

Cours d'Informatique MP2I

Antoine Domenech

2024-2025

TABLE DES MATIÈRES

Chapitre 0 – INTRODUCTION À L'INFORMATIQUE

0. Algorithmes et pseudo-codes	2
0. Problèmes	2
1. Algorithmes	3
2. Un langage de pseudo-code	4
<i>Variables et sauts conditionnels (p. 4). Fonctions (p. 6). Boucles (p. 7).</i>	
3. Convention de représentation des G.F.C.	9
4. Tableaux	10
1. Machines, programmes et langages	11
0. Machines, modèles	11
1. Programmes	12
2. Paradigmes	14

Chapitre 1 – PREUVES DE PROGRAMMES

0. Spécification d'une fonction	16
1. Terminaison	18
0. Variants et preuves de terminaison	18
1. Cas particulier des boucles Pour	20
2. Fonctions récursives	21
2. Correction	22
0. Correction partielle	22
<i>Sauts conditionnels (p. 22). Invariants de boucle (p. 22). Cas particulier des boucles Pour (p. 26).</i>	
1. Fonctions récursives	27
2. Correction totale	28

Chapitre 2 – MÉMOIRE

0. Représentation des types de base	30
0. Notion de données	30
1. Booléens	30
2. Différents types d'entiers	31
<i>Entiers non-signés (p. 31). Caractères (p. 33). Entiers signés (p. 33). Dépassements de capacité (p. 35).</i>	
3. Nombres à virgule (flottante)	36
<i>Notation mantisse-exposant pour l'écriture scientifique (p. 36). Erreurs d'approximation et autres limites (p. 37).</i>	
4. Tableaux	38
5. Chaînes de caractères	39
1. Compléments très brefs de programmation	40
0. Notion d'adresse mémoire	40
1. Portée syntaxique et durée de vie	40
<i>Définition et premiers exemples (p. 40). Exemples avancés (p. 43).</i>	
2. Organisation de la mémoire d'un programme	44
0. Bloc d'activation	44
1. Organisation de la mémoire virtuelle d'un programme	45
<i>Pile mémoire (p. 46). Tas mémoire (p. 46). À savoir faire (p. 47).</i>	

Chapitre 3 – COMPLEXITÉ

0. Complexité temporelle	50
0. Notion de complexité temporelle	50
<i>Opérations élémentaires (p. 51).</i>	
1. Notation de Landau	52
2. Exemples plus avancés	54
<i>Méthodologie (p. 54). Un exemple avec des boucles imbriquées (p. 54). Un exemple où il faut être très précis sur les itérations (p. 55). Écart incompressible entre les bornes (p. 56).</i>	
3. Pire cas, cas moyen, meilleur cas	56
4. Ordres de grandeur	57
1. Complexité spatiale	58
2. Cas particulier des fonctions récursives.....	59
0. Complexité temporelle des fonctions récursives	60
<i>Cas général : formule de récurrence (p. 60). Cas particuliers : arbre d'appels (p. 60).</i>	
1. Complexité spatiale des fonctions récursives	64
3. Complexité amortie	65
0. Méthodes de calcul	65
<i>Exemple fil rouge (p. 65). Méthode du comptable (p. 65). Méthode du potentiel (p. 67). Méthode de l'aggrégat (p. 69).</i>	
1. Remarques et compléments	70

Chapitre 4 – RÉCURSIVITÉ

0. Introduction.....	72
0. Notion de réduction	72
1. Notion de récursion	72
1. Exemples	73
0. Affichage d'un triangle	73
1. Tours de Hanoï	74
2. Exponentiation rapide	76
3. À propos des boucles	77
<i>Transformation boucle <-> récursion (p. 77). Récursivité terminale (hors-programme) (p. 78).</i>	
2. Comment concevoir une fonction récursive	78
3. Analyse de fonctions récursives	79
4. Compléments	79
0. Élimination des appels redondants	79
1. Avantages et inconvénients de la récursion	79
2. Querelle sémantique	80
3. Quand « déplier les appels » ?	80

Chapitre 5 – STRUCTURES DE DONNÉES (S1)

0. Tableaux dynamiques	82
0. Tableaux C vs listes OCaml	82
1. Fonctionnement	83
<i>Ajout dans un tableau dynamique (p. 84). Création, suppression (p. 85).</i>	
2. Analyse de complexité	85
3. Complément mi hors-programme : RETRAIT	85
1. Interfaces et types abstraits	87
0. Aspects pratiques	87
<i>Librairies (p. 87). Interface (p. 88). Étapes de la compilation (p. 90).</i>	
1. Aspects théoriques	91

2. Structures de données séquentielles	93
0. Types de base	93
1. Listes linéaire	93
<i>Implémentation par tableaux de longueur fixe (p. 93). Implémentation par tableaux dynamiques (p. 94). Implémentation par listes simplement chaînées (p. 94). Variantes des listes chaînées (p. 97).</i>	
2. Piles	97
<i>Implémentations par tableaux (p. 98). Implémentation par listes simplement chaînées (p. 99).</i>	
3. Files	100
<i>Implémentations par tableaux circulaires (p. 100). (Hors-programme) Amélioration avec des tableaux dynamiques (p. 103). Implémentations par listes simplement chaînées (p. 103). Implémentation par listes doublement chaînées (p. 104). Par double pile (p. 104).</i>	
4. Variantes des files	105
3. Aperçu des structures de données S2	105

Chapitre 6 – BONNES PRATIQUES DE PROGRAMMATION

0. Compléments sur les types de base	108
0. Évaluation paresseuse des opérateurs booléens	108
1. Détails sur le passage / renvoi des arguments	109
2. En C : instruction pré-processeur #define	109
<i>Header guards (p. 110). (Mi-HP) Utilisation pour créer des tableaux statiques (p. 110).</i>	
1. Types avancés.....	111
0. En OCaml : paires, triplets, etc	111
1. Types enregistrements	111
<i>En C (p. 111). En OCaml (p. 112).</i>	
2. Types énumérations	112
<i>(HP) En C (p. 112). En OCaml (p. 113).</i>	
3. OCaml : types construits	113
<i>Notion de type construit (p. 113). let destructurant (p. 113). Le type 'a option (p. 114).</i>	
4. Types récursifs	114
2. Les exceptions	115
0. (HP) En C	115
1. En OCaml	115
<i>Rattraper une exception (p. 115). Lever une exception (p. 116). Définir une exception (p. 116).</i>	
3. Écrire mieux	117
0. Nommer	117
1. Structurer	118
2. Documenter	118
3. Anticiper	119
4. Simplifier	119
5. Progresser	119
4. Tester mieux	120
5. Déboguer mieux	120

Chapitre 7 – RETOUR SUR TRACE

0. Fils rouges	122
0. Sudoku	122
1. n dames	122
1. Exploration exhaustive	123
0. Définition et limites	123
1. Pseudo-code général d'une exploration exhaustive	124
2. Exploration exhaustive des n dames	125
3. Amélioration de l'exploration exhaustive des n dames	127

2. Retour sur trace	128
0. Définition	128
1. Retour sur trace pour les n dames	129
2. Variantes	130
3. Complexité	131
0. Analyse théorique	131
1. Importance de l'ordre des appels	131
4. Autres exemples classiques	132
0. Cavalier d'Euler	132
1. Perles de Dijkstra	132
2. SUBSETSUM	133
3. CNF-SAT	133
5. Écriture itérative	133

Chapitre 8 – RAISONNEMENTS INDUCTIFS

0. Relations binaires (rappels de maths)	136
0. Relations d'équivalence	136
1. Relations d'ordres	138
2. Suite monotone	141
1. Ordres bien fondés et induction	142
0. Ordres bien fondés	142
1. Variants de boucle/appeal	143
2. Les inductions	144
<i>Ce qu'il faut savoir (p. 144). Formalisation (hors-programme) (p. 145).</i>	
2. Construire de nouveaux ordres	147
0. Transporter des ordres	147
1. Ordre produit	148
2. Ordre lexicographique	150
<i>Sur des éléments de même longueur (p. 150). Sur des éléments de longueurs distinctes (p. 152).</i>	
3. Clotures	153
<i>Définitions (p. 153). Engendrer un ordre (p. 156).</i>	
3. Induction structurelle	157
0. Construction inductive d'un ensemble : exemple illustratif	157
1. Induction structurelle	159
2. Un exercice final	160

Chapitre 9 – ARBRES

0. Exemples introductifs et vocabulaire	165
0. Les mobiles	165
1. Les expressions arithmétiques	166
2. Arbre d'appels récursifs	166
<i>Tours de Hanoï (p. 166). Retour sur trace (p. 167).</i>	
3. Exemple : les tries	168
4. Exemple : les arbres binaires de recherche	168
5. Exemple et parenthèse : gestion des fichiers Unix	169
<i>Inodes et fichiers (p. 169). Inodes et dossiers (p. 169). Liens physiques et symboliques (p. 170).</i>	
6. Vrac de vocabulaire	171
7. Construction inductive	172
1. Arbres binaires et implémentation	173
0. Arbres binaires	173
<i>Définition (p. 173). Implémentation (p. 176).</i>	
1. Arbres binaires stricts	176
<i>Définitions (p. 176). Implémentation (p. 177).</i>	

2. Arbres parfaits et complets	178
<i>Définitions (p. 178). Implémentation des arbres binaires complets (p. 180).</i>	
3. Peigne	180
4. Arbres quelconques	181
<i>Transformation LCRS (p. 181). Type général (p. 182).</i>	
2. Parcours d'arbres	183
0. Parcours en profondeur	183
<i>Ordre préfixe (p. 184). Parcours infixe (p. 185). Parcours postfixe (p. 186). Écriture itérative (p. 186).</i>	
1. Parcours en largeur	187
<i>Écriture itérative (p. 187). (Mi-HP) Preuve du pseudo-code (p. 188). (Mi-HP) Preuve, suite et fin (p. 189).</i>	
3. (HP) Propriétés avancées	189
0. Lemme de lecture unique	189
1. Dénombrement des arbres binaires stricts	190
4. Arbres binaires de recherche	194
0. Définition, caractérisation	194
1. Opérations	196
<i>Recherche (p. 196). Insertion (p. 197). Suppression (p. 198). Rotation (p. 200). Tri par ABR (p. 201).</i>	
2. Arbres Rouge-Noir	202
<i>Définition (p. 202). Insertion (p. 204). Suppression (p. 206).</i>	
3. Pour aller plus loin	206

Chapitre 10 – DÉCOMPOSITION EN SOUS-PROBLÈMES

0. Diviser pour Régner	209
0. Exemple introductif : le tri fusion	209
<i>Borne inférieure en pire des cas pour un tri par comparaisons (p. 209). Tri fusion (p. 209). Implémentation OCaml (p. 210). Implémentation en C (p. 211). Analyse de complexité (p. 213).</i>	
1. Principe général d'un algorithme Diviser pour Régner	214
2. Analyses de complexité	214
3. Exemple : recherche dichotomique	215
<i>La recherche dichotomique (p. 215). Une variante : la recherche d'un pic (p. 216).</i>	
4. Un autre exemple : algorithme de Karatsuba	217
<i>Présentation du problème (p. 217). Un Diviser pour Régner inefficace (p. 217). Algorithme de Karatsuba (p. 218).</i>	
5. Pire cas et cas moyen : le tri rapide	219
<i>Complexité en moyenne (p. 219). Borne inférieure en moyenne pour un tri par comparaisons (p. 219). Tri rapide (p. 220). Drapeau hollandais (p. 220). Implémentation du tri rapide en C (p. 222). Analyse de complexité (p. 223).</i>	
1. Rencontre au milieu	223
0. Problème d'optimisation associé à SUBSETSUM	223
1. Principe général d'un algorithme Rencontre au milieu	224
2. Programmation dynamique	225
0. Exemple fil rouge : plus lourd chemin dans une pyramide	225
<i>Présentation du problème (p. 225). Solution naïve en OCaml (p. 225).</i>	
1. Principe général de la Programmation Dynamique	227
2. Méthode de haut en bas	227
<i>Pour les pyramides (p. 227). Calcul de la complexité (p. 229). Pseudo-code générique (p. 230).</i>	
3. Méthode de bas en haut	231
<i>Pour les pyramides (p. 231). Calcul de la complexité (p. 231). Pseudo-code générique (p. 232). Optimisation de la mémoire (p. 232).</i>	
4. Comparaison des deux méthodes	234
5. Message d'utilité publique...	234
6. (Re)construction de la solution optimale	234
<i>Fonctionnement générique (p. 234). Pour les pyramides : mémorisation des solutions optimales (p. 235). Pour les pyramides : reconstruction de la solution optimale (p. 235).</i>	

Chapitre 11 – GRAPHS NON-PONDÉRÉS

0. Graphes non-orientés	239
0. Introduction	239
1. Définitions	240
<i>Vocabulaire (p. 240). Chemins et connexité (p. 241).</i>	
2. Graphes non-orientés remarquables	245
3. Propriétés combinatoires	246
<i>Liens entre S et A (p. 246). Bornes de connexité et d'acyclicité (p. 247). Liens avec les arbres (p. 249).</i>	
4. (Mi-HP) Coloration de graphes	251
<i>Degré chromatique (p. 251). (HP) Cas des graphes planaires (p. 252). 2-coloration de graphe (p. 253). (MPI) 3-coloration de graphe : NP-difficile (p. 254).</i>	
1. Graphes orientés.....	255
0. Définitions	255
<i>Vocabulaire (p. 255). Chemins orientés (p. 256).</i>	
1. Études des composantes fortement connexes	257
2. Implémentation.....	260
0. Matrice d'adjacence	260
1. Listes d'adjacence	261
2. Comparatif	263
3. Parcours de graphes	264
0. Parcours générique	264
<i>Parcours depuis un sommet (p. 264). Parcours complet (p. 266). Complexité (p. 266). Arbre de parcours (p. 267).</i>	
1. Parcours en largeur	270
<i>Définition et propriété (p. 270). Une optimisation : le marquage anticipé (p. 270). Implémentation en OCaml (p. 271).</i>	
2. Parcours en profondeur	272
<i>Définition et propriétés (p. 272). Pas de marquage anticipé!! (p. 274). Implémentation en OCaml (p. 275).</i>	
3. Application : calcul des composantes connexes	276
4. Application : calcul d'un ordre topologique	277
<i>Définition (p. 277). Construction d'un ordre topologique (p. 277). Détection de cycles (p. 278). Implémentation en OCaml (p. 279).</i>	
4. Graphes bipartis	281
0. Définition	281
1. Couplage (dans un graphe biparti ou non)	282

Chapitre 12 – ALGORITHMES GLOUTONS

0. Généralités	284
0. Principe général	284
<i>Définition (p. 284). Intérêt (p. 285).</i>	
1. Exemple : ordonnancement d'intervalle	285
2. Preuve d'optimalité	287
1. Plus d'exemples	288
0. Stockage de fichiers	288
1. Rendu de monnaie	288
<i>En euros (p. 289). Dans un système monétaire quelconque (p. 289).</i>	
2. Stay tuned!	289
<i>Algorithme de Huffman (p. 289). Algorithme de Kruskall (p. 290). Algorithme ID3 (p. 290).</i>	
3. Message d'utilité publique	290
2. (HP) Théorie des matroïdes	291
0. Matroïde	291
1. Algorithme glouton	291

Chapitre 13 – STRUCTURES DE DONNÉES (S2)

0. Tableaux associatifs	296
0. Type abstrait	296
<i>Motivation (p. 296). Type abstrait (p. 296).</i>	
1. Implémentation avec des listes d'associations	297
2. Implémentation avec des ABR équilibrés	298
3. Implémentation avec des tables de hachage	299
<i>Fonction de hachage (p. 299). Table de hachage (p. 300). Attention aux objets mutables! (p. 301).</i>	
4. Complexités	301
5. Tri par dénombrement	302
6. Un type d'ensemble	303
1. Tas	304
0. Définition	304
<i>Définition (p. 304). Implémentation en OCaml (p. 306).</i>	
1. Opérations	306
<i>Percolation vers le haut (p. 306). Insertion (p. 308). Percolation vers le bas (p. 309). Extraction du minimum (p. 311).</i>	
2. Complexités	312
3. Tri par tas	312
2. Files de priorité	313
0. Type abstrait	313
1. Implémentation avec des ABR équilibrés	314
2. Implémentation avec des tas	315
3. Complexités	315
4. Une opération supplémentaire	316

Chapitre 14 – LOGIQUE

0. Syntaxe des formules logiques	319
0. Formules propositionnelles	319
1. Notations et parcours	320
2. Égalités et substitutions	321
1. Sémantique des formules propositionnelles	323
0. Évaluation d'une proposition logique	323
1. Conséquence, équivalence logique	325
2. Satisfiabilité	327
2. Formes normales	329
0. Définitions	329
1. Construction de DNF/CNF	330
2. Représentation informatique d'une formule	332
3. SAT et algorithme de Quine	333
0. Problème SAT	333
1. Algorithme de Quine	334
2. Une petite parenthèse : les graphes de flots	336
4. Quantificateurs et introduction à la logique du premier ordre.....	337

Chapitre 15 – ALGORITHMIQUE DU TEXTE

Chapitre 16 – GRAPHS PONDÉRÉS

Chapitre 17 – BASES DE DONNÉES

Chapitre 0

INTRODUCTION À L'INFORMATIQUE

Notions	Commentaires
Notion de programme comme mise en oeuvre d'un algorithme. Paradigme impératif structuré, paradigme déclaratif fonctionnel, paradigme logique.	On ne présente pas de théorie générale sur les paradigmes de programmation, on se contente d'observer les paradigmes employés sur des exemples. La notion de saut inconditionnel (instruction GOTO) est hors programme. On mentionne le paradigme logique uniquement à l'occasion de la présentation des bases de données.
Caractère compilé ou interprété d'un langage.	Transformation d'un fichier texte source en un fichier objet puis en un fichier exécutable. [...]

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Graphe de flot de contrôle. [...]	[...]

Extrait de la section 1.3 du programme officiel de MP2I : « Validation, test ».

SOMMAIRE

0. Algorithmes et pseudo-codes	2
0. Problèmes	2
1. Algorithmes	3
2. Un langage de pseudo-code	4
<i>Variables et sauts conditionnels (p. 4). Fonctions (p. 6). Boucles (p. 7).</i>	
3. Convention de représentation des G.F.C.	9
4. Tableaux	10
1. Machines, programmes et langages	11
0. Machines, modèles	11
1. Programmes	12
2. Paradigmes	14

Définition 1 (src : CNRTL).

« Informatique (Subt., Fem.) : Science du traitement rationnel, notamment par des machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social. »

0 Algorithmes et pseudo-codes

0.0 Problèmes

Définition 2 (Problème).

Un problème est composé de deux éléments :

- une instance (aussi appelée entrée).
- une question ou une tâche à réaliser sur cette instance.

Exemple.

- PGCD :

Entrée : x, y deux entiers strictement positifs.

Tâche : calculer leur plus grand diviseur commun.

- TRI :

Entrée : des éléments comparables selon un ordre \leq .

Tâche : trier ces entiers par ordre \leq croissant.

- PLUS COURT CHEMIN :

Entrée : une carte de réseau de transport.

un point de départ sur ce réseau, et un d'arrivée.

Tâche : calculer un plus court chemin du départ à l'arrivée.

- PUZZLE :

Entrée : un cadre et des pièces de puzzle.

Question : peut-t-on agencer les pièces dans le cadre selon les règles ?

- #PUZZLE :

Entrée : un cadre et des pièces de puzzle.

Tâche : calculer de combien de façons il est possible d'agencer légalement les pièces dans le cadre.

- SEUIL-PUZZLE :

Entrée : un cadre et des pièces de puzzle.

un entier $n \in \mathbb{N}$

Question : Existe-t-il plus de n (inclus) façons d'agencer légalement les pièces ?

On parle de problème de :

- **décision** si la réponse est binaire (Oui/Non).
- **optimisation** si la réponse est une solution "meilleure" que tout autre.
- **dénombrement** si la réponse est un nombre de solutions.
- liste non exhaustive.

Exemple. Associer un type (ou plusieurs) à chacun des problèmes précédents.

0.1 Algorithmes

Le terme « algorithme » a de nombreux sens. La définition que j'en donne ici est celle que nous utiliserons cette année ; mais ce n'est forcément pas celle du langage courant¹. Cela ne signifie pas que l'une des deux est meilleure que l'autre : un même mot a des sens différents dans des contextes différents, c'est normal, et cela nous arrivera même entre différentes sections d'un même chapitre.

Définition 3 (Algorithme).

Un algorithme est la description finie d'une suite d'opérations résolvant un problème. On veut de plus avoir :

- Terminaison : la suite d'opération décrite est atteinte toujours sa fin. On dit que l'algorithme termine.
- Précisément définie : les opérations sont définies précisément, rigoureusement et sans ambiguïté.
- Entrée : un algorithme a 0 ou plusieurs quantités de nature connue et spécifiée qui lui sont données avant ou pendant son exécution.
- Sortie : un algorithme renvoie une ou plusieurs quantités dont le lien avec les entrées est spécifié.
- Éléментарité : les opérations décrites doivent être assez simples pour être réalisées par une personne avec un papier et un crayon en temps fini.

Remarque.

- On rapproche souvent les algorithmes des recettes de cuisine.
- Aucune notion de langage informatique n'est présente dans cette définition. Langage naturel², idéogrammes, schéma, pseudo-code, Python, C, OCaml, SQL : tout pourrait convenir.
- Attention à ne pas confondre la terminaison avec le fait que la description elle-même est finie.
- Le caractère précisément défini demande que l'on sache *quoi* faire, alors que l'élémentaire demande que l'on sache *comment* le faire.
- L'éléментарité dépend du public-cible. « Diviser un entier par un autre » est une étape élémentaire pour vous, pas pour des élèves de CE2. De même pour la précision : elle peut dépendre du contexte.
- Cette définition d'algorithme ne contient aucun critère d'efficacité en temps (ou en espace mémoire).

Exemple. L'algorithme d'Euclide pour résoudre PGCD est un exemple connu. Dedans, « assigner $x \leftarrow y$ » signifie « écraser la valeur de x en la remplaçant par celle de y ».

Algorithme 1 : Algorithme d'Euclide.

Entrées : m et n deux entiers strictement positifs

Sorties : $\text{pgcd}(m, n)$

- 1 E1 (calcul du reste) : diviser m par n , et noter r le reste
 - 2 E2 (égal à 0 ?) : si $r = 0$, renvoyer n
 - 3 E3 (réduction) : assigner $m \leftarrow n, n \leftarrow r$. Recommencer en E1.
-

1. Il me semble que de plus en plus, le terme a une connotation d'obscur, d'inconnu, de traitement caché. Ce n'est pas le contexte dans lequel je me place dans ce cours. Notez que je n'affirme pas ladite connotation est absurde.

2. Le langage naturel est la langue des humains-es.

On peut représenter cet algorithme sous une forme visuelle :

FIGURE 1 – Graphe de Flot de Contrôle de l'algorithme 1.

Exercice. Vérifier qu'il s'agit bien d'un algorithme. Le faire tourner sur l'instance $m = 15, y = 12$.

0.2 Un langage de pseudo-code

Pour utiliser une machine automatique, on doit transformer l'algorithme en une suite d'instruction que la machine comprend. La première et principale étape consiste en générale à écrire cet algorithme dans un *langage de programmation*. Ceux-ci ont de nombreuses différences, mais aussi énormément de concepts communs. On parle de **pseudo-code** pour décrire quelque chose qui ressemble à des langages de programmation, manie ces concepts communs, mais limite les détails techniques propres à chaque langage.

Le pseudo-code que je vais présenter ici est très **impératif**, c'est à dire que l'on programme en décrivant des modifications successives de la mémoire à effectuer.

0.2.0 Variables et sauts conditionnels

Définition 4 (Variable, version intuitive).

Une variable peut-être vue comme une boîte qui a :

- un identifiant, aussi appelé nom de variable.
- un contenu. Celui-ci peut-être par exemple :
 - une valeur précise indiquée.
 - le résultat d'un calcul.
 - la valeur renvoyée par un autre algorithme sur une entrée précise.

On note $n \leftarrow \alpha$ le fait que la variable n contienne dorénavant le contenu α .

Exemple.

$x \leftarrow 3$	on stocke 3 dans x
$x \leftarrow (9 - 4)$	x contient dorénavant 5 et non plus 3
$y \leftarrow 2x$	on stocke 10 dans y
$x \leftarrow y^3$	x contient dorénavant 1000 et non plus 5.

Définition 5 (Saut conditionnel).

Un saut conditionnel permet d'exécuter différentes instructions selon une valeur logique, souvent le résultat d'un test. On note :

si <i>valeur logique</i> alors instructionsDuAlors sinon instructionsDuSinon
--

Pseudo-code

G.F.C associé.

Les instructions du Alors s'effectuent si la valeur logique s'évalue à Vrai, celles du Sinon... sinon. Ces instructions sont respectivement nommées le **corps du Alors/Sinon**.

Lorsque le corps du Sinon est vide, on n'écrit tout simplement pas le Sinon (et son corps).

Exemple.

```

1  x ← 3
2  si x ≥ 2 alors
3    | x ← 3x
4    | x ← x - 1
5  sinon
6    | x ← 1.5x
7    | x ← -2x

```

À la fin de l'exécution de ce code, x contient 8.

La notion de conditionnel est central en informatique. C'est elle qui permet de faire en sorte qu'un programme puisse avoir des comportements différents en fonction des informations qui lui sont données ! Imaginez un moteur de recherche internet qui renvoie toujours les mêmes résultats, peu importe l'entrée...

Remarque.

- Les barres verticales sont très utiles à la relecture. Je vous encourage *très très fortement* à les mettre dès que vous êtes sur papier, y compris lorsque vous écrivez du vrai code.
- En anglais, Si/Alors/Sinon se dit If/Then/Else.
- Par abus du langage, on parle souvent du « corps du Si » pour désigner le corps du Alors.
- Un saut conditionnel est lui-même une instruction, et on peut donc les imbriquer :

Résumé grossier du fonctionnement de M. Domenech aux portes ouvertes.

```

1  si l'élève veut faire 12h de maths alors
2    | si l'élève aime ou veut découvrir l'informatique alors
3    | | orienter en MP2I
4    | sinon
5    | | orienter en MPSI
6  sinon
7    | orienter en PCSI (10h de maths quand même)

```

Notez que 10h vs 12h de maths n'est pas un excellent critère. Dans le groupe nominal « résumé grossier », il se pourrait que le qualificatif soit plus important que le substantif.

- Considérez l'imbrication suivante : « Si b_0 Alors Si b_1 Alors $instr_0$ Sinon $instr_1$ ». Ici, il n'est pas possible de savoir de quel Si $instr_1$ est le Sinon. On parle du problème du *Sinon pendant* (Dangling else d'Alan Turing) : il y a une ambiguïté sur l'imbrication.

Sur papier, une écriture avec retour à la ligne, indentation (décalage horizontal du corps) et barre verticale permet de l'éviter.

Les langages de programmation, eux, ont chacun une règle qui force une unique façon d'interpréter cette imbrication. Il s'agit généralement soit d'indiquer explicitement la fin du saut conditionnel, soit de la règle du parenthésage à gauche d'abord ; nous en reparlerons.

- Les sauts inconditionnels (« goto ») sont hors-programme et prohibés.

0.2.1 Fonctions

Lorsque vous avez découvert les fonctions en maths, on vous les a possiblement présenté comme des machines qui prennent quelque chose en entrée et le transforme en autre chose (en termes savants, on dit qu'elles associent un antécédant à une image). C'est la même idée en informatique.

Définition 6 (Fonction, version intuitive).

Une **fonction** est transforme des **arguments** en une nouvelle valeur. On dit aussi qu'elle transforme des **entrées** en **sorties**.

En pratique, une fonction est un bloc de pseudo-code qui attend certaines variables en entrée, décrit des modifications à faire dessus, et indique une valeur à renvoyer à la fin. On dit que les instructions contenues dans la fonction forment le **corps** de la fonction.

On dit qu'une fonction est **appelée** lorsqu'on l'utilise pour calculer la sortie associée à une entrée. On dit qu'elle **renvoie** sa sortie. On note généralement comme en maths : $f(x_0, x_1, \dots)$ est la valeur renvoyée par la fonction f sur l'entrée (x_0, x_1, \dots) .

Exemple. Voici deux fonctions. Que font-elles ?

Entrées : x une variable contenant un entier.

Sorties : L'entier 2.

1 **renvoyer** 2

Entrées : x une variable contenant un entier.

Sorties : L'entier x^2 .

1 $y \leftarrow x$

2 **renvoyer** $x * y$

Exercice. Proposer une autre façon d'écrire le corps de cette seconde fonction qui évite la création d'une variable dans la fonction.

Notez que la notion de corps induit une notion de « monde extérieur » : on distingue le (pseudo-)code qui se trouve dans la fonction du (pseudo-)code qui se trouve ailleurs.

Exemple. Indiquer ce que vaut `main()` avec les fonctions ci-dessous :

Entrées : x une variable contenant un entier.

Sorties : y un entier

1 $y \leftarrow 2x$

2 $x \leftarrow 0$

3 **renvoyer** y

Fonction `fun`

Entrées : Rien.

Sorties : Mystère.

1 $x \leftarrow 3$

2 $z \leftarrow \text{fun}(x)$

3 **renvoyer** x

Fonction `main`

Il y a deux réponses possibles : 0 (car `fun` a remis x à 0) et 3 (car `fun` ne travaille pas sur le x d'origine mais sur une copie). Cela correspond respectivement aux deux notions suivantes :

Définition 7 (Passages et effets secondaires).

Soit F une fonction et x un de ses arguments. On dit que x est :

- **appelé par référence** si toute modification de x par le corps de F modifie également le x initial (la variable qui avait été donnée en entrée à F lors de son appel). Tout se comporte comme si le corps de la fonction et le monde extérieur partagent le même x .
- **appelé par valeur** si, au contraire, toute modification de x par le corps de F ne modifie pas le x initial. Tout se comporte comme si le corps de la fonction manipule une copie du contenu x , distincte du x du monde extérieur.

Sauf mention contraire explicite dans ce cours, tous les arguments sont appelés par valeur. La mention contraire principale concernera les tableaux.

En particulier, dans l'exercice précédent, la réponse que donnerait tout langage de programmation raisonnable³ est 3.

3. Et toute étudiant-e de MP2I raisonnable ;)

Définition 8 (Effets secondaires, version courte).

L'ensemble des modifications effectuées par une fonction sur son monde extérieur est appelé les **effets secondaires** de la fonction.

Remarque.

- On parle aussi de **passage par référence/valeur**. Nous verrons également le **passage par pointeur**, qui à notre niveau ne sera pas distinct du passage par référence.
- Le passage par valeur est coûteux : l'appel de la fonction doit consommer du temps et de l'espace mémoire afin de créer une copie de l'argument concerné. Les langages de programmation réservent donc généralement le passage par valeur aux variables de taille raisonnable (comme des entiers) et passent tout ce qui est potentiellement gros (comme une collection de valeurs plus simples) par référence.
Par exemple, Python passe les entiers par valeur et les listes par référence.
- L'exemple de `fun` et `main` montre également un exemple où une fonction en appelle une autre, cela nous arrivera très fréquemment. **Le rôle principal des fonctions est de décomposer et structurer un gros code en petites unités élémentaires simples à comprendre.**
Nous verrons également qu'il peut-être très comode pour une fonction de s'appeler elle-même, ce que l'on nomme la **réursion**.

Définition 9 (Prototype, signature).

- La donnée de l'identifiant (le nom) d'une fonction ainsi que de ses types d'entrées et de sortie est appelée la **signature** de la fonction.
- Si l'on ajoute les identifiants des entrées, on parle de **prototype**.

Exemple.

- Signature de PGCD : PGCD : entier , entier \rightarrow entier
- Prototype de PGCD : PGCD : *m* entier, *n* entier \rightarrow entier

En mathématiques, on note $pgcd : \mathbb{N}^* \times \mathbb{N}^* \rightarrow \mathbb{N}^*$. C'est la signature !

0.2.2 Boucles

De même que l'informatique sans Si/Alors/Sinon n'est pas très intéressante, il nous manque pour l'instant une capacité très importante : pouvoir répéter des instructions un nombre inconnu de fois. C'est par exemple ce que faisait l'algorithme d'algorithme grâce à son étape E3 qui indiquait « retourner en E1 ». On dit que E1-E2-E3 forment une boucle.

Définition 10 (Boucle à précondition).

Une boucle à précondition, ou boucle TantQue (« While ») répète des instructions tant qu'une valeur logique (souvent une condition) est vraie :

tant que <i>valeur logique</i> faire instructions
--

Pseudo-code

G.F.C associé.

Les instructions répétées sont appelées le **corps**. On dit qu'on effectue une **itération** de la boucle lorsque l'on exécute les opérations du corps.

Exemple.

Entrées : x un entier.

```

1  $p \leftarrow 0$ 
2 tant que  $x < 436$  et  $p < 10$  faire
3   |  $x \leftarrow 2x$ 
4   |  $p \leftarrow p + 1$ 
5 renvoyer  $x$ 
    Fonction demo

```

Pour calculer à la main $\text{demo}(1)$, on peut écrire un tableau où on décrit le contenu des variables à la fin de chaque itération :

	p	x
Avant la première iter.	0	1
Fin de la première iter.	1	2
Fin de la seconde.	2	4
Fin de etc	3	8
Fin de etc	4	16
Fin de etc	5	32
Fin de etc	6	64
Fin de etc	7	128
Fin de etc	8	256
Fin de etc	9	512

Après cette dernière itération, on quitte la boucle car $x \geq 436$ et donc la valeur logique d'entrée dans la boucle devient fausse.

Remarque.

- Si la valeur logique du TantQue reste éternellement vraie, on ne quitte jamais la boucle et on effectue de sitérations à l'infini. On parle de **boucle infinie**. C'est un des bugs les plus courants.
- Plusieurs fois dans l'année, nous referons des tableaux d'itération comme celui-ci. Nous noterons avec des primes (p' et x') les quantités en fin d'itération afin d'éviter des confusions.
- On peut tout à fait faire le tableau avec les quantités en début d'itération. Cela évite la ligne « avant la première itération », mais demande souvent de rajouter une ligne à la fin « début de l'itération qui n'a pas lieu car on quitte la boucle ».

Dans ce cours et toute cette année, je travaillerai plutôt avec les fins.

Exercice. Calculer l'image de 14 par la fonction suivante :

Fonction Collatz	
Entrées : x un entier strictement positif.	
Sorties : Mystère mystère.	
1	tant que $x > 1$ faire
2	si x est pair alors
3	$x \leftarrow x/2$
4	sinon
5	$x \leftarrow 3x + 1$
6	renvoyer x

CultureG : les valeurs successives de x durant ces itérations forment ce que l'on appelle une suite de Syracuse^{4,5}. Il est conjecturé et presque prouvé⁶ que toutes les suites de Collatz contiennent tôt ou tard l'entier 1.

4. Pas Syracuse comme la ville grecque, voyons. Syracuse comme l'université de Syracuse, bien sûr. Qui tient son nom de la ville de Syracuse, New-York (l'état, pas la ville), États-Unis. Évidemment.

5. L. Collatz, lui, est un mathématicien ayant participé à populariser l'étude de ces suites.

6. « Presque » au sens où T. Tao a prouvé que cette conjecture est vraie pour presque tous les entiers. La définition de « presque tous » est subtile. Retenez qu'en pratique elle est vraie.

Définition 11 (Boucle itérative).

Une boucle itérative, ou boucle Pour (« For »), permet de répéter des instructions un nombre connu de fois tout en mémorisant le numéro de la répétition en cours. C'est en fait un simple raccourci pour une boucle TantQue :

```
pour compteur allant de 0 inclus à n
  exclu faire
    | instructionsDuCorps
```

Pseudo-code d'un For

```
compteur ← 0
tant que compteur < n faire
  | instructionsDuCorps
  | compteur ← compteur + 1
```

Pseudo-code du While équivalent

G.F.C. associé.

On parle là aussi du **corps** de la boucle.

Remarque.

- Notez que la version While et la version G.F.C. explicitent toutes deux le fait que **la mise à jour du compteur a lieu à chaque itération, en toute fin d'itération**.
- On croise souvent les boucles PourChaque, qui permettent d'effectuer le corps de la boucle sur chacun des éléments d'une collection d'objets (e.g. « PourChaque *x* entier de $\llbracket 8; 35 \rrbracket$ »). C'est par exemple ce que fait `for x in ...` de Python.

Définition 12 (break, continue).

On crée deux nouvelles instructions pour les boucles : **break** permet de quitter instantanément la boucle en cours, et **continue** permet de passer immédiatement à l'itération suivante (en ayant mis à jour le compteur).

Ces deux instructions ne s'appliquent *que* à la boucle en cours (et pas aux éventuelles sur-boucles en cas d'imbrication).

Attention, ces deux instructions sont une erreur commune de bugs des débutant·es. La plus « dangereuse » des deux, `continue`, est hors-programme pour cette raison.

0.3 Convention de représentation des G.F.C.

Le flot d'un code désigne l'enchaînement de ses instructions. Un graphe de flot de contrôle est une représentation du déroulement de l'exécution (pseudo-)code. Le but est de représenter quelle instruction peut mener à quelle instruction, c'est à dire de représenter le flot.

Dans les schémas précédents, on a utilisé les conventions suivantes :

- Le début du flot est marqué par un triangle (base en bas), et par une double flèche y menant. Au départ de la flèche, on indique les variables d'entrées.
- Les fins du flot sont marquées par un triangle (base en haut), et une double flèche en sortant. À la fin de la flèche, on indique la sortie / valeur renvoyée par cette fin du flot.

- Les instructions sont écrites dans des rectangles. On peut utiliser un même rectangle pour tout un corps.
- Le passage d'un point à un autre est représenté par une flèche simple.
- Les embranchements (sauts conditionnels, entrée/sortie de boucle) sont représentés par un "losange", coin vers le bas. Dans le losange on indique la valeur logique qui est interrogée. Sur les flèches sortantes, on indique Vrai/Faux (ou Oui/Non) pour indiquer quelle flèche correspond à quelle option.
- On représente les appels de fonction par un cercle contenant le nom de la fonction. On détaille éventuellement le GFC de la fonction à part.

Exemple. Cf G.F.C. de l'algorithme d'Euclide.

0.4 Tableaux

On a parfois besoin de stocker une très grande quantité de données, au point où créer une variable par donnée ne serait pas pratique. Par exemple, si je veux stocker la note d'anglais au bac de chaque élève de CPGE scientifique de Camille Guérin, il me faudrait $48 \times 5 = 240$ variables.

Définition 13 (Tableaux, version intuitive).

On peut penser un tableau comme un long meuble muni de L tiroirs numérotés. Chacun des tiroirs est une variable.

Un peu plus formellement, un **tableau** est défini par sa **longueur**, souvent notée L , qui est son nombre de **cases**. Chaque case se comporte comme une variable. On peut accéder à chaque case à l'aide de son indice, compris entre 0 et $L - 1$ inclus : si T est un tableau et i un indice valide, on note $T[i]$ la case d'indice i de T .

Le contenu de toutes les cases d'un même tableau doit avoir la même nature (tous des entiers, ou bien tous des nombres à virgule, etc).

Dans ce cours, si un tableau à L cases contient dans l'ordre les valeurs x_0, x_1, \dots, x_{L-1} , on le notera en pseudo-code $[x_0; x_1; \dots; x_{L-1}]$.

Lors des appels de fonction, un tableau se comporte comme si son contenu était passé par référence.

Exemple.

<pre> 1 T ← tableau de longueur 3 2 T[0] ← 42 3 T[1] ← -20 4 T[2] ← T[0] + T[1] </pre>	À la fin de l'exécution, T est $[42; -20; 22]$.
--	--

Exemple. La fonction `mainbis()` ci-dessous est la fonction constante égale à 0, car les cases du tableau se comportent comme si passées par référence :

Entrées : T un tableau d'entiers.

```

1 y ← 2 × T[0]
2 T[0] ← 0
3 renvoyer y

```

Fonction `funbis`

Entrées : Rien.

```

1 T ← tableau d'entiers non-vide
2 z ← funbis(T)
3 renvoyer T[0]

```

Fonction `mainbis`

Remarque.

- La longueur d'un tableau est fixe. Les tableaux ne sont *pas* redimensionnables. En particulier, les listes Python ne sont pas des tableaux. C'est une structure plus complexe, qui permet plus de choses mais est un peu plus lente : ce sont des tableaux dynamiques (de pointeurs), que nous verrons comment construire cette année.
- Très souvent, nous serons amenés à traiter les unes après les autres les cases d'un tableau ; on dit qu'on **parcourt** le tableau. Les boucles sont l'outil adapté pour cela :

<pre> 1 i ← 0 2 tant que i < L faire 3 traiter T[i] 4 i ← i+1 </pre>	<pre> 1 pour i allant de 0 inclus à L exclu faire 2 traiter T[i] </pre>
---	---

Avec une boucle For

Avec une boucle While

- Le contenu des cases d'un tableau T peut tout à fait être des tableaux : dans ce cas, la case $T[i]$ de T est elle-même un tableau, et on peut donc en demander la case j . On demande donc $(T[i])[j]$, que l'on préfère écrire $T[i][j]$.

1 Machines, programmes et langages

1.0 Machines, modèles

Définition 14 (Ordinateur (src : wikipedia.fr)).

Un ordinateur est un système de traitement de l'information programmable.

Exemple.

- L'ordinateur PC-prof de la salle B004. C'est le sens commun du terme « ordinateur », et c'est sur ces ordinateurs-ci que nous apprendrons à programmer.
- Vos téléphones, consoles de jeu.
- Les télévisions, voitures, vidéoprojecteurs, etc modernes.
- Un métier Jacquart.
- Certains jeux, comme Minecraft ou BabaIsYou (on peut simuler un ordinateur dedans).
- Les machines à laver modernes (si si).

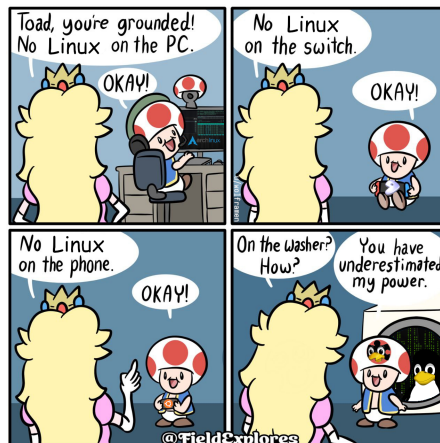


FIGURE 2 – Inspiré de faits réels.

Définition 15 (Modèle de calcul).

Un modèle de calcul est un ensemble de règles qui permettent de simuler les étapes d'un calcul. C'est souvent une abstraction d'une machine de calcul (e.g. un ordinateur) du monde réel, afin de prouver des propriétés sur la machine et son fonctionnement.

Remarque. Le modèle est une *simplification*. Par exemple, on lui suppose souvent une mémoire infinie⁷.

7. Et moi qui me trouve chanceux avec mes $2 \times 8 \times \text{Go}$ de RAM, alors que je pourrais viser l'infini...

Exemple.

- Machine à RAM : représente un ordinateur muni d'une mémoire vive (RAM) infinie.
- Machine de Turing : idem, sauf que la mémoire est un ruban qu'il faut dérouler pour aller d'un point à un autre de la mémoire. Ce modèle représente les premiers ordinateurs modernes et a eu un très fort impact théorique et historique.
- Lambda-calcul : a inspiré la famille des langages de programmation fonctionnels. Très théorique, ne correspond pas à une machine réelle.

1.1 Programmes

Voici un schéma (très) simplifié des ordinateurs modernes⁸ :

FIGURE 3 – B-A-BA de l'architecture d'un ordinateur.

Définition 16 (Programme.).

Un **programme**, aussi appelé exécutable, est une suite d'instructions à effectuer par le processeur. Ces instructions sont écrites en **langage machine**.

Remarque. Un compilateur ne peut lire *que* du langage machine, qui est une suite de 0 et de 1 !

Exemple. Ce sont des opérations très basiques. Voici un exemple "lisible" par des humains, où l'on a "traduit" les suites de 0 et de 1 du langage machine en mots-clefs (on parle de **langage assembleur**) :

<code>mov r3, #17</code>	<i>stocke 17 dans la case mémoire r3</i>
<code>add r3, r3, #42</code>	$r3 \leftarrow r3 + 42$

Le langage assembleur, et plus encore le langage machine, est très dur à lire et permet assez peu d'abstraction. C'est pour cela que nous ne programmerons pas en assembleur mais dans des langages de programmation, pensés pour être plus proches des humains.

Définition 17 (Langage de programmation).

Un langage de programmation est défini ensemble de règles syntaxiques propres. L'objectif est de rédiger des programmes de manière plus lisible que le langage machine. « Coder dans un langage de programmation » signifie « écrire un texte qui respecte les règles du langage. »

8. Non-contractuel, j'ai vraiment simplifié comme un bourrin.

Définition 18 (Compilateur).

Pour que le fichier écrit dans un langage de programmation devienne un programme, il faut le traduire en langage machine : un tel traducteur est appelé un **compilateur**.

Lorsque l'on traduit d'un langage de programmation vers un autre langage de programmation (et non vers le langage machine), on dit que l'on **transpile** (pour distinguer de compile).

Exemple.

- C, OCaml, Rust, C++, Haskell, Fortran, etc sont des
- Le langage machine est techniquement un langage de programmation (c'est même le seul qui n'a pas à être compilé).
- L'assembleur (la traduction est assez directe, il suffit⁹ de traduire mot à mot en 0-1).
- On considère souvent que les langages comme le HTML, qui ne servent pas à exprimer des calculs mais à décrire un agencement, ne sont pas des langages de programmation. Pour HTML, on parle de langage de balisage.
Cela ne signifie pas que travailler sur ces langages est moins "noble" ou plus simple. Cela signifie juste que ce n'est pas la même chose car le but (et les moyens offerts) est radicalement différent.

Remarque.

- Pour un même langage, différents compilateurs peuvent exister. Ils peuvent proposer des façons de traduire différentes, qui optimisent des critères distincts.
Par exemple, pour C, il en existe au moins 3 : gcc (que nous utiliserons), clang et compcert.
- Un compilateur peut faire des modifications à votre code afin de l'accélérer. Par exemple, si un compilateur détecte qu'une boucle ne sert qu'à calculer $\sum_{i=1}^n i$, il peut supprimer la boucle et la remplacer par $\frac{n(n+1)}{2}$.
- Différents processeurs ont des langages machines différents. Si l'on veut distribuer un logiciel, il faut donc en prévoir une version compilée par langage machine que l'on veut pouvoir supporter. C'est pour cela que vous trouvez par exemple des versions arm et des version x86 des logiciels sur leurs pages de téléchargement (en réalité il faut aussi une version par système d'exploitation : différents Linux, MacOS, Windows, etc).
- Une fois un code compilé, il n'est plus nécessaire d'avoir le compilateur (ou le fichier source du code) pour lancer le programme : seule compte la version compilée.

Définition 19 (Interpréteur).

Un interpréteur est un programme qui lit du code écrit dans un langage de programmation et l'exécute au fur et à mesure de sa lecture.

Exemple. Python, Java, OCaml (qui dispose d'un compilateur *et* d'un interpréteur).

Remarque.

- Le code n'est donc jamais traduit en un programme exécutable et a besoin de l'interpréteur pour fonctionner.
- Le fait que les lignes sont lues les unes et interprétées après les autres permet beaucoup moins d'optimisations qu'un compilateur (qui lit tout le fichier avant de compiler). Un code interprété est donc typiquement plus long qu'un code compilé (et consomme souvent plus de mémoire car il faut faire tourner l'interpréteur en plus du code).
- L'avantage est que l'on peut exécuter uniquement petit morceau du code grâce à l'interpréteur. Cela permet de tester un morceau choisi en ignorant les erreurs qui pourraient se trouver ailleurs. À l'inverse, un compilateur refusera de compiler un fichier tant qu'il reste un non-respect des règles du langage dedans.

9. Modulo simplification pédagogique.

- Un autre avantage est qu'il n'y a pas besoin de recompiler pour chaque langage processeur ! Il faut juste avoir accès à un interpréteur (lui-même un programme compilé en général) pour ce langage processeur.
- Différents interpréteurs peuvent exister pour un même langage. Python en a 3, chacun écrits dans un langage différent. Vous avez probablement utilisé sans le savoir cpython, codé en C, qui est l'interpréteur le plus courant.
- On parle de **langage interprété** (respectivement **langage compilé**) pour désigner un langage qui dispose d'un compilateur (respectivement interpréteur).

1.2 Paradigmes

Définition 20 (Paradigmes de programmation).

Il existe différentes façons de penser la programmation :

- **paradigme impératif** : programmer, c'est décrire une suite de modifications de la mémoire.
- **paradigme fonctionnel** : programmer, c'est décrire une le calcul à effectuer comme une suite d'évaluation de fonctions mathématiques.
- **paradigme logique** : programmer, c'est décrire très précisément le résultat attendu à l'aide d'un ensemble de formules logiques.

Un langage peut correspondre à différents paradigmes.

Ce sont des concepts qui se comprennent mieux en pratiquant.

Exemple.

- Pseudo-code ci-dessus, C, Python, Rust, C++, OCaml : paradigme impératif.
- OCaml, Haskell, Erlang, Elixir : paradigme fonctionnel.
- Prolog, un peu SQL : paradigme logique.

Chapitre 1

PREUVES DE PROGRAMMES

Notions	Commentaires
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine.*

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Spécification des données attendues en entrée, et fournies en sortie/retour.	On entraîne les étudiants à accompagner leurs programmes et leurs fonctions d'une spécification. Les signatures des fonctions sont toujours précisées.

Extrait de la section 1.2 du programme officiel de MP2I : « Discipline de programmation ».

SOMMAIRE

0. Spécification d'une fonction.....	16
1. Terminaison	18
0. Variants et preuves de terminaison	18
1. Cas particulier des boucles Pour	20
2. Fonctions récursives	21
2. Correction.....	22
0. Correction partielle	22
<i>Sauts conditionnels (p. 22). Invariants de boucle (p. 22). Cas particulier des boucles Pour (p. 26).</i>	
1. Fonctions récursives	27
2. Correction totale	28

0 Spécification d'une fonction

Rappels du Chapitre 0 (Introduction) :

- La signature d'une fonction est la donnée de son identifiant, du type de chacune de ses entrées et du type de sa sortie.
- Le prototype d'une fonction est la même chose à ceci près que l'on indique en plus le nom de chacun des variables.

```
1 bool est_positif(int x) {
2   return (x >= 0);
3 }
```



- Signature : $\text{est_positif} : \text{int} \rightarrow \text{bool}$
- Prototype : $\text{est_positif} : (x : \text{int}) \rightarrow \text{bool}$

Remarque.

