}$ indiquant la valeur contenue par le noeud
- $N.\text{left}$ contenant un pointeur vers le premier noeud fils
- $N.\text{right}$ contenant un pointeur vers le second noeud fils
- $N.\text{parent}$ contenant un pointeur vers le noeud parent

Notons que dans le cas général d'un arbre binaire, un noeud peut contenir 0, 1 ou 2 fils. Lorsqu'un noeud ne possède aucun fils, les deux pointeurs *left* et *right* pointent vers la valeur *null*. Si l'arbre ne contient qu'un fils, il est référencé dans *left*, tandis que *right* pointe vers *null*. Le pointeur *noeudParent* contient quant à lui la valeur *null* si nous sommes sur la racine de l'arbre.

Avec ces fonctions, nous pouvons écrire une fonction permettant de calculer la profondeur d'un arbre, en utilisant le principe *diviser pour régner*.

DEPTH($T \downarrow$) : $\llbracket T.\text{left} = \text{null} ? \text{DEPTH} \leftarrow 1 \mid \text{et } T.\text{right} = \text{null} ? \text{DEPTH} \leftarrow 1 + \text{DEPTH}(T.\text{left}) \mid \text{et } \text{DEPTH} \leftarrow 1 + \max(\text{DEPTH}(T.\text{left}), \text{DEPTH}(T.\text{right})) \rrbracket$

On montrera à titre d'exercice que cette fonction termine, et on calculera sa complexité dans le pire des cas (en terme de nombre d'appels récursifs) en fonction de la profondeur n de l'arbre. Démontrer que la fonction est correcte.

Nous supposons à présent avoir à disposition un tableau de données L que l'on souhaite structurer sous forme d'un arbre binaire complet. Toujours en utilisant la récursivité, nous pouvons écrire ce code de manière très simple :

BUILD($L \downarrow, i \downarrow, j \downarrow$) : $\llbracket T.\text{val} \leftarrow L_i; i = j ? T.\text{left} \leftarrow \text{null}; T.\text{right} \leftarrow \text{null}; \mid \text{et } T.\text{parent} \leftarrow \text{null}; n \leftarrow (\text{SIZE}(L) - 1) \div 2; T.\text{left} \rightarrow \text{BUILD}(L, i + 1, n); T.\text{right} \rightarrow \text{BUILD}(L, n + 1, j); \text{BUILD} \rightarrow T \rrbracket$

Exercice 4.6. En notant $f(t)$ le nombre de feuilles d'un arbre t et $h(t)$ sa hauteur, démontrer l'inégalité suivante :

$$h(t) + 1 \leq f(t) \leq 2^{h(t)}$$

Notons que lorsque la borne inférieure est atteinte, on parle d'arbre *peigne droit*. Si c'est la borne supérieure qui est atteinte alors on a un arbre binaire complet.

4.4.2 Parcours d'un arbre

Parcourir un arbre signifie énumérer les valeurs de ses noeuds de manière exhaustive et dans un ordre bien précis.

Le *parcours en largeur* signifie que l'on commence à épuiser complètement un niveau avant de passer au suivant. En intelligence artificielle appliquée au jeu d'échecs par exemple, le parcours en largeur consiste à étudier tous les déplacements possibles au prochain tour de jeu, puis pour chaque déplacement trouvé on étudie toutes les réponses possibles de l'adversaire et ainsi de suite. Cette stratégie est généralement adaptée au cas où l'élément recherché est proche de la racine, mais que l'on ne peut pas deviner a priori dans quelle branche.

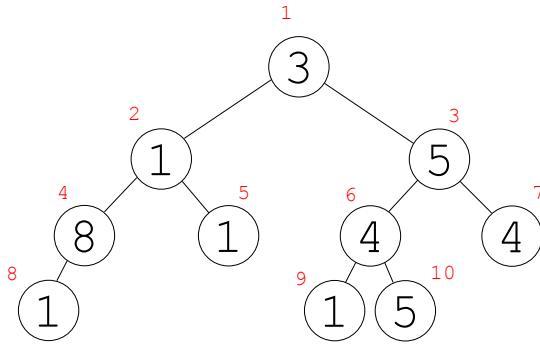


FIGURE 4.14 – Parcours en largeur d'un arbre binaire

A titre d'exemple, le parcours en largeur de l'arbre ci-dessus donne la séquence d'entiers : [3, 1, 5, 8, 1, 4, 4, 1, 1, 5].

Notons que par défaut, sur un niveau donné, on commence toujours par le noeud de gauche, mais il s'agit là d'une convention arbitraire.

Le *parcours en profondeur* signifie que l'on commence par épuiser complètement une branche avant de revenir en arrière et de tester la branche suivante. Il s'agit du cas où l'élément à rechercher est a priori situé loin de la racine, mais que l'on est capable de décrire approximativement le chemin à emprunter pour le trouver.

Le parcours en largeur de l'arbre ci-dessus donne la séquence : [1, 1, 8, 1, 1, 5, 4, 1, 9, 4].

Remarquons que dans ce type de parcours, on commence implicitement par parcourir le noeud parent avant de parcourir la branche de gauche, puis on termine par la branche de

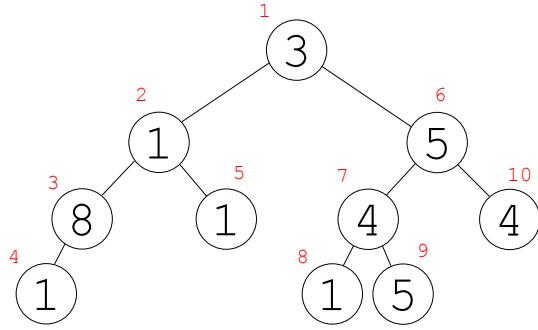


FIGURE 4.15 – Parcours en profondeur d'un arbre binaire

droite. On dit que le parcours est *préfixe*, puisque le noeud parent est parcouru en premier.

Il existe également un parcours dit *suffixe*, où le noeud est parcouru en dernier, ce qui sur notre exemple donne la séquence d'entiers : [1, 8, 1, 1, 1, 5, 4, 4, 5, 3].

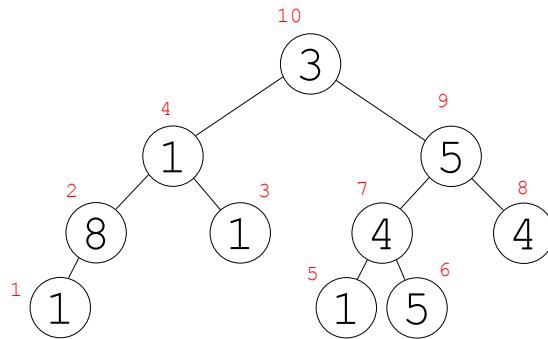


FIGURE 4.16 – Parcours en profondeur suffixe d'un arbre binaire

Enfin, on parle de parcours *infixe* lorsque le noeud parent est situé entre les deux branches descendantes de ses fils. On obtient cette fois la séquence : [1, 8, 1, 1, 3, 1, 4, 5, 5, 4].

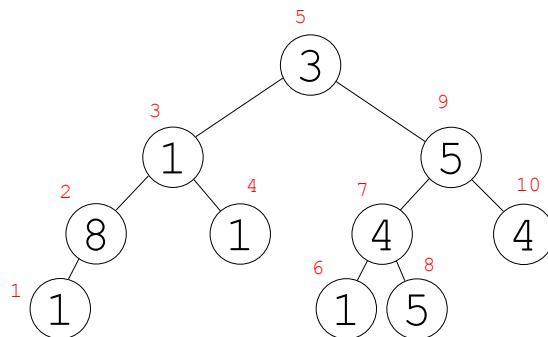


FIGURE 4.17 – Parcours en profondeur infixe d'un arbre binaire

Exemple : la recherche du chemin le plus court par l'algorithme de Dijkstra (que nous

aurons l'occasion d'étudier en détail à la fin du chapitre 5) est un parcours de graphe en largeur : on teste dans un premier temps les positions voisines du noeud de départ, puis on élargit la zone de recherche jusqu'à trouver le noeud d'arrivée.

Dans la plupart des cas, on utilise une solution hybride, consistant à amorcer un parcours en largeur de l'arbre, avant d'éliminer les branches peu prometteuses permettant ainsi de finir avec seulement quelques possibilités, sur lesquelles on effectue un parcours en profondeur. Il s'agit de la procédure dite de séparation et évaluation (*branch-and-bound*). A l'inverse, on peut aussi amorcer un parcours en profondeur si l'on a de bonnes raisons de penser que l'élément recherché se situe dans une partie bien précise de l'arbre, puis à une certaine profondeur, on utilise un parcours en largeur pour terminer la procédure de recherche.

On peut écrire une fonction permettant de rechercher un élément dans un arbre binaire suivant un parcours en largeur.

SEARCH($T \downarrow, x \downarrow$) : $\llbracket T.\text{val} = x ? \text{SEARCH} \rightarrow T; ! | _ L_0 \rightarrow T; c \leftarrow 1;$
 $\left\{ \begin{array}{l} /c \neq 0 \\ d \leftarrow c; c \leftarrow 0; \end{array} \right. ; \left\{ \begin{array}{l} ^d \\ _{i:1} \end{array} \right. L_i.\text{left} \neq \text{null} ? L_i.\text{val} = x ? \text{SEARCH} \rightarrow L_i; ! | _ L'_c \rightarrow$
 $L_i.\text{left}; c \leftarrow c + 1; | _ L_i.\text{right} \neq \text{null} ? L_i.\text{val} = x ? \text{SEARCH} \rightarrow L_i; ! | _ L'_c \rightarrow$
 $L_i.\text{right}; c \leftarrow c + 1; | _ \end{array} \right\}_i L \rightarrow L' \} \text{SEARCH} \rightarrow \text{null} \rrbracket$

Remarque : il est possible de réécrire cette fonction à l'aide d'une structure de file, permettant à chaque niveau de stocker les noeuds à décomposer au niveau suivant.

Le recherche en profondeur s'exprime quant à elle récursivement :

SEARCH_DEPTH($T \downarrow, x \downarrow$) : $\llbracket T.\text{val} = \text{null} ? \text{SEARCH_DEPTH} \rightarrow \text{null}; ! | _ T.\text{val} = x ? \text{SEARCH_DEPTH} \rightarrow T; ! | _ \text{SEARCH_DEPTH} \rightarrow \text{SEARCH_DEPTH}(T.\text{left}, x); \text{SEARCH_DEPTH} \neq \text{null} ? ! | _ \text{SEARCH_DEPTH} \rightarrow \text{SEARCH_DEPTH}(T.\text{right}, x); \rrbracket$

Exercice 4.7. Ecrire une fonction permettant de rechercher un élément dans un arbre binaire suivant un parcours en profondeur de type suffixe et infixe.

La complexité minimale de la recherche (tant en largeur qu'en profondeur) vaut 1 (si l'élément à rechercher est contenu par le premier noeud parcouru). La complexité dans le pire des cas vaut n . Dans le cas général où la position de l'élément recherché dans l'arbre suit une distribution de probabilité uniforme, la complexité en moyenne est $n/2$. Un arbre binaire ne permet donc de gagner du temps dans la recherche que sous l'hypothèse d'une distribution non-uniforme, i.e. si l'on dispose d'un a priori sur la position de l'élément.

4.4.3 Arbres binaires de recherche

Un arbre binaire de recherche (ABR) est un arbre binaire respectant la contrainte que chaque étiquette du sous-arbre gauche est inférieure ou égale à celle du noeud parent qui elle-même est inférieure ou égale à toute étiquette du sous-arbre droit.

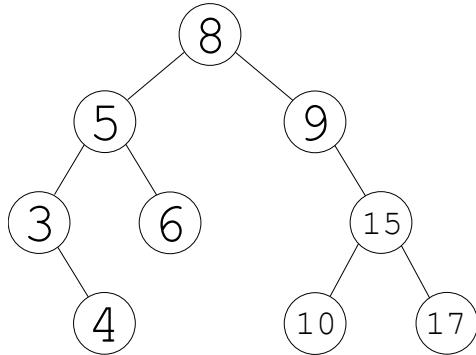


FIGURE 4.18 – Arbre binaire de recherche

La recherche d'un élément s'effectue cette fois par dichotomie :

SEARCH_ABR($T \downarrow, x \downarrow$) : $\llbracket T = \text{null} ? \text{SEARCH_ABR} \rightarrow \text{null}; ! | _ T.\text{val} = x ? \text{SEARCH_ABR} \rightarrow T; ! | _ T.\text{val} < x ? \text{SEARCH_ABR} \rightarrow \text{SEARCH_ABR}(T.\text{left}, x) | \text{SEARCH_ABR} \rightarrow \text{SEARCH_ABR}(T.\text{right}, x) _ \rrbracket$

La complexité dans le pire des cas est linéaire (le pire des cas est atteint si l'arbre est un peigne droit). La complexité moyenne est en $\Theta(\log n)$.

Exercice 4.8. Compléter le modèle avec des fonctions permettant d'ajouter un élément à un ABR, construire un ABR à partir d'une liste de valeurs et supprimer un élément d'un ABR. Indiquer la complexité de chacune de ces fonctions.

Indication : pour supprimer un noeud n , il faut d'abord le rechercher. On considère ensuite plusieurs cas suivant que n est une feuille, possède 1 ou 2 fils. Dans le premier cas, la suppression est triviale. Dans les deux autres cas, il faut analyser le problème pour trouver un noeud (ou plus exactement une feuille) de substitution qui puisse remplacer le noeud supprimé tout en conservant la propriété d'un ABR.

Définition : arbre binaire équilibré

Un arbre binaire de recherche est dit équilibré si, pour chaque noeud, la hauteur de chacun des deux sous-arbres partant de ce noeud diffère au plus de 1.

On peut alors écrire une procédure simple pour tester si un ABR est équilibré :

IS_ABR($T \downarrow$) : $\left[\begin{array}{l} T = \text{null} ? \text{ IS_ABR} \leftarrow \uparrow; ! | \downarrow d \leftarrow \text{DEPTH}(T.\text{left}) - \text{DEPTH}(T.\text{right}); \\ b \leftarrow \text{ABS}(d) \leqslant 1; \text{ IS_ABR} \leftarrow b \cap \text{IS_ABR}(T.\text{left}) \cap \text{IS_ABR}(T.\text{right}) \end{array} \right]$

Pour un arbre binaire de recherche équilibré, la recherche d'un élément se fait avec une complexité dans le pire des cas en $\Theta(\log n)$ (la démonstration est similaire au cas du tri rapide étudié au paragraphe 4.1.3).

En effet, pour un ABR équilibré, la hauteur h vérifie la double inégalité :

$$\sum_{i=0}^{h-1} 2^i \leqslant n \leqslant \sum_{i=0}^h 2^i$$

où n est le nombre de noeuds de l'arbre. Par sommation de suites géométriques, on obtient l'encadrement suivant :

$$2^h - 1 \leqslant n \leqslant 2^{h+1} - 1$$

D'où : $h \leqslant \log(n+1) \leqslant h+1$ et on vérifie bien que la hauteur de l'arbre (et donc le nombre d'étapes de recherche) est en $\Theta(\log n)$.

Remarque : le parcours infixé d'un arbre binaire de recherche permet d'effectuer un tri des éléments dans l'ordre croissant. La preuve est immédiate à partir de la propriété de base d'un ABR.

4.4.4 Exemples d'application

Dans cette section, nous passons en revue quelques-unes des nombreuses applications possibles des arbres binaires. Notons que nous avons déjà utilisé des arbres dans les chapitres précédents de ce cours, en particulier pour décrire les appels récursifs du calcul des termes de la suite de Fibonacci, ou encore pour démontrer qu'un algorithme de tri de tableaux est au mieux quasi-linéaire.

Tri par tas

On appelle tas binaire max (resp. min) une structure d'arbre binaire telle que :

- l'étiquette de chaque noeud est supérieure (resp. inférieure) à chacune des étiquettes de ses fils
- chaque niveau de l'arbre est entièrement rempli (de gauche à droite) avant d'entamer le niveau inférieur

On donne ci-dessous deux exemples de tas binaires construits à partir du même tableau de données : un tas min (à gauche) et un tas max (à droite).

```
L = [13,28,17,64,52,25,8,35,85,44]
```

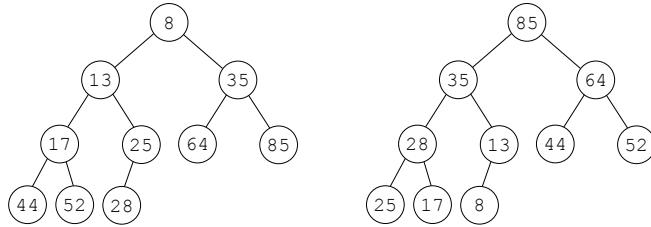


FIGURE 4.19 – Deux exemples de tas binaires min (à gauche) et max (à droite)

Remarque 1 : pour un tableau de nombres donnés, il existe généralement plusieurs tas binaires possibles⁵.

L'intérêt d'un tas binaire est de conserver l'élément minimal (ou maximal en fonction du type de tas) en haut de la pile, mais aussi de une mise-à-jour facile de la pile, en retirant, en ajoutant des éléments ou bien encore en modifiant leurs étiquettes. Le tas binaire permet donc de gérer une file de priorité, dont les priorités sont dynamiquement modifiables, contrairement aux piles et aux files introduites précédemment, dans lesquelles l'ordre de sortie des éléments était prédéterminé par leur ordre d'insertion.

Dans un tas binaire, pour insérer un élément, il suffit de le positionner au premier noeud disponible, puis de le faire remonter dans le tas par échange successifs (suivant le modèle du tri à bulles), jusqu'à ce qu'il atteigne une position permettant à la structure de respecter la contrainte de tas binaire. Cette opération s'appelle *percolation vers le haut*.

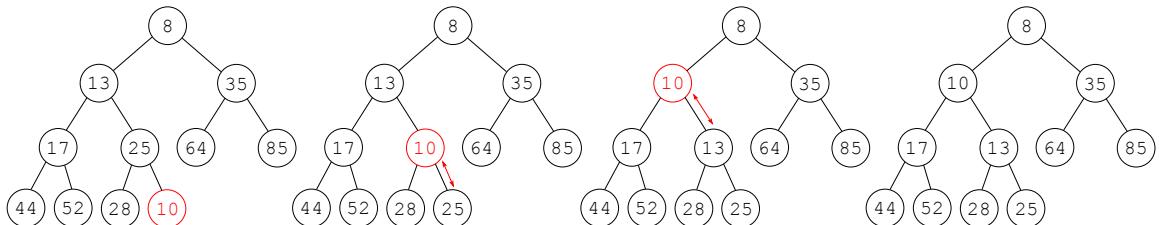


FIGURE 4.20 – Percolation vers le haut dans un tas binaire min

Après ajout de l'élément 10, il suffit de comparer cet élément à son noeud parent et de l'inverser jusqu'à ce qu'il soit à la bonne place, c'est-à-dire jusqu'à ce que l'étiquette de son noeud parent soit inférieure à la sienne (dans le cas d'un tas min).

Notons que cette opération nécessite :

- de trouver la prochaine position libre, ce qui se fait en un temps logarithmique si l'on conserve une variable (par exemple dans la racine) contenant la taille du tas binaire
- de permuter l'élément inséré jusqu'à ce qu'il soit à sa place, ce qui au plus implique un nombre de permutation égal à la profondeur de l'arbre (qui est équilibré, donc

5. Ce nombre est donné par la séquence A056971 de l'encyclopédie en ligne des suites de nombres entiers et, à titre d'exemple, vaut environ 50 milliards pour une pile de 20 éléments.

ceci se fait également en un nombre logarithmique d'étapes).

La complexité de l'insertion d'un élément dans la file de priorité est un $\Theta(\log n)$.

Notons qu'en utilisant une structure de tableau pour modéliser le tas binaire (comme nous le ferons en exercice) on peut accéder à la première position libre en temps constant.

Pour retirer la racine du tas, il suffit de procéder de la même manière, en la remplaçant par l'élément le plus en bas à droite. On effectue alors l'opération inverse de percolation vers le bas, jusqu'à obtenir une structure de tas binaire.

Remarquons qu'il y a cette fois-ci deux choix possibles pour l'échange, suivant que l'on permute l'élément avec son fils gauche ou droit. Pour optimiser l'opération, dans le cas où les deux fils sont plus petits (resp. grands) que le noeud à percoler, l'échange se fait toujours avec le plus grand (resp. petit) des deux fils dans le cas d'un tas min (resp. max).

Par exemple, dans l'illustration suivante, on retire la racine 8 et on la remplace par le dernier élément du tas, à savoir 28. L'arbre a alors perdu sa structure d'arbre binaire, ce que l'on corrige en percolant 28 vers le bas. A la première itération, seul le fils gauche de 28 est inférieur à lui, on échange donc 28 avec son fils gauche. A la seconde itération, les deux fils de 28 sont plus petits que lui, on l'échange donc avec le plus grand de ses deux fils, c'est-à-dire 25. On retrouve alors la propriété de tas binaire puisqu'à la fin du processus 28 est plus petit que son unique fils.

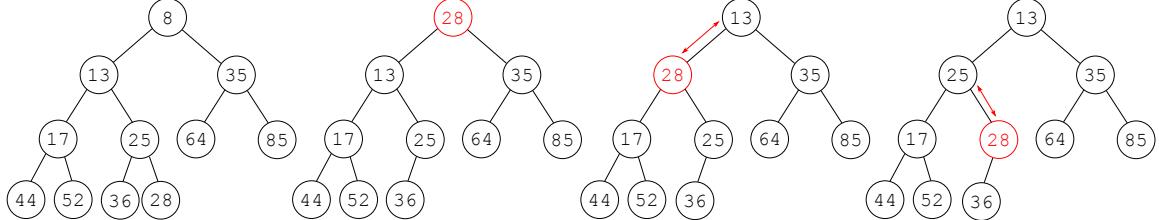


FIGURE 4.21 – Percolation vers le bas dans un tas binaire min

Cette structure donne lieu à un nouveau type de tri, où l'on considère les éléments d'un tableau un à un, et on construit un tas binaire (min ou max) de manière incrémentale. Il s'agit du *tri par tas* (on dit aussi parfois *tri par arbre*). Etant donné qu'il y a n éléments dans le tableau, la construction du tas binaire se fait en un temps $\mathcal{O}(n \log n)$. Nous avons vu qu'il était possible de retirer la racine du tas en un temps logarithmique. Extraire les n éléments du tas dans l'ordre croissant (ou décroissant) se fait donc également en un temps quasi-logarithmique et la complexité du tri qui en résulte est $\mathcal{O}(n \log n)$, ce qui est optimal pour un tri par comparaison.

Notons qu'on peut faire encore plus rapide et construire le tas en un temps linéaire. Pour cela, plutôt que de placer les éléments de manière incrémentale avec le processus de percolation, il est plus rapide de créer un arbre binaire aléatoire (en remplissant l'arbre ligne par ligne dans l'ordre d'arrivée des éléments), ce qui se fait en un temps linéaire. Il ne reste alors plus qu'à parcourir les sommets niveau par niveau en partant de la racine et en effectuant une percolation vers le bas de chaque élément. La complexité dans le pire des cas

de cette opération est égale à la somme sur chaque niveau du produit du nombre de noeuds présents sur le niveau par le nombre maximal d'échanges à effectuer par la percolation :

$$C_{max}(n) = \sum_{i=0}^{\log_2 n} i 2^{\log_2 n - i} = n \sum_{i=0}^{\log_2 n} \frac{i}{2^i}$$

En remarquant par exemple que i est dominé par 1.1^i , le terme de la somme est majoré par une suite dont la série associée est convergente. La somme est donc réduite à un $\mathcal{O}(1)$ et la construction du tas binaire se fait en un temps $\mathcal{O}(n)$. Ceci s'explique par le fait que les noeuds de bas niveau dans l'arbre sont beaucoup plus nombreux relativement et nécessitent moins d'échanges de percolation que les noeuds du haut de l'arbre. On peut donc comparer le tri par tas à un tri à bulles en cascade qui se parallélise au fur et à mesure de l'exécution de l'algorithme.

Une fois que l'on est en mesure d'effectuer les opérations de percolation dans les deux sens, il est aisément de modifier la valeur d'une étiquette. Il suffit alors simplement de percoler le noeud modifié, vers le haut ou vers le bas suivant que la valeur de l'étiquette a été augmentée ou diminuée et qu'il s'agit d'un tas min ou max.

Nous terminerons cette section en soulignant qu'il existe d'autres structures de tas, permettant de réaliser des opérations plus complexes de manière optimale. Par exemple, le tas binomial permet de fusionner deux tas, ou encore le tas de Fibonacci dans lequel l'insertion se fait en un temps constant. Tous deux diffèrent en cela qu'ils sont composés d'un ensemble d'arbres.

Codage de Huffman

Le codage de Huffman permet, à l'aide d'un arbre binaire, d'effectuer une compression sans perte d'un message.

Les symboles de l'alphabet à coder sont stockés dans les feuilles de l'arbre. Pour coder une lettre, il suffit alors simplement de trouver le chemin reliant la racine à cette lettre. Chaque bifurcation à gauche ajoute la valeur 0 au code. À l'inverse une bifurcation à droite ajoute la valeur 1. Par exemple, dans l'illustration ci-dessous, on représente un arbre binaire de codage sur l'alphabet {A,B,C,D,E,F}. Le codage de la lettre A par exemple est '00' (deux bifurcations à gauche depuis la racine) tandis que celui de la lettre D est '1001' (une bifurcation à droite, suivie de deux à gauche, puis une à droite).

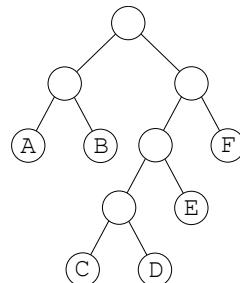


FIGURE 4.22 – Arbre de codage pour un alphabet à 6 symboles

Si l'arbre est complet, par définition la longueur de tous les symboles diffère au plus d'un bit. L'astuce du codage de Huffman est justement d'observer que les symboles n'ont généralement pas la même probabilité de survenir. Un arbre de Huffman place les symboles les moins courants aux profondeurs les plus grande de l'arbre. A l'inverse, les symboles courants (tels que le 'e' ou le 'a' en langue française) sont proche de la racine. Pour un couple (S, π) où S désigne l'alphabet et π est une loi de probabilité sur S , il existe un unique arbre de Huffman.

Le codage de Huffman permet ainsi de réduire la longueur du message, il s'agit donc d'un algorithme de compression. Il est dit non-destructif car il est complètement réversible. Enfin, on note que la structure d'arbre permet de représenter tous les symboles à la suite sans séparateur. Par exemple, le code '0110110111' donne sans équivoque le message BEEF.

On démontre qu'en analysant les probabilités d'occurrences de séquences de n symboles successifs, on minimise asymptotiquement (quand n tend vers l'infini) le nombre de bits nécessaire en moyenne au codage d'un message, qui devient alors égal à l'entropie du langage à coder. Outre la difficulté de déterminer les probabilités sur des séquences de symboles, on notera également que cette solution n'est pas viable en pratique, puisque qu'il faudrait calculer $|S|^n$ estimations, ce qui devient vite prohibitif (en terme de temps de calcul et de taille de stockage de l'arbre généré), même pour des petits alphabets.

Pour un arbre de Huffman classique, le principe consiste à définir la racine d'un arbre comme la somme des probabilités des éléments feuilles. La racine de l'arbre complet contenant une feuille par symbole, son poids est donc égal à 1. La construction s'opère alors de manière ascendante, en associant deux feuilles de poids faible pour créer un noeud de poids plus fort. On ajoute ainsi tous les symboles de manière incrémentale, jusqu'à obtenir un arbre complet. Les symboles à forte probabilité sont alors placés en dernier, c'est-à-dire proche de la racine. Pour plus détails sur la méthode de construction de l'arbre, nous renvoyons le lecteur à (Lee, 2007).

Pour une analyse lettre par lettre, l'entropie de la langue française vaut $H_F \simeq 3.95$ à comparer avec la profondeur d'un arbre binaire de 26 feuilles : $\log_2(26) = 4.70$. Ceci signifie qu'un codage de Huffman basique peut permettre de compresser un message de 100 bits en 84 bits en moyenne. Lorsque l'on considère les 256 symboles codables avec 1 octet, le gain de la compression d'un texte en langue française est bien plus spectaculaire.

Arbre d'expression arithmétique

Considérons un arbre binaire tel que chaque feuille soit un nombre (ou éventuellement un identificateur de nombre) et chaque noeud qui n'est pas une feuille est un opérateur arithmétique binaire.

On remarque alors immédiatement qu'un tel modèle d'arbre peut être utilisé pour formaliser une expression arithmétique.

Par lecture de l'arbre de bas en haut, on obtient :

$$A = 54 * 10^3 - 74 / ((81 + 11) / (89 + 78) * 28)$$

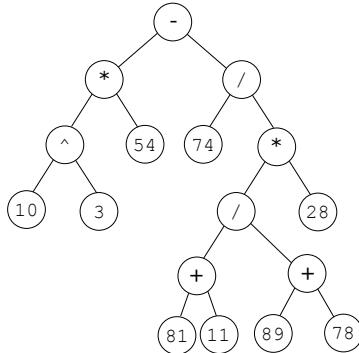


FIGURE 4.23 – Arbre binaire d'expression arithmétique

Notons que le parcours en profondeur infixe de l'arbre nous donne l'expression arithmétique sans le parenthésage :

$$10 \wedge 3 * 54 - 74 / 81 + 11 / 89 + 78 * 28$$

ce qui mathématiquement correspond à une expression mathématique différente.

Le parcours suffixe donne quant à lui :

$$10\ 3\ \wedge\ 54\ *\ 74\ 81\ 11\ +\ 89\ 78\ +\ /28\ *\ /-$$

Cette seconde formulation est appelée Notation en Polonaise Inverse (NPI) en théorie de la compilation. Elle permet d'exprimer une opération arithmétique sans ambiguïté et sans parenthésage.

En utilisant les concepts vus en théorie des arbres, indiquer le nombre maximal d'opérations arithmétiques que l'on peut formuler avec n nombres (on considère que la division par zéro est une opération valide mais retournant le résultat *null*).

Application : dans l'émission télévisé "Des chiffres et des lettres", les candidats doivent retrouver un nombre compris entre 1 et 999 en combinant 6 plaques composés des quatre opérateurs (addition, soustraction, multiplication et division). A l'aide du résultat établis ci-dessus, indiquer le nombre de combinaisons à tester si l'on souhaite faire une recherche exhaustive de l'espace des solutions⁶.

En pratique, l'arbre de recherche des solutions peut être rapidement élagué, en considérant la commutativité de l'addition et de la multiplication et en éliminant les opérations inutiles (multiplications par 1...) ou interdites (division non-entière...).

A titre d'exercice, généraliser le modèle précédent pour prendre en compte les fonctions mathématiques et les tableaux (on se limitera à des tableaux de dimension au plus égale à 2). Indication : utiliser des arbres dont certains nœuds n'ont qu'un seul fils.

6. Plus spécifiquement, la version décisionnelle de ce jeu appartient à la classe des problèmes NP-complets au même titre que le problème du sac à dos dont nous parlerons au chapitre suivant.

Arbre de décision

Pour un ensemble de données \mathcal{X} et un ensemble d'étiquettes \mathcal{Y} , on note $(x, y) \in \mathcal{X} \times \mathcal{Y}$ une donnée d'apprentissage. L'objectif de l'apprentissage statistique est, connaissant un ensemble de données d'entraînement $\mathcal{D} = \{(x_i, y_i) \mid 1 \leq i \leq n\}$, de déterminer l'étiquette y_{n+1} d'une nouvelle donnée x_{n+1} :

$$\hat{y}_{n+1} = \operatorname{argmax}_{y_{n+1} \in \mathcal{Y}} \mathbb{P}(y_{n+1} | x_{n+1}, \mathcal{D})$$

Pour simplifier l'explication, on suppose ici que $\mathcal{X} = \mathbb{R}^2$ et $\mathcal{Y} = \{\text{rouge, vert}\}$.

On dispose d'un ensemble de données d'entraînement, c'est-à-dire d'un semis de points du plan de couleur rouge ou verte. L'objectif est de déterminer, à partir de sa position, la couleur la plus probable d'un nouveau point de couleur inconnue.

Une solution simple et rapide à mettre en oeuvre consiste à déterminer le point coloré le plus proche du nouveau point, de récupérer sa couleur et de l'affecter au nouveau point. Un raffinement possible est obtenu en considérant plusieurs voisins (le nombre k de voisins à considérer est un paramètre de l'algorithme), puis d'associer au nouveau point la couleur majoritaire présente dans ces voisins. Il s'agit de la méthode des k -NN (k -Nearest Neighbors).

Pour obtenir une solution plus robuste, on utilise un arbre de décision. Chaque noeud de l'arbre est une séparation du plan en deux parties. D'un point de vue informatique, chaque noeud de l'arbre doit donc contenir, en plus des pointeurs parent et fils, une direction de coupe (horizontale ou verticale) ainsi qu'une valeur seuil.

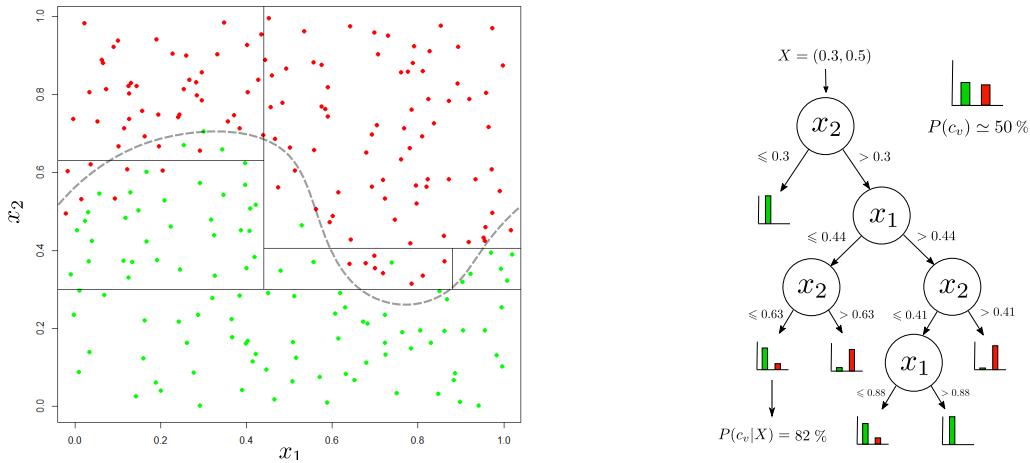


FIGURE 4.24 – Données d'entraînement et arbre de décision (à droite)

Supposons que l'on souhaite attribuer une couleur à un nouveau point : $(0.3, 0.5)$.

La probabilité a priori que la couleur de ce point soit verte sachant les données d'exemple $\mathbb{P}(y = \text{vert})$ est proche de 50% (les proportions des deux classes sont à peu près égales). La

structure d'arbre représentant une partition de l'espace au niveau de ses feuilles, elle nous permet de prendre en compte la position du point dans le processus de décision. Il suffit de suivre le cheminement de l'arbre en répondant aux questions.

Par exemple, la première intersection de l'arbre teste si $x_2 \geq 0.3$, la réponse est positive ($x_2 = 0.5$) et on se déplace donc vers le fils droit du noeud. En suivant le cheminement jusqu'à une racine de l'arbre, l'histogramme des couleurs est beaucoup plus discriminé et on obtient une probabilité a posteriori $\mathbb{P}(y = \text{vert} | x) 82\%$ que le point soit vert. On l'affecte donc à la classe verte. En effet, dans la cellule correspondante (figure de gauche ci-dessous), 82% des points appartiennent à la classe verte.

Nous verrons une structure similaire lorsque nous parlerons de l'indexation spatiale à la fin de ce chapitre.

4.5 Tables de Hachage

4.5.1 Définition

Dans toute cette section, on suppose que les indices des tableaux démarrent à 0, comme c'est le cas dans la plupart des langages informatiques.

Une table de hachage est utile dès lors que l'on souhaite former des couples de type (clé, valeur) et que la suite de clés n'est pas une suite de nombres entiers (ce qui aurait permis un référencement simple dans un tableau, où chaque ligne serait occupée par le couple dont la clé correspond au numéro de la ligne en question).

Ainsi, pour prendre un exemple simpliste, une table de hachage pourrait être requise pour optimiser les temps d'accès aux données d'un annuaire. Chaque nom de personne est une clé, et à chaque clé est associée une donnée, appelée valeur et correspondant ici au numéro de téléphone de la personne stockée dans la clé. Même lorsque les données peuvent être ordonnées (ce qui est le cas avec les chaînes de caractères, mais aussi bien sûr avec les nombres entiers, flottants...) l'accès à la valeur d'une clé particulière nécessite plusieurs itérations dichotomiques (similaire au procédé de recherche de la définition d'un mot dans le dictionnaire). Si les données ne sont pas triées, le temps d'accès à une valeur à partir de sa clé peut-être long (de l'ordre du nombre de données stockées).

Pour pouvoir permettre un accès rapide aux valeurs à partir des clés, une telle table est munie d'une fonction de hachage. Soit Ω l'ensemble des clés possibles (Ω est bien souvent de cardinal infini, c'est en particulier le cas pour des chaînes de caractères). Une fonction de hachage est alors une fonction h , sur Ω , à valeurs entières dans $\{0, 1, \dots, m - 1\}$, où m est la taille du tableau de données à garnir :

$$h : \Omega \rightarrow \{0, 1, \dots, m - 1\}$$

$$k \rightarrow h(k)$$

Remarque : pour un ensemble fini de n clés Ω et un entier m donné, il existe m^n fonctions de hachage respectant la définition ci-dessous. On note \mathcal{H} l'ensemble de ces fonctions. Quand Ω est infini, naturellement \mathcal{H} l'est aussi.

Soit une clé $k \in \Omega$, associée à la valeur v . Alors le couple (k, v) sera rangé à la ligne $h(k)$ de la table de hachage. Pour accéder aux données de la clé k , le mécanisme consiste à calculer dans un premier temps $h(k)$ puis à récupérer v à la ligne correspondante.

Cependant, en pratique, l'ensemble Ω étant de cardinal plus grand que m , la fonction h n'est pas injective et donc, à plusieurs clés k peut correspondre un même élément de $\{0, 1, \dots, m - 1\}$ (et donc une même ligne du tableau de données). On parle de *collision*. Pour contourner ce problème, il est possible d'utiliser des procédés de décalage : plus spécifiquement, lorsque l'on souhaite ajouter un couple (k, v) dans une table dont la ligne $h(k)$ est déjà occupée par une couple (k', v') , il faudra trouver un autre emplacement. On utilise alors un système de sondage⁷ (différentes positions successives sont calculées jusqu'à trouver un emplacement libre dans lequel on insère le couple), la difficulté résidant alors dans le choix d'une fonction de chaînage permettant de conserver une table remplie de manière homogène, ce qui réduit considérablement les temps d'accès. Toutes les tables (quelle que soit la fonction de hachage associée) sont tôt ou tard confrontées aux problèmes de collisions.

En conséquence, la sauvegarde de la clé associée à chaque valeur dans la table est nécessaire. Ainsi, lors de la récupération de la valeur v associée à une clé k , si la ligne $h(k)$ n'est pas occupée par un couple dont la clé est k , cela signifie que l'on est en présence d'une collision. Le mécanisme de sondage est alors mis en oeuvre jusqu'à ce qu'un couple dont la clé est k est trouvé. Bien entendu, les clés des données stockées dans une table de hachage se doivent d'être uniques. Toutes ces opérations sont transparentes pour les programmes utilisant de telles tables. Seulement, les instructions d'accès et de modifications sont exécutées en des temps quasi-indépendants de la taille de la table.

On donne à suivre l'exemple d'une table de hachage destinée à recevoir des couples (clé, valeur) de type (chaîne de caractères, entier). Prenons une table de taille $m = 10$ (extensible) et une fonction de hachage triviale, qui à chaque chaîne de caractères retourne la somme des positions des caractères ASCII utilisés, le tout modulo 10.

Ainsi, par exemple, si les codes ASCII de 't', 'e' et 's' sont respectivement 75, 54 et 74 alors la fonction de hachage appliquée à la clé `t est` renvoie la valeur 8 ($75 + 54 + 74 + 75 = 278$). La clé `t est` (associée par exemple à la valeur 13) sera alors placée en ligne 8 du tableau. L'accès à la valeur de l'objet `t est` est alors direct : il suffit de récupérer la valeur du couple situé à la ligne $h(\text{test}) = 8$.

Supposons à présent que l'on souhaite ajouter le couple (`essai`, 48) mais que la somme des positions ASCII de la chaîne `essai` soit égale à 318. Alors : $h(\text{essai}) = 8$ et on est confronté à un problème de collision. Dans un cas basique de mécanisme de sondage (ici linéaire), on balaye les positions successives de la table jusqu'à trouver un emplacement disponible.

Lors de la récupération de la valeur associée à la clé `essai`, on effectue le calcul $h(\text{essai})$

7. On parle d'adressage ouvert.

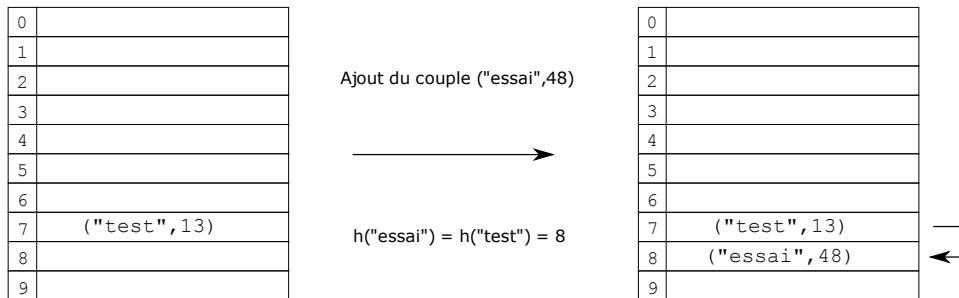


FIGURE 4.25 – Table de hachage pour des couples de type (chaîne de caractères, entier)

qui donne 8. On se place alors en position 8 dans la table. Si la ligne est vide, cela signifie que la clé `essai` n'est pas recensée. Sinon, on récupère la clé du couple présent en ligne 8. Si cette clé est égale à la clé recherchée (i.e. `essai`), alors, la valeur correspondante est retournée. Sinon, c'est que la clé `essai` a été confrontée à un problème de collision lors de son insertion dans la table de hachage. Le mécanisme de sondage permet alors de scanner les positions suivantes jusqu'à rencontrer une case vide (auquel cas la clé n'est pas recensée) ou jusqu'à trouver un couple dont la clé est égale à la clé recherchée. Ici, la fonction de sondage linéaire est une simple incrémentation. La ligne 9 est alors scannée, permettant de trouver la clé recherchée. La valeur 48 est alors retournée. La procédure n'a requis que 2 étapes alors que l'algorithme de parcours de la liste complète a une complexité moyenne de $5 (m/2)$. Avec des données triées et un algorithme de recherche de type dichotomique, la complexité est logarithmique.

Une autre solution pour gérer les collisions consiste à utiliser les listes chaînées. Dans ce cas, lorsque 2 données occupent la même cellule, à la première donnée entrée (chronologiquement) dans la table est affecté un pointeur qui référence la seconde donnée (qui elle est stockée dans un tableau externe). Lorsqu'une troisième données arrive dans cette même cellule, c'est alors la seconde donnée qui se charge de référencer cette dernière, et ainsi de suite... Avec ce système, chaque ligne du tableau est en fait une liste chaînée dont le premier élément correspond au premier couple (clé, valeur) inséré dans cette ligne du tableau. En plus de faciliter la réorganisation de la table (grâce aux propriétés des listes chaînées, étudiées en début de chapitre), cette solution offre l'avantage d'une structure dynamiquement extensible, sans avoir à changer la fonction de hachage. La fonction de sondage devient également inutile. En revanche, il est nécessaire d'allouer un second tableau pour sauvegarder les éléments des listes chaînées. D'un point de vue théorique, la gestion des collisions par liste chaînée permet également un calcul simple de la complexité en temps de l'accès aux données.

A titre d'exemple, on donne ci-dessous l'exemple d'une table de hachage référençant les codes INSEE des régions de France. La fonction de hachage choisie est le nombre de lettres de la clé modulo 10. Les collisions sont gérées par des listes chaînées.

On remarque dans un premier temps que la table n'est pas complète. En effet, il faut en moyenne $n \ln n$ insertions pour compléter une table de taille n , il s'agit de la solution au fameux problème du collectionneur. Si un philatéliste reçoit en moyenne une lettre par jour, combien de temps lui faudra-t-il pour compléter sa collection (on suppose qu'il existe $n = 2000$ timbres différents en circulation et que chaque lettre reçue est affranchie par un

0	[Auvergne-Rhône-Alpes, 84]
1	
2	
3	[Bourgogne-Franche-Comté, 27][Île-de-France, 11]
4	[Corse, 94]
5	[Hauts-de-France, 32]
6	[Pays de la Loire, 52][Provence-Alpes-Côte d'Azur, 93]
7	
8	[Bretagne, 53][Nouvelle-Aquitaine, 75]
9	[Centre-Val de Loire, 24][Grand Est, 44][Normandie, 28][Occitanie, 76]

FIGURE 4.26 – Table de hachage pour des couples de type (région, numéro INSEE)

spécimen tiré aléatoirement et uniformément parmi les timbres en circulation). La réponse est donnée par : $2000 \times \ln 2000 \simeq 42$ ans. Ici, on voit qu'il faudrait 23 régions en moyenne pour remplir notre table.

Supposons que l'on souhaite récupérer le code INSEE de l'Ile-de-France :

$$h(\text{Ile-de-France}) = 13 \bmod 10 = 3$$

On récupère alors la liste chaînée dont le premier élément est stockée à l'indice 3 de la table de hachage. On compare sa valeur (Bourgogne-Franche-Comté) à la valeur recherchée. Il n'y pas correspondance, on récupère alors le successeur dans la liste chaînée, jusqu'à trouver un couple dont la clé est Ile-de-France. Il ne reste alors plus qu'à retourner la valeur du couple (11).

Si π est une distribution de probabilité sur les clés, alors, la taille des listes chaînées est proportionnelle à l'image de π par la fonction de hachage. Dans le meilleur des cas, chaque ligne de la table contient un unique élément et l'accès se fait en un temps constant $\Theta(1)$. Dans le pire des cas, une seule ligne est occupée, et contient une liste chaînée de taille n . La complexité dans le pire des cas est alors en $\Theta(n)$.

Etudions la complexité en temps de l'accès au code INSEE des régions de France. En notant l_i la longueur de la liste chaînée débutant en ligne i de la table de hachage, on a :

$$C(n) = \sum_{i=1}^m \left(\frac{l_i + 1}{2} \right) \left(\frac{l_i}{n} \right) = \frac{1}{2n} \sum_{i=1}^m l_i^2 + \frac{l_i}{2}$$

Sur l'exemple des régions, on obtient :

$$C = \frac{1}{26} (1 + 2^2 + 1 + 1 + 2^2 + 2^2 + 4^2) + 0.5 \approx 1.69$$

Pour un nombre n fixé de données, on souhaite minimiser $\sum_i l_i^2$ sous la contrainte $\sum_i l_i = n$. Ce problème se résout trivialement avec la méthode des multiplicateurs de Lagrange et revient à annuler la dérivée de :

$$f(l_1, l_2, \dots, l_m, \lambda) = \sum_{i=1}^m l_i^2 + \lambda \left(\sum_{i=1}^m l_i - n \right)$$

On obtient la solution : $l_i = \frac{\lambda}{2}$ indépendante de i et donc similaire pour tous les l_i . La fonction f étant convexe, cet extremum global est un minimum, et on a montré que le temps d'accès à une donnée est minimal lorsque toutes les listes chaînées ont la même longueur.

Au travers de cet exemple simple, on comprend bien que la fonction de hachage doit être choisie de sorte à ce que les longueurs l_i soient les plus uniformes possibles, c'est-à-dire que l'image qu'elle renvoie de l'ensemble des clés doit être la plus uniforme possible dans l'ensemble des indices de la table.

4.5.2 Implémentation

Trois points principaux sont à prendre en compte lors de l'implantation d'un mécanisme de table de hachage :

- Le choix d'une fonction de hachage efficace, permettant une répartition homogène des entrées et limitant donc le nombre de collisions.
- Le choix d'une fonction de sondage efficace (fonctions polynomiales etc.) pour la stratégie d'adressage ouvert, ou bien un mécanisme de liste chaînée.
- Une adaptation à la croissance des données entrées dans la table (algorithmes de réhomogénéisation de la table, incrémentation de l'espace disponible en fonction du facteur de charge, c'est-à-dire du rapport maximal autorisé entre le nombre n de données entrées et la taille m de la table).

L'utilisation en pratique d'une tables de hachage nécessite quatre fonctions principales :

- **CONTAINS_KEY(k)** : retourne vrai si la clé a été recensée dans la table.
- **GET(k)** : retourne la valeur associée à la clé k .
- **PUT(k, v)** : insertion du couple (k, v) dans la table.
- **REMOVE(k)** : suppression du couple de clé k dans la table.

Exercice 4.9. Ecrire l'ensemble du code algorithmique permettant de créer une table de hachage à partir d'un tableau à deux colonnes. On supposera que les données en entrée sont des couples de type (chaînes de caractères, entier). La taille N de la table (que l'on supposera statique) est un paramètre à définir lors de sa construction.

On veillera à implémenter les quatre fonctions décrites ci-dessus, et on n'oubliera pas de gérer le cas des collisions (à l'aide d'une fonction de sondage linéaire).

4.6 Indexation spatiale

L'indexation spatiale est une extension du système d'indexation présenté dans la partie sur les tables des hachage, pour le cas de données spatialisées. Les Systèmes d'Information Géographique (SIG) ainsi que les Systèmes de Gestion de Bases de Données Géographiques (SGBD géographique) font un usage intensif de ces index pour optimiser leurs calculs.

4.6.1 Exemple d'introduction

En guise d'introduction, considérons un semis de n points du plan. Chaque point est exprimé par ses coordonnées $(x, y) \in \mathbb{R}^2$. On introduit alors un nouveau point $\mathbf{p} = (x_p, y_p)$ et on souhaite trouver le plus proche voisin de \mathbf{p} parmi les n points déjà présents.

La procédure classique consiste à parcourir les n points et pour chacun d'entre eux, calculer la distance qui le sépare du point \mathbf{p} . Durant les itérations de la boucle, on conserve la valeur de la distance minimale ainsi que l'indice du point pour lequel cette distance est atteinte. La complexité moyenne du programme exprimée en nombre de calculs de distance, vaut :

$$C_{moy}(n) = n$$

Admettons à présent que l'on souhaite effectuer cette opération pour un grand nombre de points. Par exemple, pour chaque point du semis, on souhaite savoir lequel des autres points est son plus proche voisin. Le nombre d'évaluations de distances est cette fois égal à $n(n - 1)/2$ et la complexité est quadratique. Est-il possible de faire mieux ?

La réponse est oui, si l'on utilise pas exemple un système d'indexation spatiale.

Le système le plus simple que l'on puisse imaginer est une grille de pas h . Chaque point est alors affecté à la cellule dans lequel il se trouve. Cette opération peut se faire en $\Theta(n)$.

Pour calculer le plus proche voisin d'un point, il suffit de calculer les distances séparant ce point de tous les points présents dans la cellule, ainsi que dans les 8 cellules voisines. Si aucun point n'est présent dans ces cellules, on agrandit la zone de recherche en calculant les distances avec les points compris dans les 16 cellules voisines, et ainsi de suite jusqu'à ce qu'on trouve le plus proche voisin.

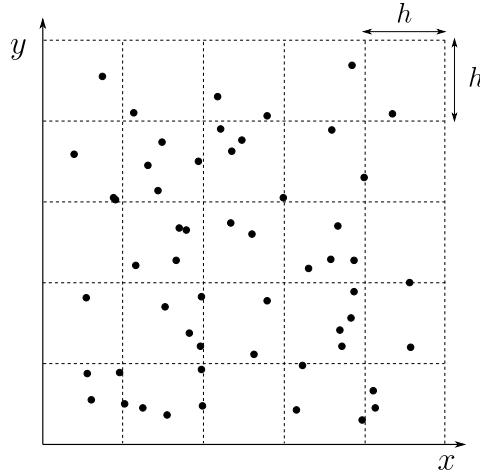


FIGURE 4.27 – Grille d’indexation spatiale sur un semis de points régulier

La complexité de la recherche est toujours linéaire, mais on peut cette fois espérer que le facteur de proportionnalité est plus faible (fonction de la valeur du pas de la grille).

En particulier, pour une distribution de point uniforme de densité λ points par cellule (avec λ suffisamment grand pour supposer que dans la très grande majorité des cas, il y a au moins un point par cellule), le temps de recherche du plus proche voisin vaut : $C(n) = 9\lambda h^2 = 9n/N$ (avec N le nombre de cellules) soit une constante de proportionnalité $9/N$ (généralement bien inférieur à 1).

Par exemple, en supposant que le nombre de points soit suffisamment grand pour pouvoir segmenter le champ d’étude avec un maillage de 50 cellules par 50 cellules, tout en assurant d’avoir au minimum un point par cellule, le temps de recherche des plus proches voisins est divisé par $2500/9 \approx 278$.

En pratique, on choisit h de sorte à minimiser les chances d’obtenir des cellules vides tout en garantissant d’avoir un minimum de points dans chaque cellule. Il s’agit donc d’un compromis : si h est trop grand, la complexité en nombre d’évaluations de distances est élevée. A l’inverse, quand h est trop petit, la grille est trop détaillée, et l’algorithme passe beaucoup de temps à changer de cellules pour élargir sa zone de recherche.

Essayons de quantifier cette observation. Admettons que le ratio entre le calcul d’une distance géométrique et le coût du chargement d’une cellule soit égal à une constante $\mu \in \mathbb{R}^+$. En notant n_c le nombre de cellules à charger pour trouver le plus proche voisin et n_g le nombre de distances à évaluer, le coût de la recherche s’écrit : $C(r) = n_c(r) + \mu n_g(r)$, où $r \in \mathbb{R}^+$ est la distance au plus proche voisin. Plus précisément, pour une distance donnée r de plus proche voisin, n_c est proportionnel à l’aire du cercle de rayon r :

$$n_c(r) = a\pi r^2$$

où a est l’aire d’une cellule du maillage. Fixons (arbitrairement) $a = 1$ et nous considérons que notre paramètre flottant est la densité de points dans le maillage (ce qui revient au

même que de supposer connue cette densité et de faire varier la taille des cellules de la grille). On obtient alors l'équation : $n_c(r) = \pi r^2$.

Le nombre de distances géométriques à calculer est lui fonction du nombre de cellules en bordure de la zone de recherche, i.e. du nombre de cellules à une distance grossièrement équivalente à celle du plus proche voisin (à l'échelle de la taille d'une cellule). Ce nombre est donc le produit de la densité de points par cellule λ multiplié par le périmètre de la zone de recherche : $n_g(r) = 2\pi\lambda r$, d'où l'expression du coût de calcul :

$$C(r) = \pi r^2 + 2\pi\lambda\mu r$$

Etant donné que r est une variable aléatoire, la complexité moyenne s'obtient en moyennant l'expression ci-dessus pour toutes les configurations possibles de tirages de points. Nous devons donc calculer la loi de la variable aléatoire r .

Le tirage de points étant par hypothèse uniforme, le nombre de points dans une zone d'aire α suit un processus de Poisson d'intensité λ :

$$P_\lambda(k; \alpha) = \frac{e^{-\lambda\alpha}}{k!} (\lambda\alpha)^k$$

La fonction de répartition de la distance au plus proche voisin R correspond à la probabilité de ne trouver aucun point dans un cercle de rayon r et s'exprime alors en fonction de $P_\lambda(0; \pi r^2)$. S'ensuit immédiatement par dérivation de F la densité de probabilité $p(r)$.

$$F(r) = \mathbb{P}(R \leq r) = 1 - P_\lambda(0; \pi r^2) = 1 - e^{-\lambda\pi r^2}$$

$$p(r) = \frac{dF(r)}{dr} = 2\lambda\pi r e^{-\lambda\pi r^2}$$

On reconnaît une distribution de Rayleigh de paramètre $\sigma = 1/\sqrt{2\pi\lambda}$.