- La façon d'écrire les prototypes « à la mathématiques » ci-dessus ressemble à ce que l'on utilisera en OCaml.
- En C, il est valide d'annoncer qu'une fonction existera¹ en déclarant uniquement sa signature ou son prototype : `bool est_positif(int);` ou `bool est_positif(int x);`.

Définition 1 (Spécification).

La **spécification** d'une fonction est la description de ce qu'elle est censée faire, c'est à dire qu'elle indique :

- Entrées :
 - pour chacune d'entre elles, son type.
 - les éventuelles conditions supplémentaires que ces entrées doivent respecter. Par exemple : « x , un entier > 3 » ou encore « T , un tableau d'entiers triés par ordre croissant ».

De telles conditions sont appelées des **pré-conditions**.
- Sortie :
 - le type de la valeur renvoyée.
 - les conditions attendues sur la valeur renvoyée (généralement en lien avec les entrées).
 - les éventuelles modifications que fait la fonction sur « le monde extérieur », c'est à dire sur des variables (ou objets mémoire) que la fonction n'a pas créé elle-même : modification d'un argument passé par référence, modification d'une variable globale, enregistrement d'un fichier sur le disque dur, affichage sur un écran, etc. On parle d'**effets secondaires** (« *side effects* »).

On appelle **post-conditions** la donnée des conditions attendues sur la valeur renvoyée et des effets secondaires.

Remarque.

- On peut voir la spécification comme un contrat entre l'utilisateur de la fonction et celle-ci : l'utilisateur de la fonction s'engage à vérifier les pré-conditions, en échange de quoi la fonction s'engage à vérifier les post-conditions.
- Si les pré-conditions ne sont pas remplies lors d'un appel, la fonction n'est tenue à rien et peut renvoyer tout et n'importe quoi. Cependant, il est pertinent de plutôt lever un message d'erreur et interrompre le programme dans ce cas (beaucoup plus simple à déboguer!).
- Quand il n'y a pas d'effets secondaires, on ne les spécifie pas (au lieu de spécifier leur absence).
 - L'anglicisme « effets de bord » pour « effets secondaires » existe.
 - On essaye de limiter les effets secondaires au strict nécessaire, et on veille à toujours être exhaustif quand on les liste. Il s'agit de l'une des sources de bogues les plus communes!

1. On s'engage à en fournir le code source tôt ou tard.

Exemple. La fonction `max` est bien spécifiée et respecte cette spécification. par contre, `apres_moi_le_deluge` a des effets secondaires mal spécifiés... et pire encore : tout à fait évitables!!

```

4  /** Maximum de deux entiers
5   * Entrées : a et b deux entiers relatifs
6   * Sortie : le maximum de a et b
7   */
8  int max(int a, int b) {
9      if (a > b) {
10         return a;
11     }
12     else {
13         return b;
14     }
15 }
16
17 /** Maximum d'un tableau
18 * Entrées : T un tableau de L entiers
19 *           L un entier positif
20 * Sorties : le maximum de T
21 * Eff. Sec. : euh?...
22 */
23 int apres_moi_le_deluge(int L, int T[]) {
24     int i = 1;
25     while (i < L) {
26         T[i] = max(T[i-1], T[i]);
27         i = i+1;
28     }
29     // à la fin, T[l-1] = max(T)
30     return T[L-1];
31 }

```

Exercice. Parmi les 4 fonctions `maxi_vX` ci-dessous, lesquelles respectent leur spécification?

```

34 /** Maximum d'un tableau
35 * Entrées : n un entier >= 1
36 *           T un tableau de n
37 *           ↪ entiers
38 * Sortie : un élément maximal
39 *           ↪ de T
40 */
41 int maxi_v0(int n, int T[]) {
42     int sortie = T[0];
43     int i = 0;
44     while (i < n) {
45         if (T[i] > sortie) {
46             sortie = T[i];
47             i = i+1;
48         }
49     }
50     return sortie;
51 }

```

```

51 /** Maximum d'un tableau
52 * Entrées : n un entier >= 1
53 *           T un tableau de n
54 *           ↪ entiers
55 * Sortie : un élément maximal
56 *           ↪ de T
57 */
58 int maxi_v1(int n, int T[]) {
59     int sortie = 666;
60     int i = 0;
61     while (i < n) {
62         sortie = max(sortie, T[i]);
63         i = i+1;
64     }
65     return sortie;
66 }

```

```

66  /** Maximum d'un tableau
67  * Entrées : n un entier >= 1
68  *          T un tableau de n
69  *          ↪ entiers
70  * Sortie : un élément maximal
71  *          ↪ de T
72  */
73  int maxi_v2(int n, int T[]) {
74      int sortie = T[0];
75      for (int i = 1; i < n; i =
76          ↪ i+1) {
77          sortie = max(sortie, T[i]);
78      }
79      return sortie;
80  }

```

```

79  /** Maximum d'un tableau
80  * Entrées : n un entier >= 1
81  *          T un tableau de n
82  *          ↪ entiers
83  * Sortie : un élément maximal
84  *          ↪ de T
85  */
86  int maxi_v3(int n, int T[]) {
87      int sortie = T[0];
88      int i = 0;
89      while (i < n) {
90          sortie = max(sortie, T[i]);
91          i = i+1;
92      }
93      return sortie;
94  }

```

1 Terminaison

Convention 2 (Notation prime).

Lorsque l'on analyse une boucle, si x désigne une valeur au début d'une itération, on notera x' la valeur associée en fin d'itération (c'est à dire juste avant le début de l'itération suivante). De même lorsque l'on analyse une fonction récursive : le prime désigne la valeur au début de l'appel récursif suivant.

Cette convention est personnelle, et même si elle est partagée par plusieurs de mes collègues, je vous recommande très chaudement de la rappeler en début d'une copie de concours avant de l'utiliser.

1.0 Variants et preuves de terminaison

Définition 3 (Terminaison).

On dit qu'une fonction termine si pour toutes entrées vérifiant les pré-conditions de la fonction, la suite d'instruction exécutée par la fonction lors d'un appel sur ces entrées est finie.

Remarque. En pratique, on aimerait aussi que la suite d'instruction soit finie même si, par erreur, on ne remplit pas les pré-conditions...

Définition 4 (Variant - semestre 1).

Un **variant de boucle** (TantQue ou Pour) est une quantité :

- entière
- minorée (resp. majorée) tant que l'on ne quitte pas la boucle.
- strictement décroissante (resp. strictement croissante) d'une itération sur l'autre.

Remarque. Le terme quantité est volontairement général : il peut s'agir de la valeur d'une variable, de la différence de deux variables, ou de toute autre opération savante impliquant au moins une variable.

Exemple.

```

103 int somme(int n) {
104     int s = 0;
105     int i = 0;
106     while (i < n+1) {
107         s = s+i;
108         i = i+1;
109     }
110     return s;
111 }

```



Dans la boucle `while` de fonction `somme` ci-contre, la quantité i est un variant. En effet, elle est :

- entière car i est un entier.
- strictement croissante car si l'on note i' la valeur de i à la fin d'une itération, on a $i' = i + 1$.
- majorée car d'après la condition de boucle, $i < n + 1$ (où n ne varie jamais).

Théorème 5 (Preuves de terminaison).

Toute boucle qui admet un variant termine.

Toute fonction dont toutes les boucles terminent *et* dont tous les appels de fonction terminent termine.

Démonstration. Traitons les deux points séparément :

- Considérons une boucle qui admet un variant V . Quitte à adapter la preuve, supposons ce variant str. décroissant minoré et notons m un tel minorant. Numérons les itérations de la boucle : $0, 1, \dots$ et notons V_0, V_1, \dots les valeurs du variant V au début de chacune de ces itérations. Comme la suite des V_i est strictement décroissante et que ses termes sont des entiers, on pour tout $i \in \mathbb{N} : V_i - 1 \geq V_{i+1}$. Il s'ensuit que si les itérations existent au moins jusqu'au rang $i = (V_0 - m) + 1$, on a $V_{(V_0 - m) + 1} \leq m - 1$. Or cela est impossible car m est un minorant : en particulier, les itérations s'arrêtent avant d'atteindre ce rang. C'est à dire que la boucle termine.
- Les seules instructions vues jusqu'à présent susceptibles de ne pas terminer sont les boucles. Si toutes les boucles de la fonction terminent, et que toutes les fonctions appelées (donc toutes leurs) boucles terminent, la fonction termine bien.

NB : nous verrons dans quelques mois la non-terminaison des appels récursifs... mais c'est un cas particulier de la non-terminaison d'un appel de fonction. L'énoncé du théorème est donc valide.

□

Exemple. La fonction `somme` de l'exemple précédent termine.

Exemple.

```

128 int sup_log_2(int x) {
129     int n = 0;
130     int p = 1;
131     while (p < x) {
132         p = 2*p;
133         n = n+1;
134     }
135     return n;
136 }

```



Dans la boucle `while` de fonction `sup_log_2` ci-contre, la quantité $V = x - p$ est un variant. En effet, elle est :

- entière car x et p sont des entiers.
- strictement décroissante. En effet, comme $p > 0$ (il l'est au début de la première itération et n'est ensuite que multiplié par 2), on a $p' > p$. Or, $x' = x$, donc $V' < V$.
- minorée car d'après la condition de boucle, $0 < x - p$.

Donc la boucle termine. Comme la fonction n'a pas d'autres boucles ni d'appels de fonctions, elle termine.

Profitions de cet exemple pour mentionner une autre façon de comprendre $\lceil \log_2 x \rceil$: c'est le nombre de fois qu'il faut diviser x par 2 pour atteindre 1 ou moins. C'est donc réciproquement l'exposant de la plus petite puissance de 2 qui atteint x ou plus.

Remarque.

- Étant donné une fonction, il peut parfois être impossible de prouver qu'elle termine et impossible de prouver qu'elle ne termine pas. On termes **savants** de MPI, on dit que « le problème de l'arrêt n'est pas décidable ».
- Ce n'est pas un problème qui se pose à notre modeste niveau.
- Les preuves de variant vous rappellent peut-être les récurrences mathématiques : il y a un lien profond entre les deux que nous apercevrons plus en détails au second semestre.

1.1 Cas particulier des boucles Pour

Pour étudier la terminaison d'une boucle Pour, on la dépie ainsi :

<pre> 1 pour <i>i</i> allant de 0 inclu à <i>n</i> exclu faire 2 corpsDeLaBoucle </pre>	\longrightarrow	<pre> 1 <i>i</i> ← 0 2 tant que <i>i</i> < <i>n</i> faire 3 corpsDeLaBoucle 4 <i>i</i> ← <i>i</i> + 1 </pre>
---	-------------------	---

C'est à dire que l'on se ramène à la boucle `while` dont le `for` est un raccourci. On fait bien attention au fait que :

- Le compteur de boucle (*i* ci-dessus) est initialisé juste avant la première itération.
- La mise à jour du compteur a lieu à la toute fin d'une itération.

Remarque.

- Parfois, la mise à jour du compteur n'est pas une simple incrémentation de 1. Ça ne change rien, on procède pareil.
 - Dans certains langages (dont le C), il est possible que le corps de la boucle Pour modifie le compteur. C'est une mauvaise pratique qu'il ne faut pas faire : il faut utiliser un `while` dans ce cas !
 - Je parlerais de « **vraie boucle Pour** » pour désigner une boucle où le compteur n'est pas modifié par le corps, où sa mise à jour est une incrémentation d'une quantité constante (pas forcément 1), et où le compteur est borné par une quantité fixée (cohérente avec la monotonie du compteur).
- Ce n'est pas un terme canonique, et je vous conseille de le redéfinir si vous souhaitez l'utiliser.

Proposition 6 (Terminaison des vraies Pour).

Une « vraie boucle Pour » admet toujours un variant. En particulier, elle termine.

Démonstration. D'après la définition de vraie boucle Pour, le compteur est un variant immédiat. □

Exemple.

```

113 int sommebis(int n) {
114     int s = 0;
115     for (int i = 0; i < n+1;
116         i = i+1) {
116         s = s+i;
117     }
118     return s;
119 }
```



La boucle pour de fonction `sommebis` ci-contre est une « vraie boucle Pour », c'est à dire que son compteur *i* est entier, strictement croissante (car $i' = i + 1$) et borné par $n + 1$ constant. La boucle termine donc.

Puisqu'il n'y a pas d'autres boucles ni d'appels de fonctions, il s'ensuit que `sommebis` termine.

1.2 Fonctions récursives

Pour prouver la terminaison d'une fonction récursive, on doit montrer que la suite d'appels récursifs termine. Pour cela, on utilise un **variant d'appels récursifs** : c'est comme un variant, sauf qu'on demande la stricte monotonie non pas d'une itération à l'autre mais d'un appel à l'autre. La minoration correspond en général au cas de base de la fonction récursive.

Exemple.

```

173  /** Renvoie x^n
174   * Entrée : x un entier
175   *          n entier >= 0
176   * Sortie : x^n
177   */
178  int exp_rap(int x, int n) {
179      assert(n >= 0);
180
181      /* Cas de base */
182      if (n == 0) { return 1; }
183      else if (n == 1) { return x; }
184
185      /* Sinon, cas récursifs */
186      int x_puiss_demi = exp_rap(x, n/2);
187      if (n % 2 == 0) {
188          return x_puiss_demi * x_puiss_demi;
189      } else {
190          return x * x_puiss_demi * x_puiss_demi;
191      }
192  }
```

Prouvons que cette fonction récursive termine. Commençons par remarquer qu'elle ne contient ni boucles ni appels à une autre fonction : il suffit de montrer que la suite d'appels récursifs termine. Pour cela, prouvons que la quantité n est un variant d'appels récursifs :

- n est un entier (ligne 179).
- n est minoré par 0 d'après les pré-conditions. Notons que si n atteint 0 ou 1 alors la fonction termine aux lignes 183-184.
- n décroît strictement d'un appel au suivant. d'après le code et le point précédent la fonction peut s'appeler récursivement si $n > 2$. Dans ce cas, elle s'appelle récursivement avec l'argument n' qui vaut $\frac{n}{2}$. On a bien $n' < n$.

Comme la suite d'appels récursifs admet un variant, elle termine, et d'après l'analyse initiale la fonction `exp_rap` termine.

Remarque.

- Dans cet exemple, le variant est un argument. Mais cela peut aussi être la différence de plusieurs arguments (par exemple pour une dichotomie) ou le nombre d'éléments d'une liste.
- Au second semestre, nous verrons les *inductions structurelles* qui permettent souvent de prouver une terminaison récursive de manière plus proche du code, et donc plus simple à comprendre.
- Il est souvent pertinent de faire la minoration avant la stricte décroissance. Par exemple, dans cette preuve, cela m'a permis d'utiliser implicitement le fait que $n \geq 0$ ce qui était nécessaire pour conclure que $n' < n$.
- S'il y a plusieurs appels récursifs, il faut prouver la terminaison de chacun d'entre eux et donc prouver la stricte monotonie pour chacun d'entre eux.

2 Correction

2.0 Correction partielle

Définition 7 (Correction partielle).

On dit qu'une fonction est **partiellement correcte** si pour toutes entrées vérifiant les pré-conditions et sur lesquelles la fonction termine, le résultat de l'appel de la fonction vérifie les post-conditions.

Remarque.

- C'est équivalent à « pour toutes entrées vérifiant les pré-conditions, soit l'appel de la fonction ne termine pas soit son résultat vérifie les post-conditions ».
- Comme pour la terminaison, on voudrait que la fonction soit bien élevée même lorsque ses pré-conditions ne sont pas remplies : le mieux est que dans ce cas elle termine rapidement en renvoyant un message d'erreur (et n'ait pas d'effets secondaires !!).

2.0.0 Sauts conditionnels

Pour prouver la correction partielle d'une fonction qui ne contient que des déclarations/modifications de variables et des sauts conditionnels, il suffit de faire une disjonction de cas sur les `if-then-else`.

Exemple. Prouvons la correction partielle de la fonction ci-dessous :

```

4  /** Maximum de deux entiers
5   * Entrées : a et b deux entiers relatifs
6   * Sortie : le maximum de a et b
7   */
8  int max(int a, int b) {
9      if (a > b) {
10         return a;
11     }
12     else {
13         return b;
14     }
15 }
```



Notons tout d'abord que comme annoncé, la fonction n'a pas d'effets secondaires (les deux arguments sont passés par valeur). Pour prouver la post-condition de la sortie, procédons comme le code (ligne 9) par disjonction de cas :

- Si $a > b$, on entre dans le `then` ligne 10 : on renvoie a , qui est bien $\max(a, b)$ comme annoncé.
- Sinon, on entre dans le `else` (ligne 13) : on renvoie b , qui est bien $\max(a, b)$ comme annoncé.

La fonction est donc partiellement correcte.

Remarque. Il est rare que, comme dans cet exemple, les sauts conditionnels demande la majeure partie du travail de la preuve : c'est généralement un point facile à prouver sur lequel on passe rapidement pour ne pas alourdir la rédaction.

2.0.1 Invariants de boucle

Convention 8 (Somme vide).

Lorsque l'on écrit $\sum_{k=a}^b \dots$, on considère que si $a > b$ alors la somme est vide est vaut 0. De même avec le symbole produit \prod : unproduit vide vaut 1.

Définition 9 (Invariant de boucle).

Un **invariant de boucle** (TantQue ou Pour) est une propriété logique qui :

- est vraie juste avant la toute première itération de la boucle (on appelle cela l'**initialisation**).
- si elle est vraie au début d'une itération, est vraie à la fin de cette même itération (on parle de **conservation**).

Remarque.

- Un invariant, comme un variant, doit impliquer des variables de la fonction pour être utile.
- C'est en fait une récurrence ! L'avantage de ce formalisme-ci par rapport à une récurrence sur les itérations est que l'on a pas besoin d'introduire un numéro d'itération superflu. L'autre avantage est que l'on compare le début d'une itération à la fin de celle-ci, là où une récurrence comparerait au début de l'itération suivante... laquelle n'existe peut-être pas.
- Comme pour les preuves de terminaison, on utilise la notation prime pour marquer une quantité en fin d'itération.
- Pour prouver la conservation dans une boucle `while`, il est souvent utile d'utiliser le fait que l'on est encore en train d'itérer la boucle et que donc la condition de la boucle est vraie : elle est une hypothèse que l'on peut appliquer.
- Une très bonne pratique est de ne pas seulement prouver que quelque chose est un invariant, mais aussi de préciser ce qu'il prouve à la fin de la dernière itération. On nomme cette étape la **conclusion**.

Exemple.

```

98  /** Somme des n premiers entiers.
99  * Entrées : n >= 0
100 * Sortie : somme des n premiers entiers de N*.
101 * Eff. Sec : aucun.
102 */
103 int somme(int n) {
104     int s = 0;
105     int i = 0;
106     while (i < n+1) {
107         s = s+i;
108         i = i+1;
109     }
110     return s;
111 }
```



Montrons que $I: \langle s = \sum_{k=0}^{i-1} k \rangle$ est un invariant.

- **Initialisation** : avant la première itération de la boucle, on a $s = 0$. On a également $i - 1 < 0$, d'où la somme de I est vide et vaut donc 0. On a bien $0 = 0$: l'invariant est initialisé.
- **Conservation** : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que $I' : \langle s' = \sum_{k=0}^{i'-1} k \rangle$.
On a :

$$\begin{cases} s' = s + i & \text{(ligne 108)} \\ i' = i + 1 & \text{(ligne 109)} \end{cases}$$

Or d'après I , on a $s = \sum_{k=0}^{i-1} k$. Donc :

$$\begin{aligned}
s' &= s + i \\
&= \left(\sum_{k=0}^{i-1} k \right) + i && \text{d'après } I \\
&= \sum_{k=0}^i k \\
&= \sum_{k=0}^{i'-1} k && \text{car } i' = i + 1
\end{aligned}$$

On a prouvé que I' est vrai : l'invariant se conserve.

- Conclusion : On a prouvé que I est un invariant. En particulier, comme à la fin de la dernière itération on a $i' = n + 1$, on a en sortie de boucle :

$$\begin{aligned}
s &= \sum_{k=0}^{n+1-1} k \\
&= \sum_{k=0}^n k
\end{aligned}$$

Remarque.

- J'ai volontairement beaucoup détaillé les étapes de calcul de cette preuve, afin de limiter les difficultés calculatoires. On aurait pu aller plus vite.
- Toutefois, je vous demande de ne pas aller plus vite que moi sur la phrase d'introduction de la conservation : « *supposons l'invariant [...] vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que [...]'* ». En effet, je veux que vous ayez chacun des éléments de cette phrase sous les yeux avant de vous lancer dans la conservation en elle-même.²

Proposition 10.

Pour prouver la correction partielle d'une fonction qui contient des boucles, on fait appel à des invariants de boucle afin de prouver qu'une certaine propriété est vraie au moment de la sortie de la boucle.

Autrement dit, les invariants jouent le rôle d'un lemme : ils *ne* prouvent *pas* la correction partielle, mais on espère pouvoir utiliser leur résultat pour en déduire la correction partielle.

Exemple. Dans l'exemple précédent, l'invariant choisi prouve qu'en sortie de boucle, on a $s = \sum_{k=0}^n k$. Comme s n'est pas modifié entre la sortie de la boucle et le `return s`, on peut conclure que la sortie renvoyée vérifie la condition annoncée. De plus, il n'y a pas d'effets secondaires (une seule entrée, passée par valeur) comme annoncés : les post-conditions sont vérifiées, donc la fonction est partiellement correcte.

Exemple. Prouvons la correction partielle de la fonction ci-dessous. On utilisera la propriété suivante pour qualifier $\lceil \log_2 x \rceil$ (avec $x \geq 1$) : c'est l'unique entier ℓ tel que $2^{\ell-1} < x \leq 2^\ell$.

2. Comme, peut-être, lors de votre apprentissage des récurrences en Terminale.

```

122  /** Partie entière supérieure de log2(x)
123   * Entrées : x entier >= 1
124   * Sortie : la partie entière supérieure du
125   *          logarithme en base 2 de x
126   * Eff. Sec : aucun.
127   */
128  int sup_log_2(int x) {
129      int n = 0;
130      int p = 1;
131      while (p < x) {
132          p = 2*p;
133          n = n+1;
134      }
135      return n;
136  }

```

On veut prouver l'encadrement voulu pour la ligne 136, c'est à dire en fin de boucle while.

Puisque l'on a quitté la boucle, on a alors $p \geq x$. En déroulant les étapes à la main, on se rend compte que I : « $p = 2^n$ » est un invariant³ : nous allons donc le prouver. Nous aurons aussi besoin de l'invariant J : « $\frac{p}{2} < x$ » qui donnera l'autre moitié de l'encadrement voulu⁴.

- Initialisation : avant la première itération, on a $p = 1$ et $n = 0$, or $1 = 2^0$ donc J est bien initialisé. De plus, d'après les pré-conditions on a $x \geq 1$. Comme $\frac{p}{2} = \frac{1}{2} < 1$, on a bien l'initialisation de I .
- Conservation : supposons les invariants I et J vrais au début d'une itération quelconque et montrons-les vrais à la fin de cette itération, c'est à dire montrons que I' : « $p = 2^{n'}$ » et que J' : « $\frac{p'}{2} < x$ ».

On a :

$$\begin{cases} p' = 2p & \text{(ligne 133)} \\ n' = n + 1 & \text{(ligne 134)} \end{cases}$$

En appliquant I dans la définition de p' , on obtient :

$$p' = 2 \times 2^n = 2^{n+1} = 2^{n'}$$

C'est à dire I' . Pour prouver J' , utilisons le fait qu'au début de chaque itération on a $p < x$. Donc d'après la définition de p' :

$$\frac{p'}{2} = p < x$$

C'est à dire J' . Les deux invariants se conservent.

- Conclusion : On a prouvé que I et J sont des invariants. En particulier, en sortie de boucle on a $p = 2^n$ et $x > \frac{p}{2} = 2^{n-1}$.

L'évaluation de J en sortie de boucle nous donne la minoration de l'encadrement attendu. Pour obtenir la majoration, remarquons qu'on quitte la boucle car $p \geq x$, c'est à dire d'après I car $2^n \geq x$.

On a ainsi prouvé l'encadrement demandé : la fonction renvoie bien $\lceil \log_2 x \rceil$. Remarquons enfin qu'elle n'a aucun effets secondaires comme annoncé : la fonction est donc partiellement correcte.

Remarque. Les invariants peuvent aussi servir à prouver une terminaison (on prouve que la minoration/majoration du variant est un invariant) ou des non-terminaisons (on prouve que la condition d'un while est toujours fausse).

3. Qui sera très utile puisque l'on veut encadrer x par des puissances de 2 : nous avons la majoration grâce à cet invariant !

4. Celui-ci est peut-être un peu moins évident : on peut le trouver en essayant de conclure avec uniquement I , on se rend alors compte qu'il nous manque la moitié de l'encadrement mais qu'elle forme un invariant pas méchant.

2.0.2 Cas particulier des boucles Pour

Comme pour la terminaison, on déplie les boucles Pour :

<pre> 1 pour <i>i</i> allant de 0 inclu à <i>n</i> exclu faire 2 corpsDeLaBoucle </pre>	\longrightarrow	<pre> 1 <i>i</i> \leftarrow 0 2 tant que <i>i</i> < <i>n</i> faire 3 corpsDeLaBoucle 4 <i>i</i> \leftarrow <i>i</i> + 1 </pre>
--	-------------------	---

En particulier :

- Le compteur est créé juste avant la première itération, et l'initialisation des invariants se fait donc avec cette valeur.
- Le compteur est mis à jour juste avant la fin de l'itération, et donc sa valeur en fin d'itération (i') n'est plus celle en début (i).
- On quitte la boucle car le compteur dépasse : en sortie de boucle, le compteur contient la valeur mise à jour du compteur (dans l'exemple ci-dessus, i vaut n en sortie de boucle).

Attention, cette valeur mise à jour n'est pas forcément la borne du `for` : pensez par exemple à `for (int i = 0; i < 3; i = i+2)` .

On se contente généralement d'affirmer cette valeur en sortie de boucle au lieu de la prouver.

Exemple. Prouvons la correction partielle de la fonction ci-dessous :

```

147 int exp_simple(int a, int n) {
148     int p = 1;
149     for (int i = 0; i < n; i = i+1) {
150         p = a*p;
151     }
152     return p;
153 }

```



On veut prouver qu'en sortie de boucle, $p = a^n$. Pour cela, montrons que $I : \langle p = a^i \rangle$ est un invariant.

- Initialisation : Juste avant la première itération, on a $p = 1$ et $i = 0$. On a bien $1 = a^0$, d'où l'initialisation.
- Hérédité : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que $I' : \langle p = a^{i'} \rangle$.

Or :

$$\begin{aligned}
 p' &= a \times p && \text{d'après ligne 151} \\
 &= a \times a^i && \text{d'après } I \\
 &= a^{i+1} \\
 &= a^{i'} && \text{d'après ligne 150}
 \end{aligned}$$

C'est à dire I' : l'invariant est conservé.

- Conclusion : On a prouvé que I est un invariant. En particulier, comme en sortie de boucle $i = n$, on a alors :

$$p = a^n$$

C'est exactement la condition attendue sur p . De plus, la fonction n'a pas d'effets secondaires, comme annoncé. Elle est donc partiellement correcte.

2.1 Fonctions récursives

Pour les fonctions récursives, on prouve les invariants par... récurrence ! Il s'agit de prouver que la spécification est vérifiée, en supposant que l'appel récursif la vérifie déjà (hérédité). Il faut bien sûr également prouver que les cas de base la vérifient (initialisation).

Exemple.

```

173  /** Renvoie x^n
174   * Entrée : x un entier
175   *          n entier >= 0
176   * Sortie : x^n
177   */
178  int exp_rap(int x, int n) {
179      assert(n >= 0);
180
181      /* Cas de base */
182      if (n == 0) { return 1; }
183      else if (n == 1) { return x; }
184
185      /* Sinon, cas récursifs */
186      int x_puiss_demi = exp_rap(x, n/2);
187      if (n % 2 == 0) {
188          return x_puiss_demi * x_puiss_demi;
189      } else {
190          return x * x_puiss_demi * x_puiss_demi;
191      }
192  }
```

Prouvons que cette fonction est partiellement correcte. Commençons par noter qu'elle n'a pas d'effets secondaires, comme demandé.

Montrons par récurrence forte sur $n \in \mathbb{N}$ la propriété H_n : « pour tout x entier, `exp_rap(x, n)` renvoie x^n ».

- **Initialisation** : si $n = 0$ (resp. si $n = 1$), la fonction renvoie 1 qui vaut bien x^0 (resp. x qui vaut bien x^1). D'où l'initialisation.
- **Hérédité** : soit n entier strictement supérieur à 1 tel que la propriété soit vraie jusqu'au rang $n-1$. Montrons-la vraie au rang n , c'est à dire montrons que `exp_rap(x, n)` renvoie x^n .

Comme $n > 1$, l'exécution ne rentre pas dans les `if-else` (ni dans le `assert`) et on va donc en ligne 187. Par hypothèse de récurrence, `x_puiss_demi` contient $x^{\lfloor \frac{n}{2} \rfloor}$.

Or, d'après le cours de maths :

- Si n est pair, on a $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$. Donc :

$$x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} = x^n$$

- Sinon, on a $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$. Donc :

$$x \cdot x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} = x \cdot x^{n-1} = x^n$$

Or, cette disjonction de cas et ces calculs sont exactement les liens 188-192 du code : la fonction renvoie donc bien dans tous les cas x^n . D'où l'hérédité.

- **Conclusion** : On a prouvé par récurrence forte que pour tout n , pour tout x , la fonction renvoie bien le résultat annoncé.

Les post-conditions sont donc vérifiées : la fonction est partiellement correcte.

Remarque. S'il y a plusieurs appels récursifs, on doit utiliser l'hypothèse de récurrence sur chacun d'entre eux.

2.2 Correction totale

Définition 11 (Correction totale).

On dit qu'une fonction est **totale**ment **correcte** lorsqu'elle termine et qu'elle est partiellement correcte.

Exemple.

- La fonction `somme` des exemples précédents est totalement correcte : on a prouvé sa terminaison page 19 et correction aux pages 23 et 24.
- La fonction `sup_log_2` aussi : terminaison page 19 et correction page 24.
- La fonction `exp_rap` aussi : terminaison page 21 et correction page 27.
- La fonction `max` termine car elle ne contient ni boucle ni appels de fonction, et on a prouvé sa correction page 22 : elle est aussi totalement correcte.
- La fonction `patience` ci-dessous est partiellement correcte mais ne termine pas, donc n'est pas totalement correcte :

```

156  /** Renvoie true
157   * Entrée : rien
158   * Sortie : true
159   */
160  bool patience(void) {
161      while (true) {
162          int sert_a_rien = 0;
163      }
164      return true;
165  }
```

Remarque. La correction totale (que ce soit la terminaison ou la correction partielle) d'une fonction dépend de la correction totale des fonctions qu'elle appelle ! Il faut donc commencer par les prouver (ou les admettre s'il s'agit de fonctions fournies par des bibliothèques que l'on a pas codées ou par l'énoncé.)

Exercice. Prouver que la fonction `maxi_v3` ci-dessous est totalement correcte :

```

79  /** Maximum d'un tableau
80   * Entrées : n un entier >= 1
81   *          T un tableau de n entiers
82   * Sortie : un élément maximal de T
83   */
84  int maxi_v3(int n, int T[]) {
85      int sortie = T[0];
86      int i = 0;
87      while (i < n) {
88          sortie = max(sortie, T[i]);
89          i = i+1;
90      }
91      return sortie;
92  }
```

Chapitre 2

MÉMOIRE

Notions	Commentaires
Représentation des flottants. Problèmes de précision des calculs flottants.	On illustre l'impact de la représentation par des exemples de divergence entre le calcul théorique d'un algorithme et les valeurs calculées par un programme. Les comparaisons entre flottants prennent en compte la précision.


Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Utilisation de la pile et du tas par un programme compilé.	On présente l'allocation des variables globales, le bloc d'activation d'un appel.
Notion de portée syntaxique et durée de vie d'une variable. Allocation des variables locales et paramètres sur la pile.	On indique la répartition selon la nature des variables : globales, locales, paramètres.
Allocation dynamique.	On présente les notions en lien avec le langage C : malloc et free [...]

Extrait de la section 5.1 du programme officiel de MP2I : « Gestion de la mémoire d'un programme ».

SOMMAIRE

0. Représentation des types de base	30
0. Notion de données	30
1. Booléens	30
2. Différents types d'entiers	31
<i>Entiers non-signés (p. 31). Caractères (p. 33). Entiers signés (p. 33). Dépassements de capacité (p. 35).</i>	
3. Nombres à virgule (flottante)	36
<i>Notation mantisse-exposant pour l'écriture scientifique (p. 36). Erreurs d'approximation et autres limites (p. 37).</i>	
4. Tableaux	38
5. Chaînes de caractères	39
1. Compléments très brefs de programmation.....	40
0. Notion d'adresse mémoire	40
1. Portée syntaxique et durée de vie	40
<i>Définition et premiers exemples (p. 40). Exemples avancés (p. 43).</i>	
2. Organisation de la mémoire d'un programme	44
0. Bloc d'activation	44
1. Organisation de la mémoire virtuelle d'un programme	45
<i>Pile mémoire (p. 46). Tas mémoire (p. 46). À savoir faire (p. 47).</i>	

 Ce chapitre, surtout sa dernière section, est rempli de simplifications.

0 Représentation des types de base

Dans cette section, nous allons voir comment les données (entiers, nombres à virgule, etc) sont représentées en mémoire. Nous n'allons *pas* voir où ces représentations sont rangées en mémoire.

Ce qui est vu dans cette section est valable en C et en OCaml.

0.0 Notion de données

Définition 1 (Donnée informatique).

Une donnée informatique est un élément fini d'information que l'on peut lire, stocker transmettre.

Définition 2.

- L'**unité élémentaire** de stockage des données est le **chiffre binaire (bit en anglais)**. Il peut prendre deux valeurs : 0 et 1.
- Un groupe de 8 bits est appelé un **octet**. Les ordinateurs modernes ne stockent pas les bits uns par uns dans la mémoire mais des octets : on dit que c'est le **plus petit adressable (byte en anglais)**.
- Les processeurs modernes, pour aller plus vite, ne font pas leurs calculs sur des octets mais sur un des groupes plus gros. On parle de **mot machine**, et ils font aujourd'hui souvent 8 octets / 64 bits

Remarque. Le terme du « plus petit adressable » provient du fait que puisque les bits sont stockés 8 par 8, on ne peut pas accéder à un des bits de l'octet sans avoir lu en même temps les autres. C'est une question de câblage électronique : les « fils » ne ciblent pas chaque bit individuellement mais des groupes de 8 bits adjacents.

Remarque. Des (suites d')octets en eux-mêmes ne veulent rien dire. Il faut connaître la façon de les déchiffrer. C'est le rôle d'un **type de données** : c'est une description de comment encoder et décoder des données d'une certaine nature. Vous connaissez déjà des types : `int` , `bool` .

0.1 Booléens

Le type booléen permet de stocker les deux valeurs logiques : Vrai (« true ») et Faux (« false »).

Proposition 3 (Encodage des booléens).

Il suffit en théorie d'1 bit pour encoder un booléen.

En C, on utilise un plus petit adressable (1 octet).

En OCaml, un mot machine (8 octets).

Remarque.

- En OCaml, *tout* est manipulé à travers un mot machine, dont les booléens.
- En C comme en OCaml, il existe des astuces pour stocker un booléen dans chacun des bits d'un octet / mot machine.

Définition 4 (Opérateurs logiques).

Voici les opérations usuelles des booléens :

x	false	true
Non(x)	true	false

(a) NON(x)

x \ y	false	true
false	false	false
true	false	true

(b) ET(x,y)

x \ y	false	true
false	false	true
true	true	true

(c) OU(x,y)

x \ y	false	true
false	false	true
true	true	false

(d) XOR(x,y)

FIGURE 2.1 – Tables (à double entrée sauf pour la négation) des opérations logiques usuelles.

Pour permettre d'alléger les écritures de formules logiques, on se donne des règles de priorité : NON est plus prioritaire que ET qui est plus prioritaire que XOR qui est plus prioritaire que OU.

Exemple. « NON false ET false OU true » s'évalue à true, car cette expression se parenthèse ainsi d'après les règles : « ((NON false) ET false) OU true »

Exercice. Évaluez NON (NON false ET NON true OU NON false ET true).

0.2 Différents types d'entiers

Il faut distinguer deux grandes familles de types d'entiers :

- les **entiers non-signés** sont des types qui permettent de manipuler uniquement des entiers positifs.
- les **entiers signés** sont des types qui permettent de manipuler des entiers relatifs.

0.2.0 Entiers non-signés

Théorème 5 (Écrire en base b).

Soit b un entier supérieur ou égal à 2 appelé **base**. Pour tout $x \in \mathbb{N}$, il existe une unique décomposition de x de la forme :

$$x = \sum_{i=0}^{N-1} c_i \cdot b^i$$

qui vérifie :

- pour tout i , $c_i \in \llbracket 0; b \rrbracket$. Les c_i sont appelés les **chiffres**.
- $a_{N-1} \neq 0$.

On appelle la donnée des c_i l'**écriture en base b** de x . Pour indiquer la base, on note généralement : ${}_b c_{N-1} c_{N-2} \dots c_1 c_0$

On dit que c_{N-1} est le **chiffre** de poids fort et que c_0 est le **chiffre de poids faible**.

Convention 6 (Base 10 implicite).

Lorsque l'on ne précise pas la base, on utilise implicitement la base 10.

Remarque.

- On parle de **binaire** pour la base 2, **décimal** pour la base 10, **hexadécimal** pour la base 16.
- En base 16, les chiffres peuvent valoir plus que 9 : on note a le chiffre correspondant au nombre décimal 10, b pour 12, ..., f pour 16.
- L'écriture $\overline{c_{N-1}c_{N-2}\dots c_1c_0}^b$ existe aussi.
- Vous êtes habitué-es à l'écriture en base 10 : $2048 = 2 \cdot 10^3 + 0 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0$.

Exemple.

$$\begin{aligned} {}^2\overline{1100\ 0010} &= 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 128 + 64 + 2 \\ &= 194 \end{aligned}$$

Définition 7 (Représentation des entiers non-signés).

Pour représenter un entier non-signé, on stocke son écriture en base b dans un ou plusieurs octets. le bit de poids faible est tout à droite. Au besoin, on complète à gauche par des 0 :

1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---

Le nombre d'octets utilisés peut être fixé par le type (4 octets pour `unsigned int` en C, 8 octets en OCaml).

Exemple. Voici l'entier 3452 représenté sur deux octets :

0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Remarque.

- Les 32 bits du type `unsigned int` C peuvent dépendre de la machine, mais 32 est la valeur la plus commune.
- En incluant `stdint.h`, on a accès en C à des types dont le nombre de bits est garanti : `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`.

Proposition 8 (Capacité des types non-signés).

Sur k bits, on peut représenter de cette façon exactement les entiers de $\llbracket 0; 2^k \rrbracket$.

⚠ Conséquence importante : on ne peut pas représenter *tous* les entiers positifs en C ni en OCaml!

Remarque. C'est long d'écrire en binaire. Pour raccourcir, on utilise souvent la base 16. En effet, 4 chiffres consécutifs en base 4 correspondent à 1 chiffre en base 16 : la traduction est donc facile¹. Convertissons par exemple ${}^2\overline{0000\ 1101\ 0111\ 1100}$

$$\begin{array}{cccc} \overline{0000} & \overline{1101} & \overline{0111} & \overline{1100} \\ \text{0} & \text{d} & \text{7} & \text{c} \end{array}$$

$$\text{Donc } {}^2\overline{0000\ 1101\ 0111\ 1100} = {}^{16}\overline{0d7c}.$$

Exercice. Réciproquement, convertissez ${}^{16}\overline{9c}$ en binaire.

1. Beaucoup plus qu'entre la base 10 et la base 2 (ou la base 16).

0.2.1 Caractères

Le type des **caractères** permet de stocker des « lettres » : une lettre de l'alphabet, un chiffre décimal, une espace, etc. Dans l'idée, il permet de stocker « une touche du clavier ».

Attention, je parle bien de lettres uniques et non de suites de lettres : ce second cas est plus compliqué, et on parle pour lui de chaînes de caractères (strings).

Définition 9 (Représentation des caractères).

Pour représenter un caractère, on le fait correspondre à un entier non-signé et on encode ensuite cet entier non-signé.

La façon la plus commune actuellement de réaliser cette correspondance est la table ASCII. Elle donne une correspondance entre 128 caractères et les 128 entiers 7-bits.

En C, le type `char` est le type des correspondances des caractères ASCII. C'est un type d'entiers non-signés, avec lequel on peut faire des calculs.

Remarque.

- ASCII signifie **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. En conséquence, les caractères qui n'intéressent pas les américains (caractères accentués par exemple) n'apparaissent pas dedans.
- Aujourd'hui, on utilise beaucoup plus de caractères que l'ASCII : nos caractères sont maintenant encodés par l'Unicode (UTF-8), qui donne des correspondances entre *beaucoup* plus de caractères et plus d'octets.
- Dans la table ASCII, les lettres de l'alphabet sont les unes après les autres (les majuscules vont de 'A'=65 à 'Z'=90, et les minuscules de 'a'=97 à 'z'=122).

Exemple. Comme les `char` sont des entiers et que les lettres se suivent, on peut écrire :

```
1 char lettre = 'a';
2 while (lettre <= 'z') {
3     fairedetrucs;
4
5     // passer à la lettre suivante
6     lettre = lettre +1;
7 }
```



0.2.2 Entiers signés

Pour représenter des entiers signés, on utilise le **complément à 2**. Pour simplifier les explications, commençons par présenter le **complément à 10**, et je noterai entre guillemets les codes (la façon de représenter les nombres) :

- 0) On fixe le nombre de chiffres que la représentation utilisera. Dans cet exemple, je travaillerai avec 3 chiffres décimaux.
- 1) La première moitié des codes possibles encodent les positifs de manière transparente :

Code	Entier représenté	Code	Entier représenté
"000"	0
"001"	1	"497"	497
"002"	2	"498"	498
...	...	"499"	499

- 2) La seconde moitié des codes encode les nombres négatifs. Pour encoder $-|x|$, on calcule $10^3 - |x|$ et on utilise le résultat comme code :

Code	Entier représenté	Code	Entier représenté
"500"	-500
"501"	-499	"997"	-3
"503"	-498	"998"	-2
...	...	"999"	-1

3) Fin.

Remarque.

- En complément à 10 sur 3 chiffres, on peut donc représenter $\llbracket -500; 500 \rrbracket$. Plus généralement, en complément à 10 sur k chiffres on peut représenter $\llbracket -10^k/2; 10^k/2 \rrbracket$.
- Le premier chiffre n'est pas un chiffre de signe au sens où il ne stocke pas le signe. Par contre, on peut en déduire le signe.
- Si l'on dispose d'un code "xyz", pour savoir s'il encode un entier positif ou strictement négatif il suffit de savoir si ce code fait partie de la première moitié des codes ou de la seconde... et donc il suffit de regarder le premier chiffre !

Et le complément à 2 ? C'est pareil mais en base 2 ! Par exemple sur 1 octet, on représente :

Code	Entier représenté	Code	Entier représenté
"0000 0000"	0	"1000 0000"	-128
"0000 0001"	1	"1000 0001"	-127
...
"0111 1110"	126	"1111 1110"	-2
"0111 1111"	127	"1111 1111"	-1

On représente souvent les compléments sous forme visuelle :

(a) Complément à 10 sur 3 chiffres

(b) Complément à 2 sur 8 chiffres

FIGURE 2.2 – Représentation visuelle des compléments

0.2.3 Dépassements de capacité

Sur des types utilisant un nombre fixé d'octets (comme en C ou en OCaml²), on ne peut donc utiliser qu'un nombre fini d'octets. Mais que se passe-t-il si, par exemple, on additionne deux nombres du type et que le résultat de l'addition est trop grand pour le type ?

Par exemple (avec des `uint8_t`), on peut poser l'addition :

$$\begin{array}{r} 1101 \quad 0011 \\ + \quad 0011 \quad 0101 \\ \hline 1 \quad 0100 \quad 1000 \end{array}$$

Le résultat de cette addition ne tient pas sur un octet. Sur la plupart des machines et des langages (dont C et OCaml), le résultat serait simplement tronqué en « enlevant ce qui déborde » et on obtiendrait ²0100 1000.

Définition 10 (Dépassement de capacité).

On parle de **dépassement de capacité (overflow)** lorsque le résultat attendu d'une opération sort de l'ensemble des valeurs représentables par son type.

On parle parfois de **surpassement** pour qualifier un dépassement « par le haut » et de **sous-passement** « par le bas ».

Proposition 11 (Gestion des dépassements entiers en C/OCaml).

Dans les deux langages au programme, les dépassements d'entiers sont gérés ainsi :

- types d'entiers non-signés : sur un type d'entiers k bits, les calculs sont effectués modulo 2^k . Cela revient à « ignorer ce qui déborde ».
- types d'entiers signés : les *encodages* sont calculés en « ignorant ce qui déborde ». Cela donne un caractère circulaire au type (cf TD).

NB : en C, cette façon de gérer n'est pas obligatoire. Elle est là sur la plupart des machines mais il est dangereux de faire un programme qui repose dessus.

⚠ Les dépassements de capacité sont une source **très** commune d'erreur. Il faut y prêter attention !

Exemple. Voici une liste d'exemples de bogues de dépassements de capacité³ :

- Explosion du vol 501 de la fusée Ariane 5, le 4 juin 1995 (à cause d'un écrêtage dans une conversion 64bits -> 16 bits et d'un mauvais choix dans le code de comment réagir à une erreur imprévue) : https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5#Conclusions.
- La machine de radiothérapie Therac-25 a causé la mort de cinq patients par violent surdosage de radiation. Parmi les causes, une variable entière qui débordait et passait sous le seuil qui déclenchait des tests. Il y a beaucoup d'autres causes, dont et surtout la façon dont le logiciel a été conçu, relu, documenté, etc : <https://fr.wikipedia.org/wiki/Therac-25#Causes>.
- Bug de l'an 2038 : dans un ordinateur, la date est stockée en nombre de secondes depuis le 1er janvier 1970. Sur les machines où cette date est stockée en complément à 2 sur 32bits (comme `int` en C...), il y aura un débordement le 19 janvier 2038 à 3h14min8s. La date deviendra alors (débordement « circulaire ») le 13 décembre 1901, 20h45 et 52s. Cf https://fr.wikipedia.org/wiki/Bug_de_l%27an_2038.
Notez que la norme demande d'utiliser un entier non signé, ce qui sur 32 bits repousse le bug à 2106.
- Et beaucoup d'autres.

2. Mais pas comme en Python

3. En réalité, un bogue n'est presque jamais uniquement dû à un dépassement de capacité. Il se combine avec d'autres manquements, que ce soit dans le code (une fonction critique ne s'assure pas du respect de ses pré-conditions), dans le processus de validation du code (le code n'a pas été assez bien testé, ni assez prouvé, ni assez bien relu) et/ou dans la gestion de l'équipe de développement.

0.3 Nombres à virgule (flottante)

On veut représenter des nombres à virgule. Remarquons tout d'abord que dans tout intervalle non-vide, il y a une quantité infinie de nombres rationnels (sans parler des réels). On ne pourra donc pas représenter tous les nombres d'un intervalle, aussi "petit" soit-il.

0.3.0 Notation mantisse-exposant pour l'écriture scientifique

Commençons à nouveau par expliquer en base 10. Un nombre $x \in \mathbb{R}$ est dit **nombre décimal** s'il existe un $p \in \mathbb{N}$ tel que $x * 10^p \in \mathbb{Z}$. Autrement dit, si il a un nombre fini de chiffres après la virgule.

Exemple.

- Tous les entiers sont décimaux.
- 0.1 est décimal car $0.1 * 10^1 = 1 \in \mathbb{Z}$.
- 0.5 est décimal car $0.5 * 10^1 = 5 \in \mathbb{Z}$.
- $\frac{1}{3}$ n'est pas décimal même si il est rationnel.
- $\sqrt{2}$ n'est pas décimal même si il est algébrique (racine d'un polynome à coefficients entiers).
- π et $\exp(1)$ ne sont pas décimaux.

Définition 12 (Écriture scientifique en base 10).

Pour tout x décimal, il existe une unique écriture de x de la forme $x = \pm m * 10^e$ avec :

- $m \in [1; 10[$ (et est décimal).
- $e \in \mathbb{Z}$.

On appelle m la mantisse et e l'exposant.

On parle parfois de **virgule flottante** pour désigner le fait qu'on « déplace » la virgule lorsqu'on passe un nombre en écriture scientifique.

Exemple.

- $243 = 2.43 * 10^2$
- $0.00759 = 7.59 * 10^{-3}$

Pour représenter un nombre décimal, on peut donc donner : le signe, la mantisse et l'exposant.

Revenons à la base 2. On va appliquer exactement cette idée : écrire en écriture scientifique, et communiquer mantisse, signe, exposant.

Définition 13 (Nombre dyadique).

Un nombre réel x est dit dyadique s'il existe un $p \in \mathbb{Z}$ tel que $x * 2^p \in \mathbb{Z}$. Autrement dit, s'il a un nombre fini de chiffres après la virgule en base 2.

Exemple.

- Tous les entiers sont dyadiques.
- 0.1 n'est **pas** dyadique.⁴
- 0.5 est dyadique car c'est 2^{-1} .
- $\frac{1}{3}, \sqrt{2}, \pi, \exp(1)$ ne sont pas dyadiques.

4. Le prouver est un petit exo de maths sympa. On peut par contre prouver que si un nombre est dyadique alors il est décimal : c'est la deuxième partie d'un petit exo sympa.

Définition 14 (Écriture scientifique en base 2).

Pour tout x dyadique, il existe une unique écriture de x de la forme $x = \pm m * 2^e$ avec :

- $m \in [1; 2[$ (et est dyadique).
- $e \in \mathbb{Z}$.

On appelle m la mantisse et e l'exposant.

L'idée est alors la suivante. On va stocker des nombres dyadiques sur 64 bits, ainsi :

- Le premier bit est un bit de signe 0 pour positif, 1 pour négatif.
- Ensuite l'exposant. On le stockera sous une forme particulière qui simplifie certains calculs.
- Enfin la mantisse. Toutefois, il est inutile de stocker le bit de poids fort de la mantisse : on sait qu'il vaut 1 puisque $m \in [1; 2[$. On stocke donc uniquement la partie après la virgule de la mantisse!⁵

On obtient le format suivant (aussi appelé **norme IEEE 754**) :

Définition 15 (Représentation des nombres dyadiques).