Si X est une variable aléatoire échantillonnée suivant une loi de Rayleigh : $X \sim \mathcal{R}(1/\sqrt{2\pi\lambda})$, le coût moyen de la recherche s'exprime (par linéarité de l'espérance) :

$$\mathbb{E}[C(r)] = \pi\mathbb{E}[X^2] + 2\pi\lambda\mu\mathbb{E}[X]$$

On peut facilement montrer que l'espérance $\mathbb{E}[X]$ vaut $1/(2\sqrt{\lambda})$. D'autre part, étant donné qu'une loi de Rayleigh $\mathcal{R}(1)$ est équivalente à la loi de la quantité $\sqrt{Y^2 + Z^2}$ où Y et Z sont deux variables aléatoires indépendantes distribuées suivant une loi normale standard, il en résulte immédiatement que, à une facteur σ près, X^2 suit une loi du χ^2 à 2 degrés de liberté

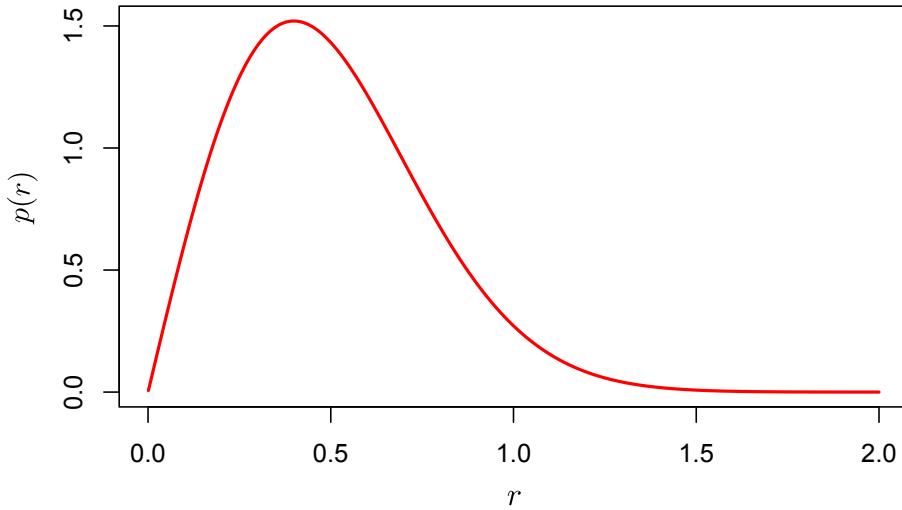


FIGURE 4.28 – Densité de probabilité de la distance au ppv pour $\lambda = 1.0$.

et donc : $\mathbb{E}[X^2] = \sqrt{2\pi}/\lambda$. On peut alors réécrire le coût moyen en fonction de la densité λ :

$$C_{moy}(\lambda) = \frac{\sqrt{2\pi}}{\lambda} + \mu\pi\sqrt{\lambda}$$

En simplifiant l'expression, et en calculant sa dérivée par rapport à λ , on obtient l'expression de la densité optimale de points par cellule :

$$\lambda^* = \frac{2}{(\mu^2\pi)^{1/3}} \approx 1.36\mu^{-2/3}$$

On vérifie bien que cette densité optimale tend vers 0 quand $\mu \rightarrow +\infty$, c'est-à-dire quand le coût de chargement d'une cellule est négligeable. En effet, dans ce cas, on peut se permettre d'avoir une grille de résolution très fine, de sorte à limiter le nombre de distances à calculer. A l'inverse, quand le coût μ est nulle (*i.e.* quand seuls les chargements de cellules comptent dans le coût d'exécution), on obtient bien le cas dégénéré où λ est infini, c'est-à-dire une seule cellule recouvrant toute l'emprise de la zone (ce cas limite n'a aucun intérêt dans le contexte de l'indexation spatiale).

En réinjectant l'expression de λ^* dans la complexité, on a :

$$C_{moy}(\lambda^*) = \frac{\sqrt{2\pi}}{2}(\mu^2\pi)^{1/3} + \sqrt{2}\mu^{2/3}\pi^{5/6} = \frac{3}{\sqrt{2}}\mu^{2/3}\pi^{5/6}$$

Le paramètre μ étant constant, nous avons montré de manière intéressante que, moyennant la connaissance de la densité des points à indexer, il est possible de construire un index

spatial de sorte à ce que la complexité moyenne de la recherche d'un plus proche voisin soit constante (donc indépendante du nombre n de points dans la base de données).

$$C_{moy}(n) = \Theta(1)$$

4.6.2 Quadtree

Le modèle simple introduit dans la section précédente n'est plus optimal dès lors que la distribution de points est fortement inhomogène. En effet, dans ce cas, il est difficile de choisir un pas de grille satisfaisant.

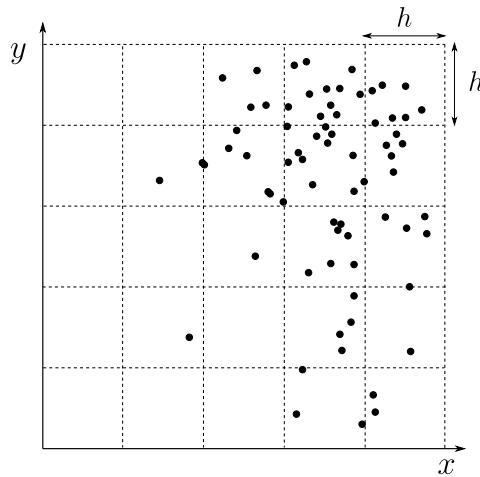


FIGURE 4.29 – Grille d'indexation spatiale sur un semis de points irrégulier

Dans cette configuration, une solution consiste à utiliser un *quadtree*.

Le principe consiste à définir une grille de niveau de résolution variable en fonction des zones. Plus la densité de points est élevée dans une zone donnée, plus on affine la résolution de la grille.

Notons que comme pour le cas des arbres de décision, la structure construite correspond à un arbre, mais cette fois-ci chaque noeud a quatre fils. On ordonne les noeuds fils de gauche à droite dans l'arbre pour un parcours dans le sens inverse des aiguilles d'une montre en partant de la cellule inférieure gauche dans l'espace des données. Ainsi par exemple, le troisième noeud en partant de la gauche, correspond à la subdivision cellulaire supérieure droite de la cellule définie par le noeud parent.

Etant donné un nouveau point, il est alors possible en suivant le cheminement de l'arbre, de trouver à quelle feuille de l'arbre appartient ce point.

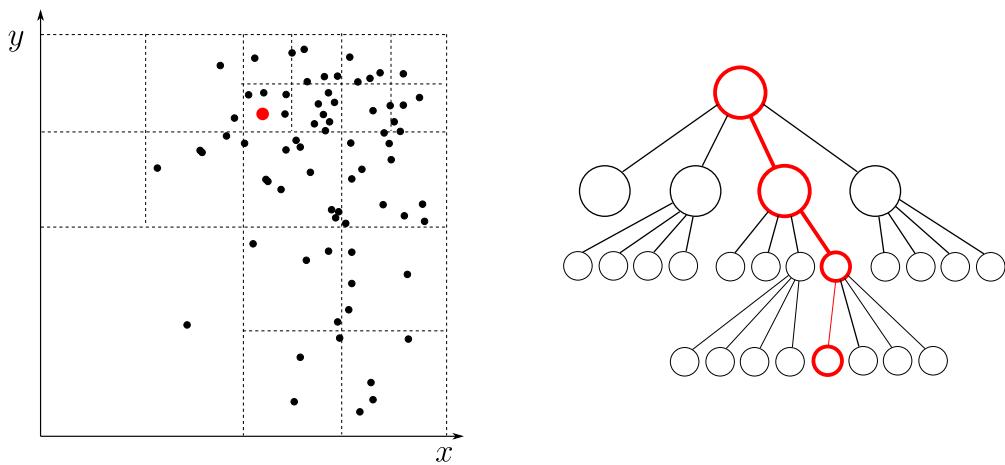


FIGURE 4.30 – Indexation spatiale par quadtree et arbre associé

La profondeur de l’arbre est un paramètre de l’algorithme que l’on peut spécifier via un nombre maximal de points par cellule. On donne par exemple ci-dessous un exemple de quadtree plus profond avec la contrainte que chaque feuille contienne au plus 3 points.

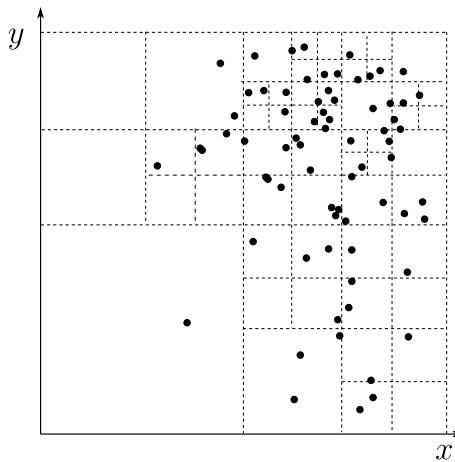


FIGURE 4.31 – Indexation spatiale par quadtree

Remarque : dans le cas d’un semis de point régulier, on retrouve une indexation qui s’approche de l’index en grille définie au début de cette section.

Remarque 2 : il existe une version tridimensionnelle de cet index, appelé *octree* et découpant chaque cube de domaine en 8 sous-domaines également cubiques.

Remarque 3 : la construction d’un quadtree se prête très bien à une approche récursive.

4.6.3 *Kd-Tree*

Il est possible de faire encore mieux que le quadtree pour des données spatialement très hétérogènes. Le *Kd-Tree* consiste, tout comme l’arbre de décision, à *couper* le semis de points à l’aide d’hyperplans séparateurs. Dans le cas de données en 2 dimensions, les sé-

parateurs sont des droites. Pour chaque coupe du domaine (qui correspond à un noeud non terminal de l'arbre), on définit une direction de coupe (horizontale ou verticale) ainsi qu'une valeur de coupe.

Le nom de cet index vient du fait qu'il est aisément généralisable en dimensions supérieurs.

Notons que l'arbre est en général plus léger (à efficacité égale) mais plus complexe à gérer, puisqu'il faut enregistrer dans chaque noeud, en plus des pointeurs vers les fils et le parent, les paramètres de la coupe (direction et valeur).

Tout comme pour le quadtree, la profondeur de l'arbre est définie par un paramètre de l'algorithme spécifiant le nombre maximal de points tolérés par cellule.

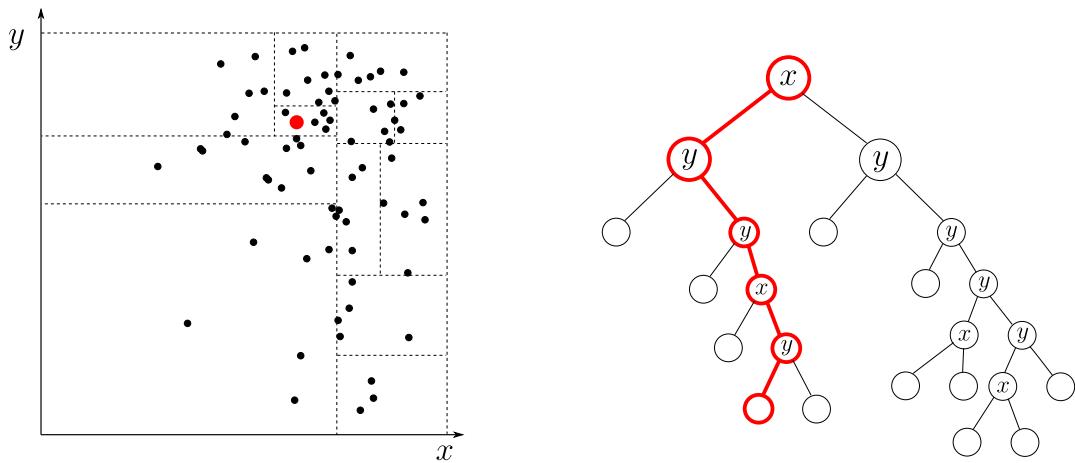


FIGURE 4.32 – Indexation spatiale par *Kd-tree* et arbre associé

Un algorithme simple de construction, consiste pour chaque découpage à choisir une direction de coupe puis à calculer la médiane des coordonnées dans cette direction (c'est-à-dire par exemple la médiane des coordonnées x_i si la coupe est définie horizontalement). La coupe est alors effectuée au niveau de la valeur de cette médiane, de sorte à obtenir un arbre équilibré. Pour choisir la direction de coupe, une solution simple et robuste préconise d'alterner suivant les directions possibles. Ainsi, en dimension 2, on alternera une fois sur deux des coupes dans les directions horizontale et verticale. En dimension 3, une fois sur 3, etc...

Remarque : dans le cas d'un semis de point régulier, on retrouve une indexation qui s'approche de l'index en grille et du quadtree.

Ceci conclut notre tour d'horizon des mécanismes d'indexation spatiale. Notons qu'il existe des algorithmes plus sophistiqués pour gérer les données non-ponctuelles (routes, bâtiments, parcelles...), où il est cette fois nécessaire de prendre en compte l'extension spatiale des objets. Citons à titre d'exemple le *R-tree* ou encore *l'Arbre de Hilbert*.

4.7 Exercices

Exercice 4.10. *Tri stupide* *

On appelle tri stupide, le tri qui consiste à permuter aléatoirement les éléments du tableau jusqu'à ce qu'ils soient triés. Ecrire cet algorithme de tri et analyser sa complexité dans le pire des cas, dans le cas moyen et dans le meilleur des cas. On prendra comme unité de complexité le nombre de comparaisons effectuées.

Cet algorithme de tri est-il en place ? Est-il stable ?

Exercice 4.11. *Fonctions de base sur les listes chaînées* *

Ecrire les algorithmes permettant de :

- tester si un élément appartient à une liste chaînée
- remplacer un élément dans la liste chaînée
- compter le nombre d'occurrences d'un élément dans la liste chaînée
- inverser deux éléments dans une liste chaînée
- faire l'union de deux listes chaînées
- faire l'intersection de deux listes chaînées
- insérer un élément avant un élément de valeur donnée
- insérer un élément après un élément de valeur donnée

Exercice 4.12. *Liste doublement chaînée circulaire* *

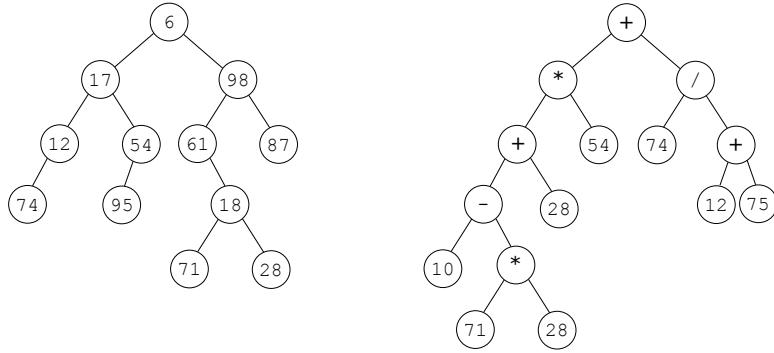
Ecrire le modèle permettant de gérer une liste doublement chaînée circulaire

Exercice 4.13. *Ordre de parcours dans un arbre* *

Pour les deux arbres suivants, lister les éléments suivant un parcours en profondeur (on donnera les parcours préfixe, suffixe et infixé) et en largeur. Quels sens peut-on donner aux parcours en profondeur infixé et suffixe de l'arbre de droite ?

Construire l'arbre associé à l'expression arithmétique suivante et transformer l'écriture en notation polonaise inverse :

$$3 * (5 + 2 * (x - 4 + 3) / (5 + y) - 4) * 2 + (1 - a) * 7 + 8$$

**Exercice 4.14.** *Tri d'une liste chaînée* ******

Ecrire un algorithme de tri par sélection prenant en entrée une liste simplement chaînée. Quelle est la complexité de cet algorithme ? On prendra comme unité de coût le nombre de référencement de pointeurs, c'est-à-dire le nombre d'appels à l'instruction \rightarrow .

Exercice 4.15. *Tours de Hanoï* ******

Réécrire l'algorithme de résolution du problème des tours de Hanoï, en considérant cette fois les tiges comme des piles de données.

Exercice 4.16. *Nombres de feuilles dans un arbre* ******

Ecrire un algorithme retournant le nombre de feuilles dans un arbre binaire.

Exercice 4.17. *Nombres de feuilles dans un arbre* ******

Ecrire un algorithme retournant le nombre de noeuds dans un arbre binaire.

Exercice 4.18. *Arbre peigne droit* ******

On appelle *arbre peigne droit* un arbre binaire tel que le nombre de feuilles f et la hauteur h vérifient la relation : $h + 1 = f$. Ecrire la fonction permettant, à partir d'un tableau de valeurs T , de créer un arbre peigne droit.

Exercice 4.19. *Vérification d'ABR* ******

Ecrire une fonction permettant de vérifier qu'un arbre binaire est un arbre binaire de recherche et calculer sa complexité algorithmique dans le pire des cas.

Exercice 4.20. *Maximum d'un ABR* ******

Ecrire une fonction permettant de trouver l'élément maximal d'un arbre binaire de recherche.

En déduire la complexité asymptotique dans le pire des cas d'un tri de liste utilisant une structure d'ABR.

Exercice 4.21. Table de hachage ★★

Dans cet exercice, on souhaite gérer des données de type (chaîne de caractères, entier) où le premier élément désigne le nom d'un livre et le second son prix en librairie.

Titre	prix
Le Tour du Monde en 80 jours	10.50
Les 10 Petits Nègres	15.00
Les Aventures de Sherlock Holmes	13.50
L'Île au Trésor	12.50
Les Misérables	16.00
Le Petit Prince	10.30
Guerre et Paix	25.00
Germinal	18.00
Le Rouge et le Noir	20.50

Q1. On range ces livres dans une table, dans l'ordre donné ci-dessus. Calculer la complexité moyenne de l'accès au prix d'un livre donné (on prendra comme unité de coût la lecture d'une cellule du tableau).

Q2. On range ces livres par ordre alphanumérique dans une table. Quelle procédure peut-on utiliser pour optimiser le temps d'accès au prix d'un ouvrage ? Indiquer la complexité moyenne associée.

On considère à présent la fonction de hachage h qui à une chaîne de caractères associe la longueur de cette chaîne modulo 10 et on utilise un sondage linéaire pour gérer les problèmes de collision. On compte les espaces dans la longueur de chaîne.

Q3. Insérer les livres dans la table de hachage ci-dessous.

Q4. On souhaite récupérer l'ouvrage intitulé "Guerre et Paix". Indiquer la séquence des étapes effectuées par l'algorithme de gestion de la table de hachage.

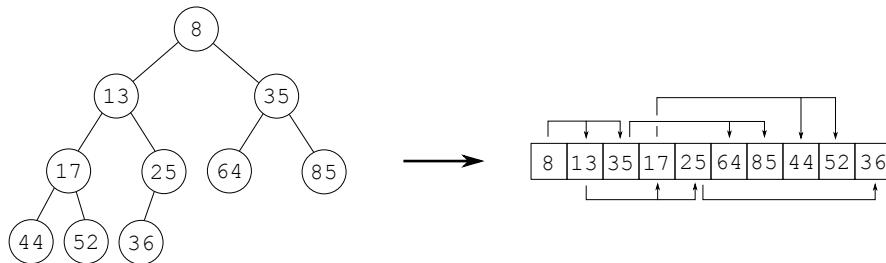
Q5. Calculer la complexité moyenne d'accès à un élément. Conclusion ?

Q6. Calculer le facteur de charge de la table.

Exercice 4.22. File de priorité dynamique ★★★

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Dans cet exercice, on s'intéresse à une file de priorité, gérée par un tas binaire min. On représente l'arbre sous forme d'un tableau :



Les éléments du tableau sont donnés dans l'ordre de parcours en largeur du tableau.

Q1. Donner l'expression analytique permettant à partir d'un noeud situé en position i dans le tableau, d'obtenir les indices de son fils gauche et de son fils droit.

Q2. A partir du résultat de la question précédente, écrire deux modules LEFT et RIGHT, prenant en entrée un indice de position i et retournant respectivement les indices des fils gauche et droit du noeud situé en position i .

Q3. Ecrire un module PARENT, prenant en entrée un indice de position i et retournant l'indice du noeud parent du noeud situé en position i .

Q4. Ecrire une fonction PERCOLATE_UP permettant d'ajouter un élément à un tas binaire min.

Q5. En déduire une fonction permettant de créer un tas binaire à partir d'un tableau non-trié. Indiquer la complexité dans le pire de cas de l'algorithme.

Q6. Remarquons à présent que le tableau non-trié correspond à un tas ne respectant pas la contrainte de tas binaire. Ecrire une fonction PERCOLATE_DOWN, permettant d'effectuer une percolation vers le bas d'un élément dont on donne la position i dans le tableau.

Q7. En utilisant la procédure PERCOLATE_DOWN écrire une fonction BUILD permettant d'obtenir un tas binaire à partir du tableau non-trié.

Q8. Ecrire une fonction REMOVE permettant de récupérer la racine du tas (tout en conservant la propriété de tas binaire)

Q9. Combiner les procédures BUILD et REMOVE pour effectuer un tri par tas d'une liste de nombres. Indiquer la complexité du tri dans le pire des cas. Ce tri est-il en place ? Stable ?

Chapitre 5

La programmation dynamique

Sommaire

5.1	Mémoïsation	204
5.2	Théorie des Equations de Bellman	206
5.2.1	Formulation du problème	206
5.3	Principe général	210
5.3.1	Recherche exhaustive	210
5.3.2	Algorithme glouton	210
5.3.3	Programmation dynamique	211
5.3.4	Extension à la recherche du plus court chemin	213
5.4	Exemple d'application : produit matriciel	214
5.4.1	Recherche exhaustive	215
5.4.2	Résolution par programmation dynamique	215
5.4.3	Etude de la complexité	218
5.5	Exemple d'application 2 : calcul d'itinéraires	219
5.5.1	Recherche de toutes les longueurs optimales	219
5.5.2	Recherche d'un chemin le plus court	220
5.6	Exemple d'application 3 : estimation de temps de trajet	226
5.6.1	Formulation du problème	226
5.6.2	Complexité de l'approche naïve	231
5.6.3	Résolution par la programmation dynamique	232
5.6.4	Complexité de la solution	234
5.7	Exercices	235

Introduction

Dans ce chapitre, nous étudions une méthode générique permettant de réduire la complexité d'un algorithme, lorsque le problème à traiter est constitué de sous-problèmes plus simples à résoudre. Contrairement à l'approche *diviser pour régner*, qui effectue une résolution du haut vers le bas, la programmation dynamique part de la résolution des problèmes élémentaires, stocke leur solution dans une structure adéquate (bien souvent un tableau de dimension 1 ou 2) puis résout un problème de difficulté supérieure en s'appuyant sur les solutions précédemment calculées et ainsi de suite jusqu'à arriver à la résolution du problème demandé. Cette approche ascendante organisée permet d'éviter les chevauchements entre les différents sous-problèmes, comme c'est le cas avec l'approche récursive descendante *diviser pour régner*.

5.1 Mémoïsation

Reprenons l'exemple d'introduction du chapitre 3, consistant à calculer les termes de la suite de Fibonacci $(f_n)_{n \in \mathbb{N}}$ par une fonction récurrente d'ordre 2 :

$$f_0 = f_1 = 1 \quad \forall n \in \mathbb{N} \quad f_{n+2} = f_{n+1} + f_n$$

On représente ci-dessous l'arbre des appels récursifs à la fonction pour calculer f_6 :

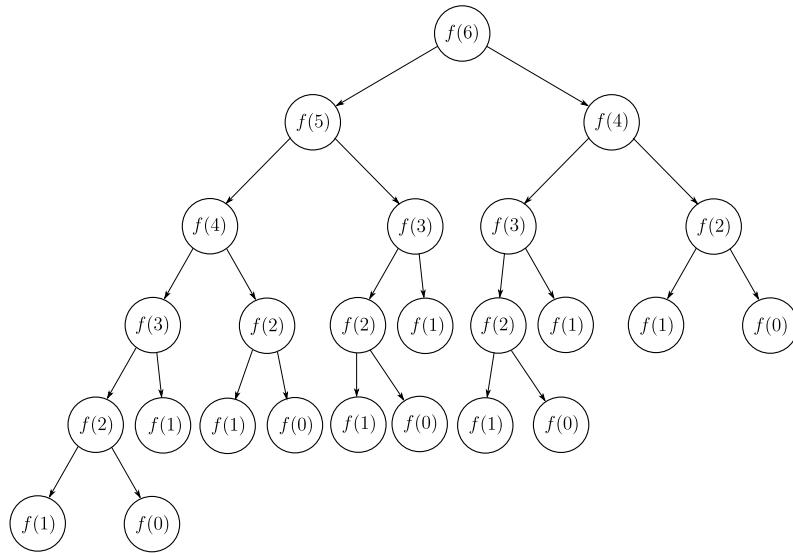


FIGURE 5.1 – Arbre des appels récursifs de la fonction de calcul de la suite de Fibonacci

Il est alors possible de remarquer que si le terme f_5 n'est évalué qu'une seule fois, ce n'est pas le cas de f_4 (qui est évalué 2 fois), ni de f_3 (3 fois) ou encore de f_2 (5 fois)... En toute logique, le nombre de ces appels est une suite de Fibonacci inversée (le terme f_{n-i} étant évalué f_i fois) si bien que la complexité de l'algorithme récursif est du même ordre que la valeur numérique de la suite elle-même.

En introduction du chapitre sur la complexité, pour éliminer cette redondance dans les appels récursifs, nous nous sommes plongés dans un espace vectoriel de dimension 2, en cherchant directement une récurrence sur le terme $F_n = [f_n, f_{n-1}]^T$, ce qui résultait en une suite géométrique dont nous pouvions trouver une expression explicite, évaluable en un nombre linéaire de multiplications¹ Notons que cette solution est spécifique au problème de Fibonacci et qu'il ne sera pas toujours possible de supprimer la redondance en se plongeant dans un espace de dimension supérieure.

D'autre part, l'expression explicite de f_n fait intervenir le nombre d'or Φ , qui est irrationnel et dont la représentation en machine est donc nécessairement incomplète. Pour obtenir la valeur (entière) exacte de f_n il est nécessaire de calculer $\Phi^n / \sqrt{5}$ à une précision meilleure que 0.5. La loi de propagation des incertitudes nous montre en particulier que le calcul du

1. Avec l'algorithme d'exponentiation rapide nous avons même atteint un temps logarithmique.

100^e terme de la suite nécessite une vingtaine de décimales, ce qui n'est pas réalisable avec le type natif de flottants en double précision ne contenant que de l'ordre de 16 décimales.

Une solution plus générique et plus intuitive consiste à mémoriser (on parle de *mémoïsation*) les calculs intermédiaires de sorte à n'effectuer chaque appel de niveau inférieur qu'une et une seule fois. Dans le cas de la suite de Fibonacci, cette mémorisation peut se faire à l'aide d'une liste de valeurs, ou bien seulement à l'aide de 2 valeurs (seules les 2 valeurs de niveau précédent sont nécessaires au calcul du niveau courant). Comme nous le verrons plus loin, dans de nombreux problèmes, la sauvegarde de l'ensemble des valeurs précédemment calculées est nécessaire aux évaluations suivantes. Pour rester le plus général possible, nous explicitons ci-dessous l'algorithme de calcul des termes de la suite de Fibonacci avec un tableau :

$$\text{FIBO}(N, \downarrow) : \left[\left[F_0 \leftarrow 1; F_1 \leftarrow 1; \left\{ \begin{array}{l} F_i \leftarrow F_{i-1} + F_{i-2} \\ i \end{array} \right. \right\}_i^N \text{FIBO} \leftarrow F_n \right] \right]$$

Remarquons que la boucle teste également la condition $N \leq 1$. En effet, pour les cas de base 0 et 1, le programme n'entre pas dans la boucle (indice i supérieur à la borne finale) et le calcul sur les termes suivants n'est pas effectué. En supposant que la valeur N passée en entrée soit positive, le programme retournera le bon résultat, y compris dans les cas de base.

Il s'agit d'un algorithme itératif avec une boucle sans débranchement ni modification du pas ou de la borne finale. La complexité (minimale, moyenne et maximale) de l'algorithme, mesurée en nombre d'additions, est donc égale au nombre de passages dans la boucle, soit une complexité linéaire.

Au travers de cet exemple simple et intuitif, nous comprenons comment réduire la complexité d'un algorithme en procédant par mémoïsation.

Définition : *mémoïsation*

On appelle mémoïsation le procédé qui consiste à enregistrer dans une table les valeurs intermédiaires calculées, de sorte à pouvoir les référencer (éventuellement plusieurs fois) au cours de l'exécution de l'algorithme.

Remarque 1 : le procédé de mémoïsation n'implique pas nécessairement une approche du bas vers le haut (*bottom-up*) dans la résolution du problème, et n'interdit pas non-plus l'utilisation de la récursivité.

Exercice 5.1. Réécrire la fonction de Fibonacci présentée ci-dessus, sous forme récursive et avec mémoïsation. Indication : cela nécessite de passer la table des valeurs sauvegardées en argument de la fonction récursive.

5.2 Théorie des Equations de Bellman

Cette partie, essentiellement théorique, a vocation à présenter la programmation dynamique dans un cadre le plus général possible, à travers le principe d'optimalité de Bellman. Le lecteur plus réceptif aux cas pratiques pourra passer directement à l'exemple applicatif proposé au 5.3, avant de revenir sur cette section, sensiblement plus délicate, mais qu'il nous a néanmoins parut intéressant d'intégrer dans le cours.

5.2.1 Formulation du problème

Soit \mathcal{X} , un ensemble discret d'états (la théorie se généralise aisément au cas des ensembles continus), ainsi qu'un pas de temps $t \in \mathcal{T} = \{1, 2, \dots, T\}$. La variable $x_t \in \mathcal{X}$ désigne alors l'état du système au temps t . On considère une fonction V :

$$\begin{aligned} V : \quad \mathcal{X} \times \mathcal{X} \times \mathcal{T} &\rightarrow \quad \mathbb{R}^+ \\ x, y, t &\rightarrow \quad V(x, y, t) \end{aligned}$$

En général, la fonction V est appelée *récompense*, mais nous nous placerons ici dans le cadre général des problèmes de minimisation, et nous parlerons donc de fonction de *coût*. Pour un couple d'états $(x, y) \in \mathcal{X}^2$, le réel positif $V(x, y, t)$ désigne le coût à payer pour passer de l'état x à l'état y à partir du pas de temps t . Notons que rien n'interdit en pratique d'étendre le modèle pour gérer les fonctions de coûts à valeurs négatives. Pour ne pas alourdir la présentation, nous supposerons ici que V est à valeurs dans \mathbb{R}^+ exclusivement. D'autre part, nous utiliserons par la suite la notation abrégée : $V_t(x, y) = V(x, y, t)$.

A chaque pas de temps t , le gestionnaire du système peut choisir parmi plusieurs actions, choix modélisé par une application Γ de l'ensemble \mathcal{X} des états vers l'ensemble des parties de \mathcal{X} (on parle de fonction *multivoque*) :

$$\Gamma : \quad \mathcal{X} \times \mathcal{T} \rightarrow \mathcal{P}(\mathcal{X})$$

où $\Gamma(x, t) \subseteq \mathcal{X}$ est l'ensemble des états accessibles au temps t depuis l'état x .

L'image $\Gamma(x, t)$ de la fonction (que nous noterons $\Gamma_t(x)$ dans ce qui suit) peut, de manière équivalente, être considérée comme l'ensemble des actions possibles au temps t et depuis l'état x . Inversement, avec un léger abus de notation, nous désignerons par $\Gamma_t^{-1}(y)$ l'ensemble des états x qui mènent potentiellement à y , en partant de l'instant $t - 1$ soit :

$$\Gamma_t^{-1}(y) = \{x \in \mathcal{X} : y \in \Gamma_{t-1}(x)\}$$

En particulier, on préférera l'attention au fait que les composées $\Gamma \circ \Gamma^{-1}$ et $\Gamma^{-1} \circ \Gamma$ n'ont pas de sens. De plus, l'indice t des fonctions Γ_t et Γ_t^{-1} renvoie systématiquement au pas de temps des états arguments de la fonction : dans la fonction Γ_t directe, l'indice t désigne le temps de départ (i.e. le temps avant l'action qui mène au changement d'état) tandis que dans Γ_t^{-1} , il désigne le temps d'arrivée (après le changement d'état).

Par définition, le système a atteint un état terminal lorsque $\Gamma_t(x) = \emptyset$. De même, par symétrie, pour un état initial x_0 on a : $\Gamma_1^{-1}(x_0) = \emptyset$. Sans perte de généralité, nous supposerons que 0 est l'unique état initial. Lorsque plusieurs états peuvent être des états initiaux (autrement dit lorsque le choix de l'état initial fait partie intégrante du processus d'optimisation), on peut toujours se ramener au cas précédent, moyennant la définition d'un état fictif 0, avec un coût de transition $V_1(0, x) = 0$ vers tous les états initiaux x .

Définition : *suite admissible*

Une suite $\mathbf{x} = (x_1, x_2, \dots, x_T)$ de \mathcal{X}^T est dite admissible pour la fonction de transition Γ si et seulement si $x_0 = 0$ et :

$$x_{t+1} \in \Gamma_t(x_t) \quad \forall t \in \{1, 2, \dots, T\}$$

Le problème se formule alors ainsi : étant donnés un ensemble d'états \mathcal{X} , une fonction de coût V et une fonction de transition Γ , trouver une suite d'états admissible optimale :

$$\mathbf{x}^* = \underset{x_t \in \mathcal{X}}{\operatorname{argmin}} \left\{ \sum_{t=1}^{T-1} V_t(x_t, x_{t+1}) \right\}$$

sous la contrainte : $x_{t+1} \in \Gamma_t(x_t)$ et $x_0 = 0$

Note : certains auteurs ajoutent un terme final $V_T(x_T)$ pour prendre en compte un coût variable, fonction de l'état d'arrivée du système. Par un raisonnement identique à celui mené plus haut sur la généralité d'un unique état initial, il est toujours possible de définir un état fictif 0, vers lequel les coûts de transitions $V_T(x_T, 0)$ de chaque état terminal seraient égaux à $V_T(x_T)$.

Dans cette définition, l'entier T est appelé *horizon* du problème. Une extension naturelle de la formulation consiste à traiter des problèmes à horizons infinis, ce qui se fait via l'introduction d'un facteur $\beta \in]0, 1[$:

$$\mathbf{x}^* = \underset{x_t \in \mathcal{X}}{\operatorname{argmin}} \left\{ \sum_{t=1}^{+\infty} \beta^t V_t(x_t, x_{t+1}) \right\}$$

Quand $\beta \rightarrow 1$, l'horizon du problème tend vers l'infini, ce qui signifie que c'est l'intérêt sur le long terme qui prime. Inversement, quand $\beta \rightarrow 0$, seuls les premiers termes de la série sont non-négligeables, et l'intérêt à court terme détermine la solution du problème. Par exemple, pour $\beta = 0.1$, chaque coût dans le futur, compte 10 fois plus dans le modèle que le coût engendré à l'époque suivante, qui lui-même compte 10 fois plus que le suivant, et ainsi de suite. Remarquons que le cas $\beta = 1$ est interdit, puisqu'il conduirait par exemple pour des fonctions de coût stationnaires, à une fonction objectif grossièrement divergente. Si V n'est pas stationnaire, on suppose que sa croissance est au plus polynomiale, permettant ainsi de supposer que le choix de β est complètement libre dans $]0, 1[$.

Ces modèles sont destinés à des problèmes relativement théoriques. Dans un cadre appliquéd, on peut toujours définir un entier t_0 tel que β^t devient négligeable pour tout $t \geq t_0$. Dans ce cours, nous ne considérerons donc que les problèmes à horizon fini.

Prenons un exemple simple : supposons que dans un immeuble à 90 étages, un ascenseur soit configuré pour parcourir en moyenne une hauteur de 3 étages (cette configuration particulière fait sens dans l'hypothèse où la tour est segmentée en une trentaine d'entreprises différentes, chacune couvrant plusieurs étages contigus). La question consiste à trouver la séquence des boutons à presser pour minimiser le coût de déplacement depuis un étage i vers un étage f . Le problème peut se modéliser ainsi : on munit un ensemble d'états $\mathcal{X} = \{0, 1, 2, \dots, 90\}$ (avec $i, j \in \mathcal{X}$) d'une fonction de coût stationnaire (donc indépendante de t) : $V(x, y) = (x - y)^2 + 4$ si $x \neq y$ et $V(x, x) = 0$. A partir de chaque étage, il est possible d'aller à tous les étages, d'où la fonction de transition : $\Gamma(x) = \mathcal{X}$ (elle aussi stationnaire). L'horizon du problème est le nombre maximal de mouvements dans une séquence directe (i.e. une séquence sans retour en arrière et sans dépasser le point d'arrivée), soit $T = f - i$.

Dans les années 50, Richard Bellman, travaillant alors pour la *RAND Corporation* (institut de recherche et de conseils pour l'U.S. Army), propose de résoudre le problème par une approche ascendante, et formule le principe suivant :

Une politique optimale possède la propriété suivante : quelques soient l'état initial et la première décision, les décisions suivantes doivent constituer une politique optimale relativement à l'état résultant de la première décision.

En somme, toute politique optimale s'appuie sur des sous-politiques optimales. Bellman donne alors le nom de *programmation dynamique* à cette nouvelle approche, où le terme programmation ne renvoie pas au code informatique², mais s'entend dans le sens militaire de programmation des entraînements, des opérations, de la logistique... On retrouve cette même terminologie dans de nombreux autres champs de recherches mathématiques, particulièrement actifs à la même époque, et dans lesquels elle est devenue synonyme d'optimisation : programmation linéaire, quadratique, en nombres entiers, etc.

En notant $v(\tau, x)$ (pour $\tau \in \mathcal{T}$, $x \in \mathcal{X}$) le coût minimal engendré par une politique finissant en x à l'instant τ , le principe s'énonce formellement comme suit :

2. Pour l'anecdote, l'expression "programmation dynamique" a été choisie comme expression bouclier, destinée à tromper un secrétaire d'état à la défense peu friand des activités de recherche académique, quant au travail réel effectué par Bellman

$$v(\tau, x) = \min_{x_t \in \mathcal{X}} \left\{ \sum_{t=1}^{\tau} V_t(x_t, x_{t+1}) : x_{t+1} \in \Gamma_t(x_t), x_{\tau} = x \right\}$$

Suivant cette définition, on a automatiquement que le coût minimal d'une politique sur l'horizon de recherche est $v(T, x)$. C'est aussi le coût d'une politique optimale. La programmation dynamique se décompose alors en deux phases. Dans la première phase (*forward*), partant de l'état initial, on calcule récursivement les coûts minimaux par sous problème $v(\tau, x)$, jusqu'à obtention de $v(T, x)$. Vient ensuite la seconde phase dans laquelle on recherche depuis l'état final (*backward*) la liste des états permettant de mener au coût minimal total $v(T, x)$. Les équations de Bellman sont alors donnés par :

Phase forward :

$$\forall x \in \mathcal{X} \quad v(0, x) = 0 \quad (\text{initialisation})$$

$$v(\tau, x) = \min_{x' \in \mathcal{X}} \left\{ v(\tau - 1, x') + V_{\tau-1}(x', x) : x' \in \Gamma_{\tau}^{-1}(x) \right\}$$

Phase backward :

$$x_T = \operatorname{argmin}_{x \in \mathcal{X}} v(T, x) \quad (\text{état final de la politique optimale})$$

$$x_{\tau-1} = \operatorname{argmin}_{x' \in \mathcal{X}} \left\{ v(\tau - 1, x') + V_{\tau-1}(x', x_T) : x' \in \Gamma_{\tau}^{-1}(x_T) \right\}$$

La phase forward s'exprime à l'aide d'une fonction min (ou max lorsque l'objectif du problème est de maximiser la quantité). La phase backward s'exprime quant à elle avec, à chaque pas de temps, l'argument du min sur la même quantité ! La procédure de calcul peut donc être optimisée en enregistrant ces arguments lors de la phase forward : c'est la mémoïsation.

Notons que nous avons ici choisi de présenter une approche *forward-backward* de la programmation dynamique, de sorte à coller au plus près des exemples applicatifs qui suivent. Dans de nombreux manuels, on rencontre l'approche (légèrement plus élégante du point de vue des notations) *backward-forward*. Le problème étant complètement réversible dans le temps (moyennant inversion des fonctions de coût et de transition), les deux approches sont équivalentes et le principe reste toujours le même : une première passe dans un sens permet de marquer la solution optimale (mémoïsation) et de calculer son coût, une seconde passe (backtracking) permet de reconstituer la solution. Dans certains problèmes, seul le coût minimal nous intéresse, et la résolution du problème ne comporte alors qu'une phase.

5.3 Principe général

Considérons le problème suivant.

On se donne un pyramide de nombres entiers naturels, dans laquelle l'étage le plus haut contient un unique nombre et tous les autres étages contiennent chacun un nombre de plus que l'étage immédiatement supérieur.

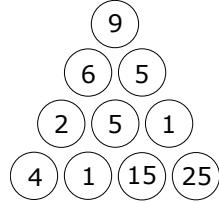


FIGURE 5.2 – Pyramide de nombres à 4 étages

L'objectif du problème est de trouver un chemin reliant le sommet de la pyramide à sa base, et minimisant la somme totale des entiers parcourus. A partir d'un noeud donné, il est possible de joindre chacun des deux noeuds situés sur l'étage inférieur. De même, excepté pour les noeuds situés sur le bord de la pyramide, chaque noeud a deux parents. Notons qu'il ne s'agit donc pas d'une structure d'arbre (cf chapitre précédent).

5.3.1 Recherche exhaustive

La solution naïve au problème consiste en une recherche exhaustive dans l'espace des solutions. Considérons une pyramide à n étages. Chaque noeud est une intersection contenant deux issues, le nombre de cheminements possibles dans le graphe est donc égal à 2^n . La complexité de l'algorithme naïf est donc exponentielle : $C(n) = \Theta(2^n)$.

5.3.2 Algorithme glouton

On appelle algorithme glouton (*greedy algorithm* en anglais) un algorithme d'optimisation qui recherche systématiquement à appliquer la meilleure solution à chaque étape d'optimisation locale.

Notons que pour certains problèmes, l'optimum global est une somme d'optimums locaux, et dans ce cas précis, un algorithme glouton permet de résoudre le problème. Sinon, la solution fournie n'est pas nécessairement optimale et l'algorithme n'est alors pas exact.

Dans le problème de la pyramide, un algorithme glouton déciderait à chaque noeud de choisir le fils (gauche ou droit) dont la valeur numérique est la plus faible. Sur l'instance de problème présenté en figure 5.2, la solution gloutonne serait :

La solution proposée est $9 + 5 + 1 + 15 = 30$ dont on vérifie aisément qu'elle n'est pas optimale.

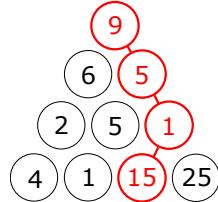


FIGURE 5.3 – Résolution par l'approche gloutonne du problème de la pyramide

En pratique, dans de nombreux problèmes, bien que n'étant pas exacte, l'approche gloutonne constitue une heuristique efficace permettant de trouver une solution acceptable en un temps rapide.

5.3.3 Programmation dynamique

L'approche par programmation dynamique consiste à remarquer que moyennant la sauvegarde de suffisamment d'informations à chaque étape, il est possible de résoudre le problème en un temps polynomial.

En effet, il suffit d'enregistrer à chaque étage, la somme minimale qui peut être atteinte en arrivant sur chaque noeud ainsi que la séquence des noeuds permettant d'arriver à cette somme. Ceci permet alors de traiter l'étage inférieur sans avoir à réétudier les noeuds supérieurs.

Remarquons tout d'abord que pour les deux premiers étages, il n'y a aucun choix possible, la somme minimale atteignable est égale à 9, à laquelle on ajoute la valeur du noeud d'arrivée. Pour chacun des deux noeuds d'arrivée, on conserve un marqueur (ici égal à 9 dans les deux cas) permettant de mémoriser l'antécédent du noeud dans le chemin optimal (figure 5.4 à gauche).

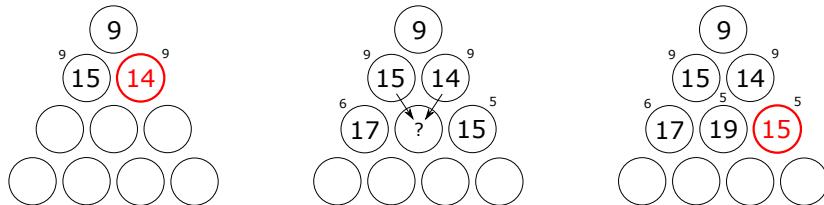


FIGURE 5.4 – Résolution du problème par programmation dynamique

On obtient alors la solution du problème de la pyramide à deux étages. La somme minimale est $9 + 5 = 14$ et l'on mémôise les marqueurs d'antécédent sur chaque noeud de sorte à pouvoir transférer la solution sur des pyramides à 3 étages ou plus. Notons que nous avons fait le choix ici de sauvegarder l'antécédent d'un noeud par sa valeur numérique mais nous aurions tout aussi bien pu enregistrer la position de ce noeud dans l'étage supérieur. Ainsi, les marqueurs 9 auraient été remplacés par des 1.

La seconde étape consiste alors à résoudre le problème à trois étages (approche *bottom-up*), en utilisant les résultats déjà calculés. Il y a trois noeuds à évaluer. La détermination des

sommes minimales que l'on peut obtenir pour un chemin finissant au niveau des noeuds gauche ou du noeud droit est immédiate : il s'agit de la somme de ce noeud avec son unique noeud parent (noeud de bord). Comme illustré sur la figure du milieu ci-dessus, les choses se compliquent pour les noeuds qui ne sont pas des noeuds de bord, i.e. ici le noeud central qui a deux parents. La somme minimale atteignable au niveau de ce noeud est alors égale à la somme de ce noeud avec le minimum de la valeur somme pré-calculée au niveau de ses deux parents. L'indice du minimum est alors sauvegardé en tant qu'antécédent et on obtient la solution du problème à trois étages : la somme minimale vaut 15 et est constituée des noeuds : 9 + 5 + 1 (figure de droite).

Jusqu'à présent, la branche droite du graphe semble plus prometteuse, mais la force de la programmation dynamique est justement de conserver les valeurs mémoisées sur la partie gauche de la pyramide jusqu'à ce que la solution de l'étage final ait été trouvée.

La propagation à l'étage 3 se fait de la même manière. Les noeuds de bords se voient affecter la somme de leur valeur et de la somme précalculée dans leur unique noeud parent, tandis que pour chacun des autres noeuds, on cherche lequel de deux noeuds parents conduit à la somme minimale et on enregistre sa valeur dans le marqueur d'antécédent.

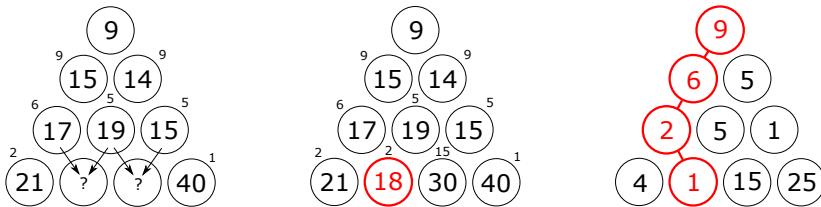


FIGURE 5.5 – Résolution du problème par programmation dynamique

Cette étape permet d'obtenir la solution pour une pyramide à 4 niveaux, et on obtient que le noeud de somme minimale vaut 18 (figure du centre). La suite des marqueurs d'antécédents nous permet alors de reconstituer le chemin de somme minimale : 9 + 6 + 2 + 1 (figure de droite) et l'on remarque que cette solution diffère de celle trouvée par l'algorithme glouton.

Pour l'implantation informatique, on note N_{ij} la valeur du i -eme noeud de l'étage j . L'étage $j = 1$ dénote le sommet de la pyramide, ainsi on a nécessairement : $1 \leq i \leq j$. Nous stockerons les solutions intermédiaires dans deux tableaux S (pour les sommes partielles) et A (pour les marqueurs d'antécédents), dont la structure est similaire à celle de N . Par souci d'efficacité, nous faisons cette fois-ci le choix de sauvegarder les antécédents par l'indice i de leur position. L'algorithme de résolution du problème par programmation dynamique s'exprime alors par :

PYRAMIDE($N, \downarrow, S \uparrow, A \uparrow$) : $\llbracket T \leftarrow \text{SIZE}(N); n \leftarrow T_2; S_{1,1} \leftarrow N_{1,1}; A_{1,1} \leftarrow \text{null};$
 $\left\{ \begin{array}{l} S_{j:2} \leftarrow S_{1,j-1}; A_{1,j} \leftarrow 1; S_{j,j} \leftarrow S_{j-1,j-1}; A_{j,j} \leftarrow j-1; \left\{ \begin{array}{l} i:2 \\ i < j \end{array} \right. \end{array} \right. \begin{array}{l} N_{i-1,j-1} \geq N_{i,j-1} ? S_{ij} \leftarrow \\ N_{ij} + S_{i-1,j-1}; A_{ij} \leftarrow i-1; | S_{ij} \leftarrow N_{ij} + S_{i,j-1}; A_{ij} \leftarrow i \end{array} \end{array} \right\}_i \}_{j-1} \rrbracket$

On peut alors écrire une fonction, permettant d'extraire la solution à partir de la table de sommes partielles et de marqueurs d'antécédents.

```
PYRAMIDE_SOL( $N, \downarrow, S \downarrow, A \downarrow, L \uparrow$ ) :  $\left[ \begin{array}{l} T \leftarrow \text{SIZE}(N); n \leftarrow T_2; \min \leftarrow S_{1,n}; \\ \text{imin} \leftarrow 1; \left\{ \begin{array}{l} \min < \min ? \quad \min \leftarrow S_{i,n}; \text{imin} \leftarrow i | i \end{array} \right. \right\}_i i \leftarrow \text{imin}; L_n \leftarrow N_{i,n}; \\ \text{PYRAMIDE\_SOL} \leftarrow S_{i,n}; \left\{ \begin{array}{l} i \leftarrow A_{ij}; L_j \leftarrow N_{i,j-1} \end{array} \right. \right\}_j \end{array} \right]$ 
```

Calculons la complexité de l'algorithme en prenant comme unité de coût le nombre d'additions. La procédure PYRAMIDE comporte une double boucle imbriquée. Pour chaque itération de la boucle externe, c'est-à-dire pour chaque étage, l'algorithme calcule i additions. Le nombre total d'additions vaut $n(n+1)/2$. La fonction PYRAMIDE_SOL récupère les indices de marqueurs pour reconstituer le chemin. Cette opération se fait en un nombre linéaire d'étapes élémentaires et ne contient aucune addition. La complexité de l'algorithme complet est donc quadratique : $C(n) = \Theta(n^2)$

5.3.4 Extension à la recherche du plus court chemin

Dans le problème de la pyramide, nous avons écrit un algorithme pour rechercher le chemin dans un graphe dont la somme des noeuds est minimale.

En considérant une instance de ce problème, il est tout-à-fait possible de transformer l'énoncé en une recherche de plus court chemin dans un graphe, où l'on attribue cette fois-ci un coût aux arcs, et non plus aux noeuds.

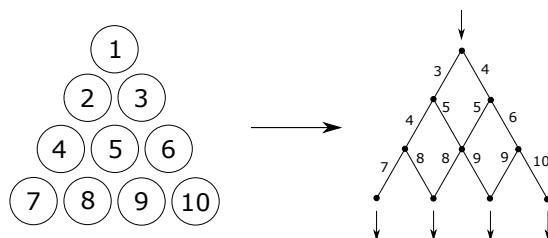


FIGURE 5.6 – Transformation du problème de la pyramide en recherche de plus court chemin

Nous remarquons donc que le problème de recherche de plus court chemin dans un graphe et celui de minimisation de la somme d'entiers positifs dans une pyramide, possèdent une composante commune. Nous verrons plus loin comment, moyennant quelques modifications, il est possible de trouver le plus court chemin d'un graphe valué à l'aide de la programmation dynamique³. Cette observation est à l'origine du célèbre algorithme de Dijkstra.

3. La même logique se trouve à la base de l'algorithme de Viterbi, utilisé dans le domaine des chaînes de Markov cachées pour débruiter un signal numérique par exemple. Dans la littérature des modèles graphiques probabilistes, il permet de trouver la configuration la plus probable d'un champ markovien, et il est connu sous le nom générique de *max-product*. Il partage de nombreuses caractéristiques avec son homologue pour l'inférence des lois marginales du champ : l'algorithme *sum-product*.

5.4 Exemple d'application : produit matriciel

Dans cette section, nous utilisons la programmation dynamique sur un exemple réel : l'optimisation du nombre de multiplications élémentaires dans un produit matriciel à n termes.

On considère n matrices réelles A_1, A_2, \dots, A_n , et on souhaite évaluer le produit :

$$P_n = A_1 \times A_2 \times \dots \times A_n = \prod_{i=1}^n A_i$$

Les dimensions des matrices A_i sont telles que le produit est valide, c'est-à-dire que l'on peut considérer avoir une liste $d_0, d_1, d_2, \dots, d_n$ d'entiers tels que la matrice A_i soit de dimension $d_{i-1} \times d_i$.

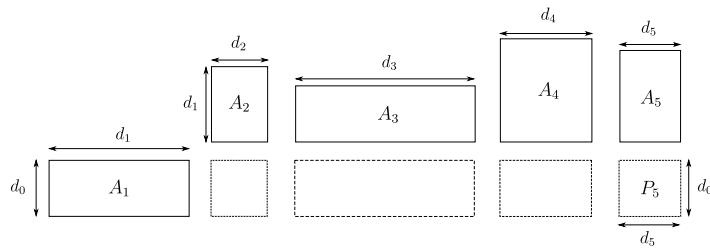


FIGURE 5.7 – Exemple de produit de 5 matrices

Par définition du produit de matrices, l'expression du calcul de P_n est associative, il est donc possible de parenthésier l'expression comme on le souhaite sans impact sur la valeur numérique de P_n .

En revanche, l'ordre dans lequel on calcule chacun des produits élémentaires détermine le nombre total de multiplications à effectuer. Prenons un exemple simple avec 3 matrices : A_1 de dimension 10×5 , A_2 de dimension 5×100 et enfin A_3 de dimension 100×50 .

$$P_n = A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3)$$

Il y a deux manières différentes de parenthésier l'expression. Chaque produit de deux matrices $A_i A_{i+1}$ entraîne $d_{i-1} d_i d_{i+1}$ multiplications.

Ainsi, le calcul de $(A_1 A_2) A_3$ nécessite : $10 \times 5 \times 100 + 10 \times 100 \times 50 = 55000$ multiplications.

Le calcul de $A_1 (A_2 A_3)$ nécessite quant à lui : $5 \times 100 \times 50 + 10 \times 5 \times 50 = 27500$ multiplications, c'est-à-dire deux fois moins que la stratégie précédente.

L'objectif du problème est donc de trouver un parenthésage qui minimise le nombre total de multiplications élémentaires à effectuer.

5.4.1 Recherche exhaustive

Notons $T(n)$ le nombre de manières de parentheser un produit de n matrices.

On peut aussi écrire P_n sous la forme : $P_n = (A_1 \dots A_i)(A_{i+1} \dots A_n)$, et en raisonnant de manière récursive, c'est-à-dire en supposant connaître le nombre de parenthésages possibles dans chacun des deux facteurs, $T(n)$ peut se décomposer en :

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

On montre que la solution de cette récurrence est la suite des nombres de Catalan :

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

A l'aide de la formule de Stirling, on obtient aisément un ordre de grandeur de la croissance de cette suite :

$$C_n = \Omega\left(\frac{4^n}{n^2}\right)$$

Plus exactement : $C_n = \Theta(4^n/n^{1.5})$. La complexité de l'algorithme de recherche exhaustive est donc exponentielle.

La question naturelle est donc de savoir si on peut faire plus rapide.

5.4.2 Résolution par programmation dynamique

L'idée consiste ici aussi à remarquer que dans le produit $(A_1 \dots A_i)(A_{i+1} \dots A_n)$, si l'on connaît les solutions optimales sur chacun des facteurs, alors la solution optimale globale **avec cette décomposition** est égale à la somme des solutions sur chacun des facteurs.