On encode des nombres dyadiques sur 64 bits de la façon suivante :

	signe s	exposant E	mantisse M
64 bits :	1 bit	11 bits	52 bits

Qui représente : $\pm m * 2^e$ avec :

- + si $s = 0$, - sinon.
- E code un entier non-signé sur 11 bits, et $E = e + 1023$. On représente donc les puissances $e \in \llbracket -1023; 1024 \rrbracket$. 1023 est appelé la **constante d'excentrement**.
- $m = 1, M$. C'est à dire que la mantisse stockée ne stocke que ce qu'il y a après la virgule. On a donc en réalité accès à 53 chiffres significatifs.

On parle de représentation en **double précision**, car autrefois on utilisait moitié moins de bits (32). Le double correspondant en C est **double**.

Définition 16 (Valeurs réservées).

Certaines paires de valeur (M, E) sont réservées :

- $M = 0...0$ et $E = 0...0$ encodent ± 0 . Par conséquent, $\pm \overline{1, 0...0} * 2^{-1023}$ n'est pas représentable.
- $M = 0...0$ et $E = 1...1$ encodent $\pm \infty$. Par conséquent, $\pm \overline{1, 1...1} * 2^{1024}$ n'est pas représentable.
- un autre M avec $E = 1...1$ encode NaN (Not a Number).

0.3.1 Erreurs d'approximation et autres limites

Cette représentation double précision a des limites. Elles proviennent du fait que puisque la mantisse est de taille finie, on manipule des approximations⁶.

- Propagation d'erreurs : toutes les erreurs listées ci-dessous se propagent lors des calculs, comme les approximations en physique.
- Beaucoup de nombres sont approximés et non exacts, dont tous les nombres non-dyadiques. Rappelons que 0.1 n'est pas dyadique : on en manipule donc un arrondi, ce qui mène à des erreurs. Si possible, il vaut lui préférer 0.125 qui est lui exact.
- Faux négatif sur les comparaisons : parfois, des comparaisons comme « $a+b=c$ » s'évaluent à **false** alors qu'elles sont mathématiquement justes. Un exemple classique est $0.1 + 0.2 == 0.3$.⁷

5. Ça a l'air de rien, mais on gagne ainsi 1 bit dans la mantisse, c'est à dire un chiffre significatif stockable de plus.

6. Comme en physique : si on a trop peu de chiffres significatifs, le résultat est peu précis.

7. En même temps, **aucun** de ces 3 nombres n'est dyadique. Partant de là, l'égalité eut été un coup de chance incroyable.

- Faux positif sur les comparaisons : de même.
- La somme de deux nombres représentables ne l'est pas forcément. Par exemple, 2^{30} l'est, 2^{-30} l'est, mais il faut un M de 60 chiffres pour stocker leur somme. En fait, il se passe un débordement de la mantisse : un flottant déborde en perdant de la précision.
- Non associativité : on n'a pas forcément $(a+b)+c == a+(b+c)$. Par exemple $(2^{-30} + 2^{30}) - 2^{30}$ ne renvoie pas le même résultat que $2^{-30} + (2^{30} - 2^{30})$.

Corollaire 17.

Quand on compare des flottants, on le fait toujours à epsilon près, avec epsilon la précision voulue!

Par exemple, au lieu de tester $a \neq b$, on teste $|a-b| > \epsilon$.

0.4 Tableaux

Définition 18 (Représentation des tableaux).

Soit τ un type qui se représente sur r bits. Un tableau T de longueur L dont les cases sont de type τ est représenté en mémoire comme :

Représentation de $T[0]$	Représentation de $T[1]$...	Représentation de $T[L-1]$
-----------------------------	-----------------------------	-----	-------------------------------

FIGURE 2.3 – Représentation du tableau T

Autrement dit, pour représenter T , on utilise $L \times r$ bits. Les r premiers bits stockent $T[0]$, les r suivants $T[1]$, etc. Les cases sont contiguës en mémoire.

Remarque.

- Si l'on connaît le type τ , on connaît sa taille r . Si l'on a accès à l'adresse mémoire du début D du tableau, on peut alors aisément calculer :
 - l'adresse de $T[0]$: c'est D .
 - l'adresse de $T[1]$: c'est $D + r$.
 - l'adresse de $T[2]$: c'est $D + 2r$.
 - Ainsi de suite. $T[i]$ a pour adresse $D + ir$.

C'est pour cela que les tableaux sont une structure de donnée aussi efficace : il est très rapide d'accès au i -*me* élément, car cet accès demande une unique calcul à partir de l'adresse du début et de l'indice.

Notez que pour que cette propriété soit vraie, il *faut* que toutes les cases du tableau aient la même taille! C'est pour cela que l'on autorise pas les mélanges de type dans un tableau.

- **Lorsque l'on passe un tableau à une fonction, tout se comporte comme si le contenu du tableau était passé par référence.**

En fait, le tableau est affaibli en pointeur : au lieu de passer le tableau par valeur, on passe à la place l'adresse de sa première case. D'après le point précédent, c'est exactement ce dont on a besoin. Mais puisqu'on a fourni à la fonction l'emplacement mémoire du tableau initial, les cases qu'elle modifie sont celles du tableau.

- Cette façon de manipuler des tableaux est celle de C. Elle a deux inconvénients : la longueur du tableau n'est pas stockée, et le tableau n'est pas redimensionnable (si ça se trouve, les cases mémoire qui suivent la fin du tableau sont déjà prises et on ne peut donc pas agrandir el tableau). OCaml (et Python) utilisent une structure qui permet de stocker la longueur du tableau en plus de ses valeurs⁸. Nous verrons avec les tableaux dynamiques comment crée des tableaux redimensionnables (comme en Python).

8. Dans l'idée, on crée une case fictive au début du tableau qui sert à stocker la longueur.

- Il faut que tous les éléments d'un tableau aient la même taille. En conséquence, si on veut faire un tableau de tableaux, il y a deux grandes façons de faire :
 - Imposer que tous les sous-tableaux aient la même longueur et le même type.
 - Ne pas stocker les sous-tableaux dans les cases, mais l'adresse de leur début (toutes les adresses ont la même taille). Il faut alors stocker les sous-tableaux ailleurs.

(a) Tableaux imbriqués

(b) Tableaux d'adresses de tableaux

FIGURE 2.4 – Les deux grandes façons de faire un tableau de tableaux.

0.5 Chaînes de caractères

Définition 19 (Chaînes de caractères).

Une chaîne de caractère est une succession de caractères, c'est à dire un texte.

On les représente généralement comme une succession de caractères contigus en mémoire. On utilise un caractère particulier, `\0` qui marque la fin (et ne veut rien dire d'autre) :

Premier caractère	Second caractère	...	Dernier caractère	<code>\0</code>
----------------------	---------------------	-----	----------------------	-----------------

Remarque. Ceci est la façon de faire de C (qui a l'avantage de la simplicité). Là aussi, on peut stocker la longueur de la chaîne en dur, et/ou faire une représentation plus compliquée qui permet plus d'opérations.

1 Compléments très brefs de programmation

1.0 Notion d'adresse mémoire

La mémoire d'un ordinateur peut être pensée comme un « très gros tableau » : c'est une succession de cases mémoires (dont la taille est un plus petit adressable), ayant chacune un indice. Celui-ci est appelé une **adresse**.

Autrement dit, une adresse indique où est rangée une donnée en mémoire.

Définition 20 (Pointeur).

Une variable qui stocke une adresse est appelée un **pointeur**.

En C, si le contenu de la case mémoire pointée est de type `type`, le type du pointeur sera `type*`.

Nous reparlerons de cette notion en TP. Tout ce qu'il faut retenir pour l'instant, c'est qu'en plus d'utiliser des variables, on peut manipuler des adresses mémoire explicites.

On parle alors d'**objet mémoire** pour désigner quelque chose qui a un contenu et une adresse.

1.1 Portée syntaxique et durée de vie

On a déjà vu comment les fonctions, if-then-else et boucles structurent un code source en différents corps (le corps de la fonction, le corps du if, etc). les lignes qui appartiennent exactement aux mêmes corps sont appelés un **bloc syntaxique**.

Remarque.

- Cette appellation provient de la *syntaxe* : un corps est délimité par quelque chose qui indique son début et quelque chose qui indique sa fin. En C, ces indicateurs sont les `{` et `}`.
- Pour les fonctions, il y a une subtilité : il faut considérer que la déclaration des arguments de la fonction est aussi dans le bloc syntaxique.
- Il y a une notion d'imbrications : par exemple, si une boucle est dans une fonction, on dira que le corps de la boucle est imbriqué dans le bloc de la fonction.

1.1.0 Définition et premiers exemples

Définition 21 (Portée).

La **portée** d'un identifiant (c'est à dire d'un nom de variable ou de fonction) est l'ensemble des endroits où cet identifiant est callable.

Proposition 22 (Portée locale vs globale).

Il y a deux types de portées :

- **locale** : si un identifiant est créé dans un bloc syntaxique, il n'est utilisable qu'à partir de sa création jusqu'à la fin de son bloc syntaxique. On peut l'utiliser dans des blocs qui sont imbriqués dans son corps.
- **globale** : si un identifiant est créé en dehors de tout bloc syntaxique (c'est à dire en dehors de toute fonction), il est utilisable de sa déclaration à la fin du fichier (peu importe les blocs).

Exemple.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int somme = -10;
6  int Lmax = 100000;
7
8
9  int somme_tableau(int T[], int len) {
10     if (len > Lmax) {
11         // faire planter le programme
12         exit(EXIT_FAILURE);
13     }
14
15     int somme = 0;
16     int indice = 0;
17     while (indice < len) {
18         int tmp = 666;
19         somme = somme + T[indice];
20         indice = indice + 1;
21     }
22
23     return somme;
24 }
25
26
27 int main(void) {
28     int T[] = {5, 85, -20};
29     int sortie_somme = somme_tableau(T, 3);
30     printf("variable globale somme : %d\n", somme);
31     printf("sortie de somme_tableau : %d\n", sortie_somme);
32
33     return EXIT_SUCCESS;
34 }

```

Dans l'exemple ci-dessus :

- Dans l'exemple ci-dessus, le bloc syntaxique du `while` de `somme_tableau` commence ligne 18 et se finit ligne 20. Ainsi, la variable `foo` n'est utilisable que entre ces lignes : c'est sa portée.
- Celui de la fonction `somme_tableau` commence ligne 9 (déclaration des arguments) et finit ligne 24. La portée de `indice` est donc de la ligne 16 à la ligne 24. On peut bien utiliser cette variable dans le bloc du `while` puisque celui-ci est imbriqué dans le bloc de la fonction.
- La variable `Lmax` a une portée globale (ligne 6 jusqu'à la fin). On peut donc l'utiliser ligne 10 (dans le bloc de la fonction).

Définition 23 (Masquage).

On parle de **masquage** lorsqu'au sein de la portée d'un identifiant, on redéfinit une nouvelle variable/fonction qui a le même identifiant. L'identifiant d'origine n'est alors plus accessible tant que l'on est dans la portée du nouveau : seul le plus « récent » est utilisable.

Exemple. Dans l'exemple précédent, il y a un masquage du `somme` global (ligne 5) par le `somme` local (ligne 15). On ne peut pas accéder au premier tant que l'on est dans la portée du second (ligne 15-24).

Définition 24 (Durée de vie).

La **durée de vie** d'un objet mémoire (par exemple une variable) est l'ensemble des endroits où cet objet existe et a une valeur définie en mémoire. Cela ne coïncide pas nécessairement avec la portée.

Exemple.

- Dans l'exemple précédent, durant le masquage, on est bien dans la durée de vie du `somme` global (il a toujours sa même valeur en mémoire, il n'a pas été supprimé). Par contre, on est pas dans sa portée à cause du masquage.
- En C, lorsque l'on crée un tableau mais que l'on ne l'a pas encore initialisé (par exemple : `double T[10]`; crée un tableau de 10 nombres à virgule sans l'initialiser), on est dans sa portée puisque l'identifiant a été créé, mais pas dans sa durée de vie puisque le contenu du tableau n'a pas de valeur définie en mémoire.

Proposition 25 (Durée de vie statique, automatique et allouée).

Il y a trois durées de vie possibles :

- **statique** : l'objet est toujours défini en mémoire. C'est le cas des variables globales. Leur valeur est stockée dans une zone particulière de la mémoire, où elles existent dès le début du fichier (avant le début de leur portée, donc).
- **automatique** : l'objet n'existe en mémoire que pendant un bloc. C'est le cas des variables locales : elles "naissent" au début de leur bloc syntaxique et "meurt" automatiquement à la fin de celui-ci.
- **allouée** : on contrôle à la main la "naissance" et la "mort" d'un objet. Cela se fait en C via les fonctions `malloc` et `free`.

Exercice. Dans l'exemple `portee.c` précédent, indiquer des variables statiques et des variables automatiques.

Remarque.

- Si on oublie de libérer la mémoire d'un objet alloué, on provoque une fuite de mémoire : la zone restera allouée jusqu'à la fin du programme, même si on ne s'en sert plus.⁹
- Certains langages de programmation ne proposent pas de contrôler à la main la "mort" des objets de durée de vie allouée. À la place, ils essaient de détecter le moment à partir duquel ils ne seront plus utilisés (et les suppriment alors). On parle de **ramasse-miette**. C'est ce que font OCaml et Python. Cela a l'avantage d'éviter les erreurs du programmeur liée à la mort des objets, mais a l'inconvénient de ralentir le programme (car il faut détecter si une variable servira encore ou non).

Proposition 26.

Les objets à durée de vie automatique ont une propriété de type LIFO (*Last In First Out*) : ce sont ceux du dernier bloc ouvert qui seront supprimés en premier.

Cette propriété provient du fait que les blocs eux-mêmes vérifient cette propriété : le bloc qui a été ouvert le plus récemment est le prochain à fermer. Vous pouvez y penser comme à des parenthèses : quand vous croisez une parenthèse fermante, elle ferme la parenthèse encore ouverte la plus récente.

⁹. Pire encore : même si on ne peut plus libérer la mémoire, car cette libération avait besoin d'une variable locale à laquelle nous n'avons plus accès.

1.1.1 Exemples avancés

Avec les objets alloués, on peut créer d'autres exemples de désynchronisation entre la durée de vie et capacité à accéder au contenu :

```
1 int* ptr = (int*)
  ↪ malloc(<taille>);
2 free(p);
3 int x = p[0];
```

Lors de la ligne 4 de ce code, l'objet alloué n'existe plus mais on peut encore y accéder via `p`.

⚠ Attention, c'est l'objet pointé par `p` qui est mort. `p`, lui, va très bien et contient toujours la même adresse (sauf qu'il n'y a plus rien à cette adresse).

```
1 int* zombi(void) {
2     int var = 42;
3     return &var;
4 }
```

Cette fonction renvoie l'adresse d'une variable locale. Or celle-ci meurt à la fin de la fonction : on récupère donc un pointeur sur de la mémoire morte.

```
1 void fuite(unsigned int n) {
2     malloc(n*sizeof(char));
3     return;
4 }
```

Après un appel à cette fonction, l'objet mémoire créé existe toujours mais on n'a plus aucun moyen d'y accéder. Il est même impossible de libérer cette mémoire car on n'a pas mémorisé le pointeur qui y mène...

```
1 double* zombieRetour(void) {
2     double tab[5] = {3.14, -4.2,
  ↪     2.71, 5.0};
3     return tab;
4 }
```

Idem qu'à gauche, sauf qu'ici ça se voit moins (surtout si on a fait du Python). Le tableau est affaibli en pointeur lorsqu'on le renvoie, c'est donc exactement la même situation.

Pour enfoncer le clou dans le cadavre des zombies, considérons le code ci-dessous et demandons au compilateur son avis :

```
1 int main(void) {
2     double* tab = zombieRetour();
3     return EXIT_SUCCESS;
4 }
```

Ligne de compilation et résultat :

```
1 MP2I/Cours/Memoire: gcc error.c -Wall -Wextra
2 error.c: Dans la fonction « zombieRetour »:
3 error.c:5:10: attention: la fonction retourne l'adresse d'une variable
  ↪ locale [-Wreturn-local-addr]
4     5 |     return tab;
5       |           ^~~
```

Voilà. Tout est dit.

2 Organisation de la mémoire d'un programme

Dans cette section, nous allons voir comment est « rangée » la mémoire dans le langage C : où vont les octets des différentes variables d'un programme.

Pour OCaml, c'est plus compliqué : la notion de pile d'appels de fonction reste valide, mais le reste est plus différents (pour faire fonctionner le ramasse-miette).

2.0 Bloc d'activation

Vous vous encourage très fortement à aller jouer avec le site ci-dessous pour visualiser la mémoire et les blocs :

<https://pythontutor.com/c.html#mode=edit>

Définition 27 (Bloc d'activation).

Les variables créées par un même bloc syntaxique sont stockées les unes après les autres en mémoire. L'espace mémoire servant à stocker ces variables est appelé **bloc d'activation**.

Il contient donc les variables créées dans ce bloc syntaxique (mais pas dans ses blocs imbriqués).

Rappelons que si le bloc syntaxique est un bloc de fonction, les arguments en font partie.

Exemple.

```

1  int f(int a, int b) {
2      int c = 10;
3      int i = 0;
4      while (i < c) {
5          int d = 42;
6          int c = 50;
7          i = i+1;
8      }
9      int e = 2;
10     return c;
11 }
```

 bloc.c

Le bloc d'activation de la fonction contient un sous-bloc d'activation (boucle for) qui s'étend des lignes 3 à 6.

On peut représenter ces deux blocs sur un schéma où on les « empile » :

FIGURE 2.5 – Blocs d'activation de `f`

Remarque.

- Un appel de fonction ouvre un nouveau bloc (celui du "code" de la fonction appelée).
- Beaucoup de langages ne font pas un bloc d'activation par bloc syntaxique mais simplement un gros bloc d'activation pour toute la fonction.

2.1 Organisation de la mémoire virtuelle d'un programme

On parle de **mémoire virtuelle** (abrégié VRAM) pour désigner la mémoire telle que le système d'exploitation la présente à un programme. C'est une version « mieux rangée » de la mémoire réelle, que le système d'exploitation s'occupe de faire correspondre avec la mémoire réelle.¹⁰

Tout comme la mémoire réelle, la mémoire virtuelle est un « tableau géant » indicé par des adresses. Du point de vue d'un programme, la mémoire virtuelle ressemble à ceci :

FIGURE 2.6 – Mémoire virtuelle vue par un processus.

Légende :

- *Code* : c'est là où les instructions du programme (son... code) sont stockées.
- *Données statiques initialisées* : une donnée statique est une donnée qui est connue à la compilation. Les données statiques initialisées sont les variables globales, ainsi que le contenu des chaînes de caractères¹¹
- *Données statiques initialisées* : hors-programme.
- *Pile mémoire* : c'est là que sont stockés les différents blocs d'activation en cours, empilés les uns après les autres.
- *Tas mémoire* : là où on stocke les objets à durée de vie allouée.

Ces deux dernières zones évoluent durant le programme, au fur et à mesure que des variables sont créées ou détruites.

10. C'est la différence entre la description que je fais à autrui de mes brouillons, et l'état réel de mes brouillons. Tel un système d'exploitation, je suis capable de lire mes brouillons fort mal organisés et de les faire correspondre en direct à quelque chose de plus présentable.

11. Une chaîne de caractère est un `char*`, c'est à dire un pointeur vers une zone où les caractères sont rangés les uns à côté des autres. Cette zone se trouve dans les données statiques initialisées.

2.1.0 Pile mémoire

Quand on entre dans un nouveau bloc syntaxique, on crée son bloc d'activation associé et on le met au bout de la pile mémoire. Quand on quitte un bloc syntaxique, on supprime son bloc d'activation.

FIGURE 2.7 – Schéma de l'évolution de la pile mémoire

Exercice. Représenter de même l'évolution de la pile mémoire lors de l'exécution du programme dont le code C est portée. c.

Proposition 28 (Organisation de la pile mémoire).

Les blocs d'activation sont stockés sur la pile mémoire suivant le principe LIFO. La pile est donc toujours remplie de manière contigu : elle n'a pas de « trou ».
La taille de la pile évolue durant l'exécution.

Remarque. La pile mémoire se comporte comme la structure de donnée pile (que nous verrons plus tard).

2.1.1 Tas mémoire

Les objets à durée de vie allouée sont stockés dans le tas. Quand le programme demande à allouer (« réserver », « faire naître ») x octets, on trouve x octets dans le tas et on les fournit au programme. Ils ne seront pas réutilisables tant que le programme n'aura pas explicitement libéré cette mémoire.

Exemple. Considérons le programme suivant, et montrons un exemple possible d'évolution du tas pour ce programme :

- 1 $x \leftarrow$ allouer 1 cases
- 2 $y \leftarrow$ allouer 3 cases
- 3 $z \leftarrow$ allouer 2 cases
- 4 Désallouer y

Proposition 29 (Non-contigüité du tas mémoire).

Le tas mémoire n'est pas rempli contigüement.

Remarque.

- Cette différence avec la pile mémoire provient de fait que l'on a pas accès à une bonne propriété comme LIFO : les objets alloués peuvent être libérés dans n'importe quel ordre.
- Il existe des stratégies pour essayer de limiter la création de « petits trous » dans le tas. On parle de stratégie d'allocation.
- Le tas mémoire n'a rien à voir avec la structure de donnée tas.

2.1.2 À savoir faire

L'objectif premier de cette section sur la mémoire d'un programme est que vous soyez capable de faire des schémas montrant l'évolution de la pile mémoire et une évolution possible du tas mémoire durant l'exécution d'un programme.¹²

Ce sont des notions importantes à avoir en tête lorsque l'on programme en C avec des pointeurs. En fait, **pour comprendre ce que l'on fait en C, il faut être capable de visualiser la mémoire.**

12. Notez que tout ce qui n'est pas marqué hors-programme est au programme. En particulier, j'ai déjà vu un sujet de concours MPI poser la question « dans quelle zone mémoire sont stockées les variables globales ? ».

Chapitre 3

COMPLEXITÉ

Notions	Commentaires
Analyse de la complexité d'un algorithme. Complexité dans le pire cas, dans le cas moyen. Notion de coût amorti.	On limite l'étude de la complexité dans le cas moyen et du coût amorti à quelques exemples simples.

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
[...] Organisation des activations sous forme d'arbre en cas d'appels multiples.	[...] Les récurrences usuelles : $T(n) = T(n-1) + an$, $T(n) = aT(n/2) + b$, ou $T(n) = 2T(n/2) + f(n)$ sont introduites au fur et à mesure de l'étude de la complexité des différents algorithmes rencontrés. On utilise des encadrements élémentaires <i>ad hoc</i> afin de les justifier ; on évite d'appliquer un théorème-maître général.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

SOMMAIRE

0. Complexité temporelle.....	50
0. Notion de complexité temporelle	50
<i>Opérations élémentaires (p. 51).</i>	
1. Notation de Landau	52
2. Exemples plus avancés	54
<i>Méthodologie (p. 54). Un exemple avec des boucles imbriquées (p. 54). Un exemple où il faut être très précis sur les itérations (p. 55). Écart incompressible entre les bornes (p. 56).</i>	
3. Pire cas, cas moyen, meilleur cas	56
4. Ordres de grandeur	57
1. Complexité spatiale	58
2. Cas particulier des fonctions récursives	59
0. Complexité temporelle des fonctions récursives	60
<i>Cas général : formule de récurrence (p. 60). Cas particuliers : arbre d'appels (p. 60).</i>	
1. Complexité spatiale des fonctions récursives	64
3. Complexité amortie	65
0. Méthodes de calcul	65
<i>Exemple fil rouge (p. 65). Méthode du comptable (p. 65). Méthode du potentiel (p. 67). Méthode de l'aggrégat (p. 69).</i>	
1. Remarques et compléments	70

0 Complexité temporelle

0.0 Notion de complexité temporelle

Définition 1 (Complexité temporelle).

La complexité temporelle d'un algorithme est une estimation du temps qu'il prend à s'exécuter. On peut la mesurer de deux grandes façons :

- En comptant le nombre de certaines opérations choisies. Par exemple, on peut exprimer la complexité en nombre d'additions effectuées. Si la ou les opérations ne sont pas choisies au hasard
- En comptant le nombre d'opérations élémentaires, c'est à dire du nombre d'étapes que prendra le calcul sur un processeur. Cette méthode impose souvent de faire des approximations pour simplifier.

Elle dépend souvent de l'entrée, on l'exprime souvent comme une fonction de l'entrée ou de la taille de l'entrée. Cette fonction est souvent nommée $C()$ ou $T()$.

Remarque.

- Généralement, ce qui nous intéresse est la complexité sur les grandes entrées. L'objectif est de prévoir (à peu près) combien de temps l'exécution d'un programme prendra, pour savoir si'il est raisonnable de le lancer ou s'il faut trouver une autre solution.
- Sauf mention contraire, une complexité est demandée en opérations élémentaires.

Exemple. Considérons la fonction ci-dessous, et calculons sa complexité en nombre d'additions $A()$:

```

9  /** Renvoie la somme des
    ↳ entiers de 0 à n */
10 unsigned somme_entiers(unsigned
    ↳ n) {
11     unsigned somme = 0;
12     unsigned i = 1;
13     while (i <= n) {
14         somme = somme + i;
15     }
16     return somme;
17 }
```

La fonction n'effectue aucune addition en dehors de sa boucle, ni aucun appel de fonctions. À chaque itération de sa boucle, elle effectue deux additions. La boucle itère n fois (car i parcourt l'ensemble $\llbracket 1; n \rrbracket$). D'où $A(n) = 2n$.

Exemple. Considérons les deux fonctions ci-dessous. On note A_{ep} la complexité en nombre d'additions de `est_premier`, et A_{spi} celle de `somme_premiers_inf`.

```

20 /** Teste si n est premier */
21 bool est_premier(int k) {
22     if (k <= 1) { return false; }
23
24     int d = 2;
25     while (d*d <= k) {
26         if (k % d == 0) { return
27             ↳ false; }
28         d = d + 1;
29     }
30     return true;
31 }
```

```

33 /** Renvoie la somme des nombres
    ↳ premiers inférieurs à x */
34 int somme_premiers_inf(int x) {
35     int somme = 0;
36     int n = 0;
37     while (n < x) {
38         if (est_premier(x)) {
39             somme = somme + x;
40         }
41         n = n + 1;
42     }
43     return somme;
44 }
```

Étudions tout d'abord `est_premier`. Elle ne fait aucune addition en dehors de sa boucle (ni aucun appel de fonction). Elle fait une addition par double. La boucle itère au plus $\lfloor \sqrt{k} \rfloor$ fois (d varie au plus de $\llbracket 2$ jusqu'à $\lfloor \sqrt{k} \rfloor + 1$ inclus). Donc $A_{ep}(k) \leq \lfloor \sqrt{k} \rfloor$.

Passons maintenant à `somme_premiers_inf`. Elle ne fait aucun appel de fonction ni addition en dehors de sa boucle. Sa boucle itère pour n allant de 0 à $x - 1$ inclus. À chaque itération, elle effectue au moins 1 addition et au plus $1 + A_{ep}(n)$. Donc au total, on peut encadrer A_{spi} ainsi :

$$\begin{aligned} \sum_{n=0}^{x-1} 1 &\leq A_{spi}(x) \leq \sum_{n=0}^{x-1} (1 + A_{ep}(n)) \\ x &\leq A_{spi}(x) \leq x + \sum_{n=0}^{x-1} (\lfloor \sqrt{n} \rfloor) \\ x &\leq A_{spi}(x) \leq x (1 + \lfloor \sqrt{x} \rfloor - 1) \end{aligned}$$

Remarque.

- Ici, on a encadré et non calculé la valeur précise de la complexité. La complexité précise de `est_premier` était en effet difficile à exprimer. C'est un cas très courant ! On va en fait essayer de *borner* les complexités, avec une borne supérieure et une borne inférieure.
- Vous trouvez peut-être que mon passage de l'avant dernière à la dernière ligne majore très grossièrement. Cela aurait pu être la cas¹, mais en l'occurrence pas tant que ça. Pour vous en convaincre, utilisez le fait que² :

$$\int_{d-1}^d \sqrt{x} dx \leq \sqrt{d} \leq \int_d^{d+1} \sqrt{x} dx$$

- L'expression de la complexité dépend de ce en fonction de quoi on l'exprime. Il y a deux grandes options :
 - Exprimer la complexité en fonction de la *valeur* des arguments (ce que l'on a fait ci-dessous).
 - Exprimer la complexité en fonction de la *taille* des arguments, c'est à dire en fonction du nombre de bits nécessaires pour représenter les arguments. Cette deuxième façon de faire prend tout son sens lorsque l'on essaye de comparer la difficulté de différents problèmes (cf MPI).
- Or, la taille des arguments est ne correspond pas à la valeur ! Penser à la taille de l'encodage binaire d'un entier, par exemple.

À retenir : il faut toujours indiquer en "fonction de quoi" on exprime la complexité.

0.0.0 Opérations élémentaires

Définition 2 (Opération élémentaire).

ne **opération élémentaire** est une opération « de base », au sens où elle n'est pas combinaison d'autres. Cela inclut par exemple :

- La lecture ou l'écriture d'une case mémoire.
- Une opération logique (NON, ET, OU, etc) ou arithmétique (+, -, %, etc) ou une comparaison du contenu de deux variables.
- Démarrer un appel de fonction ou renvoyer une valeur.
- Afficher 1 caractère.
- Et d'autres.

1. Dans beaucoup de calculs de complexité, il est facile d'obtenir une borne supérieure très large mais difficile de la rendre plus précise. Le monde est mal fait.

2. Si c'est un exo de maths trop dur pour l'instant, revenez-y au second semestre.

Remarque. Voici des exemples d'opérations qui *ne* sont pas élémentaires :

- Tester l'égalité de deux tableaux (cela demande de tester l'égalité de chacune des cases, c'est à dire de faire plusieurs tests d'égalité de variables).
- Afficher un texte (cela demande plusieurs affichages d'1 caractère).

Proposition 3.

En réalité, toutes les opérations élémentaires ne prennent pas le même temps, et le temps d'une même opération peut différer selon la machine qui l'exécute. Il est donc inutile d'essayer de compter le nombre exact d'opérations élémentaires. À la place, on cherche à compter **l'ordre de grandeur** du nombre d'opérations élémentaires.

Ainsi, au lieu de dire « on fait 47 opérations élémentaires », on dira « on effectue un nombre constant d'opérations élémentaires ».

Remarque. L'un des buts de la théorie de la complexité est d'analyser un algorithme indépendamment de la machine ou du langage qui l'exécute, afin de permettre des comparaisons des algorithmes et non des implémentations. Cela ne signifie pas que les comparaisons des langages / machines n'est pas pertinente ; ces dernières ce font juste différemment.

Exemple. Reprenons `somme_entiers`. En dehors de sa boucle, elle effectue un nombre constant d'opérations élémentaires. De même, chaque itération effectue un nombre constant d'opérations élémentaires. Or, la boucle itère n fois, donc la complexité $C(n)$ est linéaire en n .

0.1 Notation de Landau

Vous verrez ces notions plus proprement et plus en profondeur en mathématiques. Ici je m'efforce de faire une approche avant tout intuitive, quitte à piétiner certains points de rigueur.³

Définition 4 (Notations de Landau).

Soient (u_n) et (v_n) deux suites de réels. On dit que :

- (u_n) est dominé par (v_n) si quand n est grand, on a $|u_n| \leq K \cdot |v_n|$ (avec $K > 0$ une constante). Cela revient à dire que $\frac{|u_n|}{|v_n|}$ est majoré.
On note alors $u_n = O(v_n)$.
- (v_n) domine (u_n) si quand n est grand, on a $K \cdot |u_n| \leq |v_n|$ (avec $K > 0$ une constante). Cela revient à dire que $\frac{|v_n|}{|u_n|}$ est minoré (par une constante strictement positive).
On note alors $v_n = \Omega(u_n)$.
C'est en réalité équivalent au fait que u_n est dominé par v_n .
- (u_n) est de l'ordre de (v_n) si $u_n = O(v_n)$ et $v_n = O(u_n)$. Cela revient à dire que $\frac{|v_n|}{|u_n|}$ tend vers une constante strictement positive.
On note alors $u_n = \Theta(v_n)$.

En résumé : $O()$ indique l'ordre de grandeur de majoration, $\Omega()$ celui d'une minoration, et Θ les deux à la fois.

Exemple.

- $n - 50 = O(n)$; mais aussi $n + 42 = O(n)$ ou encore $3n + 12 = O(n)$
- $n + 3\sqrt{n} = O(n)$ (prendre $K = 4$ comme constante).
- $n + 3\sqrt{n} = \Omega(n)$; et donc $n + 3\sqrt{n} = \Theta(n)$.
- Pour la fonction `somme_premiers_inf`, on a prouvé que $A_{spi}(x) = O(x\sqrt{x})$ et que $A_{spi}(x) = \Omega(x)$.

Remarque.

3. J'ai essayé l'approche rigoureuse les années précédentes. Cela n'a pas marché.

- **Le but de ces notations est de ne garder que l'ordre de grandeur du terme dominant dans une majoration / minoration.** En effet, comme on l'a vu avec les opérations élémentaires, les calculs de complexité négligent souvent des « détails » de la réalité. Plutôt que de calculer une complexité fausse, on préfère se contenter de calculer l'ordre de grandeur de la vraie complexité (ou d'une borne de celle-ci).
- Il ne faut pas penser que $O(\dots)$ est une valeur précise. Deux suites différentes peuvent être dominée à l'aide d'un même $O(\dots)$. Par exemple, $(n+1)_{n \in \mathbb{N}}$ et $\left(\frac{n}{3} + 100\right)_{n \in \mathbb{N}}$ sont toutes les deux dominées par $(n)_{n \in \mathbb{N}}$, donc toutes les deux $= O(n)$.
- En maths, vous définirez l'équivalence \sim . Ce n'est pas la même chose que le $\Theta()$, ne confondez pas.
- On essaye de mettre une borne la plus « simple » possible dans le $O()$ (ou Ω ou Θ). Par exemple, on préfère $O(n^2)$ plutôt que $O(\pi n^2 - \sqrt{2.71} \log_5 n)$. Au risque de me répéter, le but est de simplifier en ne mémorisant que l'ordre de grandeur du terme dominant.

Proposition 5 (Relation usuelles de domination).

Je note ici $x \ll y$ le fait que $x = O(y)$. Soient $a, b \in \mathbb{R}_+^*$ et $c \in]1; +\infty[$. On a :

$$\log(n)^a \ll n^b \ll c^n \ll n!$$

Démonstration. Il s'agit des croissances comparées (une version plus faible des puissances comparées, pour être précis). Vous les prouverez en maths. \square

Proposition 6 (Opérations sur les dominations).

J'indique ici quelques relations utiles. Vous en verrez plus en maths.

- La domination est transitive, c'est à dire que si $u_n = O(v_n)$ et $v_n = O(z_n)$ alors $u_n = O(z_n)$.
- On peut sommer une quantité finie de dominations, c'est à dire que :
 - Si $u_n = O(y_n)$ et $v_n = O(z_n)$, alors $u_n + v_n = O(|y_n| + |z_n|)$.
 - Idem avec $\Omega()$.
 - Idem avec Θ .

En particulier, $u_n = O(z_n)$ et $v_n = O(z_n)$ entraîne $u_n + v_n = O(z_n)$.

- On peut sommer une quantité variable de dominations uniquement si elles cachent une même constante K . En conséquence, on peut par exemple sommer les dominations des itérations successives d'une boucle !
- On peut multiplier une quantité finie de dominations, c'est à dire que :
 - Si $u_n = O(y_n)$ et $v_n = O(z_n)$, alors $u_n \cdot v_n = O(|y_n| \cdot |z_n|)$.
 - Idem avec $\Omega()$.
 - Idem avec Θ .
- On peut multiplier une quantité variable de dominations uniquement si elles cachent la même constante K .

Remarque. Si cette histoire de « quantité variable de domination » vous perturbe (ou que vous ne comprenez pas pourquoi on ne peut pas forcément sommer dans ce cas), attendez le cours de maths sur le sujet.

Exemple. On a en fait déjà utilisé ces propriétés sont le dire dans le calcul de la complexité en opérations élémentaires

0.2 Exemples plus avancés

0.2.0 Méthodologie

La méthode utilisée pour ces exemples est toujours la même :

- Calculer d'abord la complexité des appels de fonction.
- Calculer la complexité en dehors des boucles.
- Pour chaque boucle, calculer la complexité d'une itération (condition + corps), puis le nombre d'itérations⁴, et en déduire la complexité de la boucle entière.
En cas de boucle imbriquées, on commence par les boucles intérieures.

0.2.1 Un exemple avec des boucles imbriquées

On considère la fonction ci-dessous :

```

47  /** Échange *a et *b */
48  void swap(int* a, int* b) {
49      int tmp = *a;
50      *a = *b;
51      *b = tmp;
52      return;
53  }
54
55
56  /** Trie tab par ordre croissant en temps quadratique */
57  void tri_bulle(int tab[], int len) {
58      int deja_trie = 0;
59      while (deja_trie < len) {
60
61          int indice = 0;
62          // Cette boucle fait remonter le plus grand élément
63          // non-encore trié et le place à sa place finale
64          while (indice + 1 < len - deja_trie) {
65              if ( tab[indice] > tab[indice+1] ) {
66                  swap( &tab[indice], &tab[indice+1] );
67              }
68              indice = indice + 1;
69          }
70
71          deja_trie = deja_trie + 1;
72      }
73      return;
74  }

```

On veut calculer sa complexité T en nombre de comparaisons en fonction de la longueur du tableau⁵. Commençons par remarquer que la fonction `swap` n'effectue aucune comparaison. On ne prendra donc pas en compte ses appels dans les calculs qui suivent.

Étudions maintenant La fonction `tri_bulle` :

- Elle n'effectue aucune comparaison en dehors de ses boucles.
- À chaque itération de la boucle intérieure (lignes 60-65), elle effectue deux comparaisons (une dans la condition, une dans le `if` du corps). Elle effectue une dernière comparaison lorsqu'elle quitte la boucle. Comme cette boucle itère $\text{len} - \text{deja_trie}$ fois, elle effectue au total $2(\text{len} - \text{deja_trie}) + 1$ comparaisons.

4. S'il n'y a pas de difficulté particulière, on peut se contenter d'affirmer le nombre d'itérations sans le justifier.

5. C'est un critère assez courant pour évaluer un tri.

- La boucle principale (ligne 56-69) effectue une comparaison dans sa condition, et contient la boucle précédente. La variable `deja_trie` parcourt $\llbracket 0; \text{len} - 1 \rrbracket$. On effectuera une dernière comparaison quand on quittera la boucle. Donc au total, cette boucle fait :

$$\begin{aligned}
 1 + \sum_{\text{deja_trie}=0}^{\text{len}-1} (2(\text{len} - \text{deja_trie}) + 1) &= 1 + \text{len} - 2 \sum_{\text{deja_trie}=0}^{\text{len}-1} (\text{len} - \text{deja_trie}) \\
 &= 1 + \text{len} + 2\text{len}^2 - 2 \sum_{\text{deja_trie}=0}^{\text{len}-1} \text{deja_trie} \\
 &= 1 + \text{len} + 2\text{len}^2 - (\text{len} - 1)(\text{len}) \\
 &= 1 + \text{len}^2
 \end{aligned}$$

- On peut en déduire que :

$$T(\text{len}) = \text{len}^2 + 1 = \Theta(\text{len}^2)$$

Remarque.

- On aurait pu prouver $T(\text{len}) = O(\text{len}^2)$ plus facilement : au lieu de donner la valeur précise du nombre de comparaisons de la boucle intérieure effectuée, on dit qu'elle effectue $O(\text{len})$. Chaque itération de la boucle principale effectue donc $O(\text{len})$ comparaisons, or cette boucle itère len fois, donc $T(\text{len}) = O(\text{len}^2)$.
- « Il y a deux boucles imbriquées donc la complexité est quadratique » n'est pas une preuve, et n'est pas toujours vrai.

0.2.2 Un exemple où il faut être très précis sur les itérations

On suppose que l'on dispose d'une fonction `affiche_entiers_l_bits` qui, en temps $\Theta(2^l)$ affiche tous les entiers pouvant s'écrire sur l bits. On considère la boucle ci-dessous :

```

187 int l = 0;
188 while (l <= n) {
189     printf("Voici les entiers binaires sur %d bits : \n\t", l);
190     affiche_entiers_l_bits(buffer, l);
191     l = l + 1;
192 }
```

(Vous pouvez ignorer le premier argument de `affiche_entiers_l_bits`, c'est un détail d'implémentation.)

Calculons la complexité de cette boucle. Notons tout d'abord qu'il y a un nombre constant d'opérations élémentaires en dehors de la boucle. (Pour la suite, je propose deux raisonnements.)

Version 1 : La boucle itère l de 0 à n inclus. Chaque itération effectue des opérations en temps constant et un appel en $\Theta(2^l)$, et est donc en $O(2^n)$. En sommant sur les itérations, on obtient une complexité T pour la boucle qui vérifie $T(n) = O(n2^n)$.

Dans cette version, on a donné une borne grossière à chaque itération, afin que la somme des itérations soit simple à calculer.

Version 2 : La boucle itère l de 0 à n inclus. Chaque itération effectue des opérations en temps constant et un appel en $\Theta(2^l)$, et l'itération a donc une complexité de l'ordre de 2^l . En sommant sur les itérations, on obtient que le tout est de l'ordre de $\sum_{l=0}^n 2^l = 2^{n+1} - 1$. D'où $T(n) = \Theta(2^n)$.

Ici, on a donné la valeur exacte de l'ordre de grandeur de chaque itération. Sommer sur les itérations a en conséquence été un peu plus compliqué, mais on a obtenu une valeur plus précise à la fin.

Remarque. L'intérêt des majorations grossières est de simplifier les calculs. On peut ainsi obtenir très rapidement des majorations qui prouvent qu'un code s'exécutera en temps raisonnable⁶.

Exercice. Reprendre la preuve de la complexité en nombre de comparaisons du tri bulle. Refaire avec une majoration grossière. Commenter.

0.2.3 Écart incompressible entre les bornes

Revenons à `est_premier` :

```

20  /** Teste si n est premier */
21  bool est_premier(int k) {
22      if (k <= 1) { return false; }
23
24      int d = 2;
25      while (d*d <= k) {
26          if (k % d == 0) { return false; }
27          d = d + 1;
28      }
29      return true;
30  }
```

Avec les calculs déjà effectués, on conclut que la complexité en additions A_{ep} vérifie $A_{ep} = \Omega(1)$ (car la boucle itère au moins une fois) et $A_{ep} = O(k)$.

On peut prouver qu'il existe des k aussi grand que l'on veut pour lesquels la boucle itère une seule fois (les entiers pairs). Similairement, il existe des k aussi grands que l'on veut pour lesquels la boucle itère \sqrt{k} fois (les entiers premiers). Il est donc impossible d'obtenir un Θ .

Remarque. Cet exemple peut-être vu comme un cas particulier de la différence entre « meilleur cas » et « pire cas ».

0.3 Pire cas, cas moyen, meilleur cas

Parfois, pour une même taille d'entrée, la complexité peut varier. `est_premier` en était un exemple, mais en voici un encore plus clair (et plus fort) :

```

202  /** Renvoie true si x dans tab,
203   * et false sinon.
204   */
205  bool mem(int x, int const tab[], int len) {
206      int indice = 0;
207      while (indice < len) {
208          if (tab[indice] == x) { return true; }
209          indice = indice + 1;
210      }
211      return false;
212  }
```

Pour len aussi grand que l'on veut, il existe des tableaux de longueur len sur lesquels cette fonction s'exécute en temps $\Theta(1)$ et d'autres sur lesquels elle s'exécute en temps $\Theta(len)$.

Autrement dit, on ne peut même pas définir la complexité comme une fonction dépendant uniquement de len ! C'est pourtant ce que l'on voudrait : on voudrait pouvoir prévoir le temps d'exécution d'une fonction à partir de critères simples, comme la longueur.

6. Vous ferez une utilisation similaire de majorations grossières en mathématiques en deuxième année, pour prouver qu'une série (ou une série de fonction) est « raisonnable ».

Définition 7 (Pire cas, meilleur cas).

- La **complexité dans le pire des cas** est le nombre *maximal* d'opérations que la fonction exécute sur une entrée d'une taille donnée.
- La **complexité dans le meilleur des cas** est le nombre *minimal* d'opérations que la fonction exécute sur une entrée d'une taille donnée.
- La **complexité dans le cas moyen** est le nombre moyen d'opérations que la fonction exécute sur une entrée d'une taille donnée. La définition de « nombre moyen » demande donc de sommer (et moyenner) sur toutes les entrées possibles de cette taille. On pondère parfois ces entrées par une probabilité afin de mieux coller à une situation pratique.

Exemple. La complexité (pire des cas) de `mem` est $\Theta(\text{len})$. Sa complexité meilleur des cas est $\Theta(1)$. Sa complexité en cas moyen... est difficile à définir, car il faut déjà définir ce que signifie moyenner sur tous les tableaux de longueur `len` lorsqu'il y en a une infinité.⁷

Convention 8.

Sauf mention contraire, toute complexité demandée est un pire des cas.

Remarque.

- **Ne confondez pas** « pire cas » avec « majoration de la complexité » et « meilleur cas » avec minoration. Ce qui est vrai, c'est que le meilleur des cas est une minoration du pire des cas. Mais il est souvent possible d'obtenir de meilleurs minoration.
- **Ne confondez pas** « meilleur cas » avec « entrée de petite taille ». On s'intéresse à la complexité sur des *grandes* entrées!⁸

0.4 Ordres de grandeur

On considère une fonction dont la complexité est $C(n)$. On dit que la fonction est :

- de complexité constante si $C(n) = O(1)$.
- polylogarithmique en n si $C(n) = O(\text{poly}(\log(n)))$ avec *poly* une fonction polynomiale.
- linéaire en n si $C(n) = O(n)$. Similaire pour quadratique en n , cubique en n , polynomiale en n .
- exponentielle en n si $C(n) = O(e^{\text{poly}(n)})$ avec *poly* une fonction polynomiale.

Notez que je précise à chaque fois « en n », pour bien rappeler en fonction de quoi la complexité est exprimée.

Il est impossible de traduire une complexité en une durée exacte à la nano seconde près : trop de facteurs liés au langage et à la machine entrent en compte. On peut cependant appliquer la méthode suivante pour avoir une approximation très grossière :

- 1) Évaluer la valeur de la complexité sur l'entrée voulue
- 2) Diviser par 1Ghz la fréquence du processeur (c'est un arrondi du nombre d'opérations par s de votre processeur).
- 3) Multiplier le tout par quelque chose entre 2 et 500 (pour prendre en compte la constante cachée du $O()$).

Si le résultat est raisonnable, votre code terminera très vite ; sinon croisez fort les doigts.

7. Les « sommes infinies » posent des difficultés mathématiques, comme vous le verrez en maths.

8. Les petites entrées terminent rapidement en pratique.

Voici un tableau des arrondis des valeurs obtenues par les étapes 1) et 2) de la méthode précédente :

$T(x) \backslash x$	10	100	1000	10000	10^6	10^9
$\Theta(\log x)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{x})$	1ns	10ns	100ns	100ns	1 μ s	1ms
$\Theta(x)$	10ns	100ns	1 μ s	10 μ s	1ms	1s
$\Theta(x \log x)$	10ns	100ns	10 μ s	100 μ s	10ms	10s
$\Theta(x^2)$	100ns	10 μ s	1ms	100ms	10min	10 ans
$\Theta(x^3)$	1 μ s	1ms	1s	10 min	10 ans	∞
$\Theta(2^x)$	1 μ s	∞	∞	∞	∞	∞

FIGURE 3.1 – Ordre de grandeurs de temps de calculs pour quelques complexités et tailles d'entrées. Non-contractuel.

1 Complexité spatiale

Dans l'idée, la complexité spatiale est l'espace utilisé par une fonction. Cependant, pour des raisons théoriques que vous verrez peut-être plus tard, on utilise un modèle extrêmement simplifié. Il a le mérite de permettre des comparaisons entre des algorithmes de manière indépendante du langage et de la machine.

Définition 9 (Complexité spatiale).

La complexité spatiale d'une fonction est le nombre de cases mémoires dont elle a besoin pour s'exécuter. On ne compte pas dedans :

- Les entrées si elles sont en lecture seule.
- La sortie si elle est en lecture seule.

Autrement dit, on compte l'espace *supplémentaire* dont la fonction a besoin.

On la note souvent E ou S . Les notions de pire, moyen et meilleur cas s'appliquent ici aussi.

Remarque.

- Comme pour la complexité temporelle, seul l'*ordre de grandeur* a véritablement un sens en pratique.
- Cette définition a des limites quand on l'applique à un langage pratique. Par exemple, si un programme C alloue n cases mémoires avec `malloc` mais ne les utilise pas, cet espace doit-il compter ? Dans de tels cas, il ne faut pas hésiter à préciser : on peut par exemple distinguer l'espace *utilisé* de l'espace *demandé*.
- Si une même zone mémoire est utilisée par plusieurs appels de fonctions (cela correspond aux arguments passés par pointeur/référence), il ne faut pas la compter plusieurs fois puisque c'est le même espace qui est utilisé !

Exemple. Toutes les fonctions vues jusqu'à présent sauf `tri_bulle` sont en espace constant car elles créent un nombre fixe de variables, dont la taille est fixe.

Par contre, `tri_bulle` lit et modifie $O(\text{len})$ cases du tableau : elles ne sont donc ni des entrées en lecture seule ni la sortie en écriture seule, et comptent donc la complexité spatiale qui est donc un $O(\text{len})$.

En fait, pour avoir une complexité spatiale variable, il faut :

- soit créer/modifier un objet dont la taille est une variable (par exemple un tableau à n cases).

- soit faire de la récurrence.

Théorème 10 (Bornes temps-espace (version simplifiée)).

On considère une fonction qui termine. On note sa complexité temporelle $T(n)$ (en opérations élémentaires) et spatiale $E(n)$ (on compte uniquement l'espace utilisé). On suppose en outre que l'ordre de $E(n)$ n'est pas constant. On a :

$$E(n) \leq T(n) \leq 2^{E(n)}$$

Démonstration. Je vais prouver l'inégalité de gauche, et donner l'intuition de la preuve de celle de droite :

- Utiliser 1 case de la mémoire demande d'y faire une lecture/écriture, et donc au moins 1 opération élémentaire. D'où $E(n) \leq T(n)$.
- L'exécution d'une fonction est déterministe. En partant d'une ligne précise, et de valeurs précises des variables, il y a une seule exécution possible. On nomme (grosso modo) « état mémoire » d'une fonction la donnée de la ligne du code où on en est ainsi que des valeurs des variables. Toutes ces informations sont stockées dans la mémoire. Par déterminisme de l'exécution, si un programme passe deux fois par exactement le même état mémoire, elle est en train de faire une boucle infinie. Or, si on utilise $E(n)$ cases, il n'y a que $2^{E(n)}$ états mémoire possibles car l'unité élémentaire est le bit. D'où $T(n) \leq 2^{E(n)}$.

□

Remarque.

- La borne supérieure est hors-programme, mais l'idée de la preuve est très intelligente. On note généralement cette borne supérieure $T(n) \leq 2^{O(E(n))}$, ce qui est plus rigoureux. J'ai ici essayé de simplifier, quitte à tordre un peu la vérité.
- Notez que la preuve de la majoration n'utilise pas l'hypothèse $E(n)$ non-constant. En fait, l'hypothèse est uniquement là car ce que l'on appelle « espace constant » d'un point de vue pratique n'est pas un espace constant d'un point de vue théorique (notamment à cause des considérations de la taille des entiers⁹). Plus d'infos à ce sujet en MPI.
- Ce théorème n'est en fait généralement même pas écrit à l'aide de E et T , mais à partir des classes de complexité en temps $DTIME$ (plus à ce sujet en MPI) et $DSPACE$ (plus à ce sujet en école).

2 Cas particulier des fonctions récursives

On voudrait analyser la complexité d'une fonction récursive. Par exemple :

Fonction Hanoï

Entrées : i : le pic de départ ; j : le pic d'arrivée ; n : le nombre de disques à déplacer