L'approche naturelle consiste donc, d'une manière similaire à celle avec laquelle nous avions déterminé la taille de l'espace des solutions dans la section précédente, à rechercher la solution optimale parmi les sous-solutions obtenues avec les $n - 1$ décompositions possibles en deux facteurs. L'approche récursive est envisageable mais risque fortement de produire une redondance notoire dans les calculs. On envisage alors une approche par programmation dynamique, où l'on cherche d'abord les solution optimales pour des produits de 3 matrices, puis pour 4, 5,... et enfin pour n matrices, en s'appuyant à chaque fois sur les solutions

précédentes.

On note $m_{i,j}$ le nombre minimal de multiplications à réaliser pour effectuer le produit partiel $P_{ij} = A_i A_{i+1} \dots A_j$.

L'objectif du problème est donc d'évaluer $m_{1,n}$. Etant donné que chaque sous-problème est défini par 2 variables, la table de mémoïsation est un tableau à 2 dimensions. De manière triviale, $m_{i,i}$ est nul et donc la diagonale du tableau est composée uniquement de 0. De même, $m_{i,j}$ n'est pas défini pour $i > j$ et donc seule la partie triangulaire supérieure (stricte) est à évaluer.

i/j	1	2	3	4	5
1	0				
2	-	0			
3	-	-	0		
4	-	-	-	0	
5	-	-	-	-	0

TABLE 5.1 – Table de mémoïsation du problème à 5 matrices

La résolution s'effectue donc en partant de la diagonale vers le coin supérieur droit jusqu'à obtention de la valeur recherchée $m_{1,n}$.

De la même manière que pour le problème de la pyramide, à chaque étape il est nécessaire d'enregistrer le produit élémentaire ayant conduit au résultat minimal. On utilise donc un second tableau de marqueurs, référençant l'indice de la découpe de produit. Par simplicité, nous fusionnerons ces deux tableaux en un seul dans l'exemple ci-dessous.

On considère le produit matriciel : $P_5 = A_1 A_2 A_3 A_4 A_5$ avec l'ensemble des dimensions : $d = (10, 50, 5, 100, 1, 20)$. On initialise la table en remplissant sa diagonale. La valeur - correspond au marqueur *null* du problème précédent, et indique qu'aucune multiplication n'a encore été effectuée.

i/j	1	2	3	4	5
1	0/-				
2	-	0/-			
3	-	-	0/-		
4	-	-	-	0/-	
5	-	-	-	-	0/-

TABLE 5.2 – Initialisation de la table de mémoïsation du problème à 5 matrices

On cherche alors à remplir la diagonale supérieure. De ce qui précède, on obtient aisément une équation de récurrence sur m_{ij} :

$$\forall i > j : m_{ij} = \min_{i \leq k < j} m_{i,k} + m_{k+1,j} + d_{i-1} d_k d_j$$

L'équation sur les marqueurs est exactement la même, si ce n'est que l'opérateur \min est remplacé par argmin .

Par exemple, le calcul de la cellule $m_{1,3}$ du tableau s'exprime par :

$$m_{1,3} = \min \left\{ m_{1,1} + m_{2,3} + d_0 d_1 d_3, m_{1,2} + m_{3,3} + d_0 d_2 d_3 \right\}$$

Le terme $m_{1,3}$ s'exprime donc uniquement à partir de termes situés sur les diagonales précédentes (dans l'ordre de parcours du tableau) et qui ont donc été précalculés.

Le remplissage de la seconde diagonale est immédiat (il s'agit du problème à 2 matrices, il n'y a donc que le terme $d_{i-1} d_k d_j$ qui intervient). Pour chaque cas, il n'y a qu'un seul découpage possible et donc une seule valeur de k . Le marqueur est donc nécessairement égal à la valeur i de la ligne concernée.

i/j	1	2	3	4	5
1	0/-	2500/1			
2	-	0/-	25000/2		
3	-	-	0/-	500/3	
4	-	-	-	0/-	2000/4
5	-	-	-	-	0/-

TABLE 5.3 – Calcul de la deuxième diagonale supérieur

On termine alors le remplissage du tableau de la même manière.

i/j	1	2	3	4	5
1	0/-	2500/1	7500/2	1250/1	1450/4
2	-	0/-	25000/2	750/2	1750/4
3	-	-	0/-	500/3	600/4
4	-	-	-	0/-	2000/4
5	-	-	-	-	0/-

TABLE 5.4 – Tableau de mémoïsation après exécution de l'algorithme

On obtient donc que le nombre optimal de multiplications vaut 1450. Pour trouver le découpage correspondant, il suffit de remonter les indices situés à droite du signe / en partant de la cellule supérieure gauche.

Le premier marqueur indique $k = 4$ et correspond donc au découpage : $(A_1 A_2 A_3 A_4) A_5$. La solution du sous-problème P_4 est contenue dans la cellule $m_{1,4}$ et donne lieu à un second découpage à l'indice $k = 1$, ce qui donne un second parenthésage : $(A_1 (A_2 A_3 A_4)) A_5$. En

continuant ainsi jusqu'à décomposition complète, on trouve le parenthésage optimal :

$$P_5 = (A_1(A_2(A_3A_4)))A_5$$

On vérifie bien que l'opération nécessite 1450 multiplications et résulte en une matrice de dimensions 10×20 .

Si l'on compare par rapport à la valeur $m_{2,3}$ qui correspond au nombre de multiplications requises par le calcul de A_2A_3 et qui vaut à lui seul 25000, on remarque que le gain de temps procuré par l'optimisation est conséquent. En particulier, le produit dans l'ordre d'arrivée des matrices s'effectue en 8700 multiplications, soit une réduction de 83% sur le nombre total d'opérations à effectuer.

Exercice 5.2.

En s'inspirant de la méthode présentée ci-dessus, calculer la situation la plus défavorable, c'est-à-dire un parenthésage qui résulte en le plus grand nombre de multiplications élémentaires. En déduire le gain de l'optimisation en comparant le résultat trouvé ci-dessus par rapport au cas défavorable.

5.4.3 Etude de la complexité

Il y a $n - 1$ diagonales à calculer. Chaque diagonale i ($i \in \{1, n - 1\}$) contient $n - i$ cellules. Chaque cellule nécessite quant à elle de calculer $2i$ produits (il faut rechercher le minimum parmi i valeurs, chacune nécessitant d'évaluer le terme $d_{i-1}d_kd_j$ soit 2 multiplications, les autres termes sont pré-calculés dans le tableau de mémoïsation).

La complexité de l'algorithme (en terme de nombres de multiplications) s'exprime alors :

$$C(n) = \sum_{i=1}^{n-1} 2i(n - i) = 2 \left(n \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2 \right)$$

Par sommation (cf annexe B.4 et exercice 3.3 page 103 si nécessaire) on obtient :

$$C(n) \underset{n \rightarrow \infty}{\sim} 2 \left(n \frac{n^2}{2} - \frac{n^3}{3} \right) = \frac{n^3}{3} \Rightarrow C(n) = \Theta(n^3)$$

La complexité de l'algorithme est donc polynomiale (plus précisément cubique), à comparer avec la complexité exponentielle de la recherche exhaustive.

5.5 Exemple d'application 2 : calcul d'itinéraires

L'un des domaines où la programmation dynamique est la plus massivement employée est très certainement la recherche d'itinéraire. En effet, si l'on cherche le plus court chemin entre deux noeuds a et b d'un graphe, alors si l'itinéraire passe par une troisième noeud c , on est certain que les sous-chemins ac et cb sont des sous-solutions optimales du problème de recherche d'itinéraire. En effet, si par exemple ac n'est pas optimal pour joindre a à c , alors c'est qu'il existe un chemin $(ac)'$ $\neq ac$ joignant a à c et tel que $|(ac)'| < |ac|$ (en notant $|\cdot|$ l'application qui à un itinéraire renvoie sa longueur). On obtient alors que :

$$|(ac)'b| = |(ac)'| + |cb| < |ac| + |cb| = |ab|$$

C'est une contradiction puisque ab est un chemin optimal par hypothèse.

5.5.1 Recherche de toutes les longueurs optimales

Il existe un algorithme très simple pour trouver les longueurs de tous les chemins optimaux sur un graphe : l'algorithme de Floyd-Warshall.

La particularité de cet algorithme est que la table de mémoïsation est aussi la table du résultat recherché.

On se donne un graphe $G(V, E)$ où V est un ensemble de sommets (*vertices*) numérotés de 1 à n et $E \subseteq V \times V$ est un ensemble d'arcs (*edges*) numérotés de 1 à m . A chaque arc e_{ij} est associé un poids w_{ij} correspondant au coût (kilométrique, tarifaire, carbone...) engendré par le parcours de l'arc reliant les sommets v_i et v_j .

L'objectif du problème est de calculer la matrice D (carrée de taille n) telle que D_{ij} contienne le coût du trajet du sommet v_i au sommet v_j .

La résolution consiste à calculer une séquence de tables D^k , où D_{ij}^k représente le coût de trajet pour relier i à j en n'utilisant que des sommets intermédiaires dans $\{1, 2, \dots, k\}$. Par définition, la solution au problème est alors : $D = D^n$.

L'initialisation est immédiate :

$$D_{ij}^0 = \begin{cases} w_{ij} & \text{si } e_{ij} \in E \\ +\infty & \text{sinon} \end{cases}$$

A chaque étape, on ajoute un nouveau sommet k dans l'ensemble des sommets intermédiaires. Chaque distance est alors mise à jour et ne peut que décroître : si le sommet k n'apporte pas d'amélioration dans le trajet entre i et j alors D_{ij} conserve sa valeur : $D_{ij}^k = D_{ij}^{k-1}$.

Sinon, la nouvelle valeur devient égale à $D_{ik}^{k-1} + D_{kj}^{k-1}$. La formule de passage s'écrit donc :

$$\forall i, j \in V \quad \forall k \in \{1, \dots n\} \quad D_{ij}^k = \min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})$$

On obtient alors directement l'algorithme permettant de calculer la table D :

FLOYD($D \Downarrow$) : $N \leftarrow \text{SIZE}(D)$; $n \leftarrow N_1$; $\boxed{\boxed{\boxed{\left\{ \begin{array}{l} \left\{ \begin{array}{l} \left\{ \begin{array}{l} \left\{ \begin{array}{l} n \\ k:1 \end{array} \right\} \left\{ \begin{array}{l} n \\ i:1 \end{array} \right\} \left\{ \begin{array}{l} n \\ j:1 \end{array} \right\} D_{ij} \leftarrow \text{MIN}(D_{ij}, D_{ik} + D_{kj}) \end{array} \right\}_j \end{array} \right\}_i \end{array} \right\}_k}}$

L'algorithme effectue une triple boucle sur n , sa complexité est donc cubique : $C(n) = \Theta(n^3)$. Il est intéressant de constater que cette complexité ne dépend que du nombre de sommets et pas du nombre d'arcs dans le réseau.

Remarque : l'approche par programmation dynamique ne fonctionne plus lorsqu'il s'agit de trouver les plus long chemins.

5.5.2 Recherche d'un chemin le plus court

L'algorithme de Dijkstra est très certainement l'algorithme de recherche d'itinéraire dans un graphe le plus célèbre. Dans cette section, après avoir expliqué le principe de la résolution du problème, nous donnons l'algorithme en langage ADL, puis nous apportons sa preuve de correction, et enfin nous étudions sa complexité en temps.

Principe

On considère un graphe connexe $G(V, E)$ (que l'on supposera orienté⁴) de cardinalités $|V| = n$ sommets et $|E| = m$ arcs. Soit $c_{ij} \in \mathbb{R}^{+*}$ le coût pour parcourir l'arc e_{ij} . Les sommets du graphe sont dénotés $\{1, 2, \dots, n\}$.

Par exemple, le graphe de la figure 5.8 possède $n = 12$ sommets et $m = 17$ arcs. D'un point de vue informatique, on le représente par une table G de tailles 17×3 , où chaque ligne contient les informations relatives à l'arc m . Par exemple, si le premier arc est e_{12} alors, on aura : $G_{1,1} = 1$, $G_{1,2} = 2$ et $G_{1,3} = c_{12}$.

L'algorithme de Dijkstra effectue un parcours en largeur du graphe, c'est-à-dire qu'il explore en premier lieu les noeuds voisins du sommet de départ, et élargit son champ de recherche de proche en proche jusqu'à trouver le noeud final (cf section 4.4.2). L'astuce consiste de plus à utiliser le principe de la programmation dynamique. En effet, comme démontré en début de section, le problème de la recherche d'itinéraires possède une structure de sous-problèmes optimaux. Si un chemin est optimal, alors tous les chemins partiels

4. Sans perte de généralité puisqu'un graphe non-dirigé peut-être *orienté* en transformant chaque arrête en deux arcs dirigés de sens opposés.

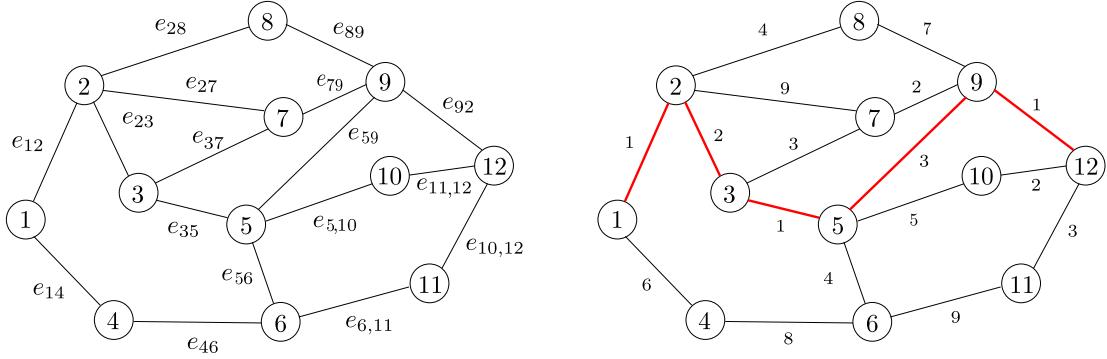


FIGURE 5.8 – Exemples de modèles de graphes non-dirigés et valués, avec un chemin optimal (à droite) entre les noeuds 1 et 12, pour un coût total de 8 unités.

entre les noeuds de l'itinéraire sont également des chemins optimaux. La programmation dynamique est donc bien adaptée pour traiter ce problème.

Dans tout ce qui suit, on note $s \in \{1 \dots n\}$ l'indice du noeud de départ (*source*) et $t \in \{1 \dots n\}$ celui du noeud d'arrivée (*target*). L'objectif du problème est de trouver une séquence d'arcs joignant s à t et dont la somme des coûts est minimale. Plus formellement, on recherche une fonction $\phi^* : \mathbb{N} \rightarrow \{1, 2, \dots, m\}$ appartenant à l'ensemble Φ des fonctions vérifiant la règle de chaînage :

$$\begin{cases} G_{\phi(1),1} = s \\ G_{\phi(p),2} = t \\ G_{\phi(k),2} = G_{\phi(k+1),1} \quad \forall k \in \{1, 2, \dots, p-1\} \end{cases}$$

Un chemin optimal est caractérisé par : $\phi^* \in \operatorname{argmin}_{\phi \in \Phi} \left\{ \sum_{k=1}^p G_{\phi(k),3} \right\}$

On initialise l'algorithme en créant une table de distances D^k , dans laquelle chaque ligne i représente le coût minimal pour aller du sommet initial s au sommet i en utilisant uniquement k sommets dans $\{1 \dots n\}$ (les k plus proches de s). A l'étape $k = 0$, seul le sommet de départ s est joignable et donc D_s^0 vaut 0 tandis que $D_i^0 = \infty$ pour tout autre sommet i . Conformément au principe de la programmation dynamique, la table D^n , obtenue après le parcourt de l'ensemble des sommets donne la solution au problème posé sous forme du coût minimal pour joindre t depuis s (bien souvent, la solution est atteinte avant d'avoir parcouru tous les sommets, puisqu'on s'arrête dès que t a été trouvé). Le coût recherché est alors lu en t -eme ligne de la table, et on complète la méthode en enregistrant une table de marqueurs A afin de pouvoir reconstituer le chemin par *backtracking*.

Définition : *backtracking*

En programmation dynamique, on appelle *backtracking* l'étape finale qui consiste à reformer la solution à l'aide de la table de mémoïsation et en effectuant un parcours inverse à celui de l'étape de résolution des sous-problèmes.

Exemples : l'étape de backtracking est en général nécessaire lorsque le problème à traiter est un problème d'optimisation. La solution est alors composée de la valeur minimale (ou maximale) atteinte, et de la configuration qui permet d'atteindre cette valeur. Ainsi, le calcul de la suite de Fibonacci, qui ne résout pas un problème d'optimisation, ne nécessite pas d'étape de backtracking. En revanche, pour le problème de la pyramide étudié en première partie, tout comme pour le problème de recherche d'itinéraire, l'étape de backtracking consiste à recréer le chemin parcouru. Dans le problème de la multiplication matricielle, elle permet d'obtenir explicitement le parenthésage à appliquer pour minimiser le nombre de multiplications élémentaires.

Algorithm

On commence par écrire un algorithme prenant en entrée la table des arcs G et retournant la table de voisinage V .

$$\text{VOISINAGE}(G, \downarrow, V \uparrow) : \left[\begin{array}{l} T \leftarrow \text{SIZE}(G); m \leftarrow T_1; n \leftarrow 0; \left\{ \begin{array}{l} \underset{i:1}{n} \leftarrow \text{MAX}(n, G_{m,1}); \\ n \leftarrow \text{MAX}(n, G_{m,2}) \end{array} \right\}_i \\ \text{creation de la table de voisinage } \textcircled{C} \left\{ \begin{array}{l} \underset{i:1}{\left\{ \begin{array}{l} \underset{j:1}{n} \left\{ \begin{array}{l} V_{ij} \leftarrow -1 \end{array} \right\}_j \end{array} \right\}_i \\ \underset{i:1}{c} \left\{ \begin{array}{l} c \leftarrow 1; \left\{ \begin{array}{l} \underset{j:1}{n} G_{j,1} = i ? \quad V_{ic} \leftarrow j; c \leftarrow c + 1 \mid \text{ } \end{array} \right\}_j \end{array} \right\}_i \end{array} \right\} \end{array} \right]$$

L'élément V_{ij} de la table calculée par le module VOISINAGE, contient l'indice (dans la table originale G) du j -eme arc partant du sommet i . Par exemple, on donne ci-dessous les tables G et V correspondant au graphe de la figure 5.8 (droite).

$$G = \begin{array}{|c|c|c|} \hline 1 & 1 & 2 & 1 \\ \hline 2 & 1 & 4 & 6 \\ \hline 3 & 2 & 3 & 2 \\ \hline 4 & 2 & 7 & 9 \\ \hline 5 & 2 & 8 & 4 \\ \hline 6 & 3 & 5 & 1 \\ \hline 7 & 3 & 7 & 3 \\ \hline 8 & 4 & 6 & 8 \\ \hline 9 & 5 & 6 & 4 \\ \hline 10 & 5 & 9 & 3 \\ \hline 11 & 5 & 10 & 5 \\ \hline 12 & 6 & 11 & 9 \\ \hline 13 & 7 & 9 & 2 \\ \hline 14 & 8 & 9 & 7 \\ \hline 15 & 9 & 12 & 1 \\ \hline 16 & 10 & 12 & 2 \\ \hline 17 & 11 & 12 & 3 \\ \hline \end{array} \quad V = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 2 & 3 & 4 & 5 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 3 & 3 & 6 & 7 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 4 & 2 & 8 & -1 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 5 & 6 & 9 & 10 & 11 & -1 & -1 & -1 & -1 & \dots \\ \hline 6 & 8 & 9 & 12 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 7 & 4 & 7 & 13 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 8 & 5 & 14 & -1 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 9 & 10 & 13 & 14 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 10 & 11 & 16 & -1 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 11 & 12 & 17 & -1 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline 12 & 15 & 16 & 17 & -1 & -1 & -1 & -1 & -1 & \dots \\ \hline \end{array}$$

Par exemple, la ligne 4 de la table V ne contient que deux éléments positifs : $V_{41} = 2$ et $V_{42} = 8$, ce qui indique que depuis le sommet 4, on ne peut emprunter que les arcs 2 et 8. Les sommets voisins de 4 peuvent alors être récupérés dans la table G : il s'agit de $G_{21} = 3$ et $G_{8,2} = 6$. Notons que dans cet exemple nous avons supposé le graphe comme étant non-orienté.

L'algorithme de Dijkstra prend en entrée la table G des arcs ainsi que les indices des sommets de départ s et d'arrivée t . La sortie de l'algorithme est composée de la distance du plus court chemin (en sortie standard via le nom de la fonction) ainsi que d'un vecteur C du parcours entre s à t , où C contient la séquence des indices d'arcs à parcourir.

Pour simplifier le code, on utilise une structure de file de priorité F (telle qu'introduite dans le chapitre précédent). Chaque entrée de F contient un couple (i, d) où i est le numéro d'un noeud et d est son ordre de priorité. L'appel de la fonction $\text{POP}(F)$ retourne le couple (i, d) où i est le noeud de priorité minimale d dans F (le noeud est physiquement retiré de la file). Il est également possible de changer la priorité d'un noeud avec la fonction $\text{UPDATE}(F, i, d)$, où d représente la nouvelle valeur de priorité à attribuer au noeud i .

REV est une fonction qui retourne l'image miroir d'une liste. $\text{GET}(V, x)$ est une fonction (à écrire à titre d'exercice) qui renvoie tous les arcs partant du sommet x à partir de la table V . Par exemple, dans le graphe de la figure 5.8, $\text{GET}(V, 6)$ retourne la liste [8, 9, 12].

DIJKSTRA($G, \downarrow, s \downarrow, t \downarrow, C \uparrow$) : $\llbracket T \leftarrow \text{SIZE}(T); m \leftarrow T_1; n \leftarrow 0;$

$$\left\{ \begin{array}{l} n \leftarrow \text{MAX}(n, G_{m,1}); n \leftarrow \text{MAX}(n, G_{m,2}) \end{array} \right\}_i$$

© création de la table de voisinage ©

VOISINAGE(G, V);

④ création de la table de mémoïzation ④
 $\left\{ \begin{array}{l} \overset{n}{D_i = \text{MAX_INT}()} \\ i:1 \end{array} \right\}_i D_s = 0;$

④ création et remplissage de la file de priorité ④
 $F \leftarrow \text{BUILD_PRIO}(); \left\{ \begin{array}{l} \overset{n}{\text{ADD}(F, i, D_i)} \\ i:1 \end{array} \right\}_i$

$\left\{ \begin{array}{l} \text{/SIZE}(F) \neq 0 \\ K \leftarrow \text{POP}(F); D_{K_1} \leftarrow K_2; \text{NEXT} \leftarrow \text{GET}(V, K_1); \end{array} \right.$

$\left\{ \begin{array}{l} \text{SIZE(NEXT)} \\ L \leftarrow \text{NEXT}_i; v \leftarrow G_{L,2}; D_v > D_{K_1} + G_{V_i,3} ? D_v \leftarrow D_{K_1} + G_{V_i,3}; \end{array} \right.$

$\text{UPDATE}(F, v, D_{K_1} + G_{V_i,3}) \quad \text{④ on enregistre l'antécédent du noeud} \quad \text{④ } A_v \leftarrow V_i; | i \quad \left\{ \begin{array}{l} K_1 = t ? \quad \text{④ on a atteint le noeud d'arrivée} \quad \text{④ } ! | i \end{array} \right\}$

④ backtracking de l'itinéraire ④
 $C_1 \leftarrow A_t; l \leftarrow 1; \left\{ \begin{array}{l} \text{/G}_{C_l,1} \neq s \\ l \leftarrow l + 1; a \leftarrow C_{l-1}; C_l \leftarrow A_{G_{a,1}} \end{array} \right\}$

④ sorties du module ④
 $\text{REV}(C); \text{DIJKSTRA} \leftarrow D_t \quad \boxed{\quad}$

Analyse de l'algorithme

L'algorithme est composé de boucles à compteur sans débranchements dans la fonction de préparation du voisinage (qui termine donc nécessairement) ainsi que d'une boucle *tant que* dans la fonction de calcul de l'itinéraire. A chaque passage dans cette boucle, un élément est retiré de la file de priorité, dont la taille est donc strictement décroissante. Etant donné qu'aucun élément n'est ajouté dans la file durant les itérations de cette boucle et que la taille de cette file est un entier nécessairement positif, il existe un nombre n tel qu'après n itérations, la file de priorité est vide. L'algorithme termine donc.

D'autre part, il est aisément de montrer par récurrence, qu'à chaque étape, D_i^k contient la distance minimale entre le sommet de départ et i en utilisant uniquement les k premiers noeuds inspectés. L'algorithme s'arrête lorsque tous les noeuds ont été inspectés (auquel cas $k = n$ et la correction est triviale puisqu'on a parcouru tout le graphe) ou bien quand la distance minimale de la file est D_t^k . Dans ce second cas de figure, cela signifie que t est joignable depuis s avec un coût D_t^k en utilisant uniquement les k premiers noeuds inspectés, et que les $n - k$ autres noeuds ne sont pas joignables avec un coût inférieur ou égal à D_t^k . La distance entre s et t est donc au moins égale à la distance trouvée qui est donc minimale pour l'ensemble des itinéraires de s à f .

Calculons à présent la complexité en temps de l'algorithme de Dijkstra (nous ne prendrons pas en compte ici la phase de préparation consistant à calculer la table de voisinage, puisqu'en pratique, le graphe est préparé une fois pour toutes, avant de l'utiliser pour calculer les itinéraires). Il suffit donc d'évaluer la contribution de la boucle *tant que*. Nous prenons comme unité de taille le couple (n, m) traduisant la taille du réseau, et le coût de l'algorithme sera évalué en nombre de comparaisons.

Nous nous intéressons à la complexité dans le pire des cas. En particulier, si le noeud d'arrivée est le dernier noeud inspecté, l'algorithme parcourt n fois la boucle (puisque un seul et unique noeud est inspecté à chaque passage). D'autre part, pour trouver le noeud à inspecter lors d'une itération donnée, il faut trouver un noeud de priorité minimale dans la file de priorité. En admettant que l'on modélise cette file à l'aide d'un tableau classique, cette opération nécessite n opérations. Au total, cela nécessite donc n^2 opérations.

Enfin, il faut comptabiliser le nombre de réajustements des distances. Un réajustement est effectué à chaque fois qu'un nouveau chemin est trouvé vers un noeud. Ce nombre est donc inférieur au nombre m d'arcs dans le graphe. La complexité s'exprime donc :

$$C_{max}(n, m) = \Theta(n^2 + m)$$

Il est possible d'exprimer cette complexité uniquement en fonction du nombre de noeuds, en remarquant qu'un graphe à n sommets contient au plus de l'ordre de n^2 arcs (un graphe orienté complet K_n , en contient exactement $n(n - 1)$, en excluant les boucles, ce qui est toujours possible durant le prétraitement). La complexité s'exprime alors :

$$C_{max}(n) = \Theta(n^2)$$

A l'inverse, il est difficile d'exprimer $C_{max}(m)$ puisque le nombre de sommets peut augmenter infiniment et indépendamment du nombre d'arcs. Toutefois, en ajoutant l'hypothèse que le graphe est connexe, on a nécessairement : $n \leq m + 1$ et donc en conséquence :

$$C_{max}(n, m) = \Theta(m^2)$$

On trouve que la complexité est également quadratique avec le nombre d'arcs.

Pour finir, remarquons qu'il est possible d'accélérer l'algorithme, en utilisant une file de priorité plus efficace, telle qu'un tas binaire, dont nous avons parlé au paragraphe 4.4.4. La construction d'un tas binaire se fait en un temps linéaire. La recherche et suppression du premier élément se fait lui en temps logarithmique. Enfin, la modification d'une valeur de priorité se fait en temps également logarithmique. On obtient alors :

$$C_{max}(n, m) = \Theta((m + n) \log n)$$

La complexité en temps de l'algorithme est alors fortement liée au nombre d'arcs du graphe. Pour un graphe complet ou presque, la recherche du plus court chemin avec l'algorithme de Dijkstra sera toujours quadratique (ou quasi-quadratique) en fonction du nombre de

sommets. En revanche, pour un graphe creux (*i.e.* quand $m = \mathcal{O}(n)$), la complexité devient quasi-linéaire. En pratique, la plupart des graphes rencontrés (réseaux de rues, etc.) sont planaires, et on pourra montrer⁵ à titre d'exercice l'inégalité $m \leq 3n - 6$, d'où $m = \mathcal{O}(n)$, ce qui est précisément une caractérisation des graphes creux. L'algorithme de Dijkstra affiche donc en général une complexité quasi-linéaire.

Remarque : Notons qu'il est possible de faire encore plus rapide en utilisant un tas de Fibonacci, dont la mise à jour d'une valeur de priorité se fait en temps constant. Le temps de calcul se réduit alors à $\Theta(n \log n + m)$. Cette complexité améliorée pourrait faire douter quant à l'intérêt de l'algorithme de Floyd-Warshall (présenté en page 219). En effet, chaque appel à l'algorithme de Dijkstra permet de calculer la distance entre une sommet donné et les $n - 1$ autres sommets du graphe. En conséquence, n appels permettent d'obtenir un résultat similaire à celui retourné par Floyd-Warshall, ce qui, en termes de complexité se chiffre à $n \times \Theta(n \log n + m)$, soit (dans le cas le plus pessimiste pour un graphe dense) une complexité cubique. Pour un graphe creux, cette seconde alternative est même plus efficace puisqu'elle résulte en une complexité en $\Theta(n^2 \log n)$. En pratique, de par ses trois boucles imbriquées pouvant être optimisées et exécutées très rapidement sur la plupart des processeurs, l'algorithme de Floyd-Warshall est bien souvent plus efficace (on parle de *tightly nested loops*). Nous avons ici une illustration des limites du modèle de complexité théorique (asymptotique) par rapport à la réalité du terrain, et dont nous avions touché un mot dans le chapitre 3 (page 105). Par ailleurs, l'implémentation de Floyd-Warshall est plus simple que celle de Dijkstra et ne nécessite pas de structures de données avancées telles que les tas binaires ou les tas de Fibonacci.

5.6 Exemple d'application 3 : estimation de temps de trajet

Pour conclure cette section, nous donnons un exemple concret de déploiement de la programmation dynamique, sur un cas d'étude réel inspiré de Wang *et al.* (2014).

Grâce aux applications mobiles collaboratives d'aide à la conduite, de gros volumes de trajectoires GPS issues de véhicules individuels deviennent disponibles. Le *map inference* est un champ de recherche opérationnelle, visant à utiliser ces trajectoires pour inférer de l'information géographique (*e.g.* réseau routier, limitations de vitesse, nombre de voies, feux tricolores...) et ainsi remplacer les méthodes cartographiques traditionnelles. Parmi les avantages indéniables du *map inference* figurent notamment une capacité de mise-à-jour rapide et peu coûteuse (en moyens humains et matériels) ainsi qu'une aptitude à inférer des éléments abstraits (zones à risque, temps de trajet, profil de vitesse recommandé...).

5.6.1 Formulation du problème

Nous nous intéressons ici à l'estimation précise des temps de parcours, en faisant l'hypothèse que le jeu complet de trajectoires a été ventilé par tranches horaires à l'aide des *timestamps*, de sorte à pouvoir fournir une estimation qui soit fonction des conditions actuelles (heure creuses, heures pleines, variations saisonnières...). Nous supposerons également que les traces GPS ont été acquises à une fréquence temporelle relativement basse

5. On pourra se placer dans le cadre des graphes simples (sans boucles et sans arêtes multiples) et invoquer la formule d'Euler

(typiquement, de l'ordre d'un point par minute). Considérons alors un itinéraire $p_0 p_1 \dots p_n$ comportant n segments de routes successifs.

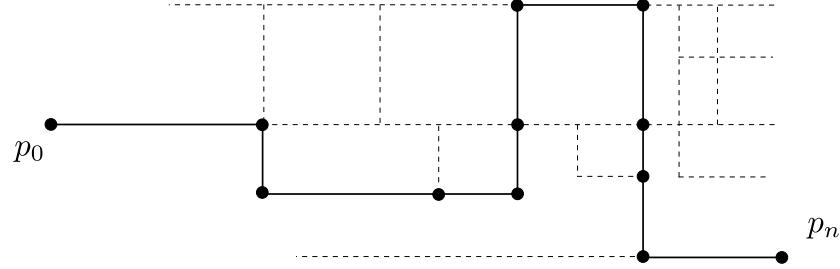


FIGURE 5.9 – Exemple d'un itinéraire sur un graphe routier

Notons que cet itinéraire est un sous-ensemble d'un graphe routier plus général. Ainsi, parmi les véhicules empruntant (partiellement) cet itinéraire, tous ne vont pas du point p_0 au point p_n (si le jeu de trajectoires est particulièrement sporadique, il peut même arriver que l'on ne dispose d'aucune trace GPS ayant parcouru l'intégralité de l'itinéraire). En revanche, en relaxant les contraintes du problème, on peut s'intéresser à l'ensemble des sections de trajectoires ayant parcouru au moins un des tronçons de l'itinéraire, dans le même sens de circulation que l'itinéraire global. Par exemple, sur la figure 5.10, la trace rouge (se déplaçant de la gauche vers la droite) apporte une information sur les tronçons $[p_1 p_3]$ et $[p_5 p_7]$, puisqu'elle y circule dans le même sens que l'itinéraire global. La trace verte, qui elle se déplace principalement dans la direction inverse, n'apporte de l'information que sur le tronçon $[p_7 p_8]$.

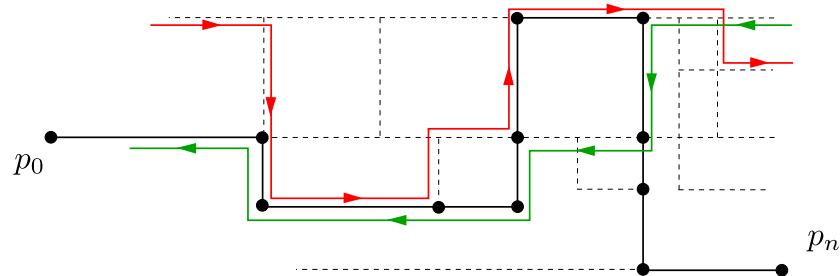


FIGURE 5.10 – Exemple d'un itinéraire partiellement parcouru par 2 trajectoires GPS

A l'issue de cette étape, moyennant abstraction du reste du réseau routier et des sections de trajectoires non pertinentes, le problème est réduit au calcul d'un temps de trajet sur un itinéraire (arbitrairement segmenté par les noeuds du graphe routier, mais d'autres schémas de segmentation sont également possibles), sur lequel on ne dispose que d'informations fragmentaires sur les temps de parcours, comme illustré sur la figure ci-après.

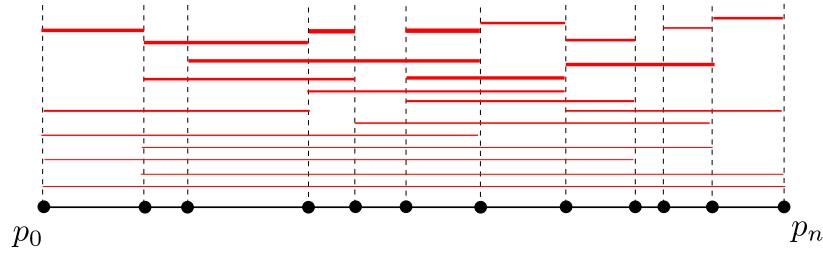


FIGURE 5.11 – Représentation schématique de l’itinéraire et des traces GPS

La figure 5.11 représente le nombre de mesures de temps de parcours dont on dispose par tronçon (l'épaisseur du trait indique ce nombre). En général, excepté pour les itinéraires longeant les grandes artères principales, plus le tronçon d'itinéraire est long, moins il a été parcouru intégralement, et donc moins on dispose de mesure de temps de parcours le concernant.

A ce stade, il est important de rappeler que la fréquence d'acquisition est supposée faible. Ainsi, la connaissance d'un temps de parcours sur un tronçon \$[p_i p_j]\$ (\$i < j\$) par exemple, n'implique pas la connaissance des temps de parcours intermédiaires \$[p_k p_l]\$ (\$i \leq k < l \leq j\$). Pour s'en convaincre, considérons la figure 5.12, où le tronçon \$[p_0 p_2]\$ est parcouru par une trace GPS ne contenant que deux mesures, aux positions \$x_0 \approx p_0\$ et \$x_1 \approx p_2\$ (datées respectivement en \$t(x_0)\$ et \$t(x_1)\$). A partir de cette unique trace, le temps de parcours \$\delta t_{02}\$ sur le tronçon \$[p_0 p_2]\$ est estimé par \$t(x_1) - t(x_0)\$, et on comprend bien qu'aucune information n'est disponible sur les durées intermédiaires \$\delta t_{01}\$ et \$\delta t_{12}\$.

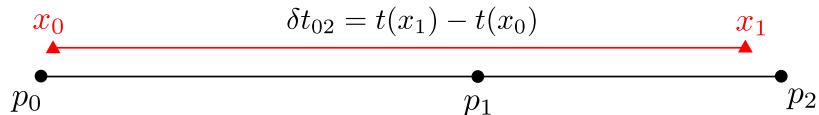


FIGURE 5.12 – Calcul du temps de parcours sur le tronçon \$[p_0 p_2]

Chaque portion de trajectoire nous donne donc une estimation \$\delta t_{ij}\$ du temps de parcours d'un tronçon \$[p_i p_j]\$, et lorsque plusieurs portions sont relevées sur le même tronçon, il paraît naturel de moyennner les temps de parcours pour obtenir une estimation plus robuste. Par exemple, si \$N_{ij}\$ estimations individuelles sont disponibles, la valeur finale moyennée du temps de parcours sur le tronçon est :

$$\delta t_{ij} = \frac{1}{N_{ij}} \sum_{k=1}^{N_{ij}} \delta t_{ij}^k$$

où \$\delta t_{ij}^k\$ est l'estimation de temps de parcours du tronçon \$[p_i p_j]\$ calculée à partir de la portion de trajectoire \$k\$.

Etant donnée une subdivision $\mathcal{S} = (s_1, s_2, \dots, s_{|\mathcal{S}|})$ de l'itinéraire (*i.e.* une liste ordonnée d'indices croissants compris dans $\{0, 1, \dots, n\}$ et telle que $s_1 = 0$ et $s_{|\mathcal{S}|} = n$), une estimation du temps de parcours total est donnée par la somme des estimations calculées sur tous les tronçons de la subdivision :

$$\delta t(\mathcal{S}) = \delta t_{s_1, s_2} + \delta t_{s_2, s_3} + \dots = \sum_{i=1}^{|\mathcal{S}|-1} \delta t_{s_i, s_{i+1}}$$

Notons que ce résultat dépend de la subdivision choisie, et que pour une itinéraire composé d'un nombre conséquent de tronçons, il existe un grand nombre de subdivisions possibles. Une démarche classique consiste alors à choisir une subdivision qui mène à l'estimation la plus robuste, c'est-à-dire en termes statistiques, une subdivision qui minimise la variance de l'estimateur δt . Calculons cette variance : en première approximation, il paraît raisonnable de supposer que les δt_{ij} sont indépendants sur des intervalles disjoints. En conséquence, on peut écrire la variance de la somme comme une somme de variances :

$$\text{Var}(\delta t(\mathcal{S})) = \text{Var}\left(\sum_{i=1}^{|\mathcal{S}|-1} \delta t_{s_i, s_{i+1}}\right) = \sum_{i=1}^{|\mathcal{S}|-1} \text{Var}(\delta t_{s_i, s_{i+1}})$$

D'autre part, chaque estimation par tronçon δt_{ij} a été obtenue en moyennant les temps de trajets estimés à partir de N_{ij} portions de trajectoires de véhicules. On peut supposer ces temps comme étant indépendants et identiquement distribués.

$$\text{Var}(\delta t_{ij}) = \frac{v_{ij}}{N_{ij}}$$

où v_{ij} est la variance de la population des temps de trajets relevés sur la portion $[p_i p_j]$, que l'on peut facilement estimer à partir des données :

$$v_{ij} = \frac{1}{n-1} \sum_{k=1}^{N_{ij}} (\delta_{ij}^k - \delta_{ij})^2$$

Le problème se formule alors ainsi : ayant au préalable calculé les variances $\text{Var}(\delta t_{ij})$ sur chaque portion $[p_i p_j]$ de l'itinéraire (*i.e.* pour tout couple d'indices tels que $i < j$), trouver la subdivision \mathcal{S} qui minimise la quantité :

$$\text{Var}(\delta t(\mathcal{S})) = \sum_{i=1}^{|\mathcal{S}|-1} \frac{v_{s_i s_{i+1}}}{N_{s_i s_{i+1}}}$$

Prenons un exemple simple pour fixer les idées. Considérons un itinéraire composé de 6 points (dénotés p_0 à p_5), représenté sur la figure 5.13. Alors, la séquence $\mathcal{S} = (0, 1, 3, 5)$ constitue une subdivision, correspondant au découpage de l'itinéraire en 3 tronçons : $[p_0p_1]$, $[p_1p_3]$ et $[p_3p_5]$. Notons que cette subdivision n'est pas unique et qu'il aurait également été possible de choisir par exemple $\mathcal{S}' = (0, 1, 2, 4, 5)$ ou encore $\mathcal{S}'' = (0, 3, 5)...$

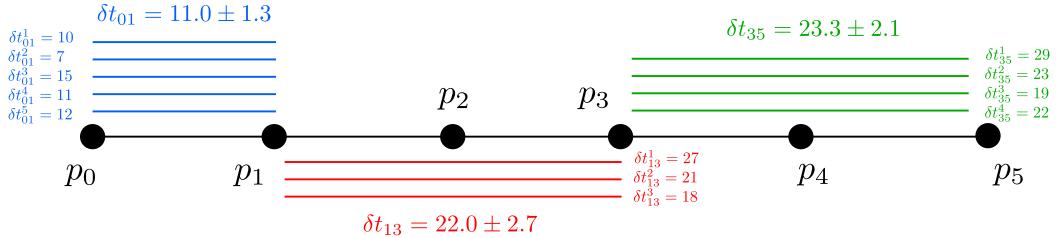


FIGURE 5.13 – Exemple d'une subdivision $(0, 1, 3, 5)$ sur un itinéraire. Toutes les valeurs numériques sont des temps de parcours exprimés en secondes.

Intéressons-nous au temps de parcours estimé à partir de cette subdivision \mathcal{S} . Chaque tronçon de \mathcal{S} est parcouru par un nombre potentiellement différent de trajectoires GPS (ici 5 traces pour le premier, 3 pour le deuxième et 4 pour le troisième), dont chacun des temps de parcours a été calculé au préalable. En moyennant ces temps, on obtient une estimation des temps de parcours par tronçon avec les écarts-types associés, et nous laissons le soin au lecteur de vérifier que les valeurs δt_{01} , δt_{13} et δt_{35} indiquées sur la figure 5.13 sont cohérentes avec les expressions données ci-dessus.

L'estimation (pour la subdivision choisie) du temps de parcours de l'itinéraire global et de son écart-type s'obtiennent alors directement :

$$\delta t = 11.0 + 22.0 + 23.3 = 56.3$$

$$\sigma_{\delta t} = \sqrt{\text{Var}(\delta t)} = \sqrt{1.3^2 + 2.7^2 + 2.1^2} = 3.7$$

Avec cette subdivision \mathcal{S} nous avons donc estimé le temps de parcours de l'itinéraire à 56.3 sec \pm 3.7 sec. Il faut alors calculer la variance obtenue avec d'autres choix de subdivision, et ne retenir que le résultat de variance minimale. En particulier, comme expliqué précédemment, le nombre maximal de passages de véhicules est en général atteint pour des tronçons de taille moyenne, relativement à la longueur totale de l'itinéraire et les variances d'estimation des moyennes de temps de trajet auront tendance à être meilleures sur ces fractions. La subdivision optimale est donc intuitivement ni trop fine, ni trop grossière.

D'un point de vue informatique : étant donnés trois matrices triangulaires supérieures strictes de nombres T_{ij} , V_{ij} et N_{ij} représentant respectivement le temps de parcours, sa variance associée et le nombre de traces sur la portion $[p_ip_j]$ ($i < j$) de l'itinéraire, trouver

une subdivision⁶ qui minimise la quantité $\text{Var}(\delta t)$ et retourner une estimation δt pour laquelle cette variance minimale est atteinte :

$$\text{Var}(\delta t(\mathcal{S})) = \sum_{i=1}^{|\mathcal{S}|-1} \frac{V_{s_i s_{i+1}}}{N_{s_i s_{i+1}}} \quad \delta t(\mathcal{S}) = \sum_{i=1}^{|\mathcal{S}|-1} T_{ij}$$

Notons qu'il est possible de simplifier le problème en pré-calculant en amont une table de données A_{ij} contenant directement le rapport des variances par portion divisées par les nombres de traces correspondants :

$$a_{ij} = \frac{V_{ij}}{N_{ij}}$$

5.6.2 Complexité de l'approche naïve

Une première solution naïve pourrait consister à parcourir l'ensemble des subdivisions possibles pour en choisir une qui mène à la variance minimale. Evaluons la complexité de cet algorithme, en prenant comme unité de taille le nombre de points de l'itinéraire et comme unité de coût le nombre d'additions flottantes et de comparaisons.

Nous devons d'abord calculer le nombre de subdivisions possibles d'un itinéraire de n points. Soit $k \in \{1, 2, \dots, n-1\}$. On montre aisément que le nombre de manières de subdiviser l'itinéraire en k tronçons correspond au nombre de manières de choisir $k-1$ éléments parmi $n-2$. D'autre part, calculer la variance sur une subdivision donnée (de k tronçons) nécessite $k-1$ additions, auxquelles il faut ajouter une comparaison (afin d'en extraire le minimum), d'où la complexité de l'algorithme de recherche exhaustive :

$$C(n) = \sum_{k=1}^{n-1} kC_{n-2}^{k-1} = \sum_{k=0}^{n-2} (k+1)C_{n-2}^k = \sum_{k=1}^{n-2} kC_{n-2}^k + \sum_{k=0}^{n-2} C_{n-2}^k$$

En écrivant 2^n sous la forme $(1+1)^n$ et à l'aide de la formule du binôme, on trouve que le second terme de l'expression ci-dessus vaut 2^{n-2} . De la même manière, on peut forcer l'apparition d'un coefficient binomial dans la première somme :

$$\begin{aligned} \sum_{k=1}^{n-2} kC_{n-2}^k &= \sum_{k=1}^{n-2} \frac{k(n-2)!}{k!(n-2-k)!} = \sum_{k=1}^{n-2} \frac{(n-2)!}{(k-1)!(n-3-(k-1))!} = (n-2) \sum_{k=1}^{n-2} C_{n-3}^{k-1} \\ &= (n-2) \sum_{k=0}^{n-3} C_{n-3}^k = (n-2)2^{n-3} \end{aligned}$$

6. Le nombre fini de subdivisions garantit l'existence d'un point minimal mais pas son unicité.

où la dernière égalité résulte d'une nouvelle utilisation de la formule du binôme. En recombinant les expressions simplifiées des deux sommes, on obtient :

$$C(n) = n2^{n-2} = \Theta(n2^n)$$

L'algorithme est donc doté d'une complexité exponentielle. En utilisant la remarque 1 de la section 3.7, on pourra montrer à titre d'exercice que cette complexité ne croît pas suffisamment vite pour appartenir à la classe des fonctions hyperexponentielles. A l'aide de la programmation dynamique, peut-on réduire cette complexité à une classe polynomiale ?

5.6.3 Résolution par la programmation dynamique

Nous allons chercher à décomposer le problème sur l'itinéraire complet, et à l'exprimer en fonction de sous-solutions calculées sur des fractions d'itinéraires.

Considérons une table $A = \{a_{ij}\}_{0 \leq i,j \leq n}$ pour un itinéraire comportant 5 tronçons (c'est-à-dire 6 sommets).

i/j	0	1	2	3	4	5
0	-	2.2	1.2	3.6	12.2	14.3
1	-	-	2.5	1.3	2.1	11.5
2	-	-	-	3.8	1.5	4.0
3	-	-	-	-	2.2	5.1
4	-	-	-	-	-	2.4
5	-	-	-	-	-	-

TABLE 5.5 – Tableau A des variances (en sec^2) de temps de trajet par tronçon

De manière similaire au problème d'optimisation des produits matriciels vu au paragraphe 5.4, on note m_{ij} la table des variances minimales que l'on peut obtenir sur un tronçon $[p_i p_j]$. Ici aussi nous ne chercherons à calculer que les termes de la partie triangulaire supérieure stricte, l'objectif du problème étant réduit à trouver la dernière cellule $m_{0,5}$. La tableau est initialisé par sa première diagonale supérieure, comme illustré ci-dessous.

Le calcul de $m_{0,2}$ est alors immédiat, puisqu'il s'agit du minimum entre $a_{0,2}$ (la variance sur le tronçon complet), et la somme de $m_{0,1}$ et $m_{1,2}$ (somme des variances sur les deux fractions d'un découpage potentiel). Le même procédé s'applique pour toute la diagonale.

A cette étape, nous avons par exemple la certitude qu'il ne sera pas possible de trouver une subdivision telle que la variance d'estimation sur le tronçon $[p_2 p_4]$ soit inférieure à 1.5 s^2 .

Le remplissage du tableau se poursuit alors diagonale par diagonale jusqu'au coin supérieur droit, à l'aide de la formule de récurrence permettant de calculer m_{ij} , sachant tous

i/j	0	1	2	3	4	5	i/j	0	1	2	3	4	5
0	-	2.2	-	-	-	-	0	-	2.2	1.2	-	-	-
1	-	-	2.5	-	-	-	1	-	-	2.5	1.3	-	-
2	-	-	-	3.8	-	-	2	-	-	-	3.8	1.5	-
3	-	-	-	-	2.2	-	3	-	-	-	-	2.2	4.6
4	-	-	-	-	-	2.4	4	-	-	-	-	-	2.4
5	-	-	-	-	-	-	5	-	-	-	-	-	-

TABLE 5.6 – Tableau m des variances minimales aux étapes 1 et 2

les termes $m_{i'j'}$ avec $i' < i$ et $j' < j$) :

$$m_{ij} = \min(a_{ij}, \min_{i < k < j} \{m_{ik} + m_{kj}\})$$

Remarquons que cette expression est très similaire à celle du paragraphe 5.4.2.

i/j	0	1	2	3	4	5
0	-	2.2	1.2	3.5	2.7	5.1
1	-	-	2.5	1.3	2.1	4.5
2	-	-	-	3.8	1.5	3.9
3	-	-	-	-	2.2	4.6
4	-	-	-	-	-	2.4
5	-	-	-	-	-	-

TABLE 5.7 – Tableau m des variances minimales après remplissage.

On trouve que la variance minimale atteignable est 5.1 s^2 . En particulier, on peut la comparer aux variances obtenues avec les deux subdivisions particulières : la subdivision grossière ne contenant qu'un intervalle (14.3 s^2) et la subdivision la plus fine contenant 5 intervalles (13.1 s^2), soit une réduction de la variance de plus de 60 % par rapport à une solution naïve.

Pour retrouver la subdivision correspondant à cette solution de variance minimale, on doit créer un tableau d'indexation b pour l'étape de backtracking. Ce tableau est rempli d'une manière similaire au tableau principal, avec les arguments des minimums :

$$b_{ij} = \operatorname{argmin}_{i < k < j} (a_{ij}, \{m_{ik} + m_{kj}\})$$

avec la convention que b_{ij} vaut -1 si le minimal a été atteint par a_{ij} .

Pour reconstituer la solution, on lit la valeur finale contenue dans la cellule $b_{0,5}$ ce qui nous donne un premier découpage de la somme opérée au niveau du point 2 : $m_{0,5} = m_{0,2} + m_{2,5}$. On procède alors de la même manière, en récupérant $b_{0,2} = -1$ (indiquant que le tronçon n'est alors plus découpé outre-mesure) et $b_{2,5} = 4$, ce qui décompose la somme en :

i/j	0	1	2	3	4	5
0	-	-1	-1	1	2	2
1	-	-	-1	-1	-1	4
2	-	-	-	-1	-1	4
3	-	-	-	-	-1	4
4	-	-	-	-	-	-1
5	-	-	-	-	-	-

TABLE 5.8 – Tableau de backtracking b permettant de reconstituer la solution.

$m_{0,5} = m_{0,2} + m_{2,4} + m_{4,5}$. Les cellules $b_{2,4}$ et $b_{4,5}$ contenant cette fois la valeur -1 , nous avons atteint le découpage optimal : $\mathcal{S}^* = (0, 2, 4, 5)$, ce qui donne une estimation : $\hat{\delta}t = \delta t_{02} + \delta t_{24} + \delta t_{45}$ avec la variance optimale associée :

$$\text{Var}(\delta t) = a_{02} + a_{24} + a_{45} = 5.1$$

soit un écart-type $\sigma_t = \sqrt{\text{Var}(\delta t)} = 2.3$ sec.

Un examen attentif du tableau des solutions révèlera que deux cheminements permettent d'atteindre la variance minimale. Ceux-ci correspondent toutefois à la même subdivision, et il n'y a donc qu'une seule solution physique au problème.

i/j	0	1	2	3	4	5
0	-	-1	(-1)	1	2	(2)/4
1	-	-	-1	-1	-1	4
2	-	-	-	-1	(-1)	(4)
3	-	-	-	-	-1	4
4	-	-	-	-	-	(-1)
5	-	-	-	-	-	-

i/j	0	1	2	3	4	5
0	-	-1	(-1)	1	(2)	2/(4)
1	-	-	-1	-1	-1	4
2	-	-	-	-1	(-1)	4
3	-	-	-	-	-1	4
4	-	-	-	-	-	(-1)
5	-	-	-	-	-	-

TABLE 5.9 – Reconstitution de la solution.

Cela met en évidence une faiblesse de l'algorithme de programmation dynamique sur ce problème, qui ne peut éviter les recouvrements partiels entre sous-problèmes. Peut-on modifier l'équation de propagation pour résoudre le problème ?

5.6.4 Complexité de la solution

Il y a $n - 2$ diagonales à calculer. Chaque diagonale $i \in \{2, 3, \dots, n - 1\}$ contient $n - i$ cellules. Pour chaque cellule, on doit trouver le minimum de $(i + 1)$ quantités, l'une étant directement lue dans la table des données, les i restantes étant évaluées à l'aide d'une unique addition, ce qui donne i additions et $i - 1$ comparaisons par cellule, d'où :

$$C(n) = \sum_{i=2}^{n-1} (2i-1)(n-i)$$

Cette équation est similaire à celle établie au paragraphe 5.4.3. Par une démarche analogue, on démontre que l'algorithme de recherche de la variance minimale du temps de parcours par programmation dynamique est doté d'une complexité polynomiale : $C(n) = \Theta(n^3)$

Conclusion

Dans ce chapitre, nous avons vu comment utiliser la programmation dynamique pour accélérer la résolution d'un problème donné. Il s'agit d'un outil puissant mais dangereux et l'on devra s'assurer que le principe d'optimalité des sous-problèmes est vérifié avant de l'employer, sous peine d'être réduit à une approche heuristique qui conduirait fatalement à une approximation (plus ou moins mauvaise) de la solution au problème posé.

Tout comme le paradigme *diviser pour régner*, elle permet lorsqu'elle est bien implantée de réduire d'au moins une classe la complexité de l'algorithme de résolution. Nous avons vu en particulier comment le calcul en temps exponentiel des termes de la suite de Fibonacci est devenu linéaire, ou bien encore comment l'ordonnancement optimal d'un produit matriciel s'est vu effectué en temps polynomial en lieu et place d'une complexité initiale également exponentielle.

Cette approche a également fait ses preuves sur des problèmes qui ont été démontrés NP-complets, c'est-à-dire pour lesquels il n'existe vraisemblablement pas de solution en temps polynomial⁷. En particulier, Held et Karp (1962) ont proposé de résoudre le fameux problème du voyageur de commerce à l'aide de la programmation dynamique. La complexité de l'algorithme, initialement factorielle, peut être réduite à un $\Theta(k^n)$, ce qui est optimal⁵ dans la classe des problèmes NP-complets (cf exercice 5.8 pour plus de détails).

5.7 Exercices

Exercice 5.3. Coefficient binomial *

On donne ci-dessous l'algorithme récursif calculant le coefficient binomial C_n^p :

```
COEFF_BINOM( $N \downarrow, P \downarrow$ ) :  $\llbracket \quad P \geq N \cup P = 0 \quad ? \quad \text{COEFF\_BINOM} \leftarrow 1; !$ 
 $|_i \quad \text{COEFF\_BINOM} \leftarrow \text{COEFF\_BINOM}(N-1, P) + \text{COEFF\_BINOM}(N-1, P-1) \quad \rrbracket$ 
```

Q1. A l'aide d'un argument simple, déterminer la complexité de cette fonction.