```

1 si  $n > 0$  alors
2    $k \leftarrow$  pic autre que  $i$  ou  $j$ 
3   HANOÏ( $i, k, n-1$ )
4   Déplacer 1 disque de  $i$  à  $j$ 
5   HANOÏ( $k, j, n-1$ )
```

Cette fonction effectue 1 déplacement par appel, mais il faut prendre en compte tous les appels... On va distinguer deux choses :

9. D'un point de vue théorique, les entiers ne peuvent pas déborder et ont une taille variable. Subséquemment, dans la théorie l'espace constant n'existe (quasiment) pas. Ce n'est pas le point de vue que nous utilisons ici.

Définition 11 (Coût local).

- Le **complexité locale à l'appel** : ce sont les ressources utilisées par l'appel en cours.
- Le **complexité des appels récursifs** : ce sont les ressources utilisées par les appels récursifs.

Pour calculer la complexité total, il faudra prendre en compte ces deux éléments.

Remarque. Le terme de « coût » est parfois utilisé comme synonyme de « complexité ». On parle donc aussi de « coût local » et de « coût récursif ».

2.0 Complexité temporelle des fonctions récursives

Proposition 12 (Complexité temporelle récursive).

La complexité temporelle d'une fonction est la somme des complexités temporelles locales de chacun des appels récursifs.

Il « suffit » donc pour la calculer d'être capable de :

- Calculer la complexité locale d'un appel.
- Sommer sur les appels.

C'est cette deuxième étape qui est généralement la plus difficile.

2.0.0 Cas général : formule de récurrence

La complexité temporelle s'exprime naturellement comme une fonction récursive. Si cette suite est une suite facile que l'on sait résoudre grâce au cours de maths, c'est gagné.

Exemple. La complexité en déplacements $D(n)$ de HANOÏ vérifie $D(n) = 2D(n-1) + 1$ et $D(0) = 0$. C'est une suite arithmético-géométrique que l'on sait résoudre, et dont la solution est $D(n) = 2^n - 1$.

Remarque.

- Lorsque l'on raisonne en opérations élémentaires, on n'a jamais une véritable égalité. Par exemple, pour HANOÏ, on obtient $T(n) = 2T(n-1) + O(1)$, c'est à dire qu'il existe $K > 0$ tel que $T(n) \leq 2T(n-1) + K$. Ici, on a une *majoration* par un terme arithmético-géométrique. On en déduit que $T(n)$ est inférieur à la suite arithmético-géométrique en question, et on obtient $T(n) = O(2^n)$. On peut faire de même pour les minoration.

- Lorsque l'on raisonne en opérations élémentaires, on fait souvent ce que j'ai fait dans le point précédent : on donne uniquement la formule de récurrence ($T(n) = 2T(n-1) + O(1)$), mais pas le cas de base. Cela signifie implicitement que le cas de base est de complexité constante, et qu'il est atteint lorsque n passe en-dessous d'une valeur fixe. On peut prouver que le choix de cette valeur fixe ne change pas l'ordre de grandeur de la complexité.

Ainsi, dans l'exemple précédent, le cas de base est atteint en $n < 1$ et a un coût $O(1)$. Changer ce seuil à $n < 4$ donnerait $T(n) = O(2^{n-3}) = O(2^n)$.

Si la complexité n'est pas une suite « gentille » que l'on sait résoudre grâce au cours de maths, on peut tout de même essayer d'observer les premiers termes de la suite et de trouver une logique et de peut-être repérer une complexité qui ressemble à un classique que l'on sait traiter. De manière générale, *essayer de se ramener à ce que l'on maîtrise* est un excellent raisonnement en sciences !

2.0.1 Cas particuliers : arbre d'appels

Rappel du cours sur la récursivité : on peut représenter la suite des appels récursifs sous la forme d'un arbre.

On peut parfois utiliser cette représentation pour facilement sommer les complexités locales des appels. Cela revient en fait à réécrire la somme des coûts des appels en faisant des « paquets » d'appels qui ont les mêmes paramètres de complexité (ces paquets sont les lignes de l'arbre si on a bien représenté l'arbre).

Exemple. Voici l'arbre des coûts des appels de HANOÏ :

FIGURE 3.2 – Arbre des coûts en déplacements des appels de HANOÏ

Pour calculer à l'aide d'un tel arbre, on procède ainsi :

- 1) On calcule le coût d'un noeud de l'arbre (= on calcule la complexité locale d'un appel).
- 2) On compte le nombre de noeuds par ligne (= le nombre d'appels qui ont les mêmes paramètres pour la complexité), et on en déduit le coût d'une ligne de l'arbre.
- 3) On somme sur les lignes pour conclure.

Les étapes 2) et 3) sont les plus difficiles, car il est aisé d'y affirmer une formule fausse. Il est donc très important d'inclure une ligne au milieu de l'arbre qui montre la forme générale d'une ligne « quelconque », car cette ligne résume le raisonnement.

Exemple. Dans l'arbre précédent, sans compter la ligne du bas (qui effectue 0 déplacements) il y a n lignes. Il y a 2^i appels sur la ligne des appels qui ont $n-i$ en entrée. Ainsi, une ligne associée à i a un coût de 2^i . Les lignes vont de $i = 0$ à $i = n-1$. En sommant sur ces lignes, on obtient $D(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$.

Remarque. Parfois, l'arbre obtenu n'a pas une forme simple à décrire. On peut alors essayer de majorer par un arbre « imple à calculer » en « rajoutant des appels là où il en manque » :

FIGURE 3.3 – Arbre d'appels et majoration pour $T(n) = T(n-1) + T(n-2)$ et cas de base constant.

Diviser pour régner : les arbres d'appels de fonctions « Diviser pour Régner » sont particuliers, mais se résolvent de manière similaire. On se place dans le cas suivant : un problème de taille n est découpé en r sous-problèmes de taille n/c . Le coût local (séparation/fusion ou cas de base) est donné par la fonction f .

On admet que **l'on peut ignorer les éventuelles parties entières dans l'équation de récurrence sans que cela ne change l'ordre du grandeur du résultat**¹⁰. L'équation de complexité est donc :

$$T(n) = r.T(n/c) + f(n)$$

Voici un schéma de l'arbre d'appel¹¹, où L est la hauteur de l'arbre d'appels :

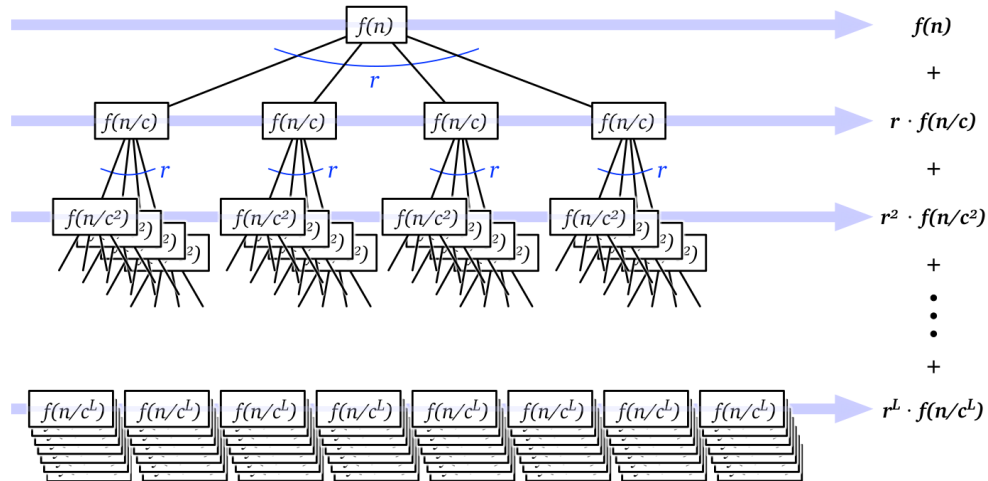


FIGURE 3.4 – Arbre de complexité diviser pour régner

Les coûts T_i de chacune des lignes sont indiqués le long de l'arbre (on a $T_i = r^i f(n/c^i)$). La complexité totale est la somme des coûts de ces lignes. Il existe 3 cas courants dans lesquels une majoration précise de cette somme est simple à calculer :

- Si les T_i décroissent (au moins) géométriquement, c'est à dire s'il existe $K > 1$ tel que $T_{i+1} \leq \frac{T_i}{K}$.
On a alors pour tout i , $T_i \leq \frac{T_0}{K^i}$ et donc :

$$\begin{aligned} T(n) &= \sum_{i=0}^L T_i \\ &\leq \sum_{i=0}^L \frac{T_0}{K^i} \\ &\leq f(n) \left(\sum_{i=0}^L \frac{1}{K^i} \right) && \text{en factorisant par } T_0 = f(n) \\ &= O(f(n)) && \text{car le terme entre parenthèses tend vers une constante positive} \end{aligned}$$

Autrement dit, si les coûts des lignes décroissent (au moins) géométriquement, le coût de la première ligne domine.

- Si les T_i croissent (au moins) géométriquement, c'est à dire s'il existe $K > 1$ tel que $T_{i+1} \geq K.T_i$.
On a alors pour tout i , $T_i \leq \frac{T_L}{K^{L-i}}$ et donc :

10. Si vous êtes accroché-es en maths, vous pouvez aller voir le lien suivant pour une preuve de cela : <https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf>

11. Issu de l'excellent *Algorithms* de J. Erikson.

$$\begin{aligned}
T(n) &= \sum_{i=0}^L T_i \\
&\leq \sum_{i=0}^L \frac{T_L}{K^{L-i}} \\
&\leq r^L f(n/c^L) \left(\sum_{i=0}^L \frac{1}{K^{L-i}} \right) && \text{en factorisant par } T_L = r^L f(n/c^L) \\
&= O(r^L f(n/c^L)) && \text{car le terme entre parenthèses tend vers une constante positive}
\end{aligned}$$

Autrement dit, si les coûts des lignes croissent (au moins) géométriquement, le coût de la dernière ligne domine. Notez que dans ce cas, il faut calculer L . C'est généralement un $O(\log_c(n))$, car c'est généralement le premier i tel que n/c^i passe en dessous du seuil du cas de base.

- Si les T_i sont constants, on a :

$$\begin{aligned}
T(n) &= \sum_{i=0}^L T_i \\
&= \sum_{i=0}^L f(n) && \text{car } \forall i, T_i = T_0 = f(n) \\
&= O(Lf(n))
\end{aligned}$$

Notez qu'ici aussi il faut calculer L .

Résumons :

Proposition 13 (Méthode de calcul Diviser pour Régner).

Pour calculer une majoration du coût temporel d'un algorithme Diviser pour Régner dont l'équation de récurrence est $T(n) = r.T(n/c) + f(n)$, on utilise la méthode suivante :

- Si les coûts des lignes de l'arbre décroissent (au moins) géométriquement de haut en bas, alors on prouve que le coût de la racine est dominant.
- Si les coûts des lignes croissent (au moins) géométriquement de haut en bas, alors on prouve que le coût des cas de base est dominant.
- Si les coûts des lignes sont constants, on les somme simplement.

Remarque.

- **Il faut savoir identifier et gérer ces 3 cas !!!!!!!** Et pour ça, pas de secret, il faut connaître les 3 cas et s'entraîner en refaisant ces calculs.
- On peut aussi obtenir les minoration associées, mais généralement on se contente d'un $O()$.
- Il existe un théorème célèbre, appelé le « théorème maître ». C'est une application de la propriété précédente au cas où f est un polynôme. Je vous déconseille fortement d'essayer de l'apprendre par coeur : vous ferez des erreurs en mémorisant ces formules¹². Vous **n'**avez **pas** le droit de l'utiliser en MP2I/MPI. Je vous le donne ci-dessous à titre informatif :

Proposition 14 (Théorème Maître (hors-programme)).

On considère une formule de récurrence de la forme $T(n) = aT(n/b) + O(n^d)$ avec comme cas de base $T(1) = 1$. Alors :

- Si $a < b^d$, alors $T(n) = O(n^d)$ (« le coût de la racine domine »).
- Si $a > b^d$, alors $T(n) = O(n^{\log_b(a)})$ (« le coût des feuilles domine »).
- Si $a = b^d$, $T(n) = O(n^d \log(n))$ (« le coût des étages est constant »).

12. Le programme officiel est d'accord avec moi, et l'a mis hors-programme pour cette raison.

2.1 Complexité spatiale des fonctions récursives

Rappel du cours sur l'organisation de la mémoire : les variables non-allouées sont créées au début de leur bloc et supprimées à la fin.

En particulier, n'existe à tout moment en mémoire que les variables des appels récursifs qui ont commencé mais n'ont pas encore terminé. Dans l'arbre d'appels, commencer un appel revient à « descendre en suivant un trait » et terminer un appel revient à « remonter en suivant un trait ». Ainsi :

Proposition 15 (Complexité spatiale récursive).

La complexité spatiale d'une fonction récursive est le maximum d'espace nécessaire pour une suite d'appels de l'appel initial à un cas de base.

Autrement dit, c'est la somme maximale des complexités locales d'appels le long d'un chemin de la racine à une feuille de l'arbre d'appels.

Remarque. Ce « Autrement dit » ignore les difficultés liées au partage de la mémoire lors d'un passage par référence/pointeur que l'on a déjà évoquées.

Exemple. La fonction HANOÏ a une complexité spatiale en (n) . En effet, le coût local de chaque appel est constant, et il y a au plus n appels actifs à tout moment¹³. Comme cette borne sur le nombre d'appels est atteinte, c'est même un $\Theta()$.

Exercice. Calculer la complexité temporelle et spatiale de la fonction `tri_fusion` ci-dessous :

```

15 (** Fusionne deux listes triées en une seule. *)
16 let rec fusionne (l0 : 'a list) (l1 : 'a list) : 'a list =
17   match (l0,l1) with
18   | [], _      -> l1
19   | _, []      -> l0
20   | h0::t0, h1::t1 -> if h0 < h1 then
21                         h0 :: (fusionne t0 l1)
22                         else
23                         h1 :: (fusionne l0 t1)
24
25
26 (** Separe la liste l en deux moitiés *)
27 let rec separe (l : 'a list) : 'a list * 'a list =
28   match l with
29   | []      -> [], []
30   | h :: [] -> [h], []
31   | h::hbis :: t -> let (u,v) = separe t in
32                     h::u, hbis::v
33
34 (** Trie l grâce à l'algorithme du tri fusion *)
35 let rec tri_fusion (l : 'a list) : 'a list =
36   match l with
37   | []      -> []
38   | h :: [] -> [h]
39   | _      -> let (u,v) = separe l in
40               fusionne (tri_fusion u) (tri_fusion v)

```



13. Cette preuve est plus convainquante avec un arbre d'appels dessiné. Je dirais même que la preuve est incomplète sans.

3 Complexité amortie

Jusqu'à présent, nous avons étudié la complexité d'un appel à une fonction. Le but désormais est d'étudier le coût d'une *suite* d'appels, et plus précisément le coût d'un appel au sein de cette suite.

Considérons l'exemple initial suivant. Un-e MP2I veut aller faire des séances de sport dans une salle. Le tarif¹⁴ est le suivant :

- Si c'est la première fois, les frais de dossiers coutent 30€. La place coûte 10€ de plus.
- Les fois suivantes, la place coûte uniquement 10€.

Question : dans une succession de séances, combien coûte chaque séance ? On peut dire que dans le pire des cas, chaque séance coûte 40€... mais on veut bien que c'est grossier : dans une succession de séances, les frais de dossiers ne sont payés qu'une seule fois. On préfère dire que dans une succession de S séances, le coût de chaque séance est $10 + \frac{30}{S}$. On *amortit* le coût de l'abonnement sur toutes les séances (on dit aussi qu'on *lisse* le coût).

Définition 16 (Complexité amortie).

On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1. Un coût amorti est la donnée de coûts fictifs \hat{C}_i tels que :

$$\sum_{i=1}^n \hat{C}_i \geq \sum_{i=1}^n C_i$$

Autrement dit, la seule condition sur un coût amorti est que la somme des coûts amorti majore le coût réel.

3.0 Méthodes de calcul

3.0.0 Exemple fil rouge

Nous allons présenter trois méthodes qui cherchent à calculer un coût amorti qui soit une majoration assez précise. Nous les appliquerons à un exemple fil rouge : les tableaux dynamiques. Initialement, un tableau dynamique est un tableau à 1 case, qui n'est pas utilisé : nous appellerons. On ajoute un élément à la fin de celui-ci via la fonction ci-dessous :

Fonction Ajout

Entrées : T un tableau dynamique ; x un élément à ajouter à la fin

```

1 si T est entièrement plein alors
2   T' ← nouveau tableau deux fois plus grand que T
3   recopier T dans T'
4   remplacer T par T'
5 len ← nombre de cases utilisées de T
6 T[len] ← x
```

L'exemple fil rouge sera donc : calculer un coût amorti d'un appel à AJOUT au sein d'une suite d'appels à AJOUT. On comptera le coût en nombre d'écriture dans un tableau effectuées.

3.0.1 Méthode du comptable

On imagine qu'au lieu de dépenser du temps ou de l'espace (ou une autre ressource), la fonction dépense des *pièces*. L'objectif de la méthode du comptable est de montrer comment pré-payer des opérations coûteuses en augmentant le prix des opérations peu coûteuses. On amortit ainsi le coût des opérations coûteuses sur les autres.

14. Fictif.

Définition 17 (Méthode du comptable).

On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1.

La **méthode du comptable** est le fait d'ajouter à chaque opération une **accumulation** a_i et une **dépense** d_i . L'accumulation consiste à poser une ou plusieurs pièces « sur » la structure, en prévision d'un paiement élevé plus tard. Une dépense consiste à utiliser des pièces déjà déposées pour payer une opération coûteuse. On doit garantir qu'à tout moment :

$$\sum_{i=1}^n a_i \geq \sum_{i=1}^n d_i$$

C'est à dire qu'on ne dépense jamais une pièce que l'on a pas encore accumulée. La méthode du comptable définit alors le coût amorti suivant :

$$\hat{C}_i = C_i + a_i - d_i$$

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned} \sum_{i=1}^n \hat{C}_i &= \sum_{i=1}^n (C_i + a_i - d_i) \\ &= \left(\sum_{i=1}^n C_i \right) + \left(\sum_{i=1}^n a_i \right) - \left(\sum_{i=1}^n d_i \right) \\ &\geq \sum_{i=1}^n C_i \end{aligned} \quad \text{car } \sum_{i=1}^n a_i \geq \sum_{i=1}^n d_i$$

Il s'agit donc bien d'un coût amorti. □

Remarque. En pratique, il faut donner les accumulations, les dépenses, et une justification convainquante du fait que l'on ne dépense pas plus que ce que l'on a accumulé.

Exemple. Calculons un coût amorti pour un AJOUT au sein d'une suite d'AJOUT sur un tableau dynamique par la méthode du comptable. On fixe les accumulations et dépenses ainsi :

- Si l'appel à AJOUT n'a pas besoin de doubler le tableau, on paye 1 pièce (pour l'écriture de x), et on stocke 2 autres pièces sur la case que l'on vient d'écrire, afin d'être plus tard capable de payer un doublement du tableau.

Ainsi, toutes les cases qui ont été écrites depuis la dernière fois que le tableau a été doublé contiennent 2 pièces posées sur elles. Autrement dit, toutes les cases écrites de la deuxième moitié du tableau contiennent 2 pièces accumulées.

Le coût amorti pour un tel appel à AJOUT est donc $\hat{C}_i = 3$.

- Sinon, il faut payer 1 (pour écrire x) et en plus payer la recopie des cases, donc payer autant de cases que le tableau en a. On va pour cela dépenser les 2 pièces accumulées sur chacune des cases de la deuxième moitié du tableau ! En dépensant ce stock, le coût de la recopie est entièrement couvert. On accumule enfin 2 pièces sur la nouvelle case, pour les mêmes raisons que précédemment.

Le coût amorti pour un tel appel à AJOUT est donc $\hat{C}_i = 3$.

On a ainsi prouvé par la méthode du comptable qu'au sein d'une suite d'appels à AJOUT, un appel à AJOUT a un coût amorti $\hat{C}_i = 3$.

FIGURE 3.5 – Schéma de l'évolution du tableau dynamique et de l'accumulation de la méthode du comptable

Remarque.

- Dans la preuve précédente, on n'a pas toujours explicitement indiqué les valeurs de d_i , C_i et a_i . Je vous conseille de les indiquer si vous avez besoin de repères clairs pour avancer votre preuve (après tout, le choix de bons a_i et d_i sont tout l'enjeu de la méthode), mais de ne pas les mettre s'ils ne feraient qu'alourdir la rédaction. Par contre, ils ne doivent jamais être ambigus : on doit pouvoir comprendre quelle serait leur valeur !
- La méthode du comptable s'applique quand on arrive à bien prévoir quelle opération peut prépayer quelle autre opération. Ici, lors de l'ajout d'un élément on pré-paye pour la recopie future de cet élément ainsi que pour la recopie d'un élément de la première moitié du tableau.

3.0.2 Méthode du potentiel

La méthode du potentiel revient à faire accumuler et dépenser un potentiel « global », au lieu de petits stocks locaux de pièces. La différence est que l'on définit le potentiel par une formule (à trouver, qui doit répondre à nos besoins), et que c'est de cette formule que l'on déduit les accumulations/dépenses.

Définition 18 (Méthode du potentiel).

On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1.

La **méthode du potentiel** est le fait d'associer à chaque état de la structure un potentiel ($P(0)$ est le potentiel initial, $P(1)$ le potentiel après la première opération, etc). On doit garantir que :

$$\forall i, P(i) \geq P(0)$$

La méthode du potentiel définit alors le coût amorti suivant :

$$\hat{C}_i = C_i + P(i) - P(i-1)$$

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned}
\sum_{i=1}^n \hat{C}_i &= \sum_{i=1}^n (C_i + P(i) - P(i-1)) \\
&= \left(\sum_{i=1}^n C_i \right) + \left(\sum_{i=1}^n P(i) - P(i-1) \right) \\
&= \left(\sum_{i=1}^n C_i \right) + P(n) - P(0) \\
&\geq \sum_{i=1}^n C_i \quad \text{car } P(n) \geq P(0)
\end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Remarque. Toute la difficulté est de trouver un bon potentiel. On veut une fonction qui :

- Augmente un peu lorsque l'on fait une opération peu coûteuse, pour « accumuler du potentiel ».
- Décroit d'un coup lors d'une opération coûteuse. Cette décroissance du potentiel va compenser le coût de l'opération : elle revient à « dépenser » le potentiel accumulé.

FIGURE 3.6 – Évolution typique d'un bon potentiel

Exemple. Calculons un coût amorti pour un AJOUT au sein d'une suite d'AJOUT sur un tableau dynamique (initialement vide) par la méthode du potentiel. Nommons E_0, \dots les états successifs du tableau, et numérotions les opérations à partir de 1 : ainsi, on passe de l'état E_{i-1} à l'état E_i par l'opération numéro i .

Un état E_i du tableau dynamique contient len_i le nombre de cases utilisées dans le tableau, ainsi que $dispo_i$ le nombre de cases dont le tableau dispose vraiment. On définit le potentiel suivant :

$$P(i) = 2len_i - dispo_i$$

Calculons le coût amorti associé :

- Si l'appel à AJOUT n'a pas besoin de doubler le tableau, le coût réel est 1 (on écrit x). Calculons la différence $P(i) - P(i-1)$. Comme le tableau n'a pas été doublé, $dispo_i = dispo_{i-1}$ et donc $P(i) - P(i-1) = 2(len_i - len_{i-1}) = 2$.

Le coût amorti pour un tel appel à AJOUT est donc $\hat{C}_i = 3$.

- Sinon, il faut payer 1 (pour écrire x) et en plus payer la recopie de toutes les cases du tableau, donc un coût total de $1 + dispo_{i-1}$. Comme le tableau a été doublé, $dispo_i = 2dispo_{i-1}$ et $len_{i-1} = dispo_{i-1}$. On a donc :

$$\begin{aligned}
P(i) - P(i-1) &= dispo_{i-1} - dispo_i + 2(len_i - len_{i-1}) \\
&= dispo_{i-1} - 2dispo_{i-1} + 2 \\
&= 2 - dispo_{i-1}
\end{aligned}$$

Le coût amorti pour un tel appel est $1 + \text{dispo}_{i-1} + 2 - \text{dispo}_{i-1}$, donc : $\hat{C}_i = 3$.

On a ainsi prouvé par la méthode du potentiel qu'au sein d'une suite d'appels à AJOUT, un appel à AJOUT a un coût amorti $\hat{C}_i = 3$.

Remarque. La méthode du potentiel s'applique lorsque l'on n'arrive pas à prévoir quelle opération pré-paye laquelle. À la place, on cherche comment doit évoluer le stock global de « pièces » et ce stock global que l'on définit.

3.0.3 Méthode de l'aggrégat

La méthode de l'aggrégat est plus mathématique. Elle consiste à calculer la somme $\sum_{i=1}^n C_i$ en la décomposant en « paquets » simples à calculer. On en déduit le coût total de la suite d'opérations, que l'on peut ensuite lisser sur les opérations.

Définition 19 (Méthode de l'aggrégat).

La **méthode de l'aggrégat** consiste à poser $\hat{C}_i = \frac{1}{n} \sum_{i=1}^n C_i$.

Toute la difficulté est donc d'avoir réussi à calculer cette somme. On peut utiliser une majoration très précise à la place de la somme.

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned} \sum_{i=0}^{n-1} \hat{C}_i &= \sum_{i=0}^{n-1} \left(\frac{1}{n} \sum_{j=0}^{n-1} C_j \right) \\ &= n \left(\frac{1}{n} \sum_{j=0}^{n-1} C_j \right) && \text{car le contenu de la seconde somme ne dépend pas de } i \\ &= \sum_{j=0}^{n-1} C_j \end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Exemple. Calculons un coût amorti pour un AJOUT au sein d'une suite d'AJOUT sur un tableau dynamique (initialement vide) par la méthode de l'aggrégat. On numérote les opérations à partir de 1. Par une récurrence immédiate, on montre qu'après l'AJOUT numéro i , le tableau contient i valeurs. On montre de même que l'on double le tableau lorsque $i - 1$ est une puissance de 2, et que dans ce cas on recopie toutes les valeurs donc $i - 1$ valeurs.

Le coût C_i d'une opération est 1 (écriture de x), plus éventuellement le doublement. Donc :

$$C_i = \begin{cases} 1 + i - 1 & \text{si } i - 1 = 2^p \text{ avec } p \in \mathbb{N} \\ 1 & \text{sinon} \end{cases}$$

Donc :

$$\begin{aligned} \sum_{i=1}^n C_i &= \sum_{i=1}^n 1 + (i - 1) \cdot \mathbb{1}_{\langle i-1 \text{ est de la forme } 2^p \rangle} \\ &= n + \sum_{p=0}^{\lfloor \log_2 n \rfloor} 2^p \\ &= n + 2^{\lfloor \log_2 n \rfloor + 1} - 1 \\ &\leq 3n \end{aligned}$$

La méthode de l'aggrégat propose alors $\hat{C}_i = \frac{3n}{n} = 3$ comme coût amorti.

Remarque. Cette méthode est à utiliser quand on connaît une propriété de la structure qui permet de faire des « paquets » agréables, ou que l'on arrive à réorganiser la somme d'une façon qui lui donne plus de sens.

3.1 Remarques et compléments

- **L'une des applications classiques de la complexité amortie est le calcul de la complexité d'une boucle qui opère à chaque itération sur une structure de données.** Voici un exemple en Python, où `append` correspond à `AJOUT` :

```

1 lst_premiers = []
2 for x in range(0, n): # pour x allant de 0 inclus à n exclu
3     if is_prime(x):
4         lst_premiers.append(x)

```



Le calcul de la complexité de cette boucle est très simple si on utilise l'analyse amortie pour affirmer que l'on peut faire comme si chacun des `append` est en $O(1)$ (puisque l'on effectue une succession de `append` sur une même liste), et assez compliqué sinon.

- **Il ne faut pas hésiter à faire un schéma** pour illustrer la structure, les accumulations/dépenses ou le potentiel. Un bon schéma permet d'écrire ensuite une preuve plus concise et plus claire.¹⁵
- Les trois méthodes de calcul reviennent finalement à trouver une façon de majorer précisément une somme. Elles sont cependant expliquées de manière différente. Prenez la méthode qui vous parle le plus (sauf si dans votre situation précise une des méthodes est clairement plus simple que les autres).
- On cherche surtout à majorer et donc à calculer un $O()$. On pourrait adapter les raisonnements pour minorer et obtenir un $\Omega()$ ou un $\Theta()$, mais c'est plus rare.
- **L'état initial de la structure étudiée n'a pas beaucoup d'importance.** Quitte à majorer, on peut ajouter des opérations initiales « fictives » qui amènent l'état initial dans un état plus agréable, sans que cela ne change l'ordre de grandeur du résultat. En effet, les calculs d'un coût amorti lissent des coûts sur un nombre très grands d'opération. Les opérations fictives auront un coût asymptotiquement nul au sein de ce très grand nombre d'opération¹⁶.
- **Un coût amorti n'a de sens qu'au sein de la suite d'opérations où on l'a calculé.** On a prouvé dans ce cours qu'*au sein d'une suite d'AJOUT*, chaque `AJOUT` a un coût amorti constant. Mais on pourrait imaginer une nouvelle opération, `DÉMON` qui remplit le tableau dynamique jusqu'à ce que son nombre d'éléments soit une puissance de 2. Avec cette nouvelle opération, chacun des `AJOUT` va doubler le tableau ; et donc le meilleur coût amorti possible pour `AJOUT` au sein d'une suite d'opérations qui alterne `AJOUT` et `DÉMON` est un coût linéaire en la longueur du tableau.
- **Un coût amorti n'a rien à voir avec la complexité moyenne**¹⁷. Ce n'est juste pas la même définition ! Un coût amorti est défini sur *une suite d'opérations*, la complexité moyenne sur *une* opération (mais en moyennant sur les entrées possibles pour cette seule opération).

15. Ce conseil s'applique à l'entiereté de l'informatique (et de la plupart des sciences). Aidez à visualiser !

16. Mathématiquement, on dit que la somme partielle d'une série qui diverge vers $+\infty$ est négligeable devant son reste.

17. J'ai d'ailleurs fait très attention à ne jamais utiliser le mot « moyenne » dans toute la section sur la complexité amortie.

Chapitre 4

RÉCURSIVITÉ

Notions	Commentaires
Réversibilité d'une fonction. Réversibilité croisée. Organisation des activations sous forme d'arbre en cas d'appels multiples.	La capacité d'un programme à faire appel à lui-même est un concept primordial en informatique. [...] On se limite à une présentation pratique de la récursivité comme technique de programmation.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

SOMMAIRE

0. Introduction	72
0. Notion de réduction	72
1. Notion de récursion	72
1. Exemples	73
0. Affichage d'un triangle	73
1. Tours de Hanoï	74
2. Exponentiation rapide	76
3. À propos des boucles	77
<i>Transformation boucle <-> récursion (p. 77). Récursivité terminale (hors-programme) (p. 78).</i>	
2. Comment concevoir une fonction récursive	78
3. Analyse de fonctions récursives.....	79
4. Compléments	79
0. Élimination des appels redondants	79
1. Avantages et inconvénients de la récursion	79
2. Querelle sémantique	80
3. Quand « déplier les appels » ?	80

0 Introduction

0.0 Notion de réduction

Rappels du cours introductif : Un **problème** est composé de deux éléments : la description d'une **instance** (une « entrée »), et une question/tâche à réaliser sur cette instance.

Définition 1 (Réduction).

On dit qu'une instance d'un problème A se *réduit* à une instance d'un problème B si résoudre cette instance de B suffit à résoudre cette instance de A .

On dit que le problème A se réduit au problème B si toute instance de A se réduit à une instance de B . On note parfois $A \preceq B$.

Remarque.

- Dit autrement, il s'agit d'utiliser le fait que l'on sait déjà résoudre un problème B afin de résoudre un problème A . C'est l'une des méthodes courantes en science : se ramener à ce que l'on sait faire !
- Vous approfondirez cette notion en MPI, notamment en imposant des conditions sur les liens entre les instances.
- Notez qu'il n'y a pas besoin de savoir *comment* B est résolu. D'un point de vue pratique, il suffit d'avoir accès à une fonction qui résout B (sans même savoir comment elle marche) afin de résoudre A .

Exemple.

- Posons A le problème de calculer le PGCD de deux entiers (une instance est la donnée de deux entiers), et B le problème de calculer le PPCM de deux entiers.
Alors A se réduit à B . En effet, soit (x, y) une instance de A . Utilisons (x, y) comme une instance de B . Or, d'après le cours de maths, on sait que $|xy| = \text{pgcd}(x, y) \cdot \text{ppcm}(x, y)$ et donc que $\text{pgcd}(x, y) = \frac{|xy|}{\text{ppcm}(x, y)}$.
Ainsi, résoudre l'instance (x, y) de B suffit à calculer $\text{pgcd}(x, y)$ et donc à résoudre l'instance (x, y) de A . Il s'ensuit que A se réduit à B .
- On peut montrer que réciproquement, PPCM se réduit à PGCD.

0.1 Notion de récursion

L'idée centrale de la récursion est de réduire une instance d'un problème à une autre instance, plus petite, de ce même problème.

Exemple.

- Le calcul d'une suite récurrente en maths : si par exemple $u_{n+1} = f(u_n)$, alors l'instance « calculer le terme $n + 1$ » se réduit à l'instance « calculer le terme n ».
- Le calcul des coefficients binomiaux : pour calculer $\binom{n}{k}$, d'après le cours de maths il suffit de calculer $\binom{n-1}{k}$ et $\binom{n-1}{k-1}$.
- Dessiner une fractale : une fractale est un dessin qui se répète dans lui-même. Ainsi, pour dessiner une fractale « grande », il faut commencer par dessiner des fractales plus « petites » dedans.

Exercice. Montrez que le problème de trouver le minimum d'un tableau se réduit au problème de trier un tableau.

Cette méthode de résolution de problèmes se généralise à l'écriture des fonctions :

Définition 2 (Fonction récursive).

Une **fonction récursive** est une fonction qui peut faire un ou plusieurs appels à elle-même. Le contenu d'une fonction récursive est composée de :

- Cas récursif : les appels que la fonction fait à elle-même, et ce qu'elle en fait.
- Cas de base : ce que fait la fonction quand elle ne peut pas s'appeler elle-même.

Exemple.

```

9  /** Renvoie x^n. */
10 double exp_naif(double x, unsigned n) {
11     if (n > 0) {
12         return x * exp_naif(x, n-1);
13     } else { // n == 0
14         return 1;
15     }
16 }

```

exemples.c

La fonction ci-contre calcule x^n (avec $n \geq 0$). Pour cela :

$$x^n = \begin{cases} x \cdot x^{n-1} & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

On peut représenter visuellement le déroulement de cette fonction sur des exemples :

(a) `exp_naif(1.5, 4)`

(b) `exp_naif(2.0, 5)`

FIGURE 4.1 – Visualisation d'appels à `exp_naif`

1 Exemples

1.0 Affichage d'un triangle

Écrivons une fonction qui affiche un triangle de n lignes d'étoiles, comme ceci :

```

***
**
*

```

(a) Triangle de $n=3$ étoiles

```

*****
****
***
**
*

```

(b) Triangle de $n = 5$

FIGURE 4.2 – Des triangles

Essayons d'écrire une fonction récursive pour ce problème. Pour cela, cherchons une « structure récursive » dans le problème.

On remarque que le triangle $n-1$ est inclus dans le triangle n :

```
*****
****
***
**
*
```

FIGURE 4.3 – Inclusion des triangles

On en déduit la solution récursive suivante :

Fonction Triangle

Entrées : $n \geq 0$

```
1 si n > 0 alors
2   Afficher une ligne de n étoiles
3   Triangle(n-1)
```

Remarque. Pour bien comprendre cette fonction, il faut la faire tourner à la main.

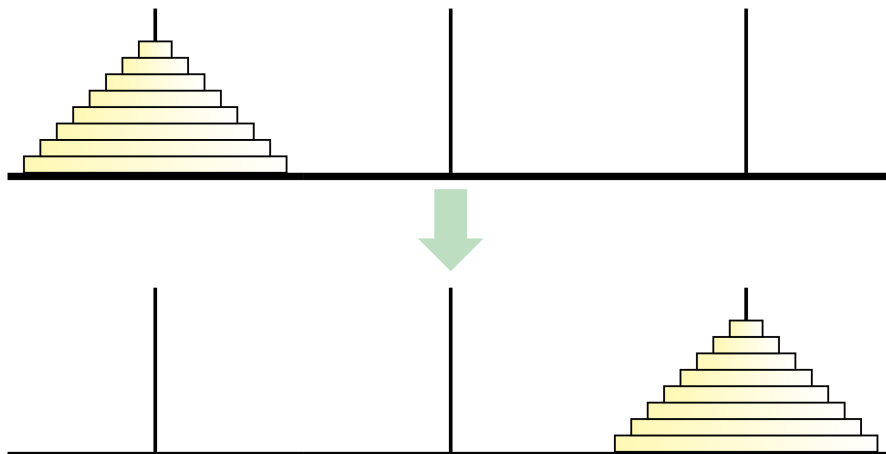
Exercice. Écrire cette fonction en C, et la tester. *Il n'y a aucune syntaxe particulière à utiliser pour la récursivité en C.*

Exercice. Faire de même mais avec un triangle "pointe en haut".

1.1 Tours de Hanoï

Les tours de Hanoï sont un casse-tête très célèbre :

- On dispose de 3 tiges (emplacements), sur lesquelles se trouvent des disques, empilés les uns sur les autres.
- Il y a un total de n disques. Ils sont caractérisés par leur diamètre, qui sont deux à deux distincts. Un disque ne peut jamais reposer sur un disque plus petit.
- On peut déplacer le disque qui est au sommet d'une pile en le plaçant au sommet d'une autre (en respectant la règle des diamètres!).
- Initialement, tous les disques sont sur une même tige. Le but est de tous les déplacer sur une autre des 3 tiges.

FIGURE 4.4 – Situation initiale et finale des tours de Hanoï avec $n=8$ disques (src : *Algorithms*, J. Erikson)

Exemple. Voici une succession de déplacements qui résout les tours de Hanoï à 3 disques :

FIGURE 4.5 – Résolution de Hanoï à 3 disques

On veut écrire une solution récursive au problème : on veut trouver une méthode pour déplacer n disques d'une tige vers une autre. Pour cela, **on suppose que l'on a des camarades très intelligent-es qui savent déjà le résoudre pour des instances plus petites**. L'objectif est de réussir à utiliser leur aide pour résoudre notre instance.

En bidouillant un peu, on réalise qu'une étape intermédiaire inévitable dans toute solution est de placer le plus gros disque (celui du fond de la pile de départ) sur la tige d'arrivée. Il faut donc réussir à enlever les autres disques d'au-dessus de lui, et comme il est plus gros que les autres il faut que la tige d'arrivée soit vide quand on le place. Autrement dit, il faut déplacer les $n-1$ premiers disques sur la tige qui n'est ni départ ni arrivée.

...¹

Mais en fait « déplacer $n-1$ premiers disques d'une tige à une autre », c'est exactement une autre instance du problème ! Nos camarades peuvent donc le résoudre ! Il nous suffit ensuite de déplacer le gros disque sur la tige d'arrivée, puis de re-déplacer les $n-1$ premiers.

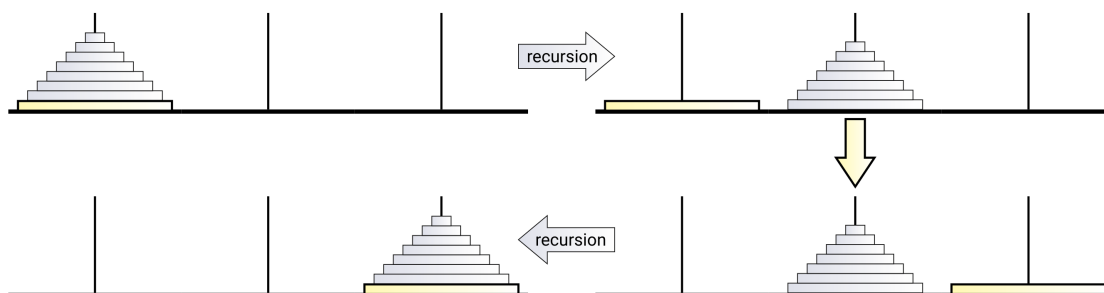


FIGURE 4.6 – Le fonctionnement récursif de l'algorithme (src : *Algorithms*, J. Erikson)

1. Insérer roulements de tambours dramatiques.

On a arrêté de réfléchir. On ne cherche SURTOUT PAS à « déplier » les appels récursifs : nos camarades sont très intelligent-es et on leur fait confiance !!!!!!!!!!!

Il reste une subtilité : quand $n = 0$, l'étape « déplacer $n - 1$ disques » n'a pas de sens. Dans ce cas, on ne *peut pas* appliquer notre méthode récursive et il faut résoudre cette instance par nous-même. Heureusement, déplacer $n = 0$ disques n'est pas trop compliqué... On obtient donc le pseudo-code suivant :

Fonction Hanoï

Entrées : i : la tige de départ; j : la tige d'arrivée; n : le nombre de disques à déplacer

```

1 si  $n > 0$  alors
2    $k \leftarrow$  tige autre que  $i$  ou  $j$ 
3   HANOÏ( $i$ ,  $k$ ,  $n-1$ )
4   Déplacer 1 disque de  $i$  à  $j$ 
5   HANOÏ( $k$ ,  $j$ ,  $n-1$ )
```

Exemple. En fait, la résolution précédente de Hanoï à 3 disques était déjà une application de cet algorithme !

Exercice. Appliquer cet algorithme récursif pour $n = 4$. *Conseil :* prévoyez beaucoup de place, et notez les appels récursifs au fur et à mesure; votre mémoire ne suffira pas.

Remarque.

- J'insiste : pour trouver cette solution, nous n'avons **pas** cherché à comprendre comment les camarades intelligent-es procèdent !! Les camarades / appels récursifs sont des « boîtes noires » qui résolvent les autres instances, sans que l'on ait à comprendre comment.
- De manière cachée, nous avons commencé par **généraliser** : on n'a pas directement résolu le problème de « déplacer n tiges du départ à l'arrivée », on a résolu le problème de « déplacer les n plus petits disques d'une tige (où ils doivent tous être) vers une autre ».
- En particulier, ce qui fait marcher cet algorithme est que l'on essaye toujours de déplacer les disques les plus petits : même si on ne le pose pas sur des tiges vides (la tige intermédiaire est rarement vide), on les pose sur des disques plus gros : tout va bien.
- Il est formateur d'essayer d'analyser la complexité de cet algorithme : cf cours complexité.

1.2 Exponentiation rapide

La fonction `exp_naïf` précédente est une implémentation de la fonction $(x, n) \mapsto x^n$. Si on compte sa complexité en nombre de multiplications $M(n)$, on obtient l'équation $M(n) = 1 + M(n-1)$ et $M(0) = 0$, donc $M(n) = n$.

Cela peut sembler bien, mais c'est en fait assez mauvais². Il existe une méthode plus maline, basée sur la remarque suivante : notons $p = \lfloor \frac{n}{2} \rfloor$ et utilisons :

$$x^n = \begin{cases} x \cdot x^p \cdot x^p & \text{si } n = 2p + 1 \text{ et } n > 0, \text{ c'est à dire si } n \text{ est impair et } > 0 \\ x^p \cdot x^p & \text{si } n = 2p \text{ et } n > 0, \text{ c'est à dire si } n \text{ est pair et } > 0 \\ 1 & \text{sinon, c'est à dire si } n = 0 \end{cases}$$

2. Vous en parlerez plus en MPI, mais faisons un brin de hors-programme : la complexité « pertinente » s'exprime en fonction de la taille des entrées. n s'écrit sur $\log_2 n$ bits, donc une complexité en $\Theta(n)$ est exponentielle en la taille de l'entrée. Et l'exponentiel, c'est beaucoup.

On obtient le code suivant :

```

22  /** Renvoie x^n. */
23  double exp_rap(double x, unsigned n) {
24      if (n > 0) {
25          int x_p = exp_rap(x, n/2); // x puissance p avec p = n/2
26          if (n % 2 == 0) { return x_p * x_p; }
27          else { return x * x_p * x_p; }
28      }
29      else {
30          return 1;
31      }
32  }

```



Ou en OCaml :

```

3  (** Renvoie x^n (avec n >= 0) *)
4  let rec exp_rap x n =
5      if n > 0 then
6          let x_p = exp_rap x (n/2) in (* x puissance p avec p = n/2 *)
7          if n mod 2 = 0 then x_p *. x_p else x *. x_p *. x_p
8      else
9          1.

```



La complexité $M(n)$ de cet algorithme-ci vérifie $M(n) \leq 2 + M(n/2)$ et $M(0) = 0$. On en déduit³ $M(n) = O(\log_2 n)$, ce qui est *bien* mieux que la complexité précédente !

Remarque.

- On être plus précis sur la complexité de cet algorithme. Notons N_0 le nombre de 0 dans l'écriture binaire de n , et N_1 celui de 1 (on a en particulier $N_0 + N_1 = \log_2 n$).

Avec ces notations, on peut⁴ montrer que $M(n) = N_0 + 2N_1$ qui est bien un $O(\log_2 n)$.

- On peut prouver qu'il faut $\Omega(\log_2 n)$ opérations élémentaires. En effet, on peut prouver qu'il *faut* lire chacun des bits de n .

Pour cela, raisonnons par l'absurde et supposons qu'il existe un algorithme totalement correct qui fonctionne sans lire tous les bits de l'entrée n . Soit $n \in \mathbb{N}$ tels qu'un bit de n ne soit pas lu. Notons m l'entier obtenu en changeant la valeur de ce bit non lu. Alors l'algorithme renvoie la même réponse sur x^m et x^n ... mais ces deux valeurs sont distinctes⁵ : l'algorithme n'est donc pas correct, absurde.

Il faut donc lire chacun des bits de n , et donc effectuer au moins $\Omega(\log_2 n)$ opérations élémentaires. Sachant cela, effectuer $O(\log_2 n)$ multiplications est très satisfaisant.

⚠ Attention, une erreur commune avec cet algorithme est d'effectuer deux appels récursifs à chaque fois (ce que l'on évite dans les codes ci-dessus en stockant le résultat de l'appel dans une variable). Si on fait cela, la complexité redevient⁶ $O(n)$.

1.3 À propos des boucles

1.3.0 Transformation boucle <-> récursion

La plupart des boucles peuvent être réécrites par une fonction récursive. L'idée est de faire une fonction qui « fait une itération », puis qui s'appelle elle-même pour faire la prochaine itération. Ainsi, les deux codes ci-dessous sont équivalents :

3. Pour cela, remarquer que $M(1) = 2$ simplifie la manipulation des log.

4. Il faut étudier l'écriture binaire de n durant la suite d'appels récursive, et utiliser le fait que le dernier bit indique la parité

5. Sauf pour $x = 0$ ou $|x| = 1$, d'accord.

6. Utiliser la méthode de calcul de complexité par arbre.

```

34
35 /** Somme des entiers jusqu'à n
   ↳ inclus. */
36 int somme_entiers(int n) {
37     int i = 0;
38     int somme = 0;
39     while (i <= n) {
40         somme = somme + i;
41         i = i + 1;
42     }
43     return i;
44 }

```

```

12
13 (** Somme des entiers jusqu'à n
   ↳ inclus *)
14 let somme_entiers n =
15     let rec boucle somme i =
16         if i <= n then
17             boucle (somme+i) (i+1)
18         else
19             somme
20     in
21     boucle 0 0

```

Ici, la boucle C a été transformée en la fonction boucle. Dans la boucle C, on a $\text{somme}' = \text{somme} + i$ et $i' = i + 1$. La version récursive fonctionne en calculant ces valeurs prime (donc en « simulant une itération »), puis en s'appelant elle-même pour simuler les itérations suivantes.

C'est une transformation classique, que l'on fera souvent lorsque l'on travaille en OCaml.

Remarque. La transformation inverse (récursion \rightarrow boucle) est possible, mais plus technique lorsqu'il y a plusieurs appels récursifs (il faut simuler la pile d'appels à la main).

1.3.1 Récursivité terminale (hors-programme)

Si une fonction récursive s'appelle *une seule fois*, et que cet appel est *la toute dernière chose* effectuée, alors il est très simple de transformer la fonction récursive en boucle. C'est une optimisation faite par le compilateur OCaml⁷ !

Cela permet de gagner :

- un peu de temps, car passer d'une itération d'une boucle à la suivante est plus rapide que d'ouvrir un appel récursif
- beaucoup d'espace, car il n'y a pas besoin d'utiliser de l'espace pour chacun des appels successifs.

⚠ Écrire une fonction récursive terminale demande parfois de modifier le code de manière peu lisible. Il faut d'abord faire une version lisible et correcte et ensuite seulement une version optimisée.

2 Comment concevoir une fonction récursive

Proposition 3 (Concevoir une solution récursive).

Pour concevoir une solution récursive à un problème sur une instance I :

- On suppose que l'on a une boîte noire magique (la récursion) qui peut résoudre toutes les instances du problème sauf I .
- On cherche un lien entre I et une ou plusieurs autres instances du problème. Souvent, cela implique de « reconnaître le problème dans le problème » (comme pour les triangles) ou de « exprimer $f(n)$ à l'aide de $f(n-1)$ ».
- On cherche les cas où le lien trouvé ne peut pas être utilisé : ce sont les cas de base, et on doit en trouver une solution sans récurrence.
- On vérifie que les instances « diminuent » d'un appel sur l'autre, afin de garantir que la suite d'appel finit par atteindre un cas de base et termine.

7. Et que gcc peut faire si on le lui demande, et que cela n'entre pas en conflit avec d'autres optimisations.

3 Analyse de fonctions récursives

Les méthodes pour analyser la terminaison, la correction et la complexité d'une fonction récursive sont dans les cours en question.

4 Compléments

4.0 Élimination des appels redondants

Lorsque l'on écrit des fonctions récursives, il arrive que certaines instances plus petites soient résolues plusieurs fois. Voici par exemple une fonction (très peu recommandée) pour calculer $\binom{n}{k}$:

```

24 (** Renvoie k parmi n *)
25 let rec binom n k =
26   if k = 0 || k = n then
27     1
28   else
29     ( binom (n-1) k ) + ( binom (n-1) (k-1) )

```

 exemples.ml

Si on applique cette méthode et que l'on essaye par exemple de calculer $\binom{5}{2}$, certains appels sont calculés plusieurs fois :

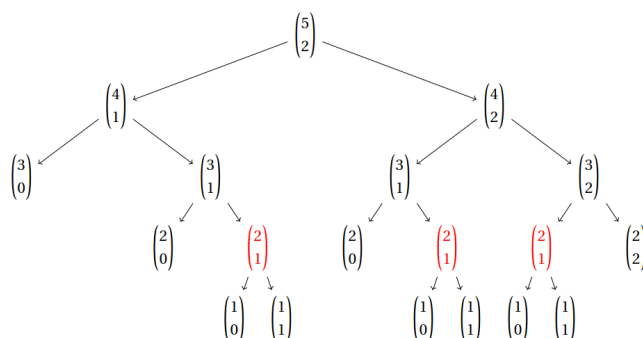


FIGURE 4.7 – Arbre d'appels de `binom 5 2`. Une redondance d'appels est mise en exergue en rouge. (src : J. Larochette)

Ces appels redondants ralentissent le fonctionnement, puisque l'on recalcule plusieurs fois la même chose. Il est plus efficace de mémoriser ces valeurs pour ne pas avoir à les recalculer ensuite : c'est l'objet de la programmation dynamique, que nous verrons au second semestre.

4.1 Avantages et inconvénients de la récursion

Avantages. La récursivité :

- peut produire un code plus lisible et plus simple à comprendre.
- produit un code souvent plus simple à prouver (car les hypothèses de récurrence / invariants sont plus simples).
- permet de résoudre des problèmes très difficiles à résoudre sans !!
- est très adaptée au travail sur les structures de données naturellement récursives (dont les arbres).
- est très adaptée aux problèmes naturellement récursifs.

Inconvénients. La récursivité :

- est plus éloignée du fonctionnement de la machine, et est ralentie par le temps nécessaire à l'ouverture d'un appel de fonction.
- utilise de l'espace mémoire en empilant les appels.
- produit *parfois* un code moins clair qu'un code itératif⁸ plus direct.

En résumé, c'est un outil extrêmement puissant. Comme tout outil, il n'est pas absolu et a ses limites d'utilisation. Comme tout bon outil, il rend simple des tâches difficiles. Un grand pouvoir implique de grandes responsabilités !

4.2 Querelle sémantique

Différents termes existent pour parler de la récursivité :

- « récursivité », « fonction récursive » sont les termes usuels en français.
- « récurrence » est également utilisé (c'est de toute façon la même notion qu'en mathématiques).
- « récursion » est une anglicisme du terme anglais « recursion », souvent utilisé.
- « induction », « fonction inductive » sont des anglicismes venant du terme anglais « induction », qui signifie « récurrence ».

À titre personnel, j'utiliserai et mélangerai toutes ces versions. Je connais des auteurs/autrices qui restreignent certains termes⁹ : adaptez-vous à ce que vous dit la personne avec qui vous interagissez.

4.3 Quand « déplier les appels » ?

J'ai insisté sur ce cours sur l'importance de ne pas « déplier les appels » lorsque l'on conçoit un algorithme récursif. Il y a cependant certains cas où déplier les appels peut-être utile :

- Pour tracer un arbre d'appel, il faut bien déplier un peu.
- Pour déboguer, il peut-être bon de déplier dans sa tête ou à la main les premiers ou les derniers appels. Cela permet de se rendre compte d'erreurs de non-respect de pré-condition, ou du besoin d'une pré-condition.

8. Un code itératif est un code non-récursif.

9. J'ai hésité à le faire aussi, mais je juge qu'à notre modeste niveau, le gain gagné par ces subtilités ne vaut pas l'effort de distinction.

Chapitre 5

STRUCTURES DE DONNÉES (S1)

Notions	Commentaires
Définition d'une structure de données abstraite comme un type muni d'opérations.	On parle de constructeur pour l'initialisation d'une structure, d'accessor pour récupérer une valeur et de transformateur pour modifier l'état de la structure. On montre l'intérêt d'une structure de données abstraite en terme de modularité. On distingue la notion de structure de données abstraite de son implémentation. Plusieurs implémentations concrètes sont interchangeables. La notion de classe et la programmation orientée objet sont hors programme.
Distinction entre structure de données mutable et immuable.	Illustrée en langage OCaml.

Extrait de la section 3.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Structure de liste. Implémentation par un tableau, par des maillons chaînés.	On insiste sur le coût des opérations selon le choix de l'implémentation. Pour l'implémentation par un tableau, on se fixe une taille maximale. On peut évoquer le problème du redimensionnement d'un tableau.
Structure de pile. Structure de file. Implémentation par un tableau, par des maillons chaînés.	

Extrait de la section 3.2 du programme officiel de MP2I : « Structures de données séquentielles ».

SOMMAIRE

0. Tableaux dynamiques	82
0. Tableaux C vs listes OCaml	82
1. Fonctionnement	83
<i>Ajout dans un tableau dynamique (p. 84). Création, suppression (p. 85).</i>	
2. Analyse de complexité	85
3. Complément mi hors-programme : RETRAIT	85
1. Interfaces et types abstraits.....	87
0. Aspects pratiques	87
<i>Librairies (p. 87). Interface (p. 88). Étapes de la compilation (p. 90).</i>	
1. Aspects théoriques	91
2. Structures de données séquentielles	93
0. Types de base	93
1. Listes linéaire	93
<i>Implémentation par tableaux de longueur fixe (p. 93). Implémentation par tableaux dynamiques (p. 94). Implémentation par listes simplement chaînées (p. 94). Variantes des listes chaînées (p. 97).</i>	
2. Piles	97
<i>Implémentations par tableaux (p. 98). Implémentation par listes simplement chaînées (p. 99).</i>	

3. Files	100
<i>Implémentations par tableaux circulaires (p. 100). (Hors-programme) Amélioration avec des tableaux dynamiques (p. 103). Implémentations par listes simplement chaînées (p. 103). Implémentation par listes doublement chaînées (p. 104). Par double pile (p. 104).</i>	
4. Variantes des files	105
3. Aperçu des structures de données S2	105

0 Tableaux dynamiques

0.0 Tableaux C vs listes OCaml

Les tableaux sont un outil très utile de la programmation impérative¹. Ils permettent de stocker beaucoup d'éléments tout en ayant un accès à chacun de ces éléments en temps constant ; mais ont un inconvénient : ils ne sont pas redimensionnables. Si une fonction utilise un tableau à 200 cases, et qu'elle découvre durant l'exécution que sur certaines entrées il en fallait en fait plus, on est bloqués². Similairement, si on se rend compte qu'il en fallait beaucoup moins, on a gaspillé de la mémoire (et à force de gaspiller de la mémoire, on n'en a plus...)

Exemple.

- On veut faire une fonction qui prend en entrée un entier `n` et renvoie le tableau de ses diviseurs. Combien de cases réserver ? Pas clair. Il est suffisant d'en réserver `n`, bien sûr, mais on sent bien que c'est beaucoup trop.
- On veut faire une fonction prend en entrée un entier `n` et calcule le tableau des nombres premiers inférieurs ou égaux à `n`. On a exactement le même problème qu'avec le point précédent...

Et pourtant, en OCaml, ces exemples ne posent pas de difficulté particulière avec des listes. Voici par exemple pour les diviseurs de `n` :

```

3  (** Construit la liste des diviseurs positifs et >=k de n .
4     * On suppose n >= 0 . *)
5  let rec construit_lst_diviseurs n k =
6      if k > n then
7          []
8      else if n mod k = 0 then
9          k :: (construit_lst_diviseurs n (k+1))
10     else
11         construit_lst_diviseurs n (k+1)
12
13
14  (** Renvoie la liste des diviseurs positifs de n .
15     * On suppose n >= 0 . *)
16  let lst_diviseurs n =
17      construit_lst_diviseurs n 1

```



Les fonctions `lst_diviseurs` et `lst_preiers` renvoient exactement les éléments attendus, et ne surallouent aucune case : les listes créées ont exactement la même taille.

Mais à l'inverse, l'inconvénient des listes est que l'on ne peut pas accéder à un élément par son indice... et quand bien même on recoderait une fonction qui calcule le *i*ème élément d'une liste, elle serait en temps $\Theta(i)$ et non en temps constant.

1. Rappel : la **programmation impérative** est la méthode de programmation consistant à écrire un programme en décrivant une succession de modifications de la mémoire. C'est ce que l'on fait en C, par exemple.

2. Ici j'utilise 200 comme exemple, et on pourrait m'objecter que `malloc` permet de créer des "tableaux" à `n` cases. C'est vrai, mais si la fonction crée un tableau à `n` cases et se rend compte qu'il en fallait n^2 sur certaines entrées spécifiques, le problème reste le même.

Les **tableaux dynamiques** sont une structure de données qui offre le meilleur des deux mondes : ils permettent un accès à chaque élément en temps constant, et sont redimensionnables ce qui permet de les agrandir quand ils ne sont plus assez grand !

Remarque. Si vous connaissez les listes Python, ce sont des tableaux dynamiques³ !

0.1 Fonctionnement

L'objectif de nos tableaux dynamiques est de :

- Pouvoir accéder aux éléments par leur indice et les modifier en temps constant.
- Pouvoir ajouter un élément *à la fin* en temps constant⁴
- (Et aussi pouvoir créer et supprimer un tableau dynamique)

L'idée est de faire un tableau volontairement trop grand et de n'en utiliser que les premières cases. Comme ça, pour ajouter un élément à la fin, il suffit d'utiliser la prochaine case libre :

FIGURE 5.1 – Ajout dans un tableau dynamique, cas facile

Définition 1 (Tableaux dynamiques, structure de base).

Un tableau dynamique est composé de 3 données :

- le tableau en lui-même. Dans ce cours, on le nommera `arr` .
- son nombre réel de cases, c'est à dire le nombre de cases du tableau qui existent dans la mémoire. Dans ce cours, on le nommera `len_max` .
- son nombre de cases utilisées. Dans ce cours, on le nommera `len` .

Les `len` éléments stockés dans le tableau dynamique sont stockés, dans l'ordre, dans les `len` premières cases de `arr` .

Remarque. En C, `arr` ne sera pas un tableau mais une zone mémoire créée avec `malloc`. Dans cette partie, j'utilise le terme « tableau » pour simplifier. De plus, mis à part la création/destruction qui est différente, un tableau et une zone allouée s'utilisent de la même façon...

Cette façon de faire répond à presque toutes nos contraintes : on peut accéder à un élément par son indice en temps constant, modifier l'élément qui se trouve à un indice donné en temps constant ; on peut aussi ajouter un élément en temps constant lorsque `len < len_max` . Reste à définir comment ajouter un élément lorsqu'il n'y a plus de cases vides.

3. En un peu plus compliqué. Les listes Python sont une petite pépite de bonnes idées mises bout à bout pour que ce soit très efficace.

4. Ou au début, c'est symétrique.

0.1.0 Ajout dans un tableau dynamique

Une première idée serait de créer un nouveau tableau ayant une case de plus, de recopier l'ancien dedans, et d'ajouter l'élément à la fin. Hélas, cela n'est pas très efficace en temps :

FIGURE 5.2 – Ne faites pas l'AJOUT comme ça !!

Avec cette méthode, à partir du moment où le tableau est rempli, chaque ajout se fait en temps $O(\text{len_max})$. Ce n'est clairement pas efficace. On pourrait essayer de l'améliorer en créant un nouveau tableau non de longueur $\text{len_max}+1$ mais de longueur $\text{len_max}+10$, cependant cela ferait qu'1 opération sur 10 serait en temps linéaire (les 9 autres en temps constant), et donc chaque paquet de 10 opérations seraient en temps $O(\text{len_max} + 9.O(1) = O(\text{len_max}))$. Bref, si l'on prévoit de faire plusieurs ajouts, cela ne fonctionne pas très bien. Idem pour d'autres valeurs fixées de 10 : 1000, 10000, etc ont tous le même problème⁵.

La solution est de ne pas agrandir en rajoutant un nombre fixe de cases, mais un nombre variable : en fait, on va simplement *doubler* la longueur maximale !

FIGURE 5.3 – AJOUT successifs, avec doublements du tableau

5. Des petit-es malin-es objecteront que avec 10^9 on ets à peu près bons, car il est rare que l'on fasse 10^9 ajouts. C'est vrai, mais : 1) cela peut arriver, et 2) vous voulez vraiment que chaque tableau dynamique demande au moins 1Go de mémoire ?

La fonction AJOUT qui ajoute un élément à un tableau dynamique est donc la suivante :

Fonction Ajout

Entrées :

- un tableau dynamique composé de `arr`, `len` et `len_max`. On les considère passés par référence.
- `x` un élément à ajouter à la fin du tableau

Sorties : rien, mais a comme effet secondaire de modifier le tableau pour y ajouter `x`

```

1 si len < len_max alors
2   arr[len] ← x
3   len ← len + 1
4 sinon
5   new_arr ← tableau de longueur 2*len_max
6   // Recopier arr dans new_arr
7   pour chaque i ∈ [0; len[ faire
8     new_arr[i] ← arr[i]
9   // Remplacer arr par new_arr
10  arr ← new_arr
11  len_max ← 2*len_max
12  // Ajouter x
13  arr[len] ← x
14  len ← len + 1

```

Remarque. Notez que les lignes 2-3 et 10-11 sont les mêmes : on peut réécrire ce code sous la forme « si `arr` est entièrement rempli, le remplacer par `new_arr` ». Ensuite, dans tous les cas, ajouter `x` ».

0.1.1 Création, suppression

Pour supprimer un tableau dynamique, il suffit de supprimer `arr`, `len` et `len_max`.

Pour créer un tableau, il suffit de créer ces trois données, avec quand même un point précis : même si l'on crée un tableau dynamique initialement vide, il faut tout de même initialiser `len_max` à au moins 1 (et donc créer `arr` comme ayant au moins 1 case). En effet, si on prend 0, comme AJOUT double la longueur maximale... deux fois zéro ça fait zéro.

0.2 Analyse de complexité

Analyser la complexité de cette version de AJOUT n'est pas aisé : la plupart des appels à la fonction coutent $O(1)$ (quand on effectue uniquement les lignes 2-3), mais certains $O(\text{len})$ (les lignes 5-11). On va en fait analyser le coût d'une *succession* d'appels à AJOUT. Faire cette analyse est l'objet de la **complexité amortie** : cf cours sur la complexité, section complexité amortie.

On y prouve que :

Proposition 2 (Complexité amortie de AJOUT).

Une succession de n ajouts coûte dans le pire des cas, au total, $O(n)$. Autrement dit, le coût amorti de AJOUT est $O(1)$.

(Précisons que cette complexité amortie est bien celle d'un pire des cas d'une succession quelconque de AJOUTE.)

Démonstration. Cf cours complexité amortie (3 preuves différentes y sont données). □

0.3 Complément mi hors-programme : RETRAIT

Notre opération AJOUT est l'équivalent du `append` de Python. Si vous connaissez Python, vous savez qu'une autre opération est souvent utilisée : `pop`, qui supprime et renvoie le dernier élément. Dans ce

cours, on la nommera RETRAIT.

Il est en soi très facile de l'implémenter : enlever l'élément d'indice $\text{len}-1$ (et faire une erreur s'il n'y a aucun élément). Mais on voudrait faire mieux : pouvoir de temps en temps réduire `len_max`, afin de récupérer la mémoire qui ne sert plus.

La première idée, qui ne marche pas, est de faire le symétrique de AJOUT : lorsque la longueur est inférieure à la moitié de la longueur maximale, créer un tableau deux fois plus petit, etc etc. En soi, l'idée est très bonne : mais elle s'agence mal avec AJOUT. En effet, juste après un AJOUT qui « double » le tableau, on obtient un tableau presque à moitié vide. Ainsi, faire un RETRAIT juste après cet AJOUT mène à une contraction du tableau, refaire un AJOUT ensuite une extension, etc : on a une suite d'opération au sein de la quelle AJOUT et RETRAIT ne seraient *pas* en temps constant amorti. Bref, ça va pas.

FIGURE 5.4 – Pourquoi il ne faut pas contracter le tableau à la moitié

Il faut en fait « éloigner » le pire cas de RETRAIT (la contraction, qui crée un tableau plus petit) du pire cas de AJOUT. Pour ce faire, une solution simple est de **diviser par 2 la longueur maximale du tableau lorsqu'il est aux 3/4 vide**, et non lorsqu'il est à moitié vide :

FIGURE 5.5 – Fonctionnement de la suppression

Fonction Retrait**Entrées :**

- un tableau dynamique composé de `arr`, `len` et `len_max`. On les considère passés par référence.

Sorties :

- si `len = 0`, indique une erreur.
- sinon, le dernier élément de `arr` (c'est à dire le plus à droite). A également pour effet secondaire modifier le tableau pour y enlever cet élément.

```

1 si len = 0 alors renvoyer une erreur

   // Retirer le dernier élément x
2 sortie ← arr[len-1]
3 len ← len-1

   // Si besoin, contracter arr
4 si len ≤ len_max/4 et len_max ≥ 4 alors
5   | new_arr ← tableau de longueur len_max/2
6   | pour chaque i ∈ [0; len] faire
7   |   | new_arr[i] ← arr[i]
8   | arr ← new_arr
9   | len_max ← len_max/2

   // Ne pas oublier de renvoyer la sortie
10 renvoyer sortie
```

Proposition 3 (Complexité amortie de RETRAIT).

En codant RETRAIT et AJOUT comme présentés ici, on obtient pour ces deux opérations une complexité amortie $O(1)$.
(Précisons que cette complexité amortie est bien celle d'un pire des cas d'une succession quelconque de RETRAIT et AJOUT.)

Démonstration. Cf TD. □

1 Interfaces et types abstraits

1.0 Aspects pratiques

Remarque. Cette sous-partie du cours contient de nombreuses approximations. Ces concepts seront définis de manière plus précise et plus approfondie en cours de génie logiciel (en école).

1.0.0 Librairies

Lorsque l'on programme, il est usuel de découper son code en plusieurs fichiers : l'idée est que chaque fichier implémente « un paquet » cohérent de fonctions.

Définition 4 (Librairie et code client).

Un fichier constitué de plusieurs fonctions ayant pour but d'être utilisées dans d'autres fichiers est appelé une **librairie**.

Un code qui utilise une librairie est appelé un **code client**.

On appelle un tel paquet une **librairie**.

Exemple.

- `stdio` est une librairie C. Son nom signifie **standard in-out** : c'est la librairie standard (c'est à dire la librairie « officielle ») qui contient les fonctions de lecture (par exemple `scanf`) et d'écriture/affichage (par exemple `printf`) (`out`).
- `stdlib` (librairie standard généraliste qui contient notamment `malloc` et `free`), `stdbool` (librairie standard qui contient le type `bool`), `stdint` (librairie standard qui contient des types d'entiers sur 8/16/32/48/64 bits) ou encore `assert` (contient la fonction `assert`) sont des librairies C.

Remarque.

- Le terme « librairie » est un anglicisme qui s'est imposé. Vous croiserez peut-être à la place la traduction **bibliothèque logicielle**.
- En OCaml, on parle plutôt de **module**. En réalité, un module est un légèrement différent d'une librairie (c'en est une version évoluée), mais je ne ferai pas la différence à notre niveau.

Exemple.

- `List` est le module des listes en OCaml.
- `Printf` est le module qui contient la fonction `printf` (et ses parentes) en OCaml.
- (et nous en verrons d'autres cette année et l'an prochain)

Pourquoi programmer en plusieurs fichiers ?

- Ne pas tout recoder à chaque fois : une fois que `stdlib` est codée, pas besoin de la recoder ! Il suffit de l'inclure pour pouvoir s'en servir. Et ça tombe bien, recoder `malloc` n'est vraiment pas un exercice facile...
- Accélérer la compilation : notre code n'appelle que les librairies dont il a besoin, et rien de plus. Ainsi, il y a moins de choses à compiler, et on compile donc plus vite.
- Garder chacun des fichiers relativement courts : il est plus simple de trouver un bogue si on sait qu'il est dans tel fichier de 2000 lignes plutôt que dans tel fichier de 50000 lignes. Idem si l'on veut aller relire ou modifier une fonction particulière.
- Segmenter et organiser le travail : on programme une chose à la fois.
On peut également donner chacun des fichiers à une équipe de programmeurs/programmeuses différentes : chaque équipe se spécialise alors dans son fichier, avance efficacement dessus (pendant que les autres avancent efficacement sur le leur), et le tout est terminé plus rapidement.

En résumé, c'est très confortable !

1.0.1 Interface

Dans votre ordinateur, vous *n'avez pas* le fichier `stdlib.c`, c'est à dire le code source de `stdlib`. Vous avez à la place uniquement une version compilée de lui-ci nommée `stdlib.o`, ainsi que son interface `stdlib.h` (sounds familiar ?).

Définition 5 (Interface).

L'**interface** (en anglais : **A**pplication **P**rogramming **I**nterface, ou **API**) d'une librairie est un fichier qui décrit et abstrait son contenu. On y trouve les signatures des variables/fonctions/types déclarés la librairie et la spécification de ceux-ci ; ainsi aussi la liste des dépendances de la librairie (c'est à dire la liste des autres librairies qu'elle utilise).

Définition 6 (Organisation des fichiers d'une librairie).

En C, les interfaces sont appelées des *headers*. Si le fichier source s'appelle `truc.c`, l'interface s'appelle `truc.h` et la version compilée du code source `truc.o`.

En OCaml, `truc.ml` a pour interface `truc.mli` et pour version compilée `truc.o`.

Exemple. Voici un extrait de `dynArray.h`, une interface pour une librairie de tableaux dynamiques que nous coderons en TP :

```

6  #include <stdlib.h>
7  #include <stdbool.h>
8  #include <stdio.h>
9  #include <limits.h>
10 #include <assert.h>
11
12
13 /** Type des tableaux dynamiques.
14  * Il n'est pas nécessaire de connaître sa définition précise.*/
15 typedef struct dynArray_s dynArray;
16
17
18
19 /* Constructeurs */
20
21 /** Crée un tableau dynamique de longueur n,
22  * dont les n cases sont initialisées à la valeur x */
23 dynArray dyn_create(unsigned len, int x);
24
25 /** Libère le contenu de d */
26 void dyn_free(dynArray* d);
27
28 /** Crée un tableau dynamique qui contient une copie
29  * des len valeurs pointées par ptr */

```



Dans cet exemple, on peut voir que la librairie `dynArray` :

- utilise les librairies `stdlib`, `stdbool` et `stdio`.
- définit un type nommé `struct dynArray_s` (aussi nommé simplement `dynArray`).
- et définit les fonctions `dyn_create`, `dyn_free` et `dynarray_of_array` dont les prototypes sont indiqués. Il y a également une documentation de ces fonctions.

Pourquoi faire une interface ?

- Lorsque l'on veut utiliser une librairie, savoir *comment* la librairie fonctionne ne nous intéresse guère. Ce que l'on veut savoir, c'est quelles sont les fonctions que l'on peut appeler et ce qu'elles font : exactement ce que l'interface nous dit.

Par exemple, vous n'êtes jamais allés voir le code source de `printf` ... et heureusement.⁶

- Une interface permet de cacher certaines fonctions, en les omettant de l'interface. C'est très utile quand le code source utilise des fonctions « auxiliaires ».

Par exemple, le code source ma fonction `dyn_create` utilise une fonction `max`. Cependant, coder `max` n'est pas le but de cette librairie, et je ne veux pas alourdir l'interface en y mettant cette fonction : je ne la mets donc pas.

C'est d'autant plus utile lorsque la fonction auxiliaire est un peu incompréhensible, car elle réalise une tâche extrêmement spécifique que l'on ne comprend qu'avec le code source du tout sous les yeux.

- Une interface décrit uniquement la signature des fonctions : en particulier, on peut changer complètement la façon précise de les implémenter sans que cela n'impacte les codes clients. En d'autres termes, on peut faire une mise à jour de la librairie sans qu'il n'y ait à adapter les codes clients : c'est plutôt très bien.

6. Si vous êtes curieux-se, `printf` appelle `vprintf` qui appelle `vfprintf` qui fait tout le travail. Le code source de celle-ci est disponible ici : <https://github.com/lattera/glibc/blob/master/stdio-common/vfprintf.c> (la fonction commence à la ligne 1236 et fait environ 400 lignes, globalement très hors-programme).

Remarque. Certaines interfaces indiquent la complexité des fonctions, d'autres noms : c'est un débat. Certains informaticiens demandent à ce que les complexités apparaissent car c'est important pour évaluer la performance du code client ; tandis d'autres demandent à ne pas les indiquer car les mettre revient généralement à forcer une implémentation spécifique et donc à s'interdire de changer le fonctionnement « sous le capot » plus tard.⁷

1.0.2 Étapes de la compilation

Cette sous-sous-partie est volontairement très vague/floue. Vous en comprendrez plus les termes avec de l'expérience.

Définition 7 (Étapes de compilations).

Un compilateur effectue les étapes suivantes quand il compile :

- Pré-compilation : une première passe qui a pour but de préparer les fichiers. C'est notamment dans cette phase qu'en C les `sizeof(type)` sont remplacés par leur valeur, et (hors-programme) que les `#define` sont résolus.
- Analyse lexicale, syntaxique et sémantique du code : il s'agit de « lire » le code et de comprendre son « organisation interne ». Cf cours de MPI pour l'analyse lexicale et syntaxique (l'analyse sémantique est hors-programme).
- Génération de code intermédiaire puis de code objet : chacun des fichiers du code est, individuellement, transformé en une version compilée de ses fonctions. Cette version compilée est le fichier `.o`. Cependant, les appels de fonctions ne sont pas encore fonctionnels dans ces `.o` : lorsqu'une fonction A appelle une fonction B, l'appel ne fonctionne pas encore : il y a à la place quelque chose comme « TODO : ici insérer un appel à B ».
- Édition de liens : les appels de fonctions sont rendus fonctionnels, et le fichier est rendu exécutable.

C'est notamment l'étape où le compilateur doit « relier » différents codes sources entre eux pour la première fois.

FIGURE 5.6 – Résumé du rôle des différents fichiers

Remarque. (Vous n'avez pas à comprendre le détail de chaque étape pour comprendre ce qui suit :)

7. À titre personnel, je suis plutôt de celles et ceux qui veulent voir les complexités ; quitte à ce que ce soit avec la mention « cette complexité pourra changer à l'avenir ». Mon avis personnel n'a cependant que très peu de valeur.

- les bibliothèques sont généralement stockées dans votre ordinateur avec uniquement le fichier objet `.o` et l'interface `.h` (ou `.mli`). Ainsi, quand on veut utiliser une bibliothèque, le compilateur n'a pas besoin de refaire toutes les premières étapes sur la bibliothèque ! Il n'aura à faire que l'édition de liens, et la compilation est donc grandement accélérée.
- (Hors-programme) Vous avez peut-être déjà entendu parler des fichiers *shared object* `.so` sous Linux, ou des `.dll` sous Windows. Ce sont une sorte particulière de fichiers objets, c'est à dire une version compilée d'une bibliothèque. C'est en réalité sous cette forme que sont distribuées les bibliothèques, plutôt que comme des simples `.o`.

1.1 Aspects théoriques

Définition 8 (Structure de données).

Une **structure de données** est une façon d'organiser des données et de les manipuler efficacement.

Exemple. En décrivant les tableaux dynamiques en section 1, nous avons décrit une structure de donnée.

Définition 9 (Type abstrait, signature).

Un **type abstrait**, aussi appelé **structure de donnée abstraite**, est une considération abstraite d'une structure de donnée. Elle est décrite par sa **signature**, c'est à dire par la donnée de :

- l'identifiant (le nom) du type abstrait.
- les autres types abstraits dont il dépend.
- les identifiants de constantes du type.
- la signature des opérations que l'on peut effectuer sur le type.

Exemple. Voici deux signatures :

(a) Interface du type abstrait Bool

(b) Interface du type abstrait Tableau Dynamique

FIGURE 5.7 – Deux exemples d'interface

Définition 10 (Signature vs documentation (syntaxe vs sémantique)).

La signature explique quelles sont les « choses » que l'on peut écrire à l'aide du type (quelles fonctions on peut appliquer et dans quels cas). Mais elle n'explique pas ce que *font* ces « choses » ! En termes scientifiques, on dit qu'une signature décrit uniquement la **syntaxe**, c'est à dire les règles d'écritures.

Pour savoir ce que font ces « choses », il faut donner de la **sémantique**, c'est à dire **documenter** les constantes et les opérations du type.

Exemple. Voici des exemples de documentation pour les 3 opérations des booléens :

- **non** : renvoie la négation d'un booléen, c'est à dire que $\text{Non}(\text{vrai}) = \text{faux}$ et réciproquement.
- **ou** : renvoie vrai si et seulement si au moins l'un de ses deux arguments vaut vrai .
- **et** : renvoie vrai si et seulement si ses deux arguments valent vrai .

Remarque.

- La documentation peut aussi prendre une forme très mathématique. C'est notamment utile lorsque l'on veut automatiser les preuves de programme. Voici par exemple un axiome vérifié par Longueur :
 $\forall n : \text{Entier}, \forall x : \text{Élément}, \text{longueur}(\text{creer}(n, x)) = n$.
 Dans la plupart des cas, sauf ambiguïté, on préfère donner une documentation en français plutôt qu'en mathématique pour des raisons évidentes de lisibilité.
- D'un point de vue pratique, une erreur de *syntaxe* est quelque chose qui est détecté à la compilation (par exemple une erreur de type), tandis qu'une erreur de *sémantique* crée un bogue dont on ne se rend compte qu'en testant le code.
- En langage naturel (en français si vous êtes francophone), une erreur de syntaxe correspond à une erreur d'orthographe/grammaire, tandis qu'une erreur de sémantique correspond à une phrase qui ne veut rien dire.

Définition 11 (Constructeur, accesseur, transformateur).

n classe les différentes opérations d'un type abstrait en :

- **constructeur** : ce sont les opérations qui créent ou détruisent une de données.
- **accesseur** : ce sont les opérations qui renvoient une information sur la structure de données (sans la modifier).
- **transformateur** : ce sont les opérations qui modifient une structure de donnée existante.

Exemple. Dans le type abstrait `Tableau dynamique`, `Créer` est un constructeur, `Longueur` un accesseur et `Ajout` un transformateur.

Remarque. Les signatures des transformateurs peuvent être écrites de deux façons : soit d'une façon impérative (ils ne renvoient pas de structure de donnée, mais modifient la structure de donnée passée en argument), soit de façon fonctionnelle (ils ne modifient pas leur argument mais renvoient le nouvel état de la structure de données).

Exemple.

- « `Ajout : Tableau Dynamique × Élément → Rien` » est une signature impérative (et la documentation doit préciser que `Ajout` modifie celui passé en argument).
- « `Ajout : Tableau Dynamique × Élément → Tableau Dynamique` » est une signature fonctionnelle (et la documentation doit préciser l'absence d'effets secondaires).

2 Structures de données séquentielles

Une structure de donnée est dite **séquentielle** lorsqu'elle range les éléments les uns après les autres, dans un certain ordre.

Exemple. Les tableaux C ou les listes OCaml sont des structures de données séquentielles.

L'objectif de cette section est de présenter différents types abstraits de structures de données séquentielles, et une ou plusieurs implémentations.

2.0 Types de base

On a déjà vu comment sont implémentés les entiers, flottants, booléens : dans ce cours, je les considérerai comme des types de base.

On a également déjà vu certains types construits à l'aide d'autres types : par exemple les pointeurs (on utilise des pointeurs vers un élément d'un autre type) ou les tableaux (des tableaux d'un autre type).

Remarque. Notez qu'il existe en fait de nombreux types de tableaux : les tableaux d'entiers, les tableaux de booléens, les tableaux de etc... On dit que le type tableau est **paramétré** par le type de son contenu.

2.1 Listes linéaire

Avant-propos : le terme « liste » est victime d'une forte polyésmie⁸. Il veut dire des choses différentes ici et là. Dans ce cours, par « liste » je désigne spécifiquement une liste linéaire :

Définition 12 (Liste linéaire).

Une **liste linéaire** est une suite finie, éventuellement vide, d'éléments repérés selon leur rang :

$\ell = [\ell_0; \dots; \ell_{n-1}]$.

Si E est l'ensemble des éléments, l'ensemble \mathbb{L} est défini récursivement par :

$$\mathbb{L} = \emptyset + E \times \mathbb{L}$$

Définition 13 (Signature de Liste linéaire).

La signature (fonctionnelle) du type abstrait Liste Linéaire est :

- Type : Liste Linéaire (abrégé List ci-dessous)
- Utilis : Entier, Élément
- Constantes : `[]` est une liste linéaire
- Opérations :
 - `longueur` : `List` → Entier
 - `acces_ieme` : `List` × Entier → Élément
 - `supprime_ieme` : `List` × Entier → List
 - `insere_ieme` : `List` × Entier × Élément → List

Les trois dernières fonctions, respectivement : renvoient l'élément d'indice pris en argument, insère un élément de sorte à ce qu'il soit à l'indice pris en argument, ou supprime l'élément d'indice pris en argument.

2.1.0 Implémentation par tableaux de longueur fixe

Pour cette implémentation, on fixe un majorant `len_max` de la longueur de toute liste linéaire.

8. Notamment à cause de Python, qui utilise le terme pour désigner ses tableaux dynamiques... et encore, le terme était déjà polyésmique avant Python.

On représente alors une liste linéaire par un tableau à `len_max` cases, dont on n'utilise que les premières pour stocker les éléments les uns après les autres. Il faut également stocker `len` la longueur réelle de la liste. Autrement dit, on fait comme des tableaux dynamiques, mais sans redimensionner :

FIGURE 5.8 – Liste linéaire dans un tableau de longueur fixe

Proposition 14 (Complexités des listes linéaires par tableaux).

ne telle implémentation permet d'obtenir les complexités suivantes :

- `[]` : se calcule en $O(1)$ s'il ne faut pas initialiser les cases du tableau, et en $O(\text{len_max})$ sinon.
- `longueur` : $O(1)$
- `acces_ieme` : $O(1)$
- `insere_ieme` : $O(\text{len} - i)$ avec i l'indice auquel on insère.
- `supprime_ieme` : $O(\text{len} - i)$ avec i l'indice auquel on supprime.

Démonstration. Les seules complexités non triviales sont les deux dernières. Je présente ici uniquement la première, l'autre étant similaire.

Pour insérer en position i , on « décale » d'une case tous les éléments d'indice $> i$: il y a $\text{len} - i$ éléments à décaler, chaque décalage se fait en temps constant (1 écriture dans une case du tableau + 1 calcul d'indice), donc tout ce décalage est en $O(\text{len} - i)$. Il ne reste qu'à écrire l'élément dans la case voulue, ce qui se fait en temps constant. \square

2.1.1 Implémentation par tableaux dynamiques

En utilisant les idées des tableaux dynamiques, on peut utiliser des tableaux "redimensionnables" pour améliorer l'implémentation précédente. Cela permet de manipuler des listes linéaires dont la valeur finit par dépasser le `len_max` initial.

Comme on l'a vu dans le chapitre sur les tableaux dynamique, en amorti les complexités restent les mêmes.

2.1.2 Implémentation par listes simplement chaînées

Définition 15 (Liste simplement chaînée).

Une **liste simplement chaînée** est une structure de données composée de **cellules** (aussi appelées **maillons**, ou parfois **noeuds**) reliées entre elles. Chaque cellule stocke deux informations :

- L'élément de la liste stockée dans cette cellule.
- Un pointeur vers la cellule de l'élément suivant.

FIGURE 5.9 – Organisation d'une liste simplement chaînée

Exemple.

FIGURE 5.10 – Une liste chaînée particulière

Remarque.

- Sur mes schémas, je ne représente pas les cellules comme étant « proprement les unes après les autres ». C'est pour vous rappeler qu'en pratique, les cellules sont allouées sur le tas mémoire, et que l'on a aucune garantie de position dans le tas mémoire.
- On sait que l'on a atteint la fin d'une liste lorsque le pointeur vers la prochaine cellule a une valeur particulière. En C, c'est généralement NULL (le pointeur qui ne pointe sur rien).
- On type souvent une liste simplement chaînée comme étant un pointeur vers une cellule. Ainsi, pour donner une liste simplement chaînée, il suffit de donner un pointeur vers la cellule de tête. Pour donner la liste vide, on donne le pointeur NULL (ou autre valeur particulière fixée).
- C'est ainsi que sont implémentées les listes en OCaml !! Cela correspond d'ailleurs à ce que l'on a vu en TP, où l'on a dit qu'une liste correspond à l'un de ces deux motifs :

```
1 | []      (* liste vide *)
2 | t :: q  (* valeur de tête suivi d'un accès à la queue *)
```



En particulier, en OCaml, passer une `'a list` en argument revient exactement à passer un pointeur, c'est à dire que cela se fait en temps constant.

Proposition 16 (Complexité des opérations sur une liste linéaire).

Cette implémentation permet d'obtenir les complexités suivantes pour une liste linéaire à len éléments :

- `[]` : $O(1)$
- `longueur` : $O(\text{len})$
- `acces_ieme` : $O(i)$
- `insere_ieme` : $O(i)$ avec i l'indice auquel on insère.
- `supprime_ieme` : $O(i)$ avec i l'indice auquel on supprime.

Démonstration. Respectivement :

- il suffit de renvoyer le pointeur NULL
- il faut parcourir toutes les cellules, en mémorisant le nombre de cellules. Chaque cellule se traite en temps constant (incrémenter le nombre de cellule vues puis aller à la suivante).
- idem, mais on ne les parcourt que jusqu'à atteindre la cellule d'indice i (où l'on renvoie alors la valeur stockée, en temps constant).
- Similaire au précédent, à ceci près que :
 - 0) On accède à la cellule d'indice i .
 - 1) On crée une nouvelle cellule, que l'on fait pointer vers la queue de la cellule d'indice i .
 - 2) on modifie le chainage de la cellule d'avant. Si l'on ne veut pas modifier la liste prise en argument, on doit recréer tout le début de la liste.

FIGURE 5.11 – Insertion dans une liste chaînée, sans modifier l'originale

- De même que le précédent.

□

Remarque. Un gros avantage des listes simplement chaînées est qu'elles réduisent la duplication en mémoire lorsque l'on « étend » de différentes façons des listes. Par exemple :

```

1 let lst = [4; 5; 6; 7; 8; 9]
2 let adora = 3 :: lst
3 let catra = 10 :: 20 :: lst
4 let glimmer = 55 :: lst

```



correspond en mémoire à :

FIGURE 5.12 – Illustration de la non-duplication des queues des listes OCaml

Cette propriété est notamment utile lorsque l'on fait un algorithme qui essaye différentes façons d'étendre une liste ; par exemple les algorithmes de retour sur trace (ou *backtracking* en anglais, que nous verrons au S2).

Remarque. (Petit complément au programme) De même, la fonction `List.append` qui concatène deux listes (c'est à dire les « colle ») est en temps et espace linéaire en la longueur de la première liste. En particulier, si on peut éviter d'y faire appel plusieurs fois, c'est (bien) mieux.

Pro-tip (au programme aussi) : `List.append l0 l1` se note aussi `l0 @ l1` .

Exemple.

```
1 let l0 = [1; 2]
2 let l1 = [2; 3; 4; 5]
3 let l2 = l0 @ l1
```



correspond en mémoire à :

FIGURE 5.13 – Illustration de la concaténation de listes OCaml

2.1.3 Variantes des listes chaînées

Voici quelques variantes qui existent (il en existe bien d'autres) :

- On peut mémoriser une « meta-donnée » en plus de la liste linéaire, typiquement un pointeur vers le début et la fin :

- La liste peut-être doublement chaînée, c'est à dire que chaque cellule retient la cellule d'avant et celle d'après.
- Le chaînage peut-être circulaire, c'est à dire que la dernière cellule pointe sur la première.
- Et bien d'autres que vous croiserez peut-être.⁹

2.2 Piles

Une pile est un cas particulier de liste linéaire. On abandonne l'idée d'insérer ou de supprimer n'importe où, et même d'accéder n'importe où. En échange, on demande une *garantie* sur l'ordre des éléments :

Définition 17 (Pile).

Une **pile** (anglais : **stack**) est une liste linéaire qui garantit que le dernier élément inséré sera le premier élément supprimé. En particulier, on ne peut pas insérer ou supprimer n'importe où. On dit qu'une pile vérifie le principe **LIFO** : Last In First Out (dernier entré, premier sorti).

9. Techniquement, on peut voir les arbres (S2) comme le cas où il peut y avoir plusieurs cellules suivantes... je préfère voir les listes chaînées comme un cas particulier des arbres.

Exemple. Une pile de linge à trier, ou une pile de copies à corriger, ou n'importe quel autre empilement de la vie réelle.

Exemple.

FIGURE 5.14 – Empilages et dépilages successifs sur une pile

Remarque. La pile mémoire est une pile.

Définition 18 (Signature des piles).

Voici la signature (impérative) des piles, un peu documentée :

- Type : Pile
- Utilise : Élément, Booléen
- Constantes : `pile_vide`
- Opérations :

<ul style="list-style-type: none"> – <code>empile</code> : Pile \times Élément \rightarrow rien – <code>depile</code> : Pile \rightarrow Élément – <code>sommet</code> : Pile \rightarrow Élément – <code>est_vide</code> : Pile \rightarrow Bool 	<p><i>Cette fonction ajoute un élément sur la pile (et la modifie donc).</i></p> <p><i>Cette fonction retire le dernier élément de la Pile (et la modifie donc) et le renvoie.</i></p> <p><i>Renvoie l'élément au « sommet » de la Pile, c'est à dire le prochain élément à sortir de la pile.</i></p>
--	--

Remarque.

- On aurait aussi pu donner une signature fonctionnelle.
- Cette propriété **LIFO** rend les piles indispensables dans la résolution de nombreux problèmes (à noter que la récursion peut fréquemment remplacer une pile).

2.2.0 Implémentations par tableaux

L'idée est de faire un tableau dynamique, où l'on empile et depile à la fin : `empiler` correspond alors à `AJOUT`, et `depiler` à `RETRAIT`.

FIGURE 5.15 – Succession d'opération sur des piles implémentées par tableaux dynamiques

Proposition 19 (Complexité des piles par tableaux dynamiques).

Cette implémentation permet d'obtenir les complexités suivantes pour une pile éléments :

- `pile_vide` : $O(1)$
- `empile` : $O(1)$ amorti
- `depile` : $O(1)$ amorti
- `sommet` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Cf la section sur les tableaux dynamiques. Pour `sommet`, il suffit de renvoyer l'élément d'indice `len-1`, et pour `est_vide` il suffit de tester si `len = 0`. \square

2.2.1 Implémentation par listes simplement chaînées

L'idée est de faire une liste simplement chaînée, où l'on empile et dépile en tête : `empiler` correspond alors à une insertion en indice 0, et `depiler` à une suppression en indice 0 (plus le fait de renvoyer l'élément).

FIGURE 5.16 – Succession d'opération sur des piles implémentées par listes simplement chaînées

Remarque. C'est cette implémentation qui est utilisée par le module `Stack` de OCaml, où le type `'a stack` est défini comme étant une `'a list ref` (c'est à dire un pointeur vers une `'a list`).

Proposition 20 (Complexité des piles par listes simplement chaînées).

Cette implémentation permet d'obtenir les complexités suivantes pour une pile :

- `pile_vide` : $O(1)$
- `empile` : $O(1)$
- `depile` : $O(1)$
- `sommet` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Cf la sous-sous-section sur les listes simplement chaînées. Pour `empiler`, `depiler` et `sommet`, ces opérations travaillent sur la cellule de tête et donc se font bien en temps $O(1)$. \square

2.3 Files

Comme les piles, les Files sont des listes linéaires particulières où l'on restreint insère, supprime et accés pour obtenir une garantie d'ordre :

Définition 21 (File).

Une **file** (anglais : **queue**) est une liste linéaire qui garantie que le premier élément inséré sera le premier élément supprimé. En particulier, on ne peut pas insérer ou supprimer n'importe où. On dit qu'une file vérifie le principe **FIFO** : **F**irst **I**n **F**irst **O**ut (premier entré, premier sorti).

Exemple. Une file d'attente dans une boulangerie^{10 11}.

Exemple.

FIGURE 5.17 – Enfilages et défilages successifs dans une file

Définition 22 (Signature des files).

Voici la signature (impérative) des files, un peu documentée :

- Type : File
- Utilise : Élément, Booléen
- Constantes : file_vide
- Opérations :

– enfile : File × Élément → rien	<i>Cette fonction fait entrer un élément dans la file (et la modifie donc).</i>
– defile : File → Élément	<i>Cette fonction fait sortir le premier élément de la File (et la modifie donc) et le renvoie.</i>
– prochain : File → Élément	<i>Renvoie le prochain élément à sortir de la file.</i>
– est_vide : File → Bool	

Remarque.

- Là encore, on aurait aussi pu donner une signature fonctionnelle.
- Cette propriété **FIFO** rend les piles indispensables dans la résolution de nombreux problèmes.

2.3.0 Implémentations par tableaux circulaires

On borne le nombre d'éléments de la file par une certaine constante len_max.

Pour utiliser des tableaux, la première idée est de faire presque comme avec des piles par tableaux dynamiques : pour enfiler on ajoute à droite, et pour défiler on retire à gauche. Cette méthode fonctionne, mais atteint vite un problème :

¹⁰. Boulangerie de Lamport =D

¹¹. Vous comprendrez la blague de la note de bas-de-page précédente en MPI. Je vous assure qu'elle est drôle.

FIGURE 5.18 – Problème de l'implémentation naïve des files dans un tableau

Pour pouvoir utiliser l'espace disponible, l'astuce est de rendre le tableau **circulaire** :

Définition 23 (Implémentation des files par tableaux circulaires).

Pour représenter une file dans un tableau circulaire à len_max cases, on mémorise :

- le tableau `arr` et sa longueur `len_max`
- `entree`, un indice du tableau qui stocke l'entrée de la file (c'est là qu'auront lieu les enfilages).
- `sortie`, un indice du tableau qui stocke la sortie de la file (c'est là qu'auront lieu les défilages).

Les éléments de la file sont alors stockés à partir de l'indice `debut` et jusqu'à l'indice `fin`, en prenant en compte la circularité (çad en calculant le prochain indice modulo len_max) :

FIGURE 5.19 – Organisation d'un tel tableau circulaire

FIGURE 5.20 – Un autre exemple où l'on voit l'aspect circulaire

Remarque. Les indices `entrée` et `sortie` sont parfois des indices de la file *inclus*, c'est à dire qu'on y trouve bien des éléments de la file, et parfois *exclus* (c'est à dire que le premier/dernier élément se trouve juste avant/après). Il faut bien lire la description de l'énoncé pour savoir !

Exemple.

FIGURE 5.21 – Succession d'enfilage et de défilage dans une file par tableaux circulaires

Remarque. Distinguer la file vide d'autres files n'est pas évident, et la façon de le faire dépend de l'implémentation. Il faut bien lire la description de l'implémentation pour savoir !

(a) File vide ou à 1 élément ?

(b) File vide ou pleine ?

FIGURE 5.22 – File vide ou non ?

Proposition 24 (Complexité des files par tableaux circulaires).

Cette implémentation permet d'obtenir les complexités suivantes pour une file :

- `file_vide` : $O(1)$ ou $O(\text{len_max})$ s'il faut initialiser le contenu des cases
- `enfile` : $O(1)$
- `defile` : $O(1)$
- `prochain` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Respectivement :

- Il suffit de créer le tableau.
- Il suffit d'écrire dans la bonne case, qui est connue grâce à `entrée`, puis de mettre à jour `entree`.
- Idem mais pour la sortie.
- Il suffit de lire la bonne case grâce à l'indice `sortie`.
- La longueur d'une file soit se déduit de l'écart entre `entrée` et `sortie`, soit est stockée (en plus de stocker les deux indices) : dans tous les cas, c'est en temps constant.

□

Proposition 25 (Longueur d'une file par tableau circulaire).

Le nombre d'éléments d'une file implémentée par tableaux circulaires peut se calculer à l'aide de l'écart entre les indices.

La formule est généralement de la forme $(\text{sortie} - \text{entree}) \bmod \text{len_max}$, avec parfois un ± 1 à rajouter au tout selon l'implémentation précise utilisée.

Exemple. Dans les exemples ci-dessous, *sortie* est l'indice (inclus) du prochain élément à sortir, et *entrée* l'indice (inclus) de l'entrée de la file (c'est à dire du dernier élément inséré).

(a) Une première file à 4 éléments

(b) Une seconde file à 4 éléments (où la circularité se voit)

FIGURE 5.23 – Exemples de calculs de longueur de file dans des tableaux circulaires

2.3.1 (Hors-programme) Amélioration avec des tableaux dynamiques

Pour échapper au fait que les files dans des tableaux circulaires sont de longueur au plus len_max , on peut les implémenter dans des tableaux dynamiques. Il faut alors faire attention lors du doublement du tableau à correctement mettre à jour *debut* et *fin*, notamment lorsque $\text{fin} < \text{debut}$.

2.3.2 Implémentations par listes simplement chaînées

Une autre implémentation possible utilise des listes simplement chaînée dont on mémorise un pointeur vers le début et la fin. Ici, on utilisera une signature impérative : on modifie la liste simplement chaînée passée en argument¹²

Exemple.

FIGURE 5.24 – Enfilages et défilages successifs dans une file par listes simplement chaînée

12. Cela sera donc différent des exemples de listes OCaml précédents, donc.

Définition 26 (Implémentation des files par listes simplement chaînée).

Pour implémenter des files à l'aide de listes simplement chaînées, on utilise :

- une liste simplement chaînée. Le prochain élément à sortir est en tête, le dernier élément inséré dans la cellule de fin de liste.
- un pointeur sortie vers la tête de la liste. Il permet de défiler.
- un pointeur entree vers la dernière cellule de fin. Il permet d'enfiler.

Pour défiler, il suffit alors de lire l'élément du maillon de tête, et de modifier le pointeur sortie pour qu'il pointe sur la queue.

Pour enfiler, on crée un nouveau maillon de fin contenant l'élément à enfiler, puis on modifie le chaînage de l'ancien maillon de fin (accessible via entree) pour le faire pointer sur le nouveau. On met ensuite à jour entree en conséquence.

Proposition 27 (Complexités des files par listes simplement chaînées).

ne telle implémentation permet d'obtenir les complexités suivantes :

- `file_vide` : $O(1)$
- `enfile` : $O(1)$
- `defile` : $O(1)$
- `prochain` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Toutes ces complexités devraient être évidentes à l'aide des descriptions des listes simplement chaînées et du fonctionnement de `enfile` et `defile`.

Précisons que pour `est_vide`, il suffit de tester si `sortie` pointe sur un maillon ou sur rien. \square

2.3.3 Implémentation par listes doublement chaînées

On peut faire l'implémentation précédente avec des listes doublement chaînées (et toujours un pointeur vers chacune de deux extrémités). Cela permet de parcourir la file dans les deux sens, et non simplement de la sortie vers l'entrée.

Remarque. C'est une version très connue, au point où plusieurs livres sur le sujet ne mentionnent pas la version simplement chaînée. Elle est pourtant très pratique ; parcourir la file dans le sens entrée->sortie ne sert pas si souvent que ça¹³.

2.3.4 Par double pile

On peut implémenter une file à l'aide de deux piles. L'idée est d'avoir une pile « d'entrée » où l'on empile les éléments enfilés, une pile de sortie où l'on dépile les éléments défilés, et lorsque cette dernière est vide on renverse la pile d'entrée sur la pile de sortie.

FIGURE 5.25 – File par double pile

13. Ne me faites pas dire ce que je n'ai pas dit : ça sert quand même. Mais à mon humble avis, pas assez souvent pour que cela justifie de ne pas présenter la version simplement chaînée.

Les opérations de la signature des files sont alors en temps constant (amorti). Plus de détails en TD.

Remarque. On peut aussi implémenter une pile à l'aide de deux files, mais c'est plus compliqué.

2.4 Variantes des files

Plusieurs variantes des files existent :

- Files à double entrée : on peut enfiler ou défiler sur les deux extrémités de la file. En anglais, on parle de **double ended queue**, souvent abrégé **dequeue**.
Pour les implémenter, on peut utiliser des tableaux circulaires ou des listes doublement chaînées.
- File de priorité : les éléments ne sortent pas de la file selon leur ordre d'entrée, mais selon une priorité qui leur est assignée. C'est typiquement ce que fait l'ordonnanceur de tâches de votre ordinateur (certains processus sont plus prioritaires à « faire avancer » que d'autres), un site internet (on priorise certaines requêtes sur d'autres¹⁴), ou un hôpital (on priorise les patient-es dont l'état est le plus critique). Nous verrons au S2 comment les implémenter.

3 Aperçu des structures de données S2

Au semestre 2, nous verrons d'autres de structure de données qui permettent de résoudre d'autres problèmes :

- Une nouvelle structure séquentielle, les dictionnaires.
- Les structures hiérarchiques (où les éléments ne sont pas à la suite mais au-dessus/dessous d'autres éléments) : les arbres (sous plusieurs formes), dont notamment les tas qui permettent de faire des files de priorité.
- Les structures relationnelles (où on stocke des relations entre éléments) : les graphes, qui permettent de faire à peu près l'entièreté de l'informatique^{15 16}

Nous verrons également et surtout plus de méthodes algorithmiques, c'est à dire de chapitres de « conception de solution pour résoudre un problème » (comme on a cherché des solutions pour Hanoï et autres problèmes dans le chapitre sur la récursion). Ces chapitres s'appuieront souvent sur des structures de données pour résoudre efficacement les problèmes.

14. De manière générale, l'organisation des réseaux informatiques ou téléphoniques sont basées sur des files d'attente et des files de priorité extrêmement optimisées; ainsi que sur des modélisations et études probabilistes poussées.

15. J'exagère un peu.

16. Mais pas tant que ça.

Chapitre 6

BONNES PRATIQUES DE PROGRAMMATION

Notions	Commentaires
Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante.	Ces annotations se font à l'aide de commentaires.
Programmation défensive. Assertion. Sortie du programme ou exception levée en cas d'évaluation négative d'une assertion.	L'utilisation d'assertions est encouragée par exemple pour valider des entrées ou pour le contrôle de débordements. Plus généralement, les étudiants sont sensibilisés à réfléchir aux causes possibles (internes ou externes à leur programme) d'opérer sur des données invalides et à adopter un style de programmation défensif. [...]
Explicitation et justification des choix de conception ou programmation.	Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent en évitant la paraphrase.

Extrait de la section 1.2 du programme officiel de MP2I : « Discipline de programmation ».

Notions	Commentaires
Jeu de tests associé à un programme.	Il n'est pas attendu de connaissances sur la génération automatique de jeux de tests ; un étudiant est capable d'écrire un jeu de tests à la main, donnant à la fois des entrées et les sorties correspondantes attendues. On sensibilise, par des exemples, à la notion de partitionnement des domaines d'entrée et au test des limites.
[...] Chemins faisables. Couverture des sommets, des arcs ou des chemins (avec ou sans cycle) du graphe de flot de contrôle.	Les étudiants sont capables d'écrire un jeu de tests satisfaisant un critère de couverture des instructions (sommets) ou des branches (arcs) sur les chemins faisables.
Test exhaustif de la condition d'une boucle ou d'une conditionnelle.	Il s'agit, lorsque la condition booléenne comporte des conjonctions ou disjonctions, de ne pas se contenter de la traiter comme étant globalement vraie ou fausse mais de formuler des tests qui réalisent toutes les possibilités de la satisfaire. On se limite à des exemples simples pour lesquels les cas possibles se décèlent dès la lecture du programme.

Extrait de la section 1.3 du programme officiel de MP2I : « Validation, test ».

Ce cours sert à deux choses :

- Donner des compléments sur les langages de programmation au programme ; compléments qui peuvent aider à écrire du code plus simple à (re)lire.

- Présenter des bonnes pratiques de programmation. On reste sur des conseils très généraux ; beaucoup de règles spécifiques dépendront de l'équipe dans laquelle vous travaillerez.

SOMMAIRE

0. Compléments sur les types de base	108
0. Évaluation paresseuse des opérateurs booléens	108
1. Détails sur le passage / renvoi des arguments	109
2. En C : instruction pré-processeur #define	109
<i>Header guards (p. 110). (Mi-HP) Utilisation pour créer des tableaux statiques (p. 110).</i>	
1. Types avancés	111
0. En OCaml : paires, triplets, etc	111
1. Types enregistrements	111
<i>En C (p. 111). En OCaml (p. 112).</i>	
2. Types énumérations	112
<i>(HP) En C (p. 112). En OCaml (p. 113).</i>	
3. OCaml : types construits	113
<i>Notion de type construit (p. 113). let destructurant (p. 113). Le type 'a option (p. 114).</i>	
4. Types récursifs	114
2. Les exceptions	115
0. (HP) En C	115
1. En OCaml	115
<i>Rattraper une exception (p. 115). Lever une exception (p. 116). Définir une exception (p. 116).</i>	
3. Écrire mieux	117
0. Nommer	117
1. Structurer	118
2. Documenter	118
3. Anticiper	119
4. Simplifier	119
5. Progresser	119
4. Tester mieux	120
5. Débuguer mieux	120

0 Compléments sur les types de base

0.0 Évaluation paresseuse des opérateurs booléens

Définition 1 (Évaluation paresseuse des opérateurs logiques).

En C et en OCaml (et Python), les opérateurs `||` et `&&` évaluent d'abord leur opérande de gauche. Selon sa valeur, ils peuvent ne même pas évaluer celui de droite :

- dans `a || b`, le terme de gauche (`a`) est évalué en premier. S'il vaut `true`, on ne calcule pas celui de droite car `true` est absorbant pour `||`.
- dans `a && b`, le terme de gauche (`a`) est évalué en premier. S'il vaut `false`, on ne calcule pas celui de droite car `false` est absorbant pour `&&`.

Exemple.