7. Sous l'hypothèse, à l'heure actuelle la plus probable, que $P \not\subseteq NP$

Q2. En utilisant le principe de la programmation dynamique, écrire un algorithme efficace du calcul du coefficient binomial.

Q3. Evaluer la complexité en temps de cet algorithme.

Exercice 5.4. *Produit matriciel* ******

Reprendre l'exemple développé au 5.4 et écrire le programme en ADL.

Exercice 5.5. *Partition d'un entier* ******

Soit $n \in \mathbb{N}$ un entier donné, dont on souhaite évaluer le nombre de partitions. On appelle partition de n une liste d'entiers (m_1, m_2, \dots, m_q) telle que : $m_i \geq 1$ et :

$$\sum_{i=1}^p m_i = n$$

De manière triviale, le nombre q d'éléments d'une partition est au plus égal à n .

Une partition est définie à une permutation près de ces éléments. Par exemple, l'entier 4 admet 5 partitions : $1 + 1 + 1 + 1$, $1 + 1 + 2$, $2 + 2$, $3 + 1$ et 4 . Dans ce problème, on s'intéresse au nombre de partitions $p(n)$ que l'on peut générer à partir d'un entier n .

On note $p(n, k)$, le nombre de partitions de l'entier n en k parties. Ainsi : $p(4, 4) = p(4, 3) = p(4, 1) = 1$ et $p(4, 2) = 2$. On observe immédiatement que la somme de ces termes donne la valeur recherchée :

$$p(n) = \sum_{k=1}^n p(n, k)$$

On peut démontrer que pour tout entier k compris entre 1 et n :

$$p(n, k) = p(n - 1, k - 1) + p(n - k, k)$$

Q1. Ecrire une fonction récursive permettant de calculer $p(n)$. Expliquer pourquoi cette fonction n'est pas optimale en terme de temps de calcul.

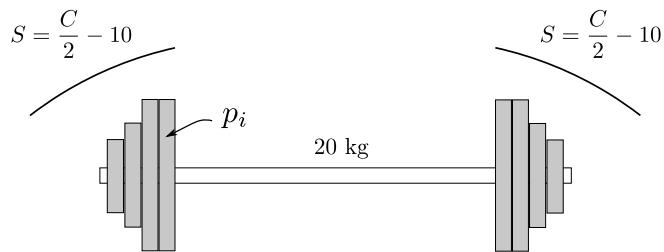
Q2. En appliquant le principe de la programmation dynamique, remplir à la main le tableau permettant de calculer le nombre de partitions de l'entier 8.

Q3. Ecrire une fonction efficace permettant de calculer $p(n)$.

Q4. Evaluer la complexité temporelle de l'algorithme.

Exercice 5.6. *Problème du powerlifter ****

On considère un système de disques de fonte : $P = (p_1, p_2, \dots, p_n)$ où $p_i \in \mathbb{N}$ représente le poids (en kg) du i -eme disque. Les disques peuvent être chargés sur les deux extrémités d'une barre olympique de 20 kg.



Un powerlifter souhaite réaliser une séance avec une charge de travail C (également exprimée en kg). L'objectif du problème est alors de constituer le poids C , en utilisant le moins de disques possibles. On suppose avoir à disposition une quantité infinie de chaque type de disque et que les valeurs p_i sont des entiers positifs (ce qui est toujours possible, moyennant un changement du système d'unités).

D'autre part, on ajoute la contrainte que la barre doit être équilibrée et parfaitement symétrique, permettant ainsi de traiter le problème sur une seule des deux extrémités de la barre, avec une charge de travail $S = (C - 20)/2$. Plus formellement, le problème s'écrit ainsi :

Pour $S \in \mathbb{N}$, trouver le vecteur $X = (x_1, x_2, \dots, x_n)$ (avec $x_i \in \mathbb{N}$) tel que :

$$X = \underset{\mathbb{N}^n}{\operatorname{argmin}} \sum_{i=1}^n x_i \quad \text{sous la contrainte : } \sum_{i=1}^n x_i v_i = S$$

Par exemple, en considérant le système de disques $(1, 5, 10, 20)$, pour la charge $C = 110$ kg on cherche la décomposition $2 * 20 + 5$, c'est-à-dire le vecteur $X = (0, 1, 0, 2)$.

On étudie dans un premier temps un algorithme glouton, consistant à retirer à chaque fois à la demi-charge de travail de la plus grande valeur possible. Par exemple pour une charge $S = 45$ kg, on retire 20 kg de S deux fois, ce qui donne $45 - 2 \times 20 = 5$, puis 5 kg (une fois), jusqu'à obtenir 0. Le vecteur solution est alors $(0, 1, 0, 2)$ pour un coût total de 3 disques (6 en comptant les deux extrémités).

Q1. Ecrire l'algorithme glouton.

Q2. A l'aide d'un exemple simple sur le système de disques $P = (1, 4, 6)$, montrer que l'algorithme glouton ne donne pas nécessairement une solution optimale.

On utilise alors la programmation dynamique pour résoudre le problème.

L'idée générale consiste à considérer un sous-problème $T(k, s)$ correspondant au nombre de disques nécessaires pour constituer la charge s en utilisant uniquement les k premiers types de disques du système : (p_1, p_2, \dots, p_k) .

Ainsi par exemple, dans le cas du système $(1, 4, 6)$, la solution du sous-problème $T(2, 6)$ vaut 3. En effet, il faut 3 disques pour composer la charge $s = 6$ en utilisant uniquement les deux premières disques $(1, 4)$ du système. En revanche, $T(3, 6)$ vaut 1, car il est à présent possible d'utiliser le troisième disque. La solution du problème est alors $T(n, s)$, c'est-à-dire le nombre minimal de disques pour former la demi-charge totale s en utilisant l'ensemble des disques du système ($k = n$).

On range alors les solutions dans un tableau à $n + 1$ lignes et à $s + 1$ colonnes. Prenons un exemple avec la charge $S = 8$:

k/s	0	1	2	3	4	5	6	7	8
{}	0	∞							
{1}	0								
{1, 4}	0								
{1, 4, 6}	0								

La première ligne du tableau est remplie avec le symbole ∞ , pour dénoter qu'une charge non-nulle ne peut être atteinte avec un système de disques vide. Lors de l'implémentation informatique, toute valeur suffisamment grande (typiquement plus grande strictement que S) pourra remplacer ce symbole.

La première colonne est quant à elle fixée à 0 (aucun disque n'est nécessaire pour former la charge $s = 0$).

Q3. Trouver une règle de passage permettant d'exprimer $T(k, s)$ en fonction de solutions pré-calculées $T(k', s')$, avec $s' \leq s$ et $k' \leq k$ (l'une des deux égalités au moins devra être stricte, sinon la fonction risque de ne pas terminer).

Indication : on pourra séparer le cas où la charge s est plus petite que la plus haute valeur v_k du sous-système de disques.

Q4. En déduire un ordre de parcours du tableau permettant de tirer pleinement profit du principe de la programmation dynamique.

Q5. Remplir le tableau ci-dessus et en déduire le nombre minimal de disques nécessaires pour former la charge $S = 8$.

Q6. On souhaite à présent avoir accès à la décomposition. Proposer une modification de la méthode précédente permettant, à l'aide d'une sauvegarde d'un marqueur judicieusement choisi, de reconstituer la décomposition. Illustrer la méthode en l'appliquant sur l'exemple de tableau ci-dessus. Comment nomme-t-on cette étape de la résolution ?

Q7. Ecrire un algorithme prenant en entrée un système de disques P (sous forme de tableau à une dimension) ainsi qu'une charge de travail à atteindre C (poids à répartir sur l'ensemble de la barre) et qui retourne la décomposition minimale de C dans le système P (le résultat sera donné sous la forme : $X = (x_1, x_2, \dots, x_n)$ où x_i désigne le nombre de fois où le disque p_i est présent sur la barre).

8) Evaluer la complexité en temps de cet algorithme.

Exercice 5.7. Problème du sac-à-dos ★★

On se donne un ensemble de n objets de valeurs (v_1, v_2, \dots, v_n) avec $v_i \in \mathbb{R}^{+*}$. Le poids de l'objet i est noté $p_i \in \mathbb{N}^*$. Notons que si les poids ne sont pas entiers, il est toujours possible de changer le système d'unités, de sorte à se ramener au cas étudié ici.

Le problème du sac-à-dos consiste à choisir un ensemble d'objets dont la somme des valeurs est maximale, avec la condition que le poids total des objets de cet ensemble n'excède pas la capacité du sac, que l'on dénote $C \in \mathbb{N}$.

Plus formellement, on cherche le vecteur $X = (x_1, x_2, \dots, x_n) \in \{0, 1\}^n$ tel que :

$$X = \underset{x_i}{\operatorname{argmax}} \sum_{i=1}^n x_i v_i \quad \text{sous la contrainte : } \sum_{i=1}^n x_i p_i \leq C$$

Q1. Ecrire un algorithme glouton, prenant en entrée deux tableaux V et P (dénotant respectivement les valeurs et les poids des objets) ainsi que la capacité C du sac. On prendra comme heuristique de choisir à chaque étape l'objet dont le rapport valeur sur poids est le plus élevé. Quelle est la complexité de cet algorithme ?

Q2. A l'aide d'un exemple simple, montrer que l'algorithme écrit à la question précédente ne retourne pas nécessairement une solution optimale.

On note $S(c, k)$ la valeur maximale que peut contenir un sac de capacité $c \in \mathbb{N}$ ($c \leq C$) en utilisant uniquement les k premiers objets.

Q3. En s'inspirant du problème du powerlifter (exercice 5.6) et en considérant le fait que la solution $S(c, k)$ est soit une la solution obtenue lorsque l'objet k n'est pas utilisé, (i.e. $S(c, k - 1)$), soit une solution utilisant l'objet k , auquel cas la capacité du sac est réduite du poids de k (i.e. $S(c - p_k, k)$), écrire une relation de récurrence entre $S(c, k)$ et $S(c', k')$, où $c' \leq c$ et $k' \leq k$. Notons que l'une des 2 inégalités au moins devra être stricte.

Q4. En utilisant la relation de récurrence établie à la question 3, et en appliquant le principe

de la programmation dynamique, écrire un algorithme permettant de résoudre le problème.

Q5. Calculer la complexité en temps de l'algorithme. On prendra comme unité de taille le nombre n d'objets à traiter.

Exercice 5.8. Problème du voyageur de commerce ***

On considère le problème du voyageur de commerce (*Traveling Salesman Problem*) :

Soit un ensemble de n points numérotés $V = \{1, 2, \dots, n\}$, qu'un représentant doit visiter en partant du point d'indice 1. Notons que le choix arbitraire de partir du sommet 1 n'enlève rien à la généralité de l'énoncé, à un réordonnancement près des indices des points. A l'issue de sa tournée, le voyageur retourne au sommet 1.

Pour tout couple (i, j) , on note $d_{ij} \in \mathbb{R}^+$ la distance séparant les points i et j . On suppose que $d_{ij} = d_{ji}$ et $d_{ii} = 0$. Par ailleurs, on notera que les réels d_{ij} ne découlent pas nécessairement d'une distance (au sens mathématique du terme) d sur l'espace V . En ce sens, on pourra éventuellement considérer (V, d) comme un espace semi-métrique.

L'objectif du problème, est de trouver un ordre de visite des points tel que la distance totale parcourue (en incluant le premier trajet depuis 1 et le retour vers 1) soit minimale.

Partie I - Première approche

Q1. Ecrire un algorithme permettant d'énumérer toutes les permutations possibles de l'ensemble $\{1, 2, \dots, n\}$. Le programme prendra en entrée un entier n et retournera une matrice M dont chaque ligne $(m_{ij})_{1 \leq j \leq n}$ donne les éléments ordonnés de la i -eme permutation possible. On pourra procéder par récurrence.

Q2. En déduire un algorithme permettant de résoudre le problème en effectuant une recherche exhaustive des chemins possibles et indiquer sa complexité en temps. On prendra comme unité de taille du problème le nombre de sommets et comme unité de coût le nombre d'additions.

Partie II - Proposition d'un algorithme efficace

Soit $i \in V$ un numéro de sommet, et $S \subset V$, tel que $i \notin S$. Notons $\delta(i, S)$ la distance minimale à parcourir en partant de i et en passant par tous les points de S puis en terminant au point 1. Ainsi par exemple, on a :

$$\delta(2, \{3, 4\}) = \min\{d_{23} + d_{34} + d_{41}, d_{24} + d_{43} + d_{31}\}$$

Q3. En remarquant que l'itinéraire solution de $\delta(i, S)$ comporte nécessairement un arc joignant i à j (avec $j \in S$), puis un itinéraire parcourant tous les sommets de $S \setminus \{j\}$ avant de revenir à 1, exprimer la relation de récurrence liant $\delta(i, S)$ et $\delta(j, S \setminus \{j\})$.

Q4. Exprimer, à l'aide de la fonction δ , la solution au problème du voyageur de commerce (*i.e.* trouver les arguments de la fonction δ qui permettent d'obtenir la valeur de la distance minimale à parcourir pour visiter tous les points de V en partant et en retournant en 1).

On remarque que $\delta(i, \{\}) = d_{i1}$ (il n'y a aucun point à visiter entre i et le retour à 1). En particulier : $\delta(1, \{\}) = 0$.

La relation de récurrence permet de lier la solution pour un ensemble S , avec une autre solution précédemment calculée (conformément au principe de la programmation dynamique) sur l'ensemble S privé d'un élément j .

Résoudre le problème nécessite donc de résoudre tous les sous-problèmes correspondant à des sous-ensembles de $S \setminus \{1\}$.

Prenons un exemple avec 4 points : $\{1, 2, 3, 4\}$.

i/S	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2, 3\}$	$\{3, 4\}$	$\{2, 4\}$	$\{2, 3, 4\}$
1								
2								
3								
4								

La table de mémoïsation doit stocker les $\delta(i, S)$ avec $i \in \{1, 2, 3, 4\}$ et $S \subseteq \{2, 3, 4\}$ (l'élément 1 est l'élément de départ, il n'est donc jamais inclus dans les décompositions).

Q5. Exprimer, en fonction de n , les dimensions de la table de mémoïsation.

Q6. En utilisant la réponse donnée à la question 3, définir un ordre de remplissage de la table de mémoïsation.

Q7. Résoudre le problème à la main, avec $n = 4$ sommets disposés sur un carré de côté unitaire. On pensera à enregistrer, en plus de la distance minimale, un marqueur permettant de reconstituer l'itinéraire à la fin de la procédure.

Q8. On suppose avoir à disposition un algorithme *powerset* prenant en entrée un ensemble fini d'indices de sommets $\Omega \subset \mathbb{N}$ et retournant l'ensemble des parties $\mathcal{P}(\Omega)$, c'est-à-dire l'ensemble des sous ensembles de Ω . L'ensemble Ω sera donné sous forme d'un vecteur, et la sortie $\mathcal{P}(\Omega)$ sera un tableau contenant autant de colonnes qu'il existe de parties dans Ω , et autant de lignes que la plus grande partie de Ω , soit $\text{card}(\Omega)$. Notons que toutes les colonnes (excepté la dernière) seront incomplètes, et qu'elles seront ordonnées par cardinal croissant.

A titre d'exemple, on donne ci-dessous la sortie de l'algorithme pour l'entrée $\Omega = \{1, 2, 3\}$:

$$\mathcal{P}(\Omega) = \begin{bmatrix} - & 1 & 2 & 3 & 1 & 2 & 1 & 1 \\ - & - & - & - & 2 & 3 & 3 & 2 \\ - & - & - & - & - & - & - & 3 \end{bmatrix}$$

On ne demande pas d'écrire l'algorithme **POWERSET**.

On suppose également avoir une fonction **GET** qui, à un indice de colonne dans la table résultat de **POWERSET** et à un indice de sommet j (inclu dans le sous-ensemble S de la colonne sélectionnée) retourne l'indice de la colonne correspondant au sous-ensemble $S \setminus \{j\}$. Pour être cohérent, ce module nécessite également de connaître la taille de l'ensemble partitionné.

Par exemple : **GET**(8, 2, 4) demande le numéro de colonne correspondant au sous-ensemble de la colonne 8 (*i.e.* $\{2, 3, 4\}$) auquel on a retiré l'élément 2, dans un tableau généré par l'appel de **POWERSET** sur l'ensemble des 4 premiers entiers. On recherche donc le numéro de colonne du sous-ensemble $\{2, 3, 4\} \setminus \{2\} = \{3, 4\}$. La réponse retournée sera **GET**(8, 2, 4) = 6.

En utilisant la fonction de partition de la question précédente, écrire l'algorithme permettant de calculer la longueur de l'itinéraire le plus court. On ne demande pas d'écrire l'algorithme de reconstitution par backtracking de cet itinéraire.

- 9) A partir de la réponse apportée à la question 3, calculer la complexité en temps de l'algorithme et la comparer à celle de l'algorithme naïf de la partie I.

Partie III - Optimalité

Dans la seconde partie de ce problème, on cherche à montrer qu'il n'existe pas d'algorithme de complexité polynomiale permettant de résoudre le problème du voyageur de commerce. On note π ce problème, et on considère sa version décisionnelle π_d :

Etant donnés un espace semi-métrique (V, d) avec $|V| = n$ et une constante $B \in \mathbb{R}^+$, existe-t-il un ordonnancement (w_1, w_2, \dots, w_n) d'éléments $w_i \in V$ distincts, tels que l'inégalité suivante soit respectée :

$$\sum_{i=1}^{n-1} d(w_i, w_{i+1}) + d(w_n, w_1) \leq B$$

On note \preceq_p la relation sur l'ensemble des problèmes définie par : $\pi_1 \preceq_p \pi_2$ si et seulement si (π_2 est résoluble en un temps polynomial $\Rightarrow \pi_1$ l'est aussi). En un sens, on peut dire que si $\pi_1 \preceq_p \pi_2$ alors, π_2 est au moins aussi difficile que π_1 .

Soit $G(V, E)$ un graphe, défini par un ensemble V de sommets, et un ensemble $E \subseteq V \times V$ d'arcs. On dit que G est hamiltonien, si et seulement si, il admet un cycle hamiltonien, *i.e.*, un chemin partant d'un sommet $v_0 \in V$ arbitraire, et passant une (et un seule) fois par chaque sommet de V avant de revenir en v_0 . On note η_d le problème de décision consistant à déterminer si un graphe est hamiltonien. On admettra⁸ le résultat suivant : il n'existe pas d'algorithme en temps polynomial permettant de résoudre le problème hamiltonien.

8. Ce résultat est correct sous l'hypothèse $P \neq NP$, et on le démontre à l'aide du théorème de Cook, et par chaînage de transformations polynomiales depuis le problème de satisfiabilité (*cf* Garey, 1991).

Formellement :

Lemme hamiltonien : Si on peut trouver un algorithme résolvant le problème de décision η_d (pour $|V| = n$), alors, il existe $\alpha > 1$ telle que sa complexité vérifie $C(n) = \Omega(\alpha^n)$.

10) Montrer que \preceq_p est un préordre sur l'ensemble des problèmes, c'est-à-dire une relation binaire, reflexive et transitive.

11) Montrer que : $\pi_d \preceq_p \pi$.

12) Montrer que le problème du voyageur de commerce π est au moins aussi difficile que le problème hamiltonien η_d : $\eta_d \preceq_p \pi_d$.

Indication : on pourra dans un premier temps montrer qu'il existe une fonction, calculable en temps polynomial, qui transforme toute instance de η_d en π_d .

13) En déduire que l'algorithme proposé en partie II est optimal en termes de classes de complexité (*cf* table 3.2).

Exercice 5.9. Modèle de Markov à états cachés ★★☆

Etant donnée une liste de variables aléatoires $X = (x_1, x_2, \dots, x_n) \in \mathcal{X}^n$ où \mathcal{X} désigne un ensemble fini (tout ce qui suit peut aisément être généralisé au cas infini, et même indénombrable) : on dit que la suite X est une chaîne de Markov homogène, si la loi de probabilité de chaque variable conditionnellement aux variables précédentes ne dépend que du résultat de la variable qui la précède immédiatement, et que cette loi de dépendance est invariable avec les états (*i.e.* stationnaire). On écrit :

$$\pi_t(x_k \mid x_1, x_2, \dots, x_{k-1}) = \pi_t(x_k \mid x_{k-1})$$

Dans le vocabulaire des modèles de Markov, les variables x_k sont appelés des états, et la propriété précédente s'énonce traditionnellement sous la forme : *le futur est indépendant du passé sachant le présent*. La fonction de deux variables π_t est appelée probabilité de transition et peut se mettre sous la forme d'une matrice $T \in \mathbb{R}^{s \times s}$, où $s = |\mathcal{X}|$ est le nombre d'états possibles et où chaque élément T_{ij} désigne la probabilité de passer de l'état i à l'état j . Imaginons par exemple un modèle de climat à trois états : {clair, nuageux, pluie}. On peut modéliser la série temporelle climatique par une matrice de transition :

$$T = \begin{pmatrix} 0.7 & 0.2 & 0.1 \\ 0.2 & 0.6 & 0.2 \\ 0.3 & 0.3 & 0.4 \end{pmatrix}$$

dans laquelle on remarque que la somme en ligne est unitaire⁹. Les valeurs plus élevées sur la diagonale indiquent qu'à chaque pas de temps, la transition la plus probable est la transition identité (ceci caractérise les systèmes dotés d'une certaine composante inertie, c'est-à-dire des systèmes ayant tendance à rester dans le même état, même en présence de légères perturbations extérieures). Notons qu'en pratique, les valeurs T_{ij} sont estimées à partir d'observations sur le phénomène à modéliser.

D'après la règle chaînage des probabilités conditionnelles, et en supposant le premier état équiprobable, la probabilité d'apparition d'une séquence X donnée est :

$$P(X) = \prod_{k=1}^{n-1} \pi_t(x_{k+1}|x_k) = \prod_{k=1}^{n-1} T_{x_i, x_{i+1}}$$

Supposons à présent que le modèle soit indirectement observé, c'est-à-dire que l'on a uniquement connaissance d'une série $Y = (y_1, y_2, \dots, y_n)$, où les observations $y_k \in \mathcal{Y}$ dépendent à chaque pas de temps de l'état courant du système x_k . Notons $r = |\mathcal{Y}|$ le nombre d'observations possibles. Tout comme pour la loi de transition, nous supposons que cette dépendance est connue, et modélisée par une probabilité conditionnelle $\pi_q(y_k|x_k)$, ou de manière équivalente par une matrice $Q \in \mathbb{R}^{s \times r}$.

Complétons notre modèle climatique, et supposons que l'observation est effectuée au moyen d'une grenouille placée dans un bocal, dans une pièce sans fenêtre. La grenouille dispose d'une échelle et on admet qu'elle a tendance à se hisser au sommet du bocal les jours de pluie. L'espace des observations contient alors deux éléments : $\mathcal{Y} = \{\text{bas}, \text{haut}\}$ avec une matrice des probabilités d'observation (on parle aussi de probabilités d'*emission*) :

$$Q = \begin{pmatrix} 0.8 & 0.2 \\ 0.5 & 0.4 \\ 0.1 & 0.9 \end{pmatrix}$$

où l'élément Q_{ij} indique la probabilité de faire une observation j du système sachant qu'il est dans l'état i . A nouveau, la somme en ligne vaut 1. Par exemple, lorsque le temps est pluvieux (état indexé par $i = 3$) la probabilité que la grenouille soit en haut de l'échelle (observation indexée par $j = 2$) vaut 0.9. Notons que l'espace des observations peut être égal à l'espace des états. Par exemple, on peut imaginer un modèle de Markov caché pour les frappes dactylographiques. Au premier ordre, on peut supposer que la probabilité d'occurrence d'une lettre dépend uniquement de la lettre précédente. D'autre part, si on admet que la probabilité de faire une faute de frappe est inversement proportionnelle à l'espace-ment entre les touches, on obtient un modèle de Markov à états cachés. L'ensemble des états et des observations sont alors égaux à l'ensemble des symboles du clavier.

Moyennant la connaissance des matrices T et Q on peut calculer la probabilité d'apparition d'une séquence d'états et d'observations :

9. Une telle matrice est appelée matrice stochastique

$$P(X, Y) = \prod_{k=1}^{n-1} \pi_t(x_{k+1} \mid x_k) \pi_q(y_k \mid x_k) = \prod_{k=1}^{n-1} T_{x_k, x_{k+1}} Q_{x_k, y_k}$$

Etant donnée une séquence connue d'observations Y on peut alors déterminer la séquence d'états la plus probable. Par exemple, dans le modèle de la grenouille, on peut rechercher à savoir le temps qu'il a fait chaque jour sans sortir de la pièce, ou encore d'un point de vue plus applicatif, on peut vouloir détecter les fautes de frappes sur le clavier d'un téléphone mobile. La solution s'obtient en maximisant la quantité précédente (que l'on appelle la *vraisemblance des données*), ce qui donne :

$$\hat{X} = \underset{X \in \mathcal{X}^n}{\operatorname{argmax}} \prod_{k=1}^{n-1} T_{x_k, x_{k+1}} Q_{x_k, y_k}$$

où on rappelle que y_k est connu à chaque pas de temps k .

Schématiquement, le problème se représente de la manière suivante :

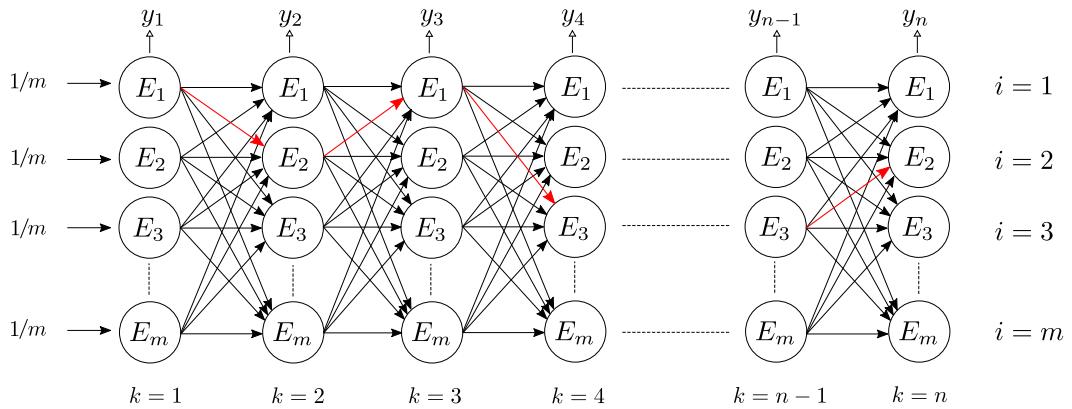


FIGURE 5.14 – Modèle de Markov homogène à états cachés discrets, sous l'hypothèse d'équiprobabilité des états initiaux, avec une trajectoire optimale (en rouge).

Dans les questions qui suivent, les complexités algorithmiques seront exprimées en nombre de multiplications.

Q1. Ecrire un algorithme de recherche de la solution optimale par force brute, c'est-à-dire en parcourant intégralement l'espace \mathcal{X}^n des séquences d'états. Quelle est la complexité de cet algorithme ?

Q2. En observant que si on est capable de trouver la solution jusqu'à un pas de temps k donné, alors la solution jusqu'à $k + 1$ s'obtient facilement à l'aide de quelques multiplications et quelques comparaisons sur les solutions jusqu'à k , écrire l'équation de récurrence permettant de résoudre le problème et déterminer un ordre de résolution.

Q3. Ecrire l'algorithme permettant de trouver la séquence d'états optimale \hat{X} .

Note : cette méthode de résolution porte le nom d'algorithme de *Viterbi*.

Q4. Calculer la complexité de cet algorithme.

Q5. En pratique, la multiplication en machine d'une longue série de flottants inférieurs à 1 peut conduire à des erreurs numériques dont la propagation au cours du processus de résolution fausse complètement la sortie de l'algorithme. Pour résoudre ce problème, on travaille en base logarithmique, en cherchant à minimiser l'opposé du log de la vraisemblance (le log étant une fonction strictement croissante, et négative sur $]0, 1]$), à condition que les probabilités des tables T et Q soient toutes strictement positives. Cette transformation a pour intérêt de transformer le produit en une somme, plus simple à manipuler, soit en notant LT et LQ les transformations logarithmiques, respectivement des tables T et Q :

$$\hat{X} = \underset{X \in \mathcal{X}^n}{\operatorname{argmin}} \left\{ - \left[\sum_{k=1}^{n-1} LT_{x_k, x_{k+1}} + LQ_{x_k, y_k} \right] \right\}$$

Modifier l'algorithme en conséquence.