```

3 let rec mem x lst =
4   match lst with
5   | t :: q -> t = x || (mem x q)
6   | [] -> false

```

 `démofl`

Dans la fonction ci-contre, le `||` de la ligne 5 est paresseux. Donc si `t = x`, on évalue pas le terme de droite du `||`. Ainsi, cela permet que dès que l'on a trouvé `x` dans la liste, on arrête le parcours de la liste.

Remarque.

- L'évaluation paresseuse permet d'économiser l'écriture d'un saut conditionnel de la forme « si terme de gauche vaut X, ne pas évaluer le terme de droite ». On peut bien sûr écrire le saut conditionnel, mais ce qui peut alléger le code est bienvenu.
- L'évaluation paresseuse consiste à tester **d'abord le terme de gauche puis ensuite (et seulement si nécessaire) le terme de droite**. Si vous voulez que le terme de droite soit évalué en premier... il faut en faire le terme de gauche.
Le compilateur ne peut pas prédire quel terme sera le plus probablement absorbant, il se contente d'appliquer une règle simple.
- Il n'est pas garanti d'avoir une évaluation paresseuse sur les multiplications d'entiers.

0.1 Détails sur le passage / renvoi des arguments

Lorsqu'on réalise un appel de fonction, l'appel a lieu par valeur. C'est à dire que l'on calcule une valeur pour chacun des arguments, et c'est ensuite cette valeur qui est recopiée dans le bloc d'appel de la fonction.

Remarque.

- ⚠ La « valeur » d'un tableau C est son adresse. Autrement dit, si une fonction modifie le contenu du tableau, elle modifie le contenu de l'original.
- En C, pour passer un struct on recopie la valeur de ses arguments.
- En OCaml, tout ce qui n'est pas un booléen ou un entier est en fait un pointeur vers son contenu (que l'on n'a pas le droit de modifier).
- Tout cela est également valable pour le *renvoi* d'arguments, ou plus généralement pour tout moment où l'on a besoin de stocker/communiquer la valeur de quelque chose.

Pourquoi c'est important? Parce que cela signifie que le calcul de la « valeur » des entrées est fait une seule fois, avant le début de l'appel.

Exemple. Si l'on exécute le code ci-dessous, l'entier 2 sera affiché une seule fois : en effet, la valeur de `foo 1` est calculée *avant* l'appel à `bar`.

```

9  (** Affiche 2*x et le renvoie *)
10 let foo x =
11   let _ = print_int (2*x) in
12   let _ = print_char '\n' in
13   2*x
14
15 (** Renvoie 3*x *)
16 let bar x =
17   (* on appelle x trois fois *)
18   x + x + x
19
20 let y = bar (foo 1)
```



Mais dans quel ordre sont évalués les arguments d'une fonction à plusieurs arguments? Il n'y a pas de garantie. Le premier argument peut-être évalué avec le second, ou l'inverse.

0.2 En C : instruction pré-processeur #define

Cette sous-section concerne *uniquement* le langage C.

En C, `#define` permet de définir une macro. Une macro est quelque chose qui est cherché-remplacé par le précompilateur : lorsque l'on écrit `#define MACRO valeur`, le pré-compilateur va remplacer toutes les occurrences de `MACRO` par `valeur`.

Dans le cadre du programme de MP2I/MPI, vous ne devez pas savoir les utiliser ; mais vous devez savoir reconnaître et comprendre leurs utilisations.

Remarque. `#define` appartient à une grande famille nommée les **instructions pré-processeur**. De manière générale, en C, toute instruction (toute "ligne") qui commence par `#` est une instruction pré-processeur. Ainsi, `#include` en est une.

0.2.0 Header guards

Si l'on inclut plusieurs fois le même fichier de manière naïve, cela mène à des erreurs : si j'inclus deux fois un fichier, je définis ses fonctions deux fois... or il est interdit de redéfinir un identifiant global (comme un nom de fonction par exemple).

La solution consiste à utiliser des **header guards**, qui garantissent que l'inclusion a lieu une seule fois. Plus précisément, ils garantissent que les inclusions suivantes n'ajoutent rien à la première. On dit qu'on *rend l'inclusion idempotente*.

En pratique, cela demande d'écrire les interfaces ainsi :

```
1  /* Un exemple d'interface
2
3  #ifndef _H_EXEMPLE_H_
4  #define _H_EXEMPLE_H_ 0
5
6  Le contenu du header ici :
7  ...
8  ...
9  ...
10 ...
11 ...
12
13 #endif
```

Il faut comprendre ce header comme ceci : « si la macro `_EXEMPLE_H` n'est pas encore définie : la définir, faire le contenu du header, fin du si ». Ainsi, la première fois que le fichier est inclus durant la compilation, la macro n'est pas encore définie et on rentre dans le Si. Les fois suivantes, la macro est déjà définie, on n'entre pas dans le Si... et on saute donc l'inclusion de la librairie (puisqu'elle est déjà incluse dans le header, est caché derrière de `#ifndef`).

0.2.1 (Mi-HP) Utilisation pour créer des tableaux statiques

Quand on crée un tableau en C avec `type T[...]`, la longueur `...` ne **doit pas** impliquer des identifiants. On ne peut donc pas écrire `int T[len]`, mais on peut écrire `int T[100 + 1]`.

Remarque. Pourquoi cette règle ? Parce que de tels tableaux sont alloués sur la pile, et qu'il est *très* utile de pouvoir prédire la taille de chacun des blocs d'activation de la pile. Il faut donc que la taille des blocs soit connue à la compilation - on dit qu'elle doit être **statique** - ce qui interdit les tableaux dont la longueur dépend d'une variable/fonction.

Toutefois, on a parfois envie d'avoir plusieurs tableaux qui ont la même longueur et de pouvoir changer dans le code source la longueur de tous ces tableaux d'un coup. De plus, donner un nom à la longueur est toujours utile : un nom de variable est plus explicite que 10, 35 200. On peut utiliser un `#define` pour cela :

```
1  #define NB_ELEM 35
2
3  type tab[NB_ELEM]; // tableau statique à NB_ELEM
```

Remarque.

- On ne peut pas utiliser de `const` à la place de la macro ici. On pourrait par contre utiliser `constexpr`, présent en C depuis la version C23.
- Cet usage des macros n'est pas mentionné dans les éléments à savoir reconnaître dans programme officiel. Pourtant, je l'ai déjà vu en concours : considérez donc qu'il faut savoir le reconnaître¹.

1 Types avancés

1.0 En OCaml : paires, triplets, etc

Cette sous-section concerne *uniquement* le langage OCaml.

On peut définir à l'aide de `x` (de type `t0`) et `y` (de type `t1`) deux expressions la paire composée de ces deux expressions : il s'agit simplement de `(x,y)` (elle est de type `t0*t1`).

On peut faire de même avec des triplets, quadruplets, etc.

1.1 Types enregistrements

Définition 2 (Enregistrement).

Un type enregistrement est un type dont les éléments sont des n-uplets. Le nombre et le type des coordonnées sont fixés, et chaque coordonnée a un identifiant. On appelle ces coordonnées des **champs**.

Par exemple, on pourrait vouloir définir le type `carte_bus` comme étant un triplet :

- `nom` : le nom d'une personne
- `nb_trajets` : le nombre de trajets restant sur la carte bus
- `age` : l'âge de la personne

Les types enregistrements sont très utiles pour définir des structures de données. En effet, une structure de données est souvent composée de plusieurs informations (par exemple, `arr`, `entree` et `sortie` pour un tableau circulaire). Utiliser un type enregistrement permet qu'au lieu de manipuler une variable par information, on manipule uniquement une « grosse » variable qui est un enregistrement.

L'autre intérêt est de pouvoir, via une interface, cacher le contenu de l'enregistrement au code client ; afin de simplifier : on préfère manipuler une file sans connaître son implémentation que d'avoir à réfléchir aux indices d'un tableau circulaire.

1.1.0 En C

Pour définir le type enregistrement donné en exemple précédemment en C :

```
1 struct carte_bus {
2     char* nom;
3     int nb_trajets;
4     unsigned age;
5 }; // Notez le ; final
```

Le type obtenu est nommé `struct carte_bus`, on peut lui donner un nouveau nom ainsi :

```
1 typedef struct carte_bus nouveau_nom;
```

1. Heureusement, même sans comprendre ce qu'est une macro, on devine assez bien ce qu'il se passe. La traduction de « define » est assez transparente.

Pour créer un élément de ce type, on peut le déclarer sans l'initialiser puis modifier le contenu de ses champs. On peut aussi initialiser tous ses champs lors de la création grâce à un initialisateur :

```
1 carte_bus moi = {.nom = "adomenech", .nb_trajets = 12, .age = 42};
```



On peut accéder (pour lire la valeur ou la modifier) à un champ `ch` d'une variable `var` via `var.ch`

1.1.1 En OCaml

Pour définir le type enregistrement donné en exemple précédemment en OCaml :

```
1 type carte_bus = {
2   nom : string;
3   nb_trajets : int;
4   age : int
5 }
```



Pour créer un élément de ce type :

```
1 let moi = {nom = "adomenech"; nb_trajets = 12; age = 42}
```



On peut lire le champ `ch` d'une variable `var` via `var.ch`

Remarque. Lorsque nous introduirons l'impératif en OCaml, nous verrons une variante des types enregistrements qui permet de créer des champs mutables.

1.2 Types énumérations

Définition 3 (Énumération).

Un type énumération est un type défini en listant exhaustivement les valeurs possibles d'un élément de ce type.

Exemple.

- On peut considérer le type booléen comme un type énuméré à deux membres : `true` et `false` (il n'est pas codé comme cela, mais c'est une bon ne introduction).

Dans les deux sous-sous-section suivantes, je présente des types énumérations qui contiennent trois membres, mais on pourra bien sûr faire de même à un, deux, quatre, cinq, six, etc.

1.2.0 (HP) En C

En C, voici comment définir un type énumération nommé `enum neveu` qui ne contient que trois valeurs `RIRI`, `FIFI` et `LOULOU` :

```
1 enum neveu = {RIRI, FIFI, LOULOU};
```



On peut bien sûr renommer en utilisant `typedef enum neveu new_name`

Remarque.

- On met des majuscules, car c'est la convention usuelle pour les macros. Le pré-compilateur va chercher-remplacer les occurrences de ces trois variables par respectivement 0, 1 et 2.
- Ils ne sont pas au programme.

1.2.1 En OCaml

En OCaml, voici comment définir un type énumération nommé `neveu` qui ne contient que trois valeurs `Riri`, `Fifi` et `Loulou` :

```
1 type neveu = Riri | Fifi | Loulou
```




On appelle Riri, Fifi et Loulou des **constructeurs** du type `neveu` (des constructeurs à 0 arguments pour être précis²).

On peut ensuite bien sûr donner ces valeurs des variables :

```
1 let a = Fifi
2 let b = Loulou
```



Remarque.  En OCaml, les majuscules sur les constructeurs sont obligatoires.

1.3 OCaml : types construits

1.3.0 Notion de type construit

En section 1, on a vu comment utiliser des constructeurs à 0 arguments pour définir une énumération en OCaml. On peut également faire des constructeurs de type prenant un argument.

Exemple. Voici par exemple un type `decision` ayant deux constructeurs. Le premier, `Indecis`, ne prend pas d'arguments; tandis que le second, `Choix`, prend en argument un booléen.

```
1 type decision = Indecis | Choix of bool
```



Les constructeurs ne peuvent prendre que 0 ou 1 argument. Si on veut leur en donner plus, il faut faire des constructeurs prenant en argument des paires, ou des triplets, etc.

Un type ainsi défini en donnant ses constructeurs s'appelle un **type construit**.

1.3.1 let destructurant

Si `p` désigne une paire, on peut utiliser un `let` pour accéder aux deux expressions de la paire. Voici par exemple des fonctions qui renvoient respectivement la première et la seconde coordonnée d'une paire :

```
1 let fst = fun paire ->
2   let (x,_) = paire in x
```



```
1 let snd = fun paire ->
2   let (_,y) = paire in y
```



On dit que de tels `let` sont des **let destructurant** : ils « ouvrent » la structure de ce qu'il y a à droite du signe `=` pour accéder à son contenu.

Pour destructurer un type construit, il faut faire une disjonction de cas :

```
1 type decision = Indecis | Choix of bool
2 let f = fun (d : decision) ->
3   match d with
4   | Indecis -> ...
5   | Choix b -> ...
```



Remarque. Les listes sont des types construits³, et c'est pour cela qu'on y applique des `match`.

2. Je vous renvoie au cours sur les structures de données pour comprendre cette appellation.

3. Le type `'a list` est certes un type construit récursif, mais un type construit quand même.

1.3.2 Le type 'a option

En informatique, on veut souvent faire des fonctions qui renvoient une solution si elle existe, et ne renvoient rien sinon.

Exemple.

- Recherche de l'indice d'un élément dans un tableau. Si l'élément est absent du tableau, il n'y a pas d'indice à renvoyer.
- Recherche d'une star dans un groupe (cf DS2). S'il n'y en a pas, il n'y a rien à renvoyer.
- Renvoyer la tête d'une liste. Si la liste est vide, il n'y a rien à renvoyer.

Dans de nombreux langages modernes, il existe un type pensé pour désigner « quelque chose, ou rien ». En OCaml, il s'agit du type `'a option`. Il est déjà codé, mais à titre informatif voici sa définition :

```
1 type 'a option = None | Some of 'a
```



C'est le type que l'on utilisera dans ces situations en OCaml. Si l'on trouve une solution, on renverra `Some la_solution`, et sinon `None`.

Remarque.

- Si on récupère un élément de type `'a option` renvoyé par une fonction, il faut bien sûr le filtrer pour faire une disjonction de cas sur s'il vaut `None` ou `Some truc`.
- À la différence du `None` de Python, ce `None`-ci a bien un type : `'a option`. Ainsi, si une fonction renvoie une `'a option` et que l'on oublie de la filtrer, le compilateur détectera une erreur *à la compilation*. Cela évite des bugs classiques, par exemple celui qui consiste à obtenir un indice via une recherche par dichotomie, puis accéder à cet indice sans avoir vérifié qu'il n'est pas `None`.⁴

1.4 Types récursifs

Que ce soit en C ou en OCaml, un type peut-être récursif : un de ses champs ou de ses constructeurs pour s'appeler lui-même :

Exemple. Voici deux exemples, un en C et un en OCaml :

```
11 struct cellule_s {
12     int elem;
13     struct cellule_s* next;
14 };
```

 list.h

```
16 type 'a list =
17     [] | (::) of 'a * 'a list
```

 list.ml

(Extrait du code source d'OCaml!)

Remarque.

- Nous verrons de nombreux types d'arbres au second semestre, qui sont des types récursifs.
- Ces types sont récursifs, on travaillera donc avec par récurrence. Nous verrons dans le chapitre sur l'induction comment étendre la récurrence dans le cas où chaque élément du type est défini à l'aide de *plusieurs* autres éléments. Par exemple :

```
1 type 'a bst = Nil | Node of ('a bst) * 'a * ('a bst)
```




- Par rapport aux types basés sur des tableaux, les types récursifs ont l'avantage d'être très flexibles dans la façon dont on « lie » des éléments entre eux et d'être très facilement redimensionnables. Ils ont l'inconvénient d'être plus difficiles à optimiser par la mémoire cache.

4. De manière générale, le compilateur d'un langage typé évite bien des erreurs.

2 Les exceptions

Une **exception** est un signal envoyé par le code qui interrompt l'exécution et attend que l'on signale que faire.

Par exemple, la fonction `int` de Python permet de convertir une donnée en entier. Si la donnée n'est pas interprétable comme un entier, elle déclenche l'exception `ValueError` ; on dit qu'on **lève** l'exception `ValueError`. Pour demander une valeur⁵ entière à l'utilisateur, on peut faire ainsi en Python :



```

1 x = 0
2 #Tant que la valeur de l'utilisateur n'est pas un entier, réessayer
3 while True:
4     try:
5         x = int(input("Entrez un entier :"))
6         break
7     except ValueError:
8         print("Vous n'avez pas entré un entier, réessayez.")

```

Dans l'exemple ci-dessus, on utilise un `try - except` : on essaye de faire un morceau de code. S'il ne lève aucune exception, tout va bien. Si une exception est déclenchée, on passe immédiatement au `except` : on *redirige exceptionnellement* l'exécution du code.

Point important : lorsqu'une exception est levée, l'exécution de la ligne de code en cours s'interrompt immédiatement, et on « remonte » dans le chemin suivi par le code jusqu'à trouvé un `except` qui gère cette exception. Ainsi, une exception peut faire quitter plusieurs appels de fonction d'un coup⁶.

Les exceptions ont deux rôles principaux :

- Signaler des erreurs, et y réagir. C'est le cas de `ValueError` présentée ci-dessus, ou encore de `DivisionByZero`.
- Sauter rapidement d'un point du code à un autre. Cela permet par exemple de quitter une imbrication d'appels récursifs très rapidement, et sans alourdir le code.


2.0 (HP) En C

Les exceptions telles que présentées ci-dessus n'existent pas en C. Il y a cependant une gestion d'erreur (rudimentaire) en C : si une fonction échoue, elle modifie une variable globale nommée `errno`. Le reste du code peut donc vérifier la valeur de `errno` pour savoir si la fonction s'est bien déroulée ou non.

2.1 En OCaml

2.1.0 Rattraper une exception

En OCaml, l'équivalent de `try - except` est `try - with` :



```

1 try
2     expr0
3 with
4     | NomDUneException -> expr1
5     | AutreExceptionEnvisage -> expr2
6     | ... etc

```

Le tout est une expression, dont le type est la valeur de `expr0` et `expr1` et `expr2` (et ainsi de suite) ; celles-ci doivent donc avoir le même type (comme dans un `then - else`).

5. Via `input` qui est l'équivalent de `scanf`.

6. De même qu'un `return` peut faire quitter plusieurs boucles imbriquées d'un coup, une exception peut faire quitter plusieurs blocs syntaxiques (boucles, fonctions, `if-else`, etc) d'un coup.

2.1.1 Lever une exception

Réciproquement, pour lever une exception, on utilise `raise NomDeLException`.

Exemple. Voici (à peu près) le code source de `List.hd` :

```
29 let hd lst =
30   match lst with
31   | [] -> raise (Failure "hd")
32   | a::_ -> a
```

 list.ml

Ici, si la liste est vide, on renvoie l'exception `Failure "hd"` (il s'agit d'une exception prenant un argument).

Voici, à titre indicatif, des exceptions courantes en OCaml :

- `Invalid_argument` : sert à signaler que l'argument donné à une fonction ne respecte pas les pré-conditions.
- `Not_found` : lorsqu'une fonction qui *doit* renvoyer quelque chose ne trouve pas la valeur à renvoyer. En général, une fonction qui essaye de renvoyer quelque chose mais peut échouer va soit renvoyer quelque chose ou lever une exception ; soit être codée comme renvoyant un type option.
- `End_of_file` : levée quand on essaye de lire un fichier alors qu'on a entièrement lu.
- `Failure` : levée par `failwith`. En fait⁷, la fonction `failwith` est définie ainsi :

```
1 let failwith msg =
2   raise (Failure msg)
```



2.1.2 Définir une exception

Pour définir (de manière globale) une exception :

```
1 exception NomDeLException
```



Il s'agit d'une déclaration globale (et non d'une expression).

Remarque.

- Une exception peut être déclarée comme prenant un argument (éventuellement un n-uplet), la syntaxe est la même que pour les constructeurs de type.
- Le nom d'une exception doit débuter par une majuscule.
- Les déclarations locales d'exceptions existent, mais ne sont pas au programme.

7. Et c'est explicitement au programme !

3 Écrire mieux

Un code est écrit une fois, modifié dix fois, et relu cent fois. Il faut donc simplifier son écriture, anticiper les modifications futures, et ne jamais négliger la lisibilité.

Proverbe de programmeur·se (issu de <https://ocaml.org/docs/guidelines>)

Toujours penser à la bonne poire qui devra déboguer et maintenir votre code : c'est le vous du futur. Prenez soin du vous du futur.

Proverbe personnel

Un·e programmeur·se passe (beaucoup) plus de temps à relire et éditer du code qu'à en écrire ; aussi créer du code de qualité n'est jamais du temps perdu⁸. Je présente ici quelques recommandations fondamentales. Vous pouvez exceptionnellement en dévier, uniquement exceptionnellement et pour de bonnes raisons.

3.0 Nommer

- **Utiliser des identifiants explicites.** Par exemple, si une variable `p` contient le nombre d'éléments pairs d'un tableau, ne l'appellez pas `p` ; appelez-le `nb_pairs`.
 - **Pour les variables locales, vous pouvez utiliser des abréviations pour raccourcir le nom ; c'est même recommandé :** une variable locale est souvent utilisée, les raccourcir permet d'accélérer la relecture.⁹ Et si jamais la·e relecteur·ice a un doute sur le sens de l'abréviation, la déclaration est sous ses yeux (une fonction est courte) pour lever le doute !
 - **Cependant, pour les identifiants de fonctions et de variables globales, il est déconseillé d'utiliser des abréviations :** en effet, elles sont déclarées loin avant dans le fichier (voire dans un autre fichier), et c'est fatigant de retourner chercher leurs définitions. Préférez un nom le plus explicite possible quitte à ce qu'il soit long.
- **Évitez les variables globales mutables.** Devoir suivre les changements de valeur d'une
- **N'utilisez pas des noms trop similaires dans une même portée.** Lorsque deux noms de variables ne diffèrent que par un ou deux caractères, on fait des erreurs d'écriture/relecture. Une variable globale, c'est *fatigant*.
- **Si dans différentes fonctions on retrouve des variables locales qui jouent le même rôle, donnez-leur le même nom.**¹⁰
- **Faites attention avec `i`, `j` ou `k`.** Ce sont de bons noms pour des variables qui sont juste des indices ; mais si c'est un indice qui a aussi un sens supplémentaire alors ce sont des noms catastrophiques. On en revient à la première règle : utiliser des noms explicites !
- **Choisissez une convention de casse et tenez-vous y.** Il existe deux grandes conventions pour écrire des noms de variables : séparer les sous-mots par des underscores (`par_exemple_cela` , on parle de **snake_case**), ou par des majuscules (`parExempleCela` , on parle de **camelCase**).

8. Même en TP à l'oral de concours : quand on découvre un bug dans une fonction écrite il y a 40min, il vaut mieux que le code soit bien écrit pour retrouver et corriger rapidement le bug.

9. Je précise : « court » ne signifie pas « une seule lettre ». Cela signifie « moins de dix lettres ».

10. Comme en maths où *f* désigne toujours les fonctions et *x* les arguments.

3.1 Structurer

- **Parenthésiez pour lever les ambiguïtés**, ou pour accélérer la relecture du code. Ne parenthésiez pas lorsque cela n'aide pas.
- **Identifiez votre code**. Python impose l'indentation (et c'est très bien) ; en C et OCaml il faut y faire attention. Voici des règles générales :
 - **À chaque entrée dans un bloc syntaxique, ajoutez un niveau d'indentation. À chaque sortie, enlevez-en un.**¹¹
 - **Une indentation mesure toujours le même nombre d'espaces (2 ou 4).**
 - **Utilisez des espaces ou des tabulations pour indenter, mais pas un mélange des deux.** Dans le cas des tabulations, les éditeurs de texte de développement proposent une option pour régler leur largeur ou pour les transformer automatiquement en espace.
 - **Une accolade fermante a le même niveau d'indentation que la ligne de l'accolade ouvrante associée.**
- **Utilisez des lignes vides pour séparer les blocs logiques de votre code.** Séparez toujours les fonctions du même nombre de lignes vides (typiquement, 2 ou 3). Dans vos fonctions, faites une ligne vide uniquement pour aider à la relecture en divisant la fonction en « paragraphes » (en groupes de lignes qui ont chacun un rôle précis). Inutile toutefois de découper une fonction si elle est déjà simple : laissez les choses simples être simples.
- **Gardez vos fonctions courtes : évitez de dépasser 20 lignes.**
- **Gardez vos lignes courtes : évitez de dépasser 80 caractères.**
- **Une ligne doit correspondre à une seule action.** N'écrivez pas $h(g(f(x)))$, c'est illisible.¹²
- **Si créer une variable rend le code plus lisible, créez une variable.** Une variable ne coûte (vraiment) pas cher¹³ ; alors qu'ajouter un nom à une quantité intermédiaire peut aider à relire le code. En particulier, évitez les « nombres magiques » : si un nombre traine dans votre code sans que l'on comprenne son sens, c'est un mauvais code.
- **Déclarez les variables locales au plus proche de leur utilisation.** Lire une déclaration d'une variable locale qui ne servira que dans 20 lignes, c'est *fatigant*.
- **Si créer une fonction rend le code plus lisible, créez une fonction.** Cela permet de donner un nom explicite au rôle d'un ou plusieurs paragraphes du code.
- **Si plusieurs morceaux de code réalisent la même tâche, créez et utilisez une fonction réalisant cette tâche.** Cela évite la redondance de code qui est source d'erreur : si plusieurs morceaux de code font la même tâche, il est vite arrivé de mettre à jour un des morceaux de code mais pas l'autre et de créer ainsi un bug.
- **Découpez votre code en plusieurs fichiers.** Chaque fichier doit correspondre à une famille de tâche (définir une structure de données, définir des tests, etc). Cela simplifiera *vraiment* la relecture et le débogage.

3.2 Documenter

- **Spécifiez vos fonctions.** Cette spécification se fait typiquement dans l'interface.
- **Utilisez les commentaires pour expliquer les morceaux de code compliqué.** Un commentaire d'une ligne pour un paragraphe de code suffit en général : il faut simplement dire *à quoi servent* ces lignes de code. Ne commentez pas ce qui est déjà simple à relire, et ne paraphrasez pas le code.
- **Annotez les boucles compliquées d'un commentaire en donnant un invariant utile.** On le fait généralement en langage naturel et non en langage mathématique.
- **Si une fonction correspond à un algorithme compliqué, vous pouvez à la débiter par un gros commentaire explicatif.**

11. Notez que c'est ainsi que Python définit ses blocs syntaxiques.

12. En programmation orientée objet, c'est d'ailleurs une des sources les plus courantes de code inutilement compliqué à relire.

13. Et le compilateur peut les enlever lors de ses optimisations du code.

3.3 Anticiper

- **Sachez ce que vous allez coder avant de le coder : de quoi aurez-vous besoin, où et comment allez-vous stocker les données, comment les manipuler, etc ; en bref comment votre code sera organisé.** Si vous n'avez aucune idée du code à écrire, sortez du brouillon et réfléchissez au brouillon.
- **Un bon code ne génère aucun Warning à la compilation.** En C, vous devez utiliser `-Wall -Wextra` pour avoir tous les warnings possibles.
- **Faites de la programmation défensive.** Utilisez des `assert` pour garantir que les pré-conditions sont vérifiées. Il faut que le non-respect des pré-conditions soit élémentaire à déboguer !
- **Garantisiez l'exhaustivité des disjonctions de cas.** Toute disjonction de cas (resp. filtrage par motif) qui n'est pas trivialement exhaustive doit se terminer par un `else` (resp. par un `| _ ->`), quitte à ce que cette branche ne contienne qu'une levée d'exception.¹⁴
- **Faites des messages d'erreurs explicites.** Les `assert` c'est très bien, mais ils donnent un message d'erreur peu lisible¹⁵. Utiliser `if` qui affiche un message d'erreur avant d'interrompre le programme ou de lever une exception, c'est encore mieux.
- **TESTEZ VOS FONCTIONS!!** Plus d'information à ce sujet dans la prochaine section.

3.4 Simplifier

- **K.I.S.S. Keep It Stupid Simple :** *gardez votre code stupidement simple*. Ne vous surestimez pas, et utilisez tous les conseils précédents pour rendre le code stupidement simple. Évitez les optimisations qui rapportent peu mais nuisent à la lisibilité. Évitez les usines à gaz, concevez plutôt des fonctions et bibliothèques élémentaires qui font une seule chose mais la font très bien.
- **Premature optimisation is the root of all evil :** *l'optimisation prématurée est la racine de tous les maux* (citation de D. Knuth). Faites d'abord un programme qui marche, et ensuite seulement un programme optimisé (et uniquement si nécessaire). N'ajoutez qu'une optimisation à la fois, de la plus simple à la plus compliquée - et testez à chaque fois ! Pour réaliser ces mises à jour successives sans pour autant perdre la version de base qui fonctionne, n'hésitez pas à faire correspondre cela à plusieurs implémentations d'une même interface.

3.5 Progresser

Vos autres conseils à vous-même ici :

14. Pensez par exemple à `List.hd` : si on lui donne une liste vide (ce que l'on est pas sensés faire), elle lève une exception.

15. Mieux vaut des `assert` que rien !!

4 Tester mieux

Les erreurs sont humaines, et arrivent : c'est normal. C'est pour s'en prévenir que l'on :

- Fait relire son code par d'autres.
- Prouve les morceaux compliqués du code.
- Fait des tests !

Un seul test ne suffit pas : il faut tout un **jeu de test**, c'est à dire un ensemble de tests qui recouvrent tous les points du code. Plus précisément, voici les pré-requis d'un bon jeu de test :

- **Chaque test est unitaire, c'est à dire qu'il teste une seule chose.** Il faut que lorsqu'un test échoue on puisse immédiatement pointer d'où provient le bug.
- **Testez une fonction avant de travailler sur la suivante.** C'est un corollaire du point précédent.
- **Il y a des tests pour les cas limites.** Si une fonction traite un tableau ou une liste, il faut vérifier que les cas particuliers du 1er élément et du dernier élément du tableau fonctionnent bien. Si une fonction est récursive, il faut vérifier que les cas de base fonctionnent bien.
- **Il y a aussi des tests pour les cas généraux.**
- **Si votre fonction doit avoir une bonne complexité, il y a des tests sur de grandes entrées.**
- **Si vous avez un vérificateur, générez des tests aléatoires et vérifiez le résultat de votre code sur ces tests.**
- **Le jeu teste toutes les façons de satisfaire ou non les conditions logiques.** Quand une condition contient des ET/OU, il faut tester toutes les façons de satisfaire ou non les clauses de la condition.
- **Le jeu couvre tous les sommets du Graphe de Flot de Contrôle, c'est à dire qu'il vérifie que toutes les lignes du code sont atteignables.** Cela permet de vérifier qu'on ne s'est pas trompés en écrivant des conditions impossibles dans des sauts conditionnels (en particulier quand on enchaîne les `else if`) et les boucles `while`.
- **Le jeu couvre tous les arcs du GFC, c'est à dire qu'il vérifie que tous les embranchements du code sont utilisables.**

Les deux derniers points sont, et de loin, les plus longs. Il est raisonnable de supposer que vous n'avez pas le temps de le faire dans des TP pensés pour 2/3/4h de programmation.

5 Déboguer mieux

Voici quelques conseils très généraux sur comment déboguer :

- **Avoir un code bien écrit, et doté de tests.** C'est le point le plus important.
- **Utilisez des `assert` pour vérifier que des conditions sont vraies.** C'est en fait le rôle principal de `assert` : déboguer. N'hésitez pas à mettre des `assert` au milieu du code pour vérifier ce que vous avez besoin de vérifier (qu'un indice est compris entre certaines bornes, qu'une certaine quantité est minorée/majorée, qu'un pointeur est non-NULL, etc).
- **Ajoutez des `print` pour voir l'état de certaines variables.** Ce n'est vraiment pas la meilleure méthode car on se retrouve vite submergé.e. On devrait préférer le prochain point, mais il n'est pas au programme en MP2I/MPI.
- **(HP) Utilisez des débogueurs comme `gdb` ou `ocamldebug`.** Je ne vais pas m'étendre dessus, mais voici deux liens si vous voulez creuser :
 - https://perso.ens-lyon.fr/daniel.hirschhoff/C_Caml/docs/doc_gdb.pdf (il y manque la fonctionnalité `frame X`, qui permet de se déplacer à l'appel `X` de la pile d'exécution)
 - <https://ocaml.org/docs/debugging#the-ocaml-debugger> (en anglais, et utilise quelques éléments avancés de OCaml)

Chapitre 7

RETOUR SUR TRACE

Notions	Commentaires
Recherche par force brute. Retour sur trace (<i>Backtracking</i>).	On peut évoquer l'intérêt d'ordonner les données avant de les parcourir (par exemple par une droite de balayage).

Extrait de la section 4.2 du programme officiel de MP2I : « Exploration exhaustive ».

SOMMAIRE

0. Fils rouges	122
0. Sudoku	122
1. n dames	122
1. Exploration exhaustive	123
0. Définition et limites	123
1. Pseudo-code général d'une exploration exhaustive	124
2. Exploration exhaustive des n dames	125
3. Amélioration de l'exploration exhaustive des n dames	127
2. Retour sur trace	128
0. Définition	128
1. Retour sur trace pour les n dames	129
2. Variantes	130
3. Complexité.....	131
0. Analyse théorique	131
1. Importance de l'ordre des appels	131
4. Autres exemples classiques	132
0. Cavalier d'Euler	132
1. Perles de Dijkstra	132
2. SUBSETSUM	133
3. CNF-SAT	133
5. Écriture itérative	133

On s'intéresse dans ce chapitre à la recherche de l'existence d'une solution parmi *plein* de possibilités.

0 Fils rouges

0.0 Sudoku

Le Sudoku est un casse-tête qui consiste à remplir une grille 9×9 en faisant en sorte que dans chaque ligne, colonne, et gros carré il y ait chaque chiffre de $\llbracket 1; 9 \rrbracket$ exactement une fois.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Une grille de Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(b) Une grille de Sudoku résolue.

FIGURE 7.1 – Grille de Sudoku. Les gros carré ont des bords en gras.

Remarque. Ceci est le 9-Sudoku, mais on peut aussi définir des Sudoku n'importe quel côté de longueur carrée : 4, (9), 16, 25, etc.

0.1 n dames

Un autre problème classique est le problème des n dames :

- ENTRÉE : n un entier strictement positif.
- TÂCHE : Trouver s'il est possible et comment placer n dames sur un échiquier $n \times n$ sans que deux dames ne soient en prise (deux dames sont en prises si elles sont alignées verticalement ou horizontalement ou diagonalement).

(a) Une prise horizontale

(b) Une prise verticale

(c) Une prise diagonale

FIGURE 7.2 – Exemples des 3 natures de prises possibles sur échiquier 5×5

1 Exploration exhaustive

1.0 Définition et limites

Définition 1 (Exploration exhaustive).

L'**exploration exhaustive** consiste à parcourir toutes les potentielles solutions d'un problème, et à vérifier les unes après les autres s'il s'agit bel et bien de solutions ou non.

On parle aussi de **force brute** (« brute force » en VO) : on essaye pas de faire un algorithme intelligent ; on utilise simplement et stupidement la force de calcul de notre ordinateur.

Exemple.

- Générer toutes les façons de parcourir une grille de Sudoku (il y en a $9^{\text{nombre_de_cases_vides}}$), puis pour chacune de ces façons tester si elle est valide ou non.
C'est *très* lent : une grille a $9 \times 9 = 81$ cases. Mettons que la grille soit simple et soit déjà à moitié remplie : cela fait $9^{40} \approx 10^{38}$ possibilités. Sur un processeur de $\sim 1\text{GHz}$, il faut $\sim 10^{29}$ secondes soit $\sim 4.10^{21}$ ans. C'est dix millions de fois l'âge de l'univers.
- Pour les n dames, il y a n dames à placer sur un damier à n^2 cases, donc n^{2n} possibilités. Avec $n = 8$ et un processeur à $\sim 1\text{GHz}$, il faut 78 heures. C'est plus raisonnable que le Sudoku, mais cela reste beaucoup.

Remarque. Le terme « force brute » est un terme général en informatique, qui désigne souvent mais pas toujours l'exploration exhaustive. L'idée des méthodes par force brute est cependant toujours la même : ne pas réfléchir et faire (aveuglément) confiance au GHz.

Proposition 2 (Inefficacité des explorations exhaustives).

Les explorations exhaustives (et plus généralement toutes les méthodes par force brute) ne fonctionnent que sur des petites entrées.

Leur principal avantage est d'être simples à coder, et de permettre de vérifier les résultats renvoyés par des algorithmes plus avancés.

Il faut pour cela découper une potentielle solution en plusieurs « choix » successifs. Ainsi, au lieu

Exemple.

- Pour le Sudoku, il faut d'abord choisir ce que l'on met dans la première case, puis choisir ce que l'on met dans la seconde, etc.
- Pour les n dames, il faut d'abord choisir où placer la première dame, puis où placer la seconde, etc.

Définition 3 (Vocabulaire).

On considère un problème dont le but est de déterminer s'il existe un uplet $(x_0, \dots, x_{\ell-1})$ vérifiant une certaine propriété. Pour construire un tel uplet, on choisit d'abord x_0 , puis x_1 , etc.

Dans ce cours, on appelle :

- **solution partielle** : c'est « une solution en cours de construction où il reste des choix à faire ». Formellement, c'est un uplet $(x_0, x_1, \dots, x_{i-1})$ avec $i - 1 < \ell$.
- **solution complète** : c'est « une solution entièrement remplie, où il n'y a plus de choix à faire ». Formellement, c'est un uplet $(x_0, \dots, x_{\ell-1})$.
- **solution valide**, ou juste **solution** : c'est une solution complète qui vérifie la propriété attendue.

De plus, si s est une solution partielle et s' une solution partielle ou complète, on dit que s' **est une extension de s** lorsque « s' a été obtenue à partir de s en faisant des choix supplémentaires ». Formellement, si $s = (x_0, x_1, \dots, x_{i-1})$ et $s' = (y_0, y_1, \dots, y_{i-1})$, s' étend s si et seulement si $j \geq i$ et $\forall k \in \llbracket 0; i \rrbracket, x_k = y_k$.

Exemple.

- Une solution partielle du Sudoku est une grille partiellement remplie. Une solution complète une grille entièrement remplie, et une valide une grille entièrement remplie qui respecte les règles.
- Une solution partielle des n dames est le placement de i dames. Une solution complète est le placement de n dames, et une valide est n dames qui ne sont pas en prise.

Remarque.

- Cette formalisation n'est pas parfaite (par exemple, elle sous-entend que toutes les solutions complètes ont la même longueur ce qui peut-être discutable), mais a l'avantage de nous donner du vocabulaire et des notations que je peux réutiliser dans tout ce cours.
- Ce n'est pas du vocabulaire « officiel », redéfinissez ces termes avant de les utiliser si vous en avez besoin en concours.

1.1 Pseudo-code général d'une exploration exhaustive

Pour coder une exploration exhaustive, la récursivité est très souvent utile : on fait un choix, puis les choix suivants sont faits récursivement.

Il faut de plus disposer d'un vérificateur, une fonction qui vérifie si une solution potentielle est bel et bien une solution ou non. On peut alors appliquer l'algorithme récursif ci-dessous :

Algorithme 10 : EXHAUSTIF

Entrées : $sol = (x_0, \dots, x_{i-1})$ une solution partielle ou complète
Sorties : Vrai si et seulement si sol peut-être étendue en une solution valide.
 Effet secondaire : afficher une solution valide si elle existe.

```

1  si  $sol$  est complète alors
2    si  $VÉRIFICATEUR(sol)$  alors
3      Afficher  $sol$ 
4      renvoyer Vrai
5    sinon
6      renvoyer Faux

// Sinon : tester toutes les extensions possibles
7  pour chaque valeur  $x_i$  possible pour le prochain choix faire
8    // Essayer avec ce  $x_i$ 
9     $sol \leftarrow sol$  étendue avec  $x_i$ 
10   si  $EXHAUSTIF(sol)$  alors
11     renvoyer Vrai

// Sinon, annuler le choix  $x_i$  et en tester un autre au prochain tour de boucle
12  $sol \leftarrow sol$  sans  $x_i$ 

// Si aucun choix n'a fonctionné, on renvoie Faux
12 renvoyer Faux
```

Remarque.

- Le point très important de cet algorithme est qu'il faut annuler un choix s'il n'a pas fonctionné ! En effet, si sol est mutable (comme dans ce pseudo-code), ne pas annuler les choix mène à des erreurs du type « un appel récursif a fait un mauvais choix, n'a pas enlevé ce choix, donc les appels récursifs suivants qui testent d'autres choix ont toujours ce mauvais choix à l'intérieur de leur sol et plantent ».
- Des écritures non-récursives sont possibles mais généralement plus techniques à rédiger. On y utilise parfois des compteurs n -aires pour représenter et parcourir des sous-ensembles.

1.2 Exploration exhaustive des n dames

Voici des extraits pertinents de `exhaustif.c` qui implémente la méthode générale présentée ci-dessus :

```

6  /*
7  Choix d'implémentation :
8  - le plateau est indicé comme une matrice n*n (avec (0,0) en haut à gauche)
9  - les n dames sont stockées dans un tableau dames.
10 - une dame absente correspond aux coordonnées (-1,-1)
11 */

```

 `exhaustif.c`

```

15 /** Une coordonnée est une paire (ligne, colonne) */
16 struct coo_s {
17     int lgn;
18     int col;
19 };
20 typedef struct coo_s coo;

```

 `ndames.h`

Et voici le retour sur trace (sans l'affichage de la solution) :

```

88 /** Renvoie true ssi il existe une façon de compléter
89  * les nb_dames déjà placées qui respecte les règles.
90  *
91  * dames est le tableau des nb_dames placées,
92  * n le nombre de dames à placer.
93  */
94 bool rec_exhaustif(coo* dames, int nb_dames, int n) {
95     if (nb_dames == n) {
96         return verificateur(dames, n);
97     }
98
99     // Parcourir toutes les façons de placer une dame
100    for (int lgn = 0; lgn < n; lgn += 1) {
101        for (int col = 0; col < n; col += 1) {
102            dames[nb_dames].lgn = lgn;
103            dames[nb_dames].col = col;
104            // Si mettre cette dame mène à une solution : GG
105            if (rec_exhaustif(dames, nb_dames+1, n)) {
106                return true;
107            }
108            // Sinon, enlever cette dame
109            dames[nb_dames].lgn = -1;
110            dames[nb_dames].col = -1;
111        }
112    }
113
114    return false;
115 }

```

 `exhaustif.c`

FIGURE 7.3 – (Partie de l') arbre des appels récursifs qui ont lieu pour $n = 3$

1.3 Amélioration de l'exploration exhaustive des n dames

Sans pour autant faire un algorithme très intelligent, on peut proposer une amélioration de la fonction précédente : d'après la règle de la prise verticale, il y a au plus une dame par colonne. On peut donc construire une solution ainsi :

- Choisir la ligne de la dame de la colonne 0.
- Puis choisir la ligne de la dame de la colonne 1.
- Etc.

On obtient le code `exhaustif-ameliore.c` dont voici des extraits pertinents :

```

6  /*
7  Choix d'implémentation :
8  - le plateau est indicé comme une matrice n*n (avec (0,0) en haut à gauche)
9  - la ième dame est en colonne i.
10 - le tableau dames_lgn stocke les lignes des dames
11 - une dame absente correspond à la ligne -1
12 - notez que le type coo ne sert plus
13 */

```

```

85 /** Renvoie true ssi il existe une façon de compléter
86  * les nb_dames déjà placées qui respecte les règles.
87  *
88  * dames_lgn est le tableau qui à l'indice d'une dame
89  * placée associe sa colonne (la dame i est en ligne i),
90  * nb_dames le nombre de dames déjà placées,
91  * n le nombre de dames à placer.
92  */
93 bool rec_exhaustif(int* dames_lgn, int nb_dames, int n) {
94     if (nb_dames == n) {
95         return verificateur(dames_lgn, n);
96     }
97
98     // Parcourir toutes les lignes pour la prochaine dame
99     for (int lgn = 0; lgn < n; lgn += 1) {
100         dames_lgn[nb_dames] = lgn;
101         if (rec_exhaustif(dames_lgn, nb_dames+1, n)) {
102             return true;
103         }
104         dames_lgn[nb_dames] = -1; // ne pas oublier d'annuler le choix !!
105     }
106
107     return false;
108 }

```

Remarquez qu'on a une boucle for imbriquée de moins ! Le gain en temps n'est pas négligeable :

Algorithme employé	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
Exhaustif	2ms	4ms	1,2s	26s	3 heures
Exhaustif amélioré	2ms	1ms	2ms	2ms	25ms

FIGURE 7.4 – Durées d'exécution sur un processeur à ~3GHz

Remarque.

- Le gain de temps spectaculaire de cette simple amélioration s'explique par le fait qu'ajoute la condition « une dame par colonne » réduit énormément les possibilités (passage à la racine !) :

Contraintes	Nombre de possibilités pour n dames	$n = 4$	$n = 7$	$n = 8$
Aucune	n^{2n}	65536	$6,8 \cdot 10^{11}$	$2,8 \cdot 10^{14}$
Une dame par case	$\binom{n^2}{n}$	1820	$8,6 \cdot 10^7$	$4,4 \cdot 10^9$
Une dame par ligne	n^n	256	823543	$1,7 \cdot 10^7$
Une dame par ligne et colonne	$n!$	24	5040	40320

FIGURE 7.5 – Nombres de façons de placer n dames sur un échiquier $n \times n$ selon différentes contraintes. Les écritures scientifiques sont données à 2 chiffres significatifs.

- Le 1ms de l'exhaustif amélioré pour $n = 5$ n'est pas une erreur : c'est un « coup de chance », il se trouve que pour $n = 5$ ma recherche récursive trouve une très vite et doit peu souvent « essayer un autre choix ».
- En effet, mes fonctions s'arrêtent dès qu'elles ont trouvé une solution : elles n'explorent donc pas *tout* l'espace des possibilités.

2 Retour sur trace

2.0 Définition

Définition 4 (Rejet et retour sur trace).

Une solution partielle est dite **rejettable** si elle ne peut pas être étendue en une solution valide. Un algorithme de **retour sur trace** est une accélération de l'exploration exhaustive qui après chaque choix essaye de détecter si la solution partielle obtenue est rejettable. Si oui, on annule ce choix, sinon on essaye récursivement d'étendre cette solution partielle.

Remarque.

- Toute la difficulté du retour sur trace est d'écrire une fonction qui détecte le plus fidèlement possible les solutions rejettables ! Dans les deux exemples fils rouge de ce cours c'est facile, dans d'autres cas plus compliqué.
- Pour accélérer la recherche, en plus d'un bon rejet il est utile d'avoir une bonne « intuition » : si l'on arrive à prévoir quel choix a le plus de chance d'être étendue en une solution valide, on peut tester ce choix en premier (et tester en dernier les choix les moins prometteurs).
- En MPI, vous étudierez les algorithmes *Branch and Bound*. Il s'agit d'une adaptation de la méthode de retour sur trace à l'optimisation.
- Le terme « trace » désigne l'historique d'un programme. Un algorithme par « retour sur trace » est donc un algorithme qui peut revenir en arrière et remettre en cause des choix précédents. Avec notre façon d'écrire le code, ce retour en arrière est très simple et géré par la sortie des appels récursifs.

Exemple.

- Pour le Sudoku, on peut rejeter une solution dès qu'elle a deux chiffres identiques sur une même ligne / colonne / grand carré. Notez que cela suffira à prouver que la grille n'est pas remplie, car si les 9 chiffres d'une ligne sont distincts alors les 9 chiffres sont présents, et idem pour les lignes et carrés.¹

1. Pour tout E fini et $f : E \rightarrow E$, f est injective si et seulement si elle est bijective (si et seulement si elle est surjective).

- Pour les n dames, on peut rejeter dès que 2 dames sont sur la même ligne / colonne / diagonale.

Voici par exemple pour $n = 4$ l'exploration que l'on obtient avec ce rejet :

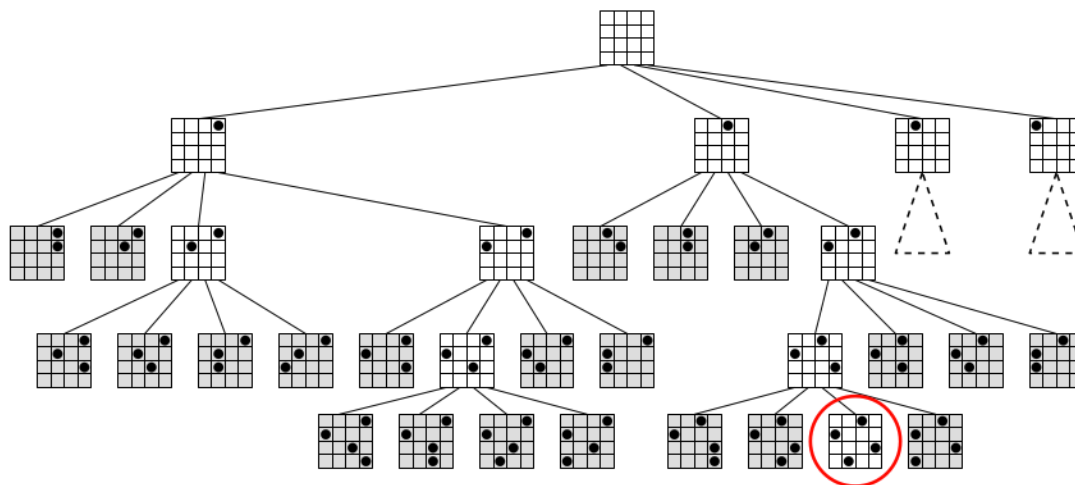


FIGURE 7.6 – Exploration en plaçant d'abord sur la première ligne, puis sur la seconde, etc. En gris les solutions partielles rejetées. La moitié non-représentée de la figure est symétrique. Src : J.B. Bianquis

Le pseudo-code générique d'une recherche exhaustive peut-être modifié ainsi pour obtenir un retour sur trace :

Algorithme 11 : RETOURSURTRACE

Entrées : $sol = (x_0, \dots, x_{i-1})$ une solution partielle ou complète

Sorties : Vrai si et seulement si sol peut-être étendue en une solution valide.

```

1 si  $sol$  est complète alors
2   | renvoyer VÉRIFICATEUR( $sol$ )

3 pour chaque valeur  $x_i$  possible pour le prochain choix faire
4   | // Essayer le choix  $x_i$ 
5   |  $sol \leftarrow sol$  étendue avec  $x_i$ 
6   | si REJET( $sol$ ) ne rejette pas la solution partielle alors
7   |   | si RETOURSURTRACE( $sol$ ) alors
7   |     | renvoyer Vrai
8   |   | // Sinon : annuler le choix  $x_i$  et en essayant un autre à la prochaine iter
8   |   |  $sol \leftarrow sol$  sans  $x_i$ 

9 renvoyer Faux
```

⚠ J'insiste : penser à défaire un choix s'il n'a pas fonctionné (lgn 8) est très important et vous évitera énormément de débogage !

2.1 Retour sur trace pour les n dames

On se base sur la version améliorée de la recherche exhaustive, à laquelle on veut ajouter un rejet. Pour cela, après chaque choix on va tester si la dame que l'on vient d'ajouter est en conflit avec une des dames précédentes. La fonction `check_dame` réalise cela. On obtient le code suivant pour le retour sur trace :



```

82  /** Renvoie true ssi il existe une façon de compléter
83      * les nb_dames déjà placées qui respecte les règles.
84      *
85      * dames_lgn est le tableau qui à l'indice d'une dame
86      * placée associe sa colonne (la dame i est en ligne i),
87      * nb_dames le nombre de dames déjà placées,
88      * n le nombre de dames à placer.
89      */
90  bool backtrack(int* dames_lgn, int nb_dames, int n) {
91      if (nb_dames == n) {
92          return verificateur(dames_lgn, n);
93      }
94
95      for (int lgn = 0; lgn < n; lgn += 1) {
96          dames_lgn[nb_dames] = lgn;
97          if (check_dame(dames_lgn, nb_dames)
98              && backtrack(dames_lgn, nb_dames+1, n) )
99              {
100                  return true;
101              }
102
103          dames_lgn[nb_dames] = -1;
104      }
105
106      return false;
107  }

```

Remarque.

- Notez l'usage de l'évaluation paresseuse du `&&` en lignes 97-98 : l'exploration récursive n'est effectuée *que* si la solution n'est pas rejetée.
- Notez la très grande similarité avec la recherche exhaustive. En fait, la recherche exhaustive est un cas particulier de retour sur trace : celui où notre fonction rejet est très très très mauvaise.
- La ligne 92 peut-être simplifiée : si l'on atteint cette ligne, c'est que l'on a réussi à placer n dames sans qu'aucune d'entre elles ne soit en prise avec une dame d'avant. Autrement dit, aucune dame n'est en prise : on peut `return true`.
- Cette fonction est *très* performante : 2ms pour $n = 8$, 72ms pour $n = 20$.

2.2 Variantes

Jusque là, le problème que nous résolvions était de la forme « existe-t-il une solution » ? Il existe en fait 4 variantes classiques pour un retour sur trace :

- Déterminer l'existence d'une solution valide.
- Renvoyer une solution valide si elle existe.
- Trouver le nombre de solutions valides.
- Renvoyer toutes les solutions valides.

Pour les variantes où il faut renvoyer une (resp. ou des) solutions, le plus simple est de faire en sorte que les appels récursifs se partagent l'adresse d'un endroit où aller recopier (resp. ajouter une copie de) une solution valide quand on en trouve une.

Attention à bien *recopier* la solution trouvée, cela évite des erreurs du type « la solution valide que j'ai trouvée a été modifiée par les appels récursifs suivants et n'est plus valide ».

Remarque. Tout ceci est très général, les choix précis à faire dépendent du langage et des choix d'implémentation.

3 Complexité

3.0 Analyse théorique

Il s'agit de calculer la complexité d'un algorithme récursif. Avec les retours sur trace, le plus simple est généralement de :

- Majorer grossièrement le coût de la fonction de rejet.
- De même pour la vérification, si elle est distincte du rejet.
- Majorer le nombre d'appels qui ont lieu par le nombre d'appels qui auraient lieu sans aucun fonction de rejet. Cela est généralement assez simple car il s'agit du nombre de noeuds d'un arbre parfait.
- Multiplier le coût des rejets et vérifications par le nombre d'appels. Par exemple

Exemple. Pour les n dames, le coût réel du rejet est $O(\text{nombre_de_dames_placees})$. On peut majorer cela par $O(n)$ pour simplifier. La vérification n'est pas nécessaire (cf remarque à propos de la ligne 92) et on l'enlève donc. Chaque appel fait au plus n appels récursifs (les n façons de placer la prochaine dame), et la profondeur de l'arbre d'appels est de n . L'arbre d'appels contient donc n^n appels. Ainsi, au total, une borne supérieure de la complexité de ce code est $O(n^{n+1})$.

Proposition 5 (Limites de l'analyse théorique).

Sur les retours sur trace, l'analyse théorique présentée ci-dessus est très limitée. Le bon outil d'analyse est la mesure du temps d'exécution en pratique.

En effet, tout l'intérêt d'un retour sur trace est de ne *pas* explorer toutes les possibilités, et beaucoup beaucoup beaucoup de solutions partielles sont rejetées.

3.1 Importance de l'ordre des appels

En plus d'un bon rejet, un point important pour accélérer un retour sur trace est de tester les appels récursifs dans le bon ordre : on veut faire en premier les choix les plus probables !

Exemple. Sur le Sudoku, on peut choisir le contenu des cases dans l'ordre, de gauche à droite et de haut en bas. Cela fonctionne, mais n'est pas le plus efficace. Par exemple, sur la grille ci-dessous, on voit que certaines cases ont 4 valeurs qui peuvent y être mises sans rejet immédiat, alors que d'autres n'en ont qu'1. Il vaut mieux compléter en premier ces dernières (pour se rapprocher plus vite d'une solution valide ou pour rejeter plus vite) !

1234	23	123	2
123	23	123	4
23	4	2	1
2	1	24	3

FIGURE 7.7 – Valeurs non-immédiatement rejetables (en rouge) pour les cases vides d'une grille de 4-Sudoku (dont les chiffres déjà placés sont en noir)

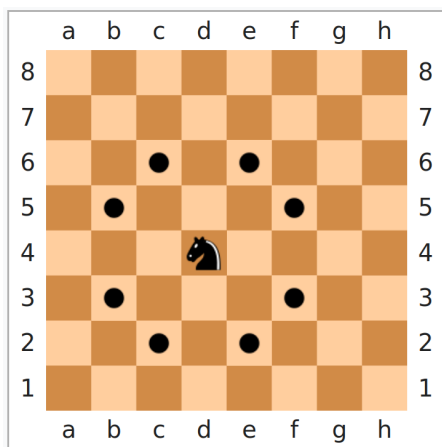
4 Autres exemples classiques

Voici d'autres exemples de retour sur trace classiques, que nous ferons en TP ou en DM².

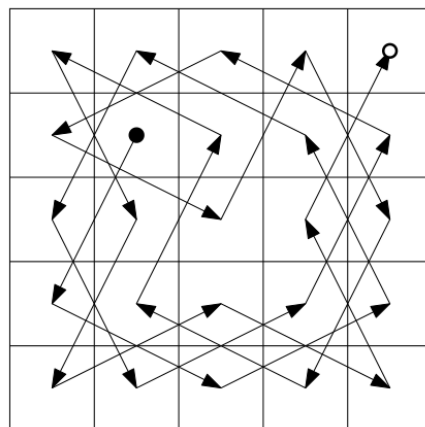
4.0 Cavalier d'Euler

Le problème du cavalier d'Euler est le suivant :

- Entrée : n la dimension du côté d'un échiquier et (i, j) les coordonnées d'une case de l'échiquier
- Tâche : déterminer si un cavalier initialement placé en (i, j) peut se déplacer sur l'échiquier de sorte à se poser sur toutes les cases mais jamais deux fois sur la même.



(a) Les 8 mouvements possibles d'un cavalier



(b) Résolution du cavalier d'Euler sur un échiquier 5×5 en partant de $(1, 1)$

FIGURE 7.8 – Autour du cavalier d'Euler

4.1 Perles de Dijkstra

Les perles de Dijkstra est un exercice historique de programmation :

- Entrée : n un entier strictement positif
- Tâche : on dispose de perles Bleu, Blanche et Rouge³. Comment en enfiler n sur un fil de sorte à ce qu'il n'y ait pas deux séquences adjacentes identiques ?

(a) Un fil avec deux séquences adjacentes identiques

(b) Un autre fil avec deux séquences adjacentes identiques

(c) Un fil avec 10 perles sans séquences adjacentes identiques

D'un point de vue plus mathématique, il s'agit de rechercher un mot ternaire sans carré. Pour en savoir plus : https://fr.wikipedia.org/wiki/Mot_sans_facteur_carr%C3%A9.

2. Le Sudoku sera un DM!

3. Comme le drapeau néerlandais, Dijkstra étant néerlandais. Vous pensiez à quel pays ?

4.2 SUBSETSUM

Le problème de la somme d'un sous-ensemble, aussi appelé SUBSETSUM, est le suivant :

- Entrée : $E \subset \mathbb{N}$ un ensemble fini d'entiers positifs et $t \in \mathbb{Z}$ une « cible » (target).
- Tâche : trouver $P \subseteq E$ tel que $\sum_{x \in P} x = t$

C'est un problème très célèbre en informatique, car c'est un exemple classique de problème NP-complet. Rendez-vous en MPI pour en savoir plus !

4.3 CNF-SAT

Dans le cours de Logique, nous verrons le problème CNF-SAT qui consiste à chercher une manière de satisfaire une formule logique. Nous y verrons l'algorithme de Quine, qui est un retour sur trace !

5 Écriture itérative

Il existe des façons impératives d'écrire les retours sur trace. **Je les décommande fortement, car il est bien plus simple de faire des erreurs ainsi.**

Il demande de pouvoir ordonner les valeurs possibles pour le i -ème choix. Étant donnée x_i une valeur possible pour le i -ème choix, on note $\text{SUIVANT}(x_i)$ la valeur à tester pour le i -ème choix si x_i n'a pas permis d'obtenir une solution valide. Quand il n'y a plus de prochaine valeur possible pour ce choix, SUIVANT renvoie une erreur ou une valeur particulière. On note aussi $\text{PREMIER}(i)$ le premier choix à tester pour le i -ème choix.

Exemple.

- pour le Sudoku, le i -ème choix est le contenu de la i -ème case, et on peut ordonner les valeurs possibles pour ce choix en posant $0 < 1 < \dots < 9$. Donc $\text{SUIVANT}(1) = 2$, etc.
- pour les n dames (avec une dame par colonne), le i -ème choix est la ligne de la i -ème dame. On les ordonne là aussi naturellement : $0 < \dots < n - 1$.

Algorithme 11 : RETOURSURTRACE IMPÉRATIF (à éviter!!)

Entrées : ℓ la longueur de solution valide voulue

Sorties : Vrai si et seulement si il existe une solution valide de longueur ℓ

```

1  sol ← tableau à  $\ell$  cases // solution partielle
2  len ← 0 // longueur de la solution partielle

   // boucle "infinie" dont on devrait finir par sortir
3  tant que Vrai faire
4      x ← SUIVANT(sol[len - 1])
5      si len =  $\ell$  alors
6          si x est une erreur alors
7              renvoyer Faux
8          sinon
9              sol[len - 1] ← x
10             si VÉRIFICATEUR(sol) alors renvoyer Vrai
               // Sinon : tour de boucle suivant
11         sinon
               // Sinon : len <  $\ell$ 
12             si x est une erreur alors
13                 len ← len - 1
14             sinon
15                 sol[len - 1] ← x
16                 si non(REJET(sol, len)) alors
17                     sol[len] ← PREMIER(len)
18                     len ← len + 1
               // Sinon : tour de boucle suivant
```

Chapitre 8

RAISONNEMENTS INDUCTIFS

Notions	Commentaires
Ensemble ordonné, prédécesseur et successeur, prédécesseur et successeur immédiat. Élément minimal. Ordre produit, ordre lexicographique. Ordre bien fondé.	On fait le lien avec la notion d'accessibilité dans un graphe orienté acyclique. L'objectif n'est pas d'étudier la théorie abstraite des ensembles ordonnés mais de poser les définitions et la terminologie.
Ensemble inductif, défini comme le plus petit ensemble engendré par un système d'assertions et de règles d'inférence. Ordre induit. Preuve par induction structurelle.	On insiste sur les aspects pratiques : construction de structure de données et filtrage par motif. On présente la preuve par induction comme une généralisation de la preuve par récurrence.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

SOMMAIRE

0. Relations binaires (rappels de maths)	136
0. Relations d'équivalence	136
1. Relations d'ordres	138
2. Suite monotone	141
1. Ordres bien fondés et induction	142
0. Ordres bien fondés	142
1. Variants de boucle/appeal	143
2. Les inductions	144
<i>Ce qu'il faut savoir (p. 144). Formalisation (hors-programme) (p. 145).</i>	
2. Construire de nouveaux ordres	147
0. Transporter des ordres	147
1. Ordre produit	148
2. Ordre lexicographique	150
<i>Sur des éléments de même longueur (p. 150). Sur des éléments de longueurs distinctes (p. 152).</i>	
3. Clotures	153
<i>Définitions (p. 153). Engendrer un ordre (p. 156).</i>	
3. Induction structurelle	157
0. Construction inductive d'un ensemble : exemple illustratif	157
1. Induction structurelle	159
2. Un exercice final	160

Chapitre non-encore rédigé.

0 Relations binaires (rappels de maths)

Dans toute cette section, A , B et E sont des ensembles.

Définition 1 (Relation binaire).

Une **relation binaire** \mathcal{R} sur $A \times B$ est une partie de $A \times B$. Intuitivement, c'est la donnée d'un ensemble de paires (a, b) tels que a et b sont « reliés » (selon un certain critère qui dépend de la relation).

Pour tous $(a, b) \in A \times B$, on note $a\mathcal{R}b$ lorsque $(a, b) \in \mathcal{R}$, c'est à dire lorsque a et b sont « reliés » ; et on note $a\not\mathcal{R}b$ sinon.

Si $A = B$, on parle de relation binaire **homogène**.

Exemple.

- Une fonction $f : A \rightarrow B$ est une relation binaire : les paires sont les $(x, f(x))$. Autrement dit, la relation est $\{(x, f(x)) \text{ t.q. } x \in A\}$.
- $\leq_{\mathbb{R}}$ est une relation binaire homogène sur \mathbb{R} .
- $\leq_{\mathbb{N}}$ est une relation binaire homogène sur \mathbb{N} .
- \in est une relation binaire sur $E \in \mathcal{P}(E)$.
- \subseteq est une relation binaire homogène sur $\mathcal{P}(E) \times \mathcal{P}(E)$.
- $=$ est une relation binaire homogène sur E .
- \vdash , la relation telle que $P \vdash Q$ signifie « la propriété P prouve la propriété Q », est une relation binaire homogène sur les formules logiques.

Dorénavant, \mathcal{R} dénote une relation binaire homogène sur E .

0.0 Relations d'équivalence

Définition 2 (Relation d'équivalence).

On dit que \mathcal{R} est :

- **réflexive** lorsque pour tout $x \in E$, $x\mathcal{R}x$.
- **symétrique** lorsque pour tout $(x, y) \in E^2$, on a $x\mathcal{R}y$ si et seulement si $y\mathcal{R}x$.
- **transitive** lorsque pour tout $(x, y, z) \in E^3$, si $x\mathcal{R}y$ et $y\mathcal{R}z$ alors $x\mathcal{R}z$.

Une relation réflexive symétrique transitive est appelée **relation d'équivalence**.

Exemple.

- $=$ sur E
- Dans \mathbb{Z} , pour tout $p \in \mathbb{Z}^*$ fixé, la relation « avoir le même reste modulo p » est une relation d'équivalence.
- « renvoyer les mêmes sorties sur les mêmes entrées » est une relation d'équivalence sur les programmes.
- Dans un graphe non-orienté, l'accessibilité est une relation d'équivalence.

FIGURE 8.1 – Un graphe non-orienté et la relation d'accessibilité

Définition 3 (Classes d'équivalence).

Soit \mathcal{R} une relation d'équivalence sur E .

Pour tout $x \in E$, on définit la **classe d'équivalence** par \mathcal{R} de x , notée $[x]_{\mathcal{R}}$, comme :

$$[x]_{\mathcal{R}} = \{y \in E \text{ t.q. } x\mathcal{R}y\}$$

Pour tout $y \in [x]_{\mathcal{R}}$, on dit que y est un **représentant** de la classe d'équivalence $[x]_{\mathcal{R}}$.

Exemple.

- Pour la congruence modulo p , les classes d'équivalence sont les éléments de $\mathbb{Z}/p\mathbb{Z}$. Les représentants canoniquement utilisés sont $0, 1, \dots, p-1$:

FIGURE 8.2 – Classes d'équivalence de la congruence modulo p .

- Pour l'équivalence des sorties des programmes, une classe d'équivalence est un ensemble de programmes qui calculent la même chose. Mais ils peuvent faire ces calculs différemment, avec des raisonnements différents voire des complexités différentes. Par exemple, il existe de nombreux algorithmes de tri mais ils fonctionnent très différemment.
- Les classes d'équivalence de l'accessibilité dans un graphe non-orienté sont les composantes connexes (C.C.). Dans un graphe orienté, on peut dire que x et y sont *mutuellement accessibles* s'il y a un chemin de x à y et de y à x . C'est une relation d'équivalence, dont les classes d'équivalence sont les composantes fortement connexes (C.F.C.) :

FIGURE 8.3 – Un graphe orienté et ses C.F.C.

Théorème 4 (Partitionnement par les classes d'équivalence).

Soit \mathcal{R} une relation d'équivalence sur E . Alors :

- Tout élément appartient à sa classe d'équivalence. Formellement, pour tout $x \in E$ on a $x \in [x]_{\mathcal{R}}$.
- Deux éléments sont en relation si et seulement si ils ont la même classe d'équivalence. Formellement, pour tout $(x, y) \in E^2$, on a $x \mathcal{R} y$ si et seulement si $[x]_{\mathcal{R}} = [y]_{\mathcal{R}}$.
- Deux classes d'équivalence sont soit égales soit disjointes. Formellement, pour tout $(x, y) \in E^2$, on a $[x]_{\mathcal{R}} = [y]_{\mathcal{R}}$ ou bien $[x]_{\mathcal{R}} \cap [y]_{\mathcal{R}} = \emptyset$.

En particulier, les classes d'équivalences de \mathcal{R} partitionnent E .

Démonstration. • Par réflexivité de \mathcal{R} .

- Soient $(x, y) \in E^2$. Procédons par double implication :

\Leftarrow Si $[x]_{\mathcal{R}} = [y]_{\mathcal{R}}$, alors $x \in [x]_{\mathcal{R}} = [y]_{\mathcal{R}}$ donc $x \mathcal{R} y$.

\Rightarrow Si $x \mathcal{R} y$, montrons l'égalité des classes. Pour tout $z \in E$, on a :

$$z \in [x]_{\mathcal{R}} \iff z \mathcal{R} x \xrightarrow{\mathcal{R} \text{ transitive}} z \mathcal{R} y \iff z \in [y]_{\mathcal{R}}$$

- Soient $(x, y) \in E^2$. Supposons que $[x]_{\mathcal{R}} \cap [y]_{\mathcal{R}} \neq \emptyset$ et soit z dans cette intersection. Donc $z \mathcal{R} x$ et $z \mathcal{R} y$. Donc par transitivité $x \mathcal{R} y$, donc d'après le point précédent : $[x]_{\mathcal{R}} = [y]_{\mathcal{R}}$. □

Exemple. On peut partitionner les programmes en fonction de ce qu'ils calculent (de leurs entrées/sorties).

0.1 Relations d'ordres

\mathcal{R} désigne encore une relation homogène sur E .

Définition 5 (Relation d'ordre).

On dit que \mathcal{R} est :

- **antiréflexive** lorsque pour tout $x \in E$, $x \not\mathcal{R} x$.
- **antisymétrique** lorsque pour tout $(x, y) \in E^2$ avec $x \neq y$, on n'a pas à la fois $x \mathcal{R} y$ et $y \mathcal{R} x$.

Une relation réflexive, antisymétrique et transitive est appelée **relation d'ordre** (aussi appelée **relation d'ordre large**).

Une relation antiréflexive, antisymétrique et transitive est appelé **relation d'ordre stricte**.

Remarque. Une autre définition de l'antisymétrie est « pour tout $(x, y) \in E^2$, $x \mathcal{R} y$ et $y \mathcal{R} x$ implique $x = y$ ».

Exemple.

- $\leq_{\mathbb{Z}}$ est une relation d'ordre large sur \mathbb{Z} .
- $<_{\mathbb{Z}}$ est une relation d'ordre strict sur \mathbb{Z} .
- \subseteq est une relation d'ordre large sur $\mathcal{P}(E)$.
- \subsetneq est une relation d'ordre stricte sur $\mathcal{P}(E)$.
- \vdash n'est pas une relation d'ordre, puisque deux propriétés équivalentes se prouvent mutuellement.

Définition 6 (Ordre partiel, ordre total).

Soit \mathcal{R} une relation d'ordre sur E .

Soient $(x, y) \in E^2$. On dit que x et y sont comparables par \mathcal{R} lorsque $x\mathcal{R}y$ ou $y\mathcal{R}x$.

On dit que \mathcal{R} est **total** si tous éléments x et y sont comparables. Dans le cas contraire, on dit que l'ordre est **partiel**.

Remarque. En particulier, un ordre total est un ordre large puisque x et x doivent être comparables.

Exemple.

- $\leq_{\mathbb{Z}}$ est un ordre total.
- $<_{\mathbb{Z}}$ est un ordre partiel (mais « presque » total).
- \subseteq est un ordre partiel.
- $=$ sur E est un ordre partiel (c'est même le plus petit ordre large).
- La relation \emptyset (celle qui ne met *aucun* élément en relation) est un ordre strict partiel (c'est même le plus petit). Très très très partiel même !
- Sur \mathbb{C} , la relation $\{z_0, z_1 \text{ t.q. } \operatorname{Re}(z_0) \leq \operatorname{Re}(z_1) \text{ et } \operatorname{Im}(z_0) \leq \operatorname{Im}(z_1)\}$ est une relation d'ordre partiel. On verra bientôt qu'il s'agit de l'ordre produit sur \mathbb{C} . Visuellement, c'est l'ordre qu'un dit qu'un complexe est inférieur à un ordre s'il est « en dessous à gauche » dans le plan complexe :

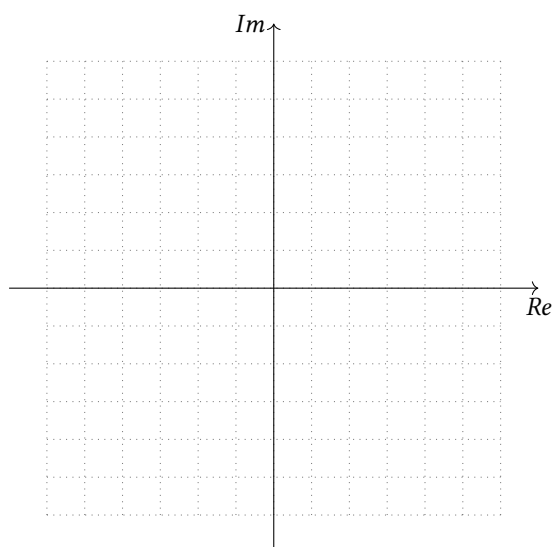


FIGURE 8.4 – Ordre produit sur \mathbb{C}

Définition 7 (Restriction et élargissement d'un ordre).

Si \mathcal{R} un ordre strict, posons $\mathcal{T} = \mathcal{R} \cup \{(x, x) \text{ t.q. } x \in E\}$ (c'est à dire \mathcal{R} à laquelle on ajoute la réflexivité). Alors \mathcal{T} est le plus petit ordre large (au sens de \subseteq) qui contient \mathcal{R} . On dit que c'est **l'ordre large associé à \mathcal{R}** .

Réciproquement, si \mathcal{R} est un ordre strict, posons $\mathcal{T} = \mathcal{R} \setminus \{(x, x) \text{ t.q. } x \in E\}$ (c'est à dire \mathcal{R} à laquelle on enlève la réflexivité). Alors \mathcal{T} est le plus grand ordre strict contenu dans \mathcal{R} . On dit que c'est **l'ordre strict associé à \mathcal{R}** .

Démonstration. C'est un exercice de maths, mais faisons-le (un peu) quand même.

- Montrons d'abord que \mathcal{T} est un ordre. Il est :
 - Réflexif (par définition).
 - Anti-symétrique car pour tous $(x, y) \in E^2$ avec $x \neq y$, si $x\mathcal{T}y$ alors $x\mathcal{R}y$ (car $\mathcal{T} \setminus \mathcal{R} = \{(x, x) \text{ t.q. } x \in E\}$). De même si $y\mathcal{R}x$. On conclut par antisymétrie de \mathcal{R} .

- Transitif car on a transitive pour trois x, y, z distincts par transitivité de \mathcal{R} , et les cas d'égalité se gèrent très bien.

Montrons maintenant qu'il s'agit du plus petit ordre large. Soit \mathcal{T}' un ordre large contenant \mathcal{R} . Comme il est large, \mathcal{T}' est réflexif et contient donc $\{(x, x) \text{ t.q. } x \in E\}$. Il contient également par définition \mathcal{R} , et contient donc tout \mathcal{T} . D'où le résultat.

- La construction réciproque est similaire.

□

Définition 8 (Prédécesseur, prédécesseur strict, prédécesseur immédiat).

Soit \mathcal{R} un ordre sur E , et $(x, y) \in E^2$. On dit que :

- x est un **prédécesseur** de y si $x\mathcal{R}y$.
Si de plus $x \neq y$, on dit que c'est un **prédécesseur strict** de y .
Si en plus de $x \neq y$ il n'existe aucun z tel que $x\mathcal{R}z$ et $z\mathcal{R}y$, on dit que x est un **prédécesseur immédiat** de y .
- y est un **successeur** de x si x précède y .
Si de plus $y \neq x$, on dit que y est un **successeur strict**.
Si de plus x précède immédiatement y , on dit que y est un **successeur immédiat** de x .

Ces notions se visualisent très bien sur un schéma. On représente tous les éléments, et on met une flèche de x vers y si x précède immédiatement y .

Exemple. Deux exemples de schémas d'ordres :

$$(a) \leq \text{ sur } \mathbb{N}$$

$$(b) \subseteq \text{ sur } \mathcal{P}(\llbracket 0; 2 \rrbracket)$$

FIGURE 8.5 – Deux ordres et les schémas associés

Remarque.

- \subseteq est un exemple d'ordre pour lequel il n'y a pas unicité d'un prédécesseur immédiat.
- Même si un élément admet des prédécesseurs, il peut n'admettre aucun prédécesseur immédiat : c'est par exemple le cas dans les ordres denses, comme $\leq_{\mathcal{R}}$ sur \mathbb{R} .

Définition 9 (Ensemble ordonné).

Si \mathcal{R} est un ordre sur E , on dit que (E, \mathcal{R}) est un **ensemble ordonné**. Si l'ordre est total, on dit qu'il s'agit d'un **ensemble totalement ordonné**.

Théorème 10.

Tout ensemble peut-être muni d'un ordre total.

Démonstration. Admis. □

Définition 11 (Élément minimal, minimum).

Un élément est dit minimal s'il n'a aucun prédecesseur strict.

Un élément minimal (resp. maximal) est un **minimum** (resp. **maximum**) s'il est de plus comparable à tous les éléments.

Proposition 12 (Unicité du minimum).

Le minimum (resp. maximum) de E , s'il existe, est unique.

Démonstration. Supposons que E admette m et m' deux minimas, et montrons que $m = m'$. Par définition d'un minimum, m et m' sont comparables. Par définition d'un minimum, on a $m \leq m'$ et $m' \leq m$ donc par anti-symétrie $m = m'$. □

Définition 13 (Minorant, inf).

- Soit $A \subseteq E$ et $m \in E$. On dit que m est un **minorant** (resp. **majorant**) de A si m est plus petit (resp. plus grand) que tous les éléments de A .
Formellement, $m \in E$ est un minorant de A si pour tout $a \in A$, on a $m \leq a$ (resp. $a \leq m$).
- S'il existe, on appelle $\inf(A)$ (resp. $\sup(A)$) le plus grand des minorants (resp. le plus petit des majorants).

Exemple. Dans $(\mathcal{P}(\llbracket 0; 2 \rrbracket), \subseteq)$, en posant $A = \{\{0\}; \{2\}\}$ on a $\sup(A) = \{0; 2\}$.

0.2 Suite monotone

Dans cette partie, (E, \leq_E) et (F, \leq_F) sont deux ensembles ordonnés (pas forcément totalement ordonnés). On note $<_E$ et $<_F$ les ordres stricts associés.

Définition 14 (Fonction croissante).

On dit qu'une fonction $f : E \rightarrow F$ est :

- **croissante** si pour tous $(x, y) \in E^2$ on a $x \leq_E y$ qui implique $f(x) \leq_F f(y)$. Si de plus $f(x) <_F f(y)$, on dit que la fonction est **strictement croissante**.
- **décroissante** si pour tous $(x, y) \in E^2$ on a $x \leq_E y$ qui implique $f(y) \leq_F f(x)$. Si de plus $f(y) <_F f(x)$, on dit que la fonction est **strictement décroissante**.

On dit qu'une fonction est monotone (resp. strictement monotone) si elle est croissante ou décroissante (resp. strictement croissante ou strictement décroissante).

Définition 15 (Suite croissante).

Soit (u_n) une suite d'éléments de E , finie ou infinie. On note I l'ensemble des indices de la suite, avec $I \subseteq \mathbb{N}$. On dit que :

- (u_n) est croissante (resp. strictement croissante) si $f : I \rightarrow E$ est croissante (resp. strictement croissante). Plus intuitivement, (u_n) est croissante si pour tout i , $u_i \leq_E u_{i+1}$.
- (u_n) est décroissante (resp. strictement décroissante) si $f : I \rightarrow E$ est décroissante (resp. strictement décroissante). Plus intuitivement, (u_n) est décroissante si pour tout i , $u_{i+1} \leq_E u_i$.

1 Ordres bien fondés et induction

Le but de la section est de montrer que la raison pour laquelle les preuves par récurrence fonctionnent est que la véracité du cas de base se « transmet » de proche en proche à tous les cas. Autrement dit, on peut faire des récurrences si et seulement si toute suite décroissante finit par tomber sur un cas de base.

(a) Récurrence sur \mathbb{N} (b) Induction sur $\mathcal{P}(\llbracket 0; 2 \rrbracket)$

FIGURE 8.6 – Illustration de la récurrence

1.0 Ordres bien fondés

Dans cette sous-section, (E, \leq) est un ensemble ordonné.

Définition 16 (Ordre bien fondé).

On dit que \leq est un **ordre bien fondé** sur E si toute partie non-vide de E admet un élément minimal.

Exemple. Visualisons cela dans quatre cas :

(a) \mathbb{N} muni de l'ordre usuel(b) \mathbb{N} avec l'ordre usuel restreint aux éléments pairs(c) \mathbb{N} muni de \geq n'est pas bien fondé(d) $\mathcal{P}(\llbracket 0; 2 \rrbracket)$ muni de \subseteq

FIGURE 8.7 – Ordres bien fondés ou non

Théorème 17.

(\mathbb{N}, \leq) est bien fondé.

Démonstration. Admis. Le prouver demande de savoir ce qu'est \mathbb{N} , c'est à dire comment \mathbb{N} est construit : c'est compliqué. \square

Théorème 18 (Caractérisation des ordres bien fondés).

\leq est bien fondé sur E si et seulement si il n'existe pas de suite infinie strictement décroissante dans (E, \leq) .

Démonstration. Procédons par double inclusion :

- \Rightarrow) Par contraposition : soit $(u_n)_{n \in \mathbb{N}}$ une suite infinie strictement décroissante, montrons que \leq n'est pas bien fondé. Pour cela, considérons la partie $A = \{u_n \text{ t.q. } n \in \mathbb{N}\}$: elle n'admet pas d'élément minimal puisque pour tout $u_i \in A$, on a $u_{i+1} < u_i$.
- \Leftarrow) Supposons qu'il n'y a pas de suite infinie strictement décroissante et procédons par l'absurde : soit $A \subseteq E$ sans élément minimal et $a_0 \in A$. Comme A est sans élément minimal, il existe $a_1 < a_0$. De même, il existe $a_2 < a_1$ et ainsi de suite : par axiome du choix dépendant¹, on obtient une suite (a_n) strictement décroissante. \square

Remarque. C'est généralement le critère le plus simple à manipuler pour montrer qu'un ordre est bien fondé ou non, comme par exemple pour les 4 exemples précédents.

Intuitivement, les ordres bien fondés sont importants car ce sont les ordres qui ont des « cas de bases » (les fameux éléments minimaux demandés dans la définition). Or, les cas de base sont capitaux pour faire des récurrences !

1.1 Variants de boucle/appel

Une première application de cette notion d'ordre bien fondé est la notion de variant !

Définition 19 (Variants (S2)).

Un variant de boucle (resp. d'appels récursifs) est une quantité qui décroît strictement d'une itération à l'autre (resp. d'un appel à l'autre) selon un ordre bien fondé.

Proposition 20.

Une boucle (resp. une suite d'appels récursifs) qui admet un variant itère un nombre fini de fois (resp. contient un nombre fini d'appels).

Démonstration. Si elle itérerait un nombre infini de fois, les valeurs successives du variant donnerait une suite infinie strictement décroissante selon un ordre bien fondé : absurde. \square

Exemple.

- Une suite d'entiers minoré strictement décroissant donne un variant.
- Une suite d'entiers majoré strictement croissant donne un variant.
- Une suite d'ensembles finis décroissante pour \subseteq , c'est à dire tels que $E_{i+1} \subsetneq E_i$, donne un variant² ! Autrement dit, il n'y a pas besoin de dire que le cardinal de l'ensemble est un variant : l'ensemble lui-même est un variant !
- Plus d'exemples avec l'ordre produit et lexicographique !

1. Et non par récurrence. C'est subtil, on s'en fiche à notre niveau, mais c'est important : on ne fait pas encore de récurrence (on est en train de la construire!).

2. Cf SLOWSELECT vu en DS, ou le retour sur trace de SUBSETSUM.

1.2 Les inductions

1.2.0 Ce qu'il faut savoir

Définition 21 (Preuve par induction).

Soit I une propriété portant sur des éléments de E . Pour prouver I par induction, on doit prouver :

- Initialisation : montrer que tous les éléments minimaux pour \leq vérifient I .
- Hérédité : soit x non-minimal. On doit montrer que si tous les prédécesseurs stricts de x vérifient I , alors x le vérifie aussi.

Sur (\mathbb{N}, \leq) , l'induction s'appelle récurrence forte.

Théorème 22 (Induction ssi bien fondé).

Une preuve par induction (basée sur l'ordre \leq) prouve qu'une propriété est vraie pour tous les éléments de E si et seulement si l'ordre \leq est bien fondé.

Exemple. Un mobile de Calder est une succession de « barre » qui chacune portent soit un objet, soit une autre barre :

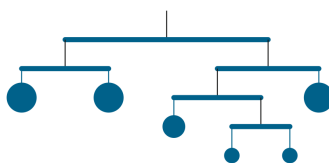


FIGURE 8.8 – Un mobile de Calder (src : Balabonski, Conchon Filliâtre, Nguyen, Sartre)

Autrement dit, un mobile de Calder est :

- Soit un objet
- Soit une barre reliant deux mobiles de Calder.

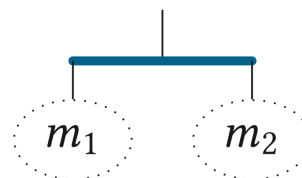


FIGURE 8.9 – Construction inductive des mobiles de Calder

On veut montrer que dans tout mobile de Calder m , le nombre $obj(m)$ d'objets est supérieur de 1 au nombre $bar(m)$ de barres. On admet que l'ordre « $m_0 \preccurlyeq m_1$ lorsque m_0 apparaît dans m_1 » est un ordre bien fondé (on verra plus tard qu'il s'agit de l'ordre structurel). Procédons par induction :

- Initialisation : les cas de base sont les mobiles qui ne contiennent pas d'autres mobiles, autrement dit les mobiles réduits à des objets. Pour un tel mobile, il y a 0 barre et 1 objet : la propriété est bien initialisée.
- Hérédité : soit m un mobile non-réduit à un objet et supposons la propriété vraie pour tout autre mobile $m' \preccurlyeq m$. Comme m n'est pas réduit à un objet, m est une barre reliant deux mobiles m_1 et m_2 .

Donc $obj(m) = obj(m_1) + obj(m_2)$; et $bar(m) = bar(m_1) + bar(m_2) + 1$ (le +1 correspond à la barre reliant m_1 et m_2). En appliquant l'hypothèse d'induction à $obj(m_1)$ et $obj(m_2)$ on obtient :

$$\begin{aligned}
obj(m) &= obj(m_1) + obj(m_2) \\
&= bar(m_1) + 1 + bar(m_2) + 1 \\
&= bar(m) + 1
\end{aligned}$$

D'où l'hérédité.

- **Conclusion** : on a prouvé par induction que pour tout mobile de Calder m , le nombre $obj(m)$ d'objets est supérieur de 1 au nombre $bar(m)$ de barres.

Remarque. Notez que ce n'est pas une récurrence : on n'a pas travaillé sur des entiers ! L'hérédité ne consistait pas à prouver « $H_{n-1} \Rightarrow H_n$ ». En fait, la notion de « passer d'un entier au suivant » ne marche pas sur les mobiles : un mobile a deux prédecesseurs immédiats et non un seul.

1.2.1 Formalisation (hors-programme)

Savoir ce qu'est une preuve par induction n'est pas simple. Il existe en fait une façon ensembliste de formaliser l'induction. Nous allons la détailler sur un exemple ; c'est assez abstrait.

Exemple. Prouvons que pour tout $n \in \mathbb{N}$: $\sum_{k=0}^n k = \frac{n(n+1)}{2}$

- **V0 : version basique**

- **Initialisation** : Montrons que $P(0)$ est vraie. Si $n = 0$, on a $\sum_{k=0}^0 k = 0$ et $\frac{n(n+1)}{2} = 0$. Donc $I(0)$ est vraie.
- **Hérédité** : Supposons la propriété vraie pour un rang $n \in \mathbb{N}$, et montrons la vraie au rang $n + 1$. On a :

$$\begin{aligned}
\sum_{k=0}^{n+1} k &= (n+1) + \sum_{k=0}^n k \\
&= n+1 + \frac{n(n+1)}{2} \quad \text{d'après } I(n) \\
&= \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}
\end{aligned}$$

D'où l'hérédité.

- **Conclusion** : On a donc montré par récurrence que pour tout $n \in \mathbb{N}$, $I(n)$ est vraie.

- **V1 : comme V0, mais par récurrence forte**

Comme V0, mais l'on remplace la ligne d'introduction de l'hérédité par : « Soit $n \in \mathbb{N}^*$ tel que pour tout $m < n$, $I(m)$ soit vrai ; et montrons la vraie au rang $n + 1$. »

- **V2 : on généralise et fusionne les notions d'initialisation et d'hérédité en une seule notion**

Montrons par récurrence sur $n \in \mathbb{N}$ que $I(n)$: « $\sum_{k=0}^n k = \frac{n(n+1)}{2}$ » est vraie. Pour cela, montrons que pour tout $n \in \mathbb{N}$ la propriété suivante est vraie :

$$h_n : [\forall m < n, I(m)] \implies I(n)$$

Soit $n \in \mathbb{N}$. On procède par disjonction de cas.

- Si n est minimal dans (\mathbb{N}, \leq) , alors $n = 0$. Le terme gauche de l'implication est donc trivialement vrai³. Il faut alors montrer $I(0)$. Et on refait comme l'initialisation de V0/V1.

3. Un « Pour tout » qui quantifie sur l'ensemble vide est automatiquement vrai. De même, un « Il Existe » qui quantifie sur l'ensemble vide est automatiquement Faux.

- Sinon, $n \neq 0$ (car 0 est le seul élément minimal de (\mathbb{N}, \leq)). Pour montrer l'implication, on suppose que le terme gauche est vrai : $\forall m < n, I(m)$. Il faut alors montrer $I(n)$. Et on refait comme l'hérédité de V1.

Conclusion : Par principe de récurrence, on en déduit que pour tout $n \in \mathbb{N}$ $I(n)$ est vrai.

• **V3 : on se ramène à des ensembles**

Notons pour $n \in \mathbb{N}$, $I(n) : \ll \sum_{k=0}^n k = \frac{n(n+1)}{2} \gg$. Notons $P = \{n \in \mathbb{N} \text{ t.q. } I(n)\}$. On veut montrer que $P = \mathbb{N}$. Pour cela, montrons que pour tout $n \in \mathbb{N}$ la propriété suivante est vraie :

$$\text{Hered}_n : [\forall m < n, m \in P] \implies n \in P$$

Et on continue comme V2, jusqu'à la conclusion exclue.

Conclusion : Par principe de récurrence, on en déduit que $P = \mathbb{N}$.

Mais qu'est-ce donc finalement que le principe de récurrence ? De cette V3, on en déduit⁴ l'expression ensembliste suivante du principe de récurrence :

Définition 23 (Principe d'induction).

Soit (E, \leq) ensemble ordonné. On appelle principe d'induction sur E la propriété suivante (pas forcément vraie !!) :

$$\begin{aligned} &\text{Pour toute partie } P \text{ de } E, \text{ on a } (\forall x \in E, \text{Hered}_x) \implies P = E \\ &\text{où } \text{Hered}_x : [\forall y < x, y \in P] \implies x \in P \end{aligned}$$

Si cette propriété est vraie, on dit que (E, \leq) vérifie le principe d'induction.

Ce que dit cette propriété, c'est que si l'appartenance à un ensemble s'hérite de ses (éventuels) prédécesseurs stricts, alors cet ensemble est l'espace tout entier. Le « cas de base » de la récurrence correspond alors au cas où il n'y a pas de prédécesseur strict, qui appelle généralement à une preuve distincte (comme en V2 ci-dessus).

- Plus généralement, si $\Phi(x)$ est la propriété à prouver vraie pour chacun des x de E , alors prouver utiliser le principe d'induction revient à poser $P = \{x \in E \mid \Phi(x)\}$ et utiliser le principe pour prouver $P = E$.
- Vous noterez que cette façon de faire des récurrences correspond à des récurrences fortes. En fait, on peut prouver que tout résultat prouvable par récurrence forte l'est pas récurrence simple, et réciproquement (quitte à transformer un peu le résultat). Ainsi, récurrence forte et simple (et double, et triple, etc) ont exactement le même pouvoir de preuve.
En mathématiques, il peut être attendu de vous que vous utilisiez la « récurrence la plus simple ». En informatique, je ne vous en voudrai jamais de faire une récurrence forte au lieu d'une simple, surtout lorsque vous faites une induction (récurrence hors de \mathbb{N}).
En résumé : faites des inductions fortes en informatique, c'est plus simple (vous avez accès à tous les antécédants).
- Le principe d'induction n'est pas toujours vrai ! En fait, pour bien fonctionner il a besoin qu'il existe des « cas de base » : voilà le lien avec les ordres bien fondés.⁵

Théorème 24.

(E, \leq) vérifie le principe d'induction si et seulement si \leq est un ordre bien fondé sur E .

Démonstration. DM/TD étoilé. Procédez par double implication. Le sens direct est le plus simple puisque vous y avez accès à des récurrences ! \square

4. Pas au sens de « on prouve », mais au sens de « On comprend que le principe de récurrence est ».

5. On peut formaliser une notion de récurrence sans cas de base, qui est rigoureuse et correcte : on appelle cela de la co-induction. C'est très largement hors de notre portée.

Remarque. Le terme de « récurrence » est souvent réservé à \mathbb{N} et le terme « induction » couvre les « récurrences hors de \mathbb{N} ». C'est le choix que j'ai fait dans ce cours car j'ai vu des concours faire ce choix ; mais à titre personnel je ne vois pas l'intérêt de distinguer les deux notions.⁶ On parle aussi de récurrence noethérienne pour parler de récurrence sur un ordre bien fondé, et d'ordre nothérien pour l'ordre.

2 Construire de nouveaux ordres

Nous avons vu comment faire des inductions (récurrences hors de \mathbb{N}). Mais comme nous venons de le dire, il faut pour cela disposer de relations d'ordre bien fondées pour « descendre » jusqu'à un cas de base : voyons comment en obtenir.

Dans toute cette section, (E, \leq_E) et (F, \leq_F) sont deux ensembles ordonnés (et les ordres partiels associés sont $<_E$ et $<_F$).

2.0 Transporter des ordres

Dans cette sous-section, on veut « transporter » un ordre via une restriction ou une fonction.

Proposition 25 (Restriction d'un ordre bien fondé).

Si \leq_E est bien fondé, et si $A \subseteq E$, alors la restriction de \leq_E à A est un ordre bien fondé sur A .

Démonstration. Une suite strictement décroissante de (A, \leq_A) est une suite strictement décroissante de (E, \leq_E) . \square

Exemple. $(\llbracket k; +\infty \rrbracket, \leq)$ est bien fondé car il s'agit de la restriction à $\llbracket k; +\infty \rrbracket$ de l'ordre usuel bien fondé sur \mathbb{N} .

Proposition 26.

Si \leq_E est bien fondé, et si $f : E \rightarrow F$ est injective, alors on peut munir $f(E)$ d'un ordre bien fondé : il suffit de « transporter » les comparaisons de E vers $f(E)$.

Démonstration. Pour tout $y \in f(E)$, notons $f^{-1}(y)$ son unique antécédant. On définit \leq ordre sur F par $y \leq y'$ lorsque $f^{-1}(y) \leq_E f^{-1}(y')$. Autrement dit, on « transporte » \leq_E à l'aide de l'injection.

Avec cette construction, toute suite infinie strictement décroissante dans $f(E)$ correspond à une suite infinie strictement décroissante dans E . D'où le résultat. \square

Exemple. $(\llbracket -\infty; k \rrbracket, \geq)$ est bien fondé car il s'agit de l'image de l'exemple précédent par la fonction injective $f : x \mapsto x$.

Remarque. En pratique, on ne devrait pas attendre de vous que vous puissiez prouver aussi rigoureusement qu'un ordre est bien fondé. Une justification convainquante de pourquoi il n'existe pas de suite infinie strictement décroissante suffira. Toutefois, les règles de l'on construction que l'on donne ici sont bonnes à avoir en tête puisqu'elles permettent de trouver en un coup d'oeil un ordre bien fondé.

Proposition 27.

Tout ensemble dénombrable (c'est à dire en bijection avec \mathbb{N}) peut être muni d'un ordre bien fondé.

Démonstration. La définition de dénombrable donne une bijection donc une injection depuis (\mathbb{N}, \leq) . \square

⁶ Je soupçonne, sans preuve, que la co-existence de « récurrence » et « induction » en français provient du fait que « induction » est le terme anglais pour « récurrence », et que les inductions bien fondées ont gagné en usage ce dernier demi-siècle, où les articles de recherche sont écrits... en anglais.

Remarque. Cependant, l'ordre bien fondé correspondant n'est pas toujours pratique. Lorsque vous verrez en maths la tête de la bijection entre \mathbb{Z} et \mathbb{N} ou entre \mathbb{Q} et \mathbb{N} , vous comprendrez que cet ordre bien fondé n'est pas agréable...

2.1 Ordre produit

Dans cette sous-section, on veut munir $E \times F$ d'un ordre.

Définition 28 (Ordre produit).

Soient (e_0, f_0) et (e_1, f_1) deux paires de $E \times F$. On pose $(e_0, f_0) \leq_{\Pi} (e_1, f_1)$ lorsque $e_0 \leq_E e_1$ et $f_0 \leq_F f_1$. Cette relation \leq_{Π} est un ordre sur $E \times F$, appelé **ordre produit**.

Démonstration. Pour prouver que \leq_{Π} est un ordre, il faut le montrer réflexif, antisymétrique, transitif. Les trois points s'obtiennent immédiatement via le fait que \leq_E et \leq_F ont les bonnes propriétés. \square

Exemple. $\mathbb{C} = \mathbb{R} \times i\mathbb{R}$, on peut donc munir \mathbb{C} de l'ordre produit (cf partie précédente).

L'ordre produit est pratique puisqu'il permet de faire des inductions !

Proposition 29 (Un produit d'ordres bien fondés est bien fondé).

Si (E, \leq_E) et (F, \leq_F) sont bien fondés, alors $(E \times F, \leq_{\Pi})$ est bien fondé. Autrement dit, le produit d'ordres bien fondés est bien fondé.

Démonstration. Il est ici plus confortable de revenir à la définition d'ordre bien fondé (hélas)⁷. Soit $X \subseteq E \times F$ non-vide.

Notons A l'ensemble des premières coordonnées de X : $A = \{e \in E \mid \exists f \in F, (e, f) \in X\}$. Alors A admet un élément minimal puisque c'est une partie non-vide de E et que \leq_E est bien fondé. Nommons le a .

Notons $B(a)$ l'ensemble des secondes coordonnées pouvant être associées à A dans X : $B(a) = \{f \in F \mid (a, f) \in X\}$. De même, $B(a)$ admet un élément minimal b .

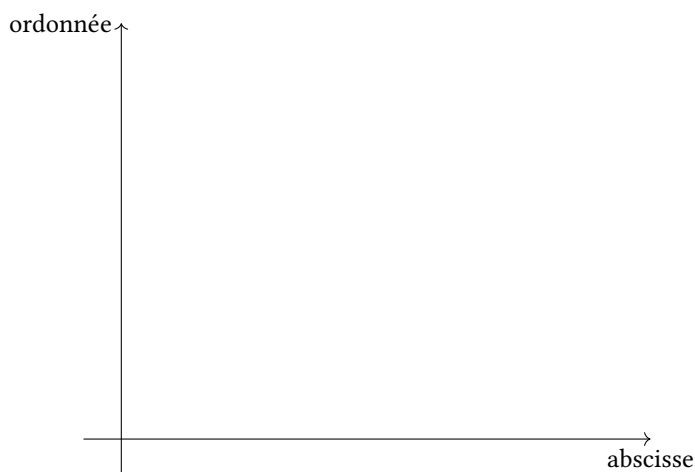


FIGURE 8.10 – Preuve avec l'ordre produit Abscisses \times Ordonnées.

Montrons que (a, b) est minimal dans X . Soit (a', b') un prédécesseur de (a, b) par l'ordre produit, et montrons qu'il s'agit en fait de (a, b) .

7. Si vous trouvez une version qui utilise la caractérisation et n'est pas un chaos dénotationnel, je suis fort curieux.

$(a', b') \leq_{\Pi} (a, b)$ implique que $a' \leq_E a$, $b' \leq_F b$ et $(a, b) \neq (a', b')$. Par minimalité de a , comme $a' \in A$, $a' \leq_E a$ implique $a = a'$. Mais alors $b' \in B(a)$ et le même raisonnement donne $b = b'$. Donc $(a, b) = (a', b')$.

Donc X admet bien un élément minimal, d'où le résultat. \square

Exemple.

- $(\mathbb{N}^2, \leq_{\Pi})$ est bien fondé. En particulier, une paire d'entier (x, y) où « parfois x décroît (et y reste constant), parfois y décroît (et x reste constant) » est un variant de boucle!
- Pour tout $\text{len} \in \mathbb{N}$ fixé, $([0; \text{len}], \leq)$ et $([0; \text{len}], \geq)$ sont bien fondés donc l'ensemble produit est bien fondé. En particulier, cela donne une autre façon de prouver la terminaison de la dichotomie!