Annexe A

Syntaxe ADL

Opérateur	Symbol	Type	Entrée(s)	Sortie	Priorité
Exponentiation	\wedge	binaire	numériques	numérique	1
Multiplication	*	binaire	numériques	numérique	2
Division	/	binaire	numériques	numérique	2
Division entière	\div	binaire	numériques	numérique	2
Concaténation	&	binaire	alphanum.	alphanum.	2
Addition	+	binaire	numériques	numérique	3
Soustraction	-	binaire	numériques	numérique	3
Moins monadique	-	unaire	numérique	numérique	3
Supérieur	>	binaire	numériques	booléenne	4
Supérieur ou égal	\geq	binaire	numériques	booléenne	4
Egal	=	binaire	-	-	4
Different	\neq	binaire	-	-	4
Inférieur	<	binaire	numériques	booléenne	4
Inférieur ou égal	\leq	binaire	numériques	booléenne	4
Négation	\bar{a}	unaire	booléenne	booléenne	5
Conjonction	\cap	binaire	booléennes	booléenne	6
Nand	\boxtimes	binaire	booléennes	booléenne	6
Disjonction	\cup	binaire	booléennes	booléenne	7
Nor	\boxdot	binaire	booléennes	booléenne	7
Xor	\oplus	binaire	booléennes	booléenne	7

TABLE A.1 – Liste des opérateurs à une ou deux opérandes

Nom	Symbol
Identificateur	$A, I, V, x, \text{VAL} \dots$
Identificateur dimensionné	$V_i, M_{ij}, \text{TAB}_{ijk}, L_{i+j,k*k} \dots$
Affectation	$x \leftarrow 2$
Référencement	$x \rightarrow y$
Opérateurs arithmétiques	$\wedge, *, /, \div, +, -, \%$
Opérateurs booléens	$\cap, \cup, \boxtimes, \boxplus$
Opérateurs de comparaison	$\leq, <, \geq, >, =, \neq$
Négation booléenne	\bar{a}
Test conditionnel	cond ? action1 action2 ;
Test simple	cond ? action ;
Test multiple	val $\not\exists \dots \not\forall$
Boucle à compteur	$\left\{ \begin{array}{l} \text{actions} \\ i:\text{ini} \\ \text{fin} \end{array} \right\}_i$
Boucle à compteur à pas unitaire	$\left\{ \begin{array}{l} \text{actions} \\ i:\text{ini} \\ \text{fin} \end{array} \right\}_i$
Boucle <i>tant que</i>	$\left\{ \begin{array}{l} \text{actions} \\ / \text{cond} \end{array} \right\}$
Boucle <i>jusqu'à ce que</i>	$\left\{ \begin{array}{l} \text{actions} \\ / \text{cond} \end{array} \right\}$
Boucle infinie	$\left\{ \begin{array}{l} \text{actions} \end{array} \right\}$
Sortie de boucle	!, !n
Passage à l'itération suivante	$\uparrow, \uparrow n$
Retour à l'itération courante	$\downarrow, \downarrow n$
Sortie du module	!
Module	MODULE(args) : $\llbracket \text{actions} \rrbracket$
Passage par valeur	MODULE(..., $x \downarrow$, ...)
Passage par référence	MODULE(..., $x \uparrow$, ...)
Sortie de module	MODULE(..., $x \uparrow$, ...)
Sortie de fonction	MODULE $\leftarrow x$

TABLE A.2 – Liste des symboles autorisés par ADL

Nom	Arguments	Sortie	Description
MIN	$a, b \in \mathbb{R}$	$c \in \mathbb{R}$	retourne le minimum de a et b
MAX	$a, b \in \mathbb{R}$	$c \in \mathbb{R}$	retourne le maximum de a et b
SQRT	$a \in \mathbb{R}^+$	$b \in \mathbb{R}$	retourne \sqrt{a}
FLOOR	$a \in \mathbb{R}$	$b \in \mathbb{Z}$	retourne la partie entière $E(a)$
ABS	$a \in \mathbb{R}$	$b \in \mathbb{R}^+$	retourne ma valeur absolue $ a $
EXP	$x \in \mathbb{R}$	$y \in \mathbb{R}^+$	retourne l'exponentielle e^x
LOG	$x \in \mathbb{R}^{+*}$	$y \in \mathbb{R}$	retourne le logarithme naturel $\ln x$
SIN	$x \in \mathbb{R}$	$y \in [-1, 1]$	retourne $\sin x$
COS	$x \in \mathbb{R}$	$y \in [-1, 1]$	retourne $\cos x$
TAN	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\tan x$
ASIN	$x \in [-1, 1]$	$y \in [0, 2\pi]$	retourne $\arcsin x$
ACOS	$x \in [-1, 1]$	$y \in [0, 2\pi]$	retourne $\arccos x$
ATAN	$x \in \mathbb{R}$	$y \in [0, 2\pi]$	retourne $\arctan x$
SINH	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\sinh x$
COSH	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\cosh x$
TANH	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\tanh x$
ASINH	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\text{arsinh } x$
ACOSH	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\text{arccosh } x$
ATANH	$x \in \mathbb{R}$	$y \in \mathbb{R}$	retourne $\text{arctanh } x$
MOD	$a, b \in \mathbb{Z}$	$r \in \{0, \dots, b - 1\}$	retourne $r \mid a = bq + r$, $q \in \mathbb{Z}$
RAND	-	$y \in [0, 1]$	retourne $y \sim \mathcal{U}([0, 1])$
SIZE	Tenseur T	Vecteur V	retourne les tailles de T
SIZE	Vecteur V	$n \in \mathbb{N}$	retourne la taille de V

TABLE A.3 – Liste des fonctions prédéfinies en ADL

Annexe B

Rappels de suites et séries

Dans cette annexe, $(u_n)_{n \in \mathbb{N}}$ désigne une suite de réels, à laquelle on associe la série de terme général S_n :

$$S_n = \sum_{k=0}^n u_k \quad \forall n \in \mathbb{N}$$

Enfin, on note S_{ij} la somme partielle entre les éléments i et j :

$$S_{ij} = S_j - S_{i-1} = \sum_{k=i}^j u_k \quad \forall n \in \mathbb{N}$$

B.1 Suites et séries arithmétiques

Une suite arithmétique est définie par la données de u_0 et par la relation de récurrence :

$$u_{n+1} = u_n + b \quad \forall n \in \mathbb{N}$$

Le réel b est appelé la raison de la suite arithmétique.

Le terme général de la série S_{ij} associée est :

$$S_{ij} = (j - i + 1) \frac{u_i + u_j}{2} \quad (i \leq j)$$

En particulier : $S_n = \frac{1}{2}(n + 1)(u_0 + u_n)$

On a également : $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$

B.2 Suites et séries géométriques

Une suite géométrique est définie par la données de u_0 et par la relation de récurrence :

$$u_{n+1} = qu_n \quad \forall n \in \mathbb{N}$$

Le réel $q \neq 1$ est appelé la raison de la suite géométrique.

Le terme général de la série S_{ij} associée est :

$$S_{ij} = u_0 \frac{q^i - q^j + 1}{1 - q} \quad (i \leq j)$$

En particulier : $S_n = u_0 \frac{1 - q^{n+1}}{1 - q}$

Si $|q| < 1$ la série converge et on a : $S_n \xrightarrow[n \rightarrow +\infty]{} \frac{1}{1-q}$

B.3 Suites arithmético-géométriques

Une suite arithmético-géométrique est définie par la données de u_0 et par la relation de récurrence :

$$u_{n+1} = qu_n + b \quad \forall n \in \mathbb{N}$$

Avec : $q \neq 1$ et $b \neq 0$ (sinon on se ramène à un des cas traités dans les deux sections précédentes).

On obtient le terme de rang n de la suite en posant : $r = \frac{b}{1-q}$:

$$u_n = q^n(u_0 - r) + r \quad \forall n \in \mathbb{N}$$

On obtient alors une expression simple de la série associée :

$$S_n = (u_0 - r) \frac{1 - q^{n+1}}{1 - q} + nr$$

Et donc, la somme partielle vaut :

$$S_{ij} = (u_0 - r) \frac{q^i - q^{j+1}}{1 - q} + (j - i + 1)r \quad (i \leq j)$$

B.4 Somme des puissances

On s'intéresse dans cette partie au terme :

$$S_k(n) = \sum_{m=1}^n m^k \quad k \in \mathbb{N}$$

De manière triviale, S_0 est une somme de 1 donc :

$$S_0(n) = n$$

On reconnaît en S_1 la série associée à une suite arithmétique de raison 1 et donc :

$$S_1(n) = \frac{n(n + 1)}{2}$$

Pour le terme général, on peut exprimer S_k sous forme d'une relation de récurrence forte, à l'aide de tous les termes précédents $S_{k-1}, S_{k-2}, \dots, S_1$ et S_0 . On pourra trouver une démonstration détaillée dans (M. Volle, 2014) :

$$(k + 1)S_k = (n + 1)^{k+1} - 1 - \sum_{p=2}^{k+1} C_{k+1}^p S_{k+1-p}$$

où C_n^p désigne le coefficient binomial p parmi n .

En analyse de complexité, la valeur exacte de S_n ne nous intéresse pas trop. En revanche, grâce à la relation de récurrence, on montre facilement que :

$$S_k(n) = \Theta(n^{k+1})$$

Preuve : soit P_k la proposition : $S_i(n) = \Theta(n^{i+1}) \quad \forall i \leq k$ (récurrence forte). On a montré précédemment que $S_0(n) = n = \Theta(n)$, donc la proposition est vérifiée au rang $k = 0$. Supposons que P_k est vérifiée pour un certain rang $k \in \mathbb{N}$. Alors d'après la relation de récurrence sur S_k et sous l'hypothèse P_k , on peut écrire :

$$S_k(n) = \Theta((n+1)^{k+1}) + \sum_{p=2}^{k+1} \Theta(n^{k+2-p}) = \Theta((n+1)^{k+1}) + \sum_{p=1}^k \Theta(n^p) = \Theta(n^{k+1})$$

On peut obtenir le même résultat directement par comparaison série-intégrale. En effet, la fonction $f_k : x \rightarrow x^k$ étant monotone croissante sur \mathbb{R}^+ :

$$\int_0^n f_k(x) dx \leq \sum_{i=1}^n f_k(i) \leq \int_1^{n+1} f_k(x) dx$$

D'où par encadrement sur les équivalents : $S_k(n) \sim \frac{n^{k+1}}{k+1}$ et par suite : $S_k(n) = \Theta(n^{k+1})$.

B.5 Suites récurrentes d'ordre multiple

On appelle suite récurrente linéaire d'ordre p toute suite $(u_n)_{n \in \mathbb{N}}$ définie par p valeurs initiales et une relation de récurrence de la forme :

$$u_{n+p} = a_{p-1}u_{n+p-1} + a_{p-2}u_{n+p-2} + \dots + a_1u_{n+1} + a_0u_n = \sum_{k=0}^{p-1} a_k u_{n+k}$$

Par exemple, une suite récurrente linéaire d'ordre 2 s'exprime sous la forme :

$$u_0 = a \quad u_1 = b \quad u_{n+2} = a_1u_{n+1} + a_2u_n$$

Résolution directe

Définition :

On appelle polynôme caractéristique de la suite $(u_n)_{n \in \mathbb{N}}$, le polynôme défini par :

$$P(X) = X^p - a_0 - a_1X - a_2X^2 - \dots - a_{p-1}X^{p-1}$$

Il s'agit d'un polynôme de degré p .

Théorème

Si P est un polynôme scindé, c'est-à-dire qu'on peut l'écrire sous la forme : $P(X) = \prod_{i=1}^n (X - r_i)$, avec (r_1, r_2, \dots, r_n) les racines de P de degrés de multiplicité $(\alpha_1, \alpha_2, \dots, \alpha_n)$, alors, toute suite sous la forme d'une somme de suites dont le terme général est :

$$R(n) = Q_i(n)r_i^n$$

avec : Q_i un polynôme de degré strictement inférieur à α_i , est solution de la récurrence.

Exemple : suite récurrente linéaire d'ordre 2

Si le polynôme a deux racines distinctes r_1 et r_2 , la solution s'exprime sous la forme (si $\lambda_1 \neq 0$ et $\lambda_2 \neq 0$) :

$$u_n = \lambda_1 r_1^n + \lambda_2 r_2^n = \Theta(\max(r_1, r_2)^n)$$

Si le polynôme a une racine double r alors la solution s'exprime sous la forme (si $\lambda_1 \neq 0$) :

$$u_n = (\lambda_1 n + \lambda_2) r^n = \Theta(nr^n)$$

dans les deux cas λ_1 et λ_2 sont déterminés grâce aux conditions initiales.

Résolution matricielle

En exprimant une nouvelle suite vectorielle $(\mathbf{V}_n)_{n \in \mathbb{N}}$ sous la forme :

$$\mathbf{V}_n = \begin{pmatrix} u_{n+p-1} \\ u_{n+p} \\ \vdots \\ u_n \end{pmatrix}$$

On peut alors aisément trouver une matrice $\mathbf{A} \in \mathbb{R}^{p \times p}$ telle que la relation de récurrence devienne une relation linéaire d'ordre 1 : $\mathbf{V}_{n+1} = \mathbf{AV}_n$

En posant \mathbf{V}_0 le vecteur contenant les conditions initiales de la suite, on obtient une expression explicite du terme de rang n de la suite :

$$\mathbf{V}_n = \mathbf{A}^n \mathbf{V}_0$$

Il suffit alors de récupérer dans \mathbf{V}_n la composante recherchée u_n .

D'un point de vue algorithmique, cette méthode permet d'accélérer notamment le calcul d'une suite linéaire d'ordre multiple. D'un point de vue mathématique, elle permet de calculer la complexité d'une fonction définie par une récurrence linéaire multiple (c'est le cas de la suite de Fibonacci dans sa version récursive par exemple). Il reste encore à être capable d'évaluer \mathbf{A}^n .

Si \mathbf{A} est diagonalisable, on peut réécrire \mathbf{V}_n sous la forme :

$$\mathbf{V}_n = \mathbf{P}\mathbf{D}^n\mathbf{P}^{-1}\mathbf{V}_0$$

où \mathbf{P} est la matrice de passage et \mathbf{D} est une matrice diagonale.

Notons que pour l'obtention de la complexité, le calcul de la matrice de passage n'est pas nécessaire. En supposant que les valeurs propres de \mathbf{A} sont indiquées dans l'ordre décroissant : $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_p$, on obtient immédiatement l'ordre de grandeur :

$$\mathbf{V}_n = \Theta(\lambda_1^n)$$

B.6 Série harmonique

En analyse de complexité, il arrive fréquemment d'avoir à exprimer le terme général de la série harmonique, par exemple pour le calcul de la complexité moyenne du tri par sélection :

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \sum_{k=1}^n \frac{1}{k}$$

Une solution simple pour obtenir un ordre de grandeur explicite de H_n est d'encadrer la somme à l'aide de deux intégrales dont on sait évaluer l'expression :

$$\int_i^{i+1} \frac{dx}{x} \leq \frac{1}{k} \leq \int_{i-1}^i \frac{dx}{x} \Rightarrow \int_2^{n+1} \frac{dx}{x} \leq H_n \leq \int_1^n \frac{dx}{x}$$

$$\ln(n+1) - \ln 2 \leq H_n \leq \ln n$$

Par encadrement, on obtient alors : $H_n \underset{n \rightarrow +\infty}{\sim} \ln n$ et donc :

$$H_n = \Theta(\log n)$$

B.7 Formule de Stirling

La formule de Stirling est une équivalence célèbre et très pratique, décrivant le comportement asymptotique de la factorielle quand n devient infiniment grand. On pourra trouver une démonstration de ce résultat sous forme d'exercice dans (Gourdon, 2018).

$$n! \underset{n \rightarrow +\infty}{\sim} n^n e^{-n} \sqrt{2\pi n}$$

Exemple d'application : ordre de complexité de $\log n!$

Aucune des deux fonctions ne s'approche de 1 quand n tend vers l'infini, on peut donc les composer à gauche par le logarithme naturel dans la relation d'équivalence :

$$\ln n! \underset{n \rightarrow +\infty}{\sim} \ln(n^n e^{-n} \sqrt{2\pi n})$$

$$\ln n! \underset{n \rightarrow +\infty}{\sim} n \ln(n) - n + \ln(\sqrt{2\pi n}) \underset{n \rightarrow +\infty}{\sim} n \ln(n)$$

On obtient alors immédiatement : $\ln n! = \Theta(n \ln(n))$ et par suite :

$$\log_b n! = \frac{\ln n!}{\ln b} = \Theta(n \ln(n)) \text{ et donc } \log n! \in \Theta(n \log(n))$$

L'ordre de complexité est donc quasi-linéaire.

Une formulation alternative de l'approximation de Stirling :

$$\ln n! = n \ln n - n + \mathcal{O}(\ln n)$$

On obtient également un corollaire sur les séries de terme général logarithmique :

$$\sum_{i=1}^n \log i = \Theta(n \log n)$$

B.8 Nombres de Catalan

On part de la relation de récurrence établie dans la section 5.4 traitant de l'optimisation des produits matriciels par la programmation dynamique :

$$c_0 = 1 \quad \text{et pour } n \geq 1 \quad c_n = \sum_{i=1}^{n-1} c_i c_{n-i}$$

Dans un mécanisme similaire à celui évoqué au sujet de la transformation de Fourier, le produit de Cauchy (*i.e.* la convolution discrète) dans l'espace des suites et le produit classique de l'espace des fonctions génératrices, sont analogues. On en déduit donc immédiatement une équation sur la série génératrice associée à la suite $(c_n)_{n \in \mathbb{N}}$:

$$f(x) = \sum_{n=0}^{\infty} c_n x^n = 1 + xf(x)^2$$

où les termes additionnels proviennent du décalage entre le produit de Cauchy formel (défini au rang $n - 1$) et la définition de la suite par récurrence (au rang n).

La résolution de l'équation fonctionnelle nous donne immédiatement l'expression analytique de la série génératrice associée à la suite :

$$f(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

dont on peut montrer que le rayon de convergence vaut $1/4$. Notons que f est prolongée par continuité en 0 avec $f(0) = 1$, ce qui justifie *a posteriori* le choix d'une solution du trinôme par rapport à l'autre.

En théorie, la série génératrice f permet de retrouver les termes de la suite, à l'aide de la relation inverse $c_n = f^{(n)}(0)$. Ici, nous allons plutôt procéder par identification directe à partir du développement en série entière de $\sqrt{1 - 4x}$.

Rappel :

$$(1 + x)^\alpha = 1 + \sum_{n=1}^{\infty} \frac{\alpha(\alpha - 1)\dots(\alpha - n + 1)}{n!} \cdot x^n$$

Avec le changement de variable $x \rightarrow -4x$ et pour $\alpha = \frac{1}{2}$, on obtient :

$$f(x) = \frac{1}{2} + \sum_{k=0}^{\infty} \frac{4^n}{2^n n!} [1 \times 3 \times 5 \times \dots \times (2n - 3)] x^n$$

En identifiant terme à terme, on retrouve la formulation explicite des nombres de Catalan :

$$c_n = \frac{2^n}{(n+1)!} \frac{(2n)!}{2 \times 4 \times 6 \dots \times 2n} = \frac{1}{n+1} \frac{(2n)!}{n!n!} = \frac{1}{n+1} \binom{2n}{n}$$

Index

- Ackermann, 78
Affectation, 16, 24
Agrawal-Kayal-Saxena, 105
Al-Khwârizmî, 7
Algorithm de tri, 156, 161
Algorithm glouton, 210
Arbre, 171
Arbre binaire de recherche, 177
Arbre de décision, 184, 194, 195
Arbre de Hilbert, 196
Arbre peigne droit, 174, 177, 198
- Babbage, Charles, 12
Bachmann, Paul, 101
Backtracking, 221, 222
Bellman, 206, 208
Booléen, 14
Bouchon, Basile, 12
Boucle, 25
Boucle à compteur, 27
Boucle combinée, 31
Boucle infinie, 31
Boucle jusqu'à ce que, 30
Boucle tant que, 29
Boyer-Moore, 51
Brainware, 8
- Calcul de pi, 61, 145
Catalan, 215, 257
Chaine de caractères, 49
Chaitin, Gregory, 69
Coût d'exécution, 96
Codes d'erreur, 40
Coefficient binomial, 87, 231, 235, 253
Commentaires, 37
Convolution matricielle, 51, 125
Copie, 167
Coppersmith-Winograd, 122
Courbes de niveau, 54
- Débranchement de module, 40
- Décidabilité, 68
Déclaration, 15
Décomposition décimale, 87
De Morgan (lois), 22
Dichotomie, 62, 77, 116, 177
Dijkstra, Edger, 34, 67, 100, 175, 213, 220
Disjonction, 25
Disjonctions multiples, 26
Diviser pour régner, 47, 116
Douglas-Peucker, 143, 159
- Entrées/sorties de module, 38
Euclide (algorithme), 7, 73, 85
Evaluation d'un polynôme, 105
Exponentiation rapide, 47, 91, 92, 95, 96, 105, 116, 117, 120, 134, 204
Expressions algébriques, 18
- Factorielle, 27, 28, 45, 46, 82, 86, 95, 96, 104, 110, 111
Fibonacci (suite), 29, 47, 76, 90–92, 95, 110, 113, 114, 117, 133, 178, 204, 205, 222, 235, 256
Fibonacci (tas), 181, 226
File, 62, 170, 199
Floyd-Warshall, 219, 226
Fonctions, 23, 44
Fourier, 122, 147
Frege, Gottlob, 7
- Gödel, Kurt, 68, 69
Go to, 34
- Héron, 29
Hachage, 185
Huffman, 181
Hyperexponentiel, 103, 232
- Identificateur dimensionné, 15
Identificateur scalaire, 14
Identificateurs, 13

- Indécidabilité de la terminaison, 69
- Indexation spatiale, 190
- Indice de Jaccard, 50
- Invariant de boucle, 84
- Jeu de la vie, 63
- Karnaugh, 23
- Kd-Tree, 195
- Landau, Edmund, 100
- Leibniz, Gottfried Wilhelm, 7
- Liste chaînée, 167
- Lucas, Edouard, 131
- Médiane, 126
- Médiane des médianes, 126, 159
- Mémoïsation, 204
- Méthode, 169
- Map inference, 226
- Marching squares, 56
- Markov, 213, 243
- Master Theorem, 112, 119, 131
- Matrice creuse, 46
- McCarthy (fonction), 83
- Modèle Numérique de Terrain, 54
- Modules, 38
- Modulo, 24
- Monte-Carlo, 61, 146, 152
- Morris (fonction), 83
- Multiplication égyptienne, 47
- Multiplication de grands entiers, 63
- Multiplication de Karatsuba, 116, 146
- Multiplication matricielle, 38, 40, 91, 92, 104, 116, 222
- Multiplication matricielle par Strassen, 121, 135
- Multiplication par FFT, 147
- Nombre d'or, 114
- Normalisation d'un vecteur, 45
- Notation en Polonaise Inverse, 183
- Opérateurs, 18
- Paramètres actuels, 41
- Paramètres formels, 41
- Parcours en largeur, 174, 220
- Parcours en profondeur, 174
- Parité d'un nombre, 49
- Passage à l'itération suivante, 36
- Passage par référence, 43
- Passage par valeur, 42
- Percolation, 179
- Pile, 170
- Plus proches voisins, 138
- Pointeur, 165
- Problème de l'arrêt, 69
- Problème du voyageur de commerce, 235, 240
- Procédures, 44
- Quadtree, 194
- R-Tree, 196
- Récursivité, 74
- Récursivité croisée, 49
- Réitération, 33
- Recherche dans une liste, 48, 62, 77, 107
- Rice (théorème), 71
- Salamin et Brent (algorithme), 146
- Sortie de boucle, 32
- Sortie de module, 40
- Squelette, 34
- Stirling (formule), 103, 105, 162, 215, 257
- Syracuse, 30, 31
- Tableau, 14
- Takeuchi (fonction), 83
- Tas binaire, 178
- Tas binomial, 181
- Temps de trajet, 226
- Test conditionnel, 25
- Test de primalité, 105
- Test de primalité , 130
- Toom-Cook (algorithme), 147
- Tours de Hanoï, 79, 112, 131, 170, 198
- Traces GPS, 226
- Transposition matricielle, 43
- Tri à bulles, 135, 157, 160, 179, 181
- Tri à peigne, 160
- Tri en place, 156
- Tri fusion, 47, 82, 88, 116
- Tri fusion , 117
- Tri introspectif, 160
- Tri par comparaison, 161
- Tri par dénombrement, 162
- Tri par insertion, 134
- Tri par paquets, 163
- Tri par sélection, 29
- Tri par tas, 178
- Tri rapide, 157

- Tri stable, 157
- Turing, Alan, 7, 68, 69
- Type abstrait de données, 168
- Types de données, 13
- Unité de taille, 95
- Variant de boucle, 73
- Viterbi, 213, 246
- Wirth, Niklaus, 152

Bibliographie

Samuel Balmand, Cours d'Informatique à l'Ecole Nationale des Sciences Géographiques, 2015

Patrick Bosc, Marc Guyomard, Laurent Miclet. Conception d'Algorithmes : Principes et 150 exercices non-corrigés, 2016

François Bouillé, La Formulation simple et rationnelle des Algorithmes avec le Langage ADL, 2006

Francis Cottet, Traitement des signaux et acquisition de données, éditions Dunod, 2015

Delacourt, Phan Luong, Poupet, TD 8 : Tables de hachage. Algorithmique et structures de données (Année 2011/2012)

Jean-Paul Delahaye, Logique informatique et paradoxes, Belin - Pour la Science, 1995

Edsger Dijkstra : Go To Statement Considered Harmful, Communications of the ACM 11, 3, pp-147-148, Mars 1968

Alan Edelman, Eigenvalue roulette and random test matrices. In Marc S. Moonen, Gene H. Golub, and Bart L. R. De Moor, editors, Linear Algebra for Large Scale and Real-Time Applications, NATO ASI Series, pages 365–368. 1992.

Séverine Fratani, Cours Logique et Calculabilité, L3 informatique, 2013

Michael R. Garey, David S. Johnson. Computers and Intractability, A Guide to the Theory of NP-Completeness. Editions Freeman, 1991

Xavier Gourdon, Les maths en tête : Analyse. Editions Ellipse, 2008.

Bertrand Hauchecorne, Les contre-exemples en mathématiques, édition ellipses, 2007

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. Computer Vision and Pattern Recognition. Décembre 2015.

Michael Held, Richard M. Karp. A Dynamic Programming Approach to Sequencing Problems. Journal of the Society for Industrial And Applied Mathematics, Vol. 10, No. 1, pp 196-210, Mars 1962

Philippe Lac, Malika More. Complexité. Cours à l'IREM de Clermont-Ferrand, 2010

Joseph Lee. Huffman Data Compression, MIT Undergraduate Journal of Mathematics, 2007.

Ran Raz. On the complexity of matrix product. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing (pp. 144-151). ACM. 2002.

Djamal Rebaïne. Programmation dynamique (chapitre 5) : www.uqac.ca/rebaine/8INF806/Chapitre5propreprogrammationdynamique.pdf

Vincent Simonet, Les deux points les plus proches, cours - travaux pratiques, filière Mathématiques - Physique - Science de l'Ingénieur - option informatique, 2001

Cristina Sirangelo, Préparation à l'option Informatique de l'agrégation de mathématiques, 2014

Daniel A. Spielman, Teng Shang-Hua, Smoothed analysis : an attempt to explain the behavior of algorithms in practice. Commun. ACM 52.10 (2009) : 76-84.

Jean-Marc Vincent, Equations de récurrence et complexité : Le Master Theorem. Cours à l'Université de Grenoble-Alpes, 2010

Michel Volle, Somme d'entiers élevés à une puissance quelconque, 2014

Yilun Wang, Yu Zheng, Yexiang Xue. Travel time estimation of a path using sparse trajectories. Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pp 25-34, 2014

Niklaus Wirth, Algorithms + Data Structures = Programs. Prentice Hall series in automatic computation, novembre 1975