```

9  /** Renvoie true ssi x est présent dans une des len cases de arr
10  * Pré-conditions : arr doit être trié par ordre croissant.
11  */
12  bool mem_opt(int x, int arr[], int len) {
13      int lo = 0;
14      int hi = len-1;
15
16      // Invariant : si x est présent, il est dans arr[lo:hi[
17      while (lo < hi) {
18          int mid = (lo+hi)/2;
19          if (arr[mid] == x) { return true; }
20          else if (arr[mid] < x) { hi = mid; }
21          else { lo = mid+1; }
22      }
23
24      return false;
25  }
```

Pour prouver la terminaison, on peut prouver que (lo, hi) est un variant! C'est une preuve plus simple que celle que nous avons fait au S1 (où on a montré que $hi - lo$ est un variant, ce qui était un peu plus dur *mais* avait l'avantage de bien expliquer l'algorithme).

Proposition 30 (Non totalité de l'ordre produit).

Il suffit que $|E| > 1$ et que $|F| > 1$ pour que l'ordre produit \leq_{Π} ne soit pas total.

Démonstration. Supposons que $|E| > 1$ et que $|F| > 1$. Remarquons tout d'abord que s'il existe e_0 et e_1 non comparables dans E , alors pour tout $f \in F$ les paires (e_0, f) et (e_1, f) ne sont pas comparables par \leq_{Π} dans $E \times F$ et donc \leq_{Π} n'est pas total. De même dans F .

Supposons désormais E et F totalement ordonnés, et soient $e_0 <_E e_1$ deux éléments distincts de E et $f <_F f_1$ deux éléments distincts de F . Alors (e_0, f_1) et (e_1, f_0) ne sont pas comparables par \leq_{Π} . En effet :

- $e_0 <_E e_1$ implique $(e_1, f_0) \not\leq_{\Pi} (e_0, f_1)$
- $f_0 <_F f_1$ implique $(e_0, f_1) \not\leq_{\Pi} (e_1, f_0)$.

\square

En d'autres termes, l'ordre produit n'est presque jamais total. Si un ordre total est requis, il ne faut pas prendre un ordre partiel.

Exemple. Pour trier un tableau de points⁸ de \mathbb{Z}^2 , il faut un ordre total : l'ordre produit ne peut pas être utilisé. : l'ordre produit n'est pas suffisant pour cela.

8. Cf exercice sur les milieux dans un nuage de points, ou sur les 2 plus proches points dans un nuage.

Définition 31 (Ordre produit sur grands produits).

Soient E_1, \dots, E_n n ensembles munis d'ordres \leq_1, \dots, \leq_n . On définit l'ordre produit \leq_Π sur $E_1 \times \dots \times E_n$ comme $(e_1, \dots, e_n) \leq_\Pi (e'_1, \dots, e'_n)$ si et seulement si pour tout $i \in \llbracket 1; n \rrbracket$, $e_i \leq_0 e'_i$

Démonstration. Comme pour l'ordre produit sur $E \times F$. □

Remarque. C'est en fait l'ordre produit obtenu par récurrence en remarquant que $E_1 \times \dots \times E_n = (E_1 \times \dots \times E_{n-1}) \times E_n$

2.2 Ordre lexicographique

Dans cette sous-section, on veut munir $E \times F$ d'un ordre qui palie à la non-totalité de l'ordre produit : c'est l'ordre lexicographique. Le nom provient de son origine : c'est l'ordre utilisé pour comparer des mots entre eux. On compare la première lettre, puis la seconde si les premières sont égales, etc.

2.2.0 Sur des éléments de même longueur

Définition 32 (Ordre lexicographique).

Soient (e_0, f_0) et (e_1, f_1) paires de $E \times F$.

On définit la relation d'ordre \leq_{lex} sur $E \times F$ en posant $(e_0, f_0) \leq_{lex} (e_1, f_1)$ lorsque $e_0 <_E e_1$ ou que $e_0 = e_1$ et $f_0 \leq_F f_1$.

On appelle \leq_{lex} **ordre lexicographique**, ou produit lexicographique de \leq_E et \leq_F .

Autrement dit, on compare la première coordonnée pour décider qui est le plus petit. Si elle fait un ex-aequo, on compare la seconde.

Démonstration. Il faut prouver que \leq_{lex} est un ordre. Soient (e_0, f_0) , (e_1, f_1) et (e_2, f_2) paires de $E \times F$. On a bien :

- Réflexivité : $e_0 \leq_E e_0$ donc $(e_0, f_0) \leq_{lex} (e_0, f_0)$.
- Anti-symétrie : supposons $(e_0, f_0) \leq_{lex} (e_1, f_1)$ et $(e_1, f_1) \leq_{lex} (e_0, f_0)$.
Distinguons deux cas. Si $e_0 = e_1$, alors les deux inégalités de \leq_{lex} donnent respectivement $f_0 \leq_F f_1$ et $f_1 \leq_F f_0$. Donc $f_0 = f_1$, donc $(e_0, f_0) = (e_1, f_1)$.
Sinon, les deux inégalités donnent $e_0 <_E e_1$ et $e_1 <_E e_0$. Donc $e_0 = e_1$, contradiction avec le « Sinon ».
D'où l'antisymétrie.
- Un peu laborieux à cause des disjonctions de cas sur les comparaisons des premières coordonnées. Rien de très profond.

□

Exemple.

- C'est le tri des mots du dictionnaires (ici les deux mots ont deux lettres). Par exemple, $ba \leq bb$ et $ba \leq ab$.
- C'est l'ordre utilisé en DS pour trier un nuage de points !

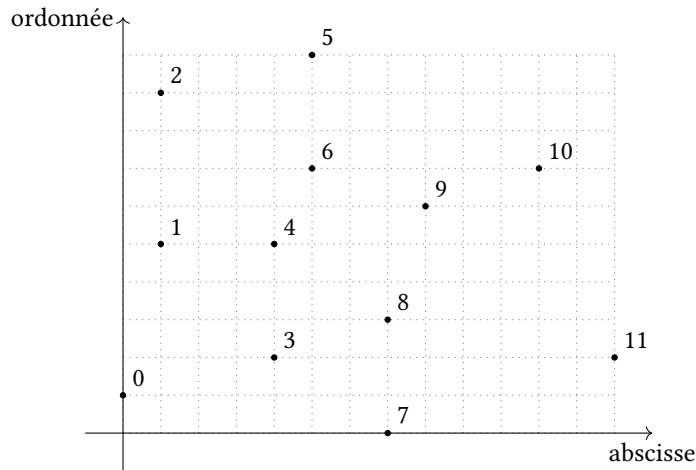


FIGURE 8.11 – Deux suites croissantes dans le plan muni de l'ordre lexicographique « Abscisse \times Ordonnée ».

Proposition 33 (Un produit lexicographique d'ordres bien fondés est bien fondé).

Si (E, \leq_E) et (F, \leq_F) sont bien fondés, alors $(E \times F, \leq_{lex})$ est bien fondé.

Autrement dit, le produit lexicographique d'ordres bien fondés est bien fondé.

Démonstration. On procède de manière similaire à la preuve précédente. Soit $X \subseteq E \times F$ non-vide, montrons que X admet un élément minimal.

On construit $A, a, B(a)$ et b comme dans la preuve précédente.

Montrons que (a, b) est minimal dans X . Soit (a', b') un prédécesseur de (a, b) par l'ordre lexicographique, et montrons qu'il s'agit en fait de (a, b) .

Tout d'abord, par minimalité de a , il est impossible que $a' <_E a$. Donc $a = a'$, donc $b' \leq_F b$. Mais par minimalité de b , on a donc $b' = b$.

D'où le résultat. \square

Notons que l'exemple d'éléments non comparables dans \mathbb{C} par l'ordre produit était $1 + 0i$ et $0 + i$. Ces deux éléments sont tout à fait comparables par l'ordre lexicographique : $i \leq 1$ (comparaison de la première coordonnée).

Proposition 34.

Si \leq_E et \leq_F sont totaux, alors \leq_{lex} est total.

Démonstration. Supposons \leq_E et \leq_F totaux. Soient (e_0, f_0) et (e_1, f_1) paires de $E \times F$.

Par hypothèse, e_0 et f_0 sont comparables puisque \leq_E est total. On distingue deux cas :

- Si $e_0 = e_1$, alors soit $f_0 \leq_F f_1$ et donc $(e_0, f_0) \leq_{lex} (e_1, f_1)$, soit (par totalité de \leq_F) $f_1 \leq_F f_0$ et donc $(e_1, f_1) \leq_{lex} (e_0, f_0)$. Ainsi, (e_0, f_0) et (e_1, f_1) sont comparables.
- Sinon, on a soit $e_0 <_E e_1$ et donc $(e_0, f_0) \leq_{lex} (e_1, f_1)$, soit $e_1 \leq_E e_0$ et donc $(e_1, f_1) \leq_{lex} (e_0, f_0)$. Ainsi, (e_0, f_0) et (e_1, f_1) sont comparables.

\square

Ainsi, l'ordre lexicographique permet d'avoir un ordre total et bien fondé !

Exemple. La fonction d'Ackermann^{9 10 11} est définie par :

9. En réalité de Rozsa Péter, celle d'Ackermann avait trois variables.

10. C'est une fonction « monstrueuse » faite pour croître très très vite et casser certains espoirs en informatique fondamentale. Ne lui cherchez pas de sens profond.

11. Vous la croiserez peut-être en MPI, où l'on peut mentionner sa réciproque qui croît très très lentement (le logarithme est à supraluminique à côté). Mais je doute que vous fassiez plus qu'en entendre parler.

$$A : (n, x) \in \mathbb{N} \times \mathbb{N}^* \mapsto \begin{cases} x + 1 & \text{si } n = 0 \\ A(n - 1, 1) & \text{si } x = 0 \\ A(n - 1, A(n, x - 1)) & \text{sinon} \end{cases}$$

Prouver par induction dans $(\mathbb{N} \times \mathbb{N}^*, \leq_{\text{lex}})$ que pour tous n et x on a $A(n, x) > x$.

- **Initialisation** : si $n = 0$, $A(n, x) = x + 1 > x$.
- **Hérédité** : Soit (n, x) avec $n \neq 0$ tel que la propriété soit vraie pour tout $(n', x') <_{\text{lex}} (n, x)$, montrons-la vraie pour (n, x) . Distinguons deux cas :
 - si $x = 0$, alors $A(n, 0) = A(n - 1, 1) > 1 > 0$.
 - sinon :

$$\begin{aligned} A(n, x) &= A(n - 1, A(n, x - 1)) \\ &> A(n, x - 1) \\ &\geq A(n, x - 1) + 1 \\ &> x - 1 + 1 \\ &> x \end{aligned}$$

Ainsi, dans les deux cas la propriété est héréditaire.

- **Conclusion** : Pour tout (n, x) , $A(n, x) > x$.

Définition 35 (Ordre lexicographique sur grands produits).

Soient E_1, \dots, E_n n ensembles munis d'ordres \leq_1, \dots, \leq_n . On définit l'ordre lexicographique \leq_{lex} sur $E_1 \times \dots \times E_n$ comme $(e_1, \dots, e_n) \leq_{\text{lex}} (e'_1, \dots, e'_n)$ si et seulement si il existe $i \leq n$ tel que les i premières coordonnées soient égales et que la $i + 1$ ème (si elle existe) vérifie $e_{i+1} <_{i+1} e'_{i+1}$.

Formellement, $e = (e_1, \dots, e_n) \leq_{\text{lex}} (e'_1, \dots, e'_n) e'$ si et seulement si $e = e'$ ou il existe $i \in \llbracket 1; n \rrbracket$ tel que pour tout $1 \leq j \leq i$, $e_j = e'_j$ et $e_{i+1} <_{i+1} e'_{i+1}$.

Démonstration. Comme pour l'ordre lexicographique sur $E \times F$. □

Remarque. C'est en fait l'ordre lexicographique obtenu par récurrence en remarquant que $E_1 \times \dots \times E_n = (E_1 \times \dots \times E_{n-1} \times) E_n$

2.2.1 Sur des éléments de longueurs distinctes

Quand on compare des mots dans des livres, on veut aller plus loin : on veut comparer des mots de longueurs différentes. En pratique, on dit que « si l'on a fini de parcourir un mot et pas l'autre, le plus court est plus petit » ; donc *baba* est plus petit que *babaurhum*.

Cela se code très bien : voici par exemple un code OCaml qui compare lexicographiquement deux listes (qui peuvent être de longueur différents) :

```

4  (** Renvoie -1 si e < e', 0 si = et 1 sinon *)
5  let rec cmp_lex e e' =
6    match (e, e') with
7    | [], [] -> 0
8    | [], _ -> -1
9    | _, [] -> 1
10   | t::q, t'::q' ->
11     if t < t' then -1
12     else if t > t' then 1
13     else cmp_lex q q'

```

 Lex.ml

Cependant, autoriser les uplets à être de taille différente et aussi grande que l'on veut brise la bonne fondaison :

Proposition 36.

Un ordre lexicographique sur un ensemble d'uplets de longueur variable et non bornée peut ne pas être bien fondé.

Démonstration. Posons $\Sigma = \{a, b\}$ ordonné par $a < b$. et Σ^+ l'ensemble des n -uplets sur Σ pour tout $n \in \mathbb{N}^*$:

$$\Sigma^+ = \{(a); (b); (a, a); (a, b); (b, a); (b, b); (a, a, a); (a, a, b); \dots\}$$

Pour simplifier, je note ab au lieu de (a, b) .

La suite suivante est infinie strictement décroissante :

$$b <_{\text{lex}} ab <_{\text{lex}} aab <_{\text{lex}} aaab <_{\text{lex}} \dots$$

□

2.3 Clotures

2.3.0 Définitions

Dans cette sous-section, on se donne E est un ensemble et \mathcal{R} est une relation binaire homogène sur E . On se demande si on peut déduire un ordre de \mathcal{R} .

Dans tout cette sous-section, lorsque l'on compare des relations (« \mathcal{R} est plus petite/grande que \mathcal{T} ») la comparaison utilisée est l'inclusion.

Définition 37 (sur-relation).

On appelle **sur-relation** de \mathcal{R} une relation \mathcal{T} sur E telle que $\mathcal{R} \subseteq \mathcal{T}$.

Réciproquement, on dit dans ce cas que \mathcal{R} est une **sous-relation** de \mathcal{T} .

Remarque. Nous avons implicitement utilisé cette notion lorsque l'on a donné le lien entre ordre large et strict.

Définition 38 (Cloture).

La **cloture réflexive** de \mathcal{R} , notée \mathcal{R}^0 , est la plus petite sur-relation réflexive de \mathcal{R} .

On définit de même la **cloture symétrique** \mathcal{R}^S et la **cloture transitive** \mathcal{R}^+ comme étant les plus petites sur-relations de \mathcal{R} symétrique et transitive respectivement.

On note \mathcal{R}^* la **cloture réflexive transitive**, i.e. la plus petite sur-relation de \mathcal{R}^0 qui est à la fois réflexive et transitive.

Démonstration. Il faudrait démontrer que ces notions existent bel et bien, c'est à dire que la plus petite sur-relation réflexive/symétrique/transitive existe.

Il existe des sur-relations réflexive, symétrique ou transitive car la relation pleine (qui met tous les éléments en relation) est réflexive, symétrique et transitive. L'existence de la plus petite est pour l'instant admis. □

Remarque.

- On appelle aussi une cloture une *fermeture* (réflexive, transitive, symétrique, etc).
- La cloture antisymétrique n'est pas définissable. En effet, si une relation \mathcal{R} n'est pas antisymétrique, elle contient un couple (x, y) tel que $x\mathcal{R}y$ et $y\mathcal{R}x$, donc toutes ses sur-relations contiennent ce couple (x, y) et aucune d'entre elles ne peut être antisymétrique. Nous verrons plus tard une condition plus générale sur l'antisymétrie.

FIGURE 8.12 – Graphe de la congruence modulo p dans \mathbb{Z}

- La clôture réflexive transitive de \mathcal{R} est par définition la plus petite relation réflexive transitive contenant \mathcal{R} , mais c'est aussi la clôture transitive de la clôture symétrique de \mathcal{R} , ou encore la clôture symétrique de la clôture transitive de \mathcal{R} . On peut de même ajouter une clôture réflexive dans n'importe quel ordre. (*admis*)
- En termes de schéma/graphe d'une relation, la clôture réflexive signifie que l'on peut « faire du sur place », symétrique que « si on peut aller de a à b on peut aller de b à a » et transitive que « on peut suivre une suite de flèches ».

Exemple.

- La clôture transitive réflexive de $\{(n, n+1), n \in \mathbb{N}\}$ est l'ordre usuel sur \mathbb{N} .
- La clôture réflexive symétrique transitive de la relation vide est l'égalité.

Proposition 39.

La clôture réflexive symétrique transitive de \mathcal{R} est la plus petite relation d'équivalence contenant \mathcal{R} .

Démonstration. On admet pour l'instant l'existence de cette plus petite sur-relation d'équivalence.

Soit \mathcal{R}' une relation d'équivalence contenant \mathcal{R} . Alors \mathcal{R}' est réflexive symétrique transitive et contient \mathcal{R} , donc elle est contenue dans la clôture réflexive symétrique transitive de \mathcal{R} par définition.

Autrement dit, la clôture est minimale parmi les sur-relations d'équivalence. Par unicité d'un plus petit élément, on conclut. \square

Proposition 40.

On peut expliciter les relations \mathcal{R}^0 , \mathcal{R}^S et \mathcal{R}^* :

- $\mathcal{R}^0 = \mathcal{R} \cup \{(x, x) \text{ s.t. } x \in E\}$
- $\mathcal{R}^S = \mathcal{R} \cup \{(y, x) \text{ s.t. } x\mathcal{R}y\}$
- Pour $i \in \mathbb{N}^*$, posons $\mathcal{R}^i = \mathcal{R}^{i-1} \cup \{(x, y) \text{ t.q. } \exists c_0, \dots, c_i \text{ avec } c_0 = x, c_i = y \text{ et } \forall 0 \leq k < i, c_k\mathcal{R}c_{k+1}\}$. Alors :
 - $\mathcal{R}^+ = \bigcup_{i \in \mathbb{N}^*} \mathcal{R}^i$
 - $\mathcal{R}^* = \mathcal{R}^0 \cup \mathcal{R}^+$

En particulier, ces constructions prouvent que ces clôtures existent.

Démonstration. • La relation \mathcal{R}^0 donnée est une relation réflexive, et elle inclut bien \mathcal{R} . De plus, toute sur-relation réflexive de \mathcal{R} contient au moins \mathcal{R} et les paires identiques, c'est à dire au moins \mathcal{R}^0 . D'où la minimalité.

• Même raisonnement.

• On remarque que $\mathcal{R} = \mathcal{R}^1$. Ainsi, la relation \mathcal{R}^+ donnée inclut bien \mathcal{R} . Montrons maintenant qu'elle est transitive :

Si $x\mathcal{R}^+y$ et $y\mathcal{R}^+z$, alors $\exists i, x\mathcal{R}^i y$ et $\exists j, y\mathcal{R}^j z$.

C'est à dire :

$\exists c_0, \dots, c_i$ avec $c_0 = x$, $c_i = y$ et $\forall 0 \leq k < i, c_k \mathcal{R} c_{k+1}$

et $\exists c'_0, \dots, c'_j$ avec $c'_0 = y$, $c'_j = z$ et $\forall 0 \leq k < j, c'_k \mathcal{R} c'_{k+1}$.

On en déduit que $x \mathcal{R}^{i+j} z$ par la séquence $x = c_0, \dots, c_i = y = c'_0, \dots, c'_j = z$ de longueur $i + j$. Donc $x \mathcal{R}^+ z$, d'où la transitivité.

Pour la minimalité, si \mathcal{T} est une sur-relation transitive de \mathcal{R} , on montre par récurrence sur i que tous les \mathcal{R}^i sont inclus dans \mathcal{T} .

- Similaire aux deux premiers.

□

Remarque.

- La définition de \mathcal{R}^+ se comprend mieux en français : \mathcal{R}^i est l'ensemble des paires peuvent être « reliées » par un chemin de i applications de \mathcal{R} . Par exemple, si $(x, y) \in \mathcal{R}^3$, on a un chemin de la forme $x \mathcal{R} u \mathcal{R} v \mathcal{R} y$. Notez que dans cet exemple x et y ne sont pas forcément en relation par \mathcal{R} : on affirme juste qu'il y a un chemin de longueur au plus 3 (et donc qu'ils sont en relation dans la cloture transitive).

FIGURE 8.13 – $\mathcal{P}(\llbracket 0; 4 \rrbracket)$ et les chemins de longueur 3

- Ces constructions s'expriment bien en français : pour rendre une relation réflexive on ajoute les paires identiques, pour la rendre symétrique on symétrise, et pour la rendre transitive on « suit les chemins » (c'est à dire qu'on ajoute la transitivité).

2.3.1 Engendrer un ordre

La question est maintenant : comment construire des relations d'ordre avec cela ?

Définition 41 (Acyclicité d'une relation).

Soit $l \in \mathbb{N}$. Un cycle de longueur l pour \mathcal{R} est une suite finie $(u_n)_{n \in \llbracket 0; l \rrbracket}$ d'éléments deux à deux distincts telle que pour tout n , $u_n \mathcal{R} u_{n+1}$ et telle que $u_{l-1} \mathcal{R} u_0$.

On dit que \mathcal{R} est sans cycle s'il n'y a pas de cycle de longueur au moins 2 pour \mathcal{R} .

Cette notion se comprend très bien sur un schéma :

FIGURE 8.14 – Un cycle

Exemple. Si on considère $(\mathbb{Z}/3\mathbb{Z}, \leq)$ et que l'on définit une relation d'ordre sur cet ensemble par :

$$0 \leq 1, \text{ et } 1 \leq 2 \text{ et } 2 \leq 0$$

alors 0, 1, 2 est un cycle de longueur 3.

Exemple. Notons $|$ la relation de divisibilité, c'est à dire que $a|b$ signifie $a|b$. Si l'on considère que tout entier divise 0, alors $(\mathbb{N}, |)$ contient de nombreux cycles. Sinon, la relation est acyclique car $a|b \implies a \leq b$.

FIGURE 8.15 – Cycles de la division de 0

Remarque. On pourrait aussi définir un cycle avec la contrainte alternative : pour tout n , $u_{n+1} \mathcal{R} u_n$. C'est une question de conventions. L'important est que ce soit toujours dans le même sens.

Proposition 42 (Cloture réflexive transitive d'une relation acyclique).

Si \mathcal{R} est sans cycle, alors sa cloture réflexive transitive \mathcal{R}^* est la plus petite relation d'ordre contenant \mathcal{R} .

Démonstration. Il y a deux points à prouver : que c'est une relation d'ordre, et que c'est la plus petite. Commençons par le second.

Toute relation d'ordre contenant \mathcal{R} est en particulier réflexive et transitive. Par définition de la cloture réflexive transitive, \mathcal{R}^* est bien la plus petite.

Montrons maintenant que c'est bien une relation d'ordre. Comme elle est par définition réflexive et transitive, il reste à montrer qu'elle est antisymétrique.

Soient x et y tels que $x\mathcal{R}^*y$ et $y\mathcal{R}^*x$. Par construction de \mathcal{R}^* , notons i et j les premiers rangs tels que $x\mathcal{R}^iy$ et $y\mathcal{R}^jx$. Si i ou j sont non-nuls, alors cela fournit un cycle de \mathcal{R} : absurde. Donc $i = j = 0$, i.e. $x = y$. La relation est donc bien antisymétrique. \square

Définition 43 (Ordre engendré).

Soit \mathcal{R} une relation sans cycle sur E . On appelle **ordre engendré par \mathcal{R}** la cloture réflexive transitive de \mathcal{R} .

Exemple. L'ordre engendré par la relation $\{(n, n+1) \mid n \in \mathbb{N}\}$ est l'ordre usuel sur \mathbb{N} .

Remarque.

- Pour définir un ordre, il suffit parfois de donner les prédécesseurs immédiats de chaque élément (ou ses successeurs immédiats). Nous en verrons les conditions en TD.
- Les deux grosses applications des ordres engendrés sont l'ordre topologique (que nous verrons dans quelques mois) et l'induction structurelle.

3 Induction structurelle

3.0 Construction inductive d'un ensemble : exemple illustratif

On reprend l'exemple des mobiles de Calder. On suppose pour simplifier que les objets suspendus sont des entiers :

```
4 type objet = int
5 type mobile = Obj of objet | Bar of mobile * mobile
```



Ainsi, le mobile ci-dessous correspond à la déclaration OCaml qui l'accompagne :

FIGURE 8.16 – Un mobile et son code OCaml

On veut savoir quels sont précisément les éléments de type `mobile`. Notons \mathcal{O} l'ensemble des objets, et on veut définir M l'ensemble des mobiles.

Un mobile est obtenu par des applications successives des règles de construction. Notons f_M la fonction qui « applique une étape de la construction à un ensemble X de mobiles déjà construits » :

$$f_T : X \mapsto \mathcal{O} \cup \{\text{Bar}(m_1, m_2) \mid m_1 \in X, m_2 \in X\}$$

Autrement dit, si l'on dispose de X un ensemble de mobiles, les mobiles que l'on peut obtenir à partir de X sont :

- Les objets (sans utiliser les mobiles de X donc).

- Deux mobiles de X reliés par une barre.

On peut alors nommer les étapes successives de construction :

- $M_0 = f_M(\emptyset) = \emptyset$
- Pour $n \in \mathbb{N}$, $M_{n+1} = M_n \cup M_T(M_n)$

Ainsi, M_n sont les mobiles que l'on obtient en *au plus* $n + 1$ étapes de constructions.

Définition 44.

L'ensemble M des mobiles est :

$$M = \lim_{n \rightarrow +\infty} M_n = \bigcup_{n=0}^{+\infty} M_n$$

C'est à dire l'ensemble des arbres que l'on obtient en partant des cas de bases et en appliquant un nombre fini de fois le constructeur récursif `Bar`.

Remarque. Cette définition exclut les expressions infinies puisque le constructeur n'est appliqué qu'un nombre fini de fois. En conséquence, les listes infinies ne peuvent pas exister puisqu'une liste est une suite d'application du constructeur `::` à partir du cas de base `[]`.¹²

Il existe une autre façon de voir M , qui n'utilise pas cette construction par récurrence :

Démonstration. On procède en deux temps : on montre que M est plus petit que tout point fixe, puis que c'est un point fixe.

- si X est un point fixe de f_M , alors $f_M(X) = X$. En particulier, $M_0 = \emptyset \subseteq X$. Mais alors $f_M(M_0) \subseteq f(X) = X$. Ainsi, par récurrence sur i , pour tout i , $M_i \subseteq X$. Il s'ensuit que $M \subseteq X$
- Pour tout $n \in \mathbb{N}^*$, $f_M(M_n) = M_{n+1}$. Donc $f_M(\bigcup_{n=0}^{+\infty} M_n) = \bigcup_{n=1}^{+\infty} M_n$. D'où $f_M(M) \subseteq M$. Mais $M_0 = \emptyset \subseteq M_1$. Donc $\bigcup_{n=0}^{+\infty} M_n \subseteq f_M(M)$, donc $f_M(M) = M$.

□

Le type des mobiles était plutôt simple en cela qu'il n'y avait qu'un seul cas de base, et un seul cas récursif. Mais on peut tout à fait définir des cas plus compliqués, comme par exemple :

```
1 type expr =
2   | Int of int
3   | Var of string
4   | Add of expr * expr
5   | Mul of expr * expr
6   | Modulo of expr * expr
```



Ce type permet de représenter récursivement des expressions arithmétiques pouvant faire apparaître des entiers, des variables nommées, des additions, soustractions ou modulus. Par exemple, « $((x+1)(x-1)) \bmod 3$ » correspond à :

12. (Hors-Programme) Toutefois, OCaml supporte partiellement les listes infinies... pour peu qu'elles soient un « cycle », c'est à dire un facteur fini qui se répète à l'infini. On peut par exemple faire `let rec l = 0::1::l`. On retombe assez vite sur la notion de co-induction, déjà évoquée mais de niveau M2.

FIGURE 8.17 – Représentation arborescente d’une expression arithmétique

3.1 Induction structurelle

Soit E un ensemble obtenu par une construction inductive¹³. Par exemple, les mobiles de Calder ou les expressions arithmétiques ou les listes OCaml.

On note \mathcal{R}_X la relation homogène sur X définie par : « $x \mathcal{R}_X y$ lorsque y est construit en appliquant un constructeur à (entre autres) x ».

Exemple.

- Dans les mobiles, $m_1 \mathcal{R} \text{Bar}(m_1, m_2)$
- Dans les suites OCaml, $[2; 0; 3] \mathcal{R} [1; 2; 0; 3]$. De même, $[2; 0; 3] \mathcal{R} [5; 2; 0; 3]$.

Proposition 45.

\mathcal{R}_X est sans cycle.

Démonstration. Admis. □

Corollaire 46 (ordre structurel).

On peut étendre \mathcal{R}_X en un ordre, appelé « ordre structurel ».

Intuitivement, cet ordre dit que $x \leq y$ si et seulement si x a été utilisé dans la construction récursive de y .

Démonstration. Il s’agit de la fermeture réflexive transitive d’une relation acyclique. □

Théorème 47.

L’ordre structurel est bien fondé.

Démonstration. Idée la preuve sur l’exemple des mobiles : à chaque mobile, on associe le plus petit n tel que le mobile appartienne à M_n . On peut alors montrer qu’une suite infinie strictement décroissante de mobiles correspond à une suite infinie strictement décroissante de $n \in \mathbb{N}$: impossible. □

Et voilà le travail ! Nous pouvons maintenant faire des inductions *sur les membres d’un type inductif eux-mêmes* !

13. ou récursive, c’est synonyme

Convention 48 (Méthodologie des inductions structurelles).

Pour faire une induction structurelle, on doit :

- **Énoncer clairement la propriété** à prouver et le fait qu'on procède par induction structurelle.
- Traiter comme **cas de base** tous les cas de base du type (les éléments produits par des constructeurs non-récursifs).
- Traiter l'hérédité, qui consiste à **prendre un élément qui n'est pas un cas de base, supposer la propriété vraie pour tous ses sous-éléments, et en déduire la propriété vraie pour lui**.
- **Conclure** par principe d'induction structurelle.

Exemple. Nous avons en fait déjà fait un exemple d'induction structurelle sur les mobiles ! Et nous en ferons *beaucoup* sur les arbres.

Remarque. ⚠ Certains rapports de jury sont dubitatifs sur les rédactions des inductions structurelles. Faites-y très attention. Notamment, définissez précisément qui sont les « sous-éléments ».

3.2 Un exercice final

Voici une fonction récursive. Prouver qu'elle termine et qu'elle est totalement correcte :

```

17  (** Fusionne les deux listes triées l0 et l1
18     * en une seule liste triée.
19     *)
20  let rec merge l0 l1 =
21    match (l0 , l1) with
22    | [] , [] -> []
23    | [] , _ -> l1
24    | _ , [] -> l0
25    | t0::q0 , t1::q1 ->
26      if t0 <= t1 then t0 :: (merge q0 l1)
27      else t1 :: (merge l0 q1)

```



Chapitre 9

ARBRES

Notions	Commentaires
Définition inductive du type arbre binaire. Vocabulaire : noeud, noeud interne, racine, feuille, fils, père, hauteur d'un arbre, profondeur d'un noeud, étiquette, sous-arbre.	La hauteur de l'arbre vide est -1 . On mentionne la représentation d'un arbre complet dans un tableau.
Arbre. Conversion d'un arbre d'arité quelconque en un arbre binaire.	La présentation donne lieu à des illustrations au choix du professeur. Il peut s'agir par exemple d'expressions arithmétiques, d'arbres préfixes (<i>trie</i>), d'arbres de décision, de dendrogrammes, d'arbres de classification, etc.
Parcours d'arbre. Ordre préfixe, infixe et postfixe.	On peut évoquer le lien avec l'empilement de blocs d'activation lors de l'appel à une fonction récursive.
[...] Arbre binaire de recherche. Arbre bicolore.	On note l'importance de munir l'ensemble des clés d'un ordre total.

Extrait de la section 3.3 du programme officiel de MP2I : « Structures de données hiérarchiques ».

Notions	Commentaires
Implémentation interne [des fichiers] : blocs et noeuds d'index (<i>inode</i>).	On présente le partage de blocs (avec liens physiques ou symboliques) et l'organisation hiérarchique de l'espace de nommage.

Extrait de la section 5.2 du programme officiel de MP2I : « Gestion des fichiers et entrées-sorties ».

SOMMAIRE

0. Exemples introductifs et vocabulaire.....	165
0. Les mobiles	165
1. Les expressions arithmétiques	166
2. Arbre d'appels récursifs	166
<i>Tours de Hanoï (p. 166). Retour sur trace (p. 167).</i>	
3. Exemple : les tries	168
4. Exemple : les arbres binaires de recherche	168
5. Exemple et parenthèse : gestion des fichiers Unix	169
<i>Inodes et fichiers (p. 169). Inodes et dossiers (p. 169). Liens physiques et symboliques (p. 170).</i>	
6. Vrac de vocabulaire	171
7. Construction inductive	172
1. Arbres binaires et implémentation.....	173
0. Arbres binaires	173
<i>Définition (p. 173). Implémentation (p. 176).</i>	
1. Arbres binaires stricts	176
<i>Définitions (p. 176). Implémentation (p. 177).</i>	
2. Arbres parfaits et complets	178
<i>Définitions (p. 178). Implémentation des arbres binaires complets (p. 180).</i>	

3. Peigne	180
4. Arbres quelconques	181
<i>Transformation LCRS (p. 181). Type général (p. 182).</i>	
2. Parcours d'arbres.....	183
0. Parcours en profondeur	183
<i>Ordre préfixe (p. 184). Parcours infixe (p. 185). Parcours postfixe (p. 186). Écriture itérative (p. 186).</i>	
1. Parcours en largeur	187
<i>Écriture itérative (p. 187). (Mi-HP) Preuve du pseudo-code (p. 188). (Mi-HP) Preuve, suite et fin (p. 189).</i>	
3. (HP) Propriétés avancées	189
0. Lemme de lecture unique	189
1. Dénombrement des arbres binaires stricts	190
4. Arbres binaires de recherche	194
0. Définition, caractérisation	194
1. Opérations	196
<i>Recherche (p. 196). Insertion (p. 197). Suppression (p. 198). Rotation (p. 200). Tri par ABR (p. 201).</i>	
2. Arbres Rouge-Noir	202
<i>Définition (p. 202). Insertion (p. 204). Suppression (p. 206).</i>	
3. Pour aller plus loin	206

0 Exemples introductifs et vocabulaire

Définition 1 (Arbres).

Un **arbre** est une structure de données hiérarchique constituée de $n \in \mathbb{N}$ **noeuds**, éventuellement étiquetés. S'il n'est pas vide ($n \geq 1$), les noeuds sont structurés de la manière suivante :

- Un noeud particulier r est appelé la **racine** de l'arbre.
- Les $n - 1$ noeuds restants sont partitionnés en $k \geq 0$ sous-ensemble disjoints qui forment k arbres, appelés **sous-arbres** de r .
- La racine r est reliée à la racine de chacun des sous-arbres. Ces liens sont appelés des **arêtes**.
- S'il n'y a pas de sous-arbres (si $k = 0$), on dit que r est une **feuille**.

FIGURE 9.1 – Un arbre

Remarque.

- Autrement dit, un arbre est un type inductif! Un arbre est soit constitué d'une racine reliée à des sous-arbres (éventuellement vides), soit vide.
- Il n'y a pas une seule définition des arbres, mais plein de variations autour de cette définition.

0.0 Les mobiles

Voici un type pour les mobiles de Calder :

```
4 type objet = int
5 type mobile = Obj of objet | Bar of mobile * mobile
```



Ici, les noeuds sont les barres ou les objets. Vous noterez qu'il n'autorise pas les mobiles vides : le cas de base est l'objet, qui est donc une feuille.

FIGURE 9.2 – Un mobile de Calder décomposé comme un arbre

0.1 Les expressions arithmétiques

Voici un type pour des expressions arithmétiques (sans division) :

```

4 type expr =
5   | Int of int
6   | Var of string
7   | Add of expr * expr
8   | Sub of expr * expr
9   | Mul of expr * expr
10  | Div of expr * expr
11  | Mod of expr * expr
12

```

 arith.ml

Ici, les noeuds sont des entiers, variables ou des opérations. Les feuilles correspondent aux entiers et aux variables. Notez que l'on pourrait fusionner tous les cas inductifs en un seul :

```

19 type expr =
20   | Int of int
21   | Var of string
22   | Fun of (int -> int -> int) * expr * expr
23     (* ex : Fun ((+), Int 3, Int 2)
24       correspond à Add (Int 3, Int 2) *)

```

 arith.ml

FIGURE 9.3 – Une expression arithmétique décomposée comme un arbre

0.2 Arbre d'appels récursifs

0.2.0 Tours de Hanoï

Voici un pseudo-code résolvant les tours de Hanoï :

Fonction Hanoï

Entrées : i : la tige de départ ; j : la tige d'arrivée ; n : le nombre de disques à déplacer

```

1 si n > 0 alors
2   k ← tige autre que i ou j
3   HANOÏ(i, k, n-1)
4   Déplacer 1 disque de i à j
5   HANOÏ(k, j, n-1)

```

Et voici la forme de l'arbre d'appels associé pour $n = 3$:

FIGURE 9.4 – Arbre d'appels pour $n = 3$

On remarque que le déroulé de l'algorithme récursif consiste à se déplacer dans l'arbre. On appelle cela un **parcours** de l'arbre.

0.2.1 Retour sur trace

Voici l'arbre des plateaux des n dames explorés par un retour sur trace qui place les dames ligne par ligne et rejette dès que deux dames sont en prise :

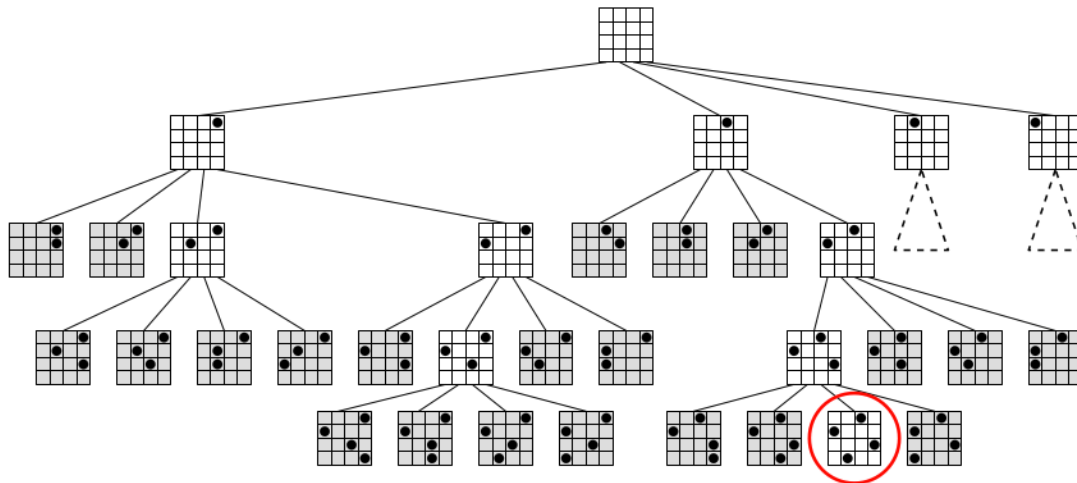


FIGURE 9.5 – Exploration en plaçant d'abord sur la première ligne, puis sur la seconde, etc. En gris les solutions partielles rejetées. La moitié non-représentée de la figure est symétrique. Src : J.B. Bianquis

Remarque.

- Si l'on code l'algorithme de sorte à ce qu'il s'arrête dès qu'il a trouvé une solution, on arrête le parcours de l'arbre dès que l'on rencontre le plateau valide entouré en rouge.
- Le rejet dans un retour sur trace est aussi appelé un **élagage** : on « coupe » des « branches » de l'arbre.

0.3 Exemple : les tries

Un **trie** est un arbre qui permet de représenter un ensemble de mots, en fusionnant leurs préfixes : on étiquette les arêtes de l'arbre par des lettres, et un mot stocké dans le trie correspondant à un chemin (descendant) de la racine jusqu'à une feuille.

FIGURE 9.6 – Trie contenant { dragon, drame, drap, droide, droite, drone, demon, de, degat, demo, degel, demi }

0.4 Exemple : les arbres binaires de recherche

Un arbre binaire de recherche est un arbre où chaque noeud a au plus deux sous-arbres. On range des valeurs dans les noeuds. L'étiquette d'un noeud est plus grande que celles de son sous-arbre gauche et plus petite que celles de son sous-arbre droit : cela permet d'y effectuer des recherches plutôt efficaces, « par dichotomie ».

(a) Un arbre binaire de recherche pour E .

(b) Un autre arbre binaire de recherche pour E .

FIGURE 9.7 – Deux arbres binaires de recherche pour $E = \llbracket 1; 12 \rrbracket$

0.5 Exemple et parenthèse : gestion des fichiers Unix

0.5.0 Inodes et fichiers

Un disque dur sert à contenir des fichiers, et est naturellement linéaire : les différentes zones d'un disque dur sont correspondent à différentes « adresses », les unes après les autres. Plus précisément, un disque dur est une suite de **blocs**, et un fichier est réparti sur plusieurs blocs.

Chaque fichier correspond en fait à :

- Des métadonnées (e.g. date de dernière modification)
- L'adresse des blocs du disque dur contenant le contenu du fichier.

Ces informations dans une structure qui s'appelle un **inode** (information **node**).

FIGURE 9.8 – Schéma (très simplifié) d'un inode indiquant des blocs d'un disque dur.

Remarque.

- Le nom d'un fichier n'est pas stocké dans l'inode du fichier, mais dans celui du dossier parent. Cf suite du cours.
- Une conséquence intéressante de cette façon de gestion des fichiers est que faire un couper-coller est quasi-immédiat : on ne recopie absolument pas tout le fichier, on ne déplace que l'inode (qui est très petit). Ainsi, couper-coller un fichier de 20 Go et un fichier de 10ko est à peu près aussi long.
- La remarque précédente ne s'applique pas à la copie : on veut que la copie d'un fichier soit physiquement différente de l'original (c'est à dire qu'il ne s'agisse pas des mêmes blocs du disque dur), de sorte à ce que modifier la copie ne modifie pas l'original (et réciproquement).
- En réalité, rien ne garantit que les blocs d'un fichier soient consécutifs : un inode pointe sur tous les blocs. Si un seul inode n'est pas assez long pour contenir tous les inodes, il y a une gestion par liste chaînée d'inodes¹ : un inode indique où trouver les premiers blocs du fichier, et où trouver l'inode de la suite.

0.5.1 Inodes et dossiers

Les dossiers sont des fichiers particuliers : les dossiers sont des inodes dont les blocs pointés sont... les inodes du contenu du dossier².

FIGURE 9.9 – Schéma (très simplifié et un peu faux) de l'inode d'un dossier.

1. Je simplifie. Plus généralement, toute cette sous-partie sur les inodes tient plus de la vulgarisation ; et a avant tout pour but de vous faire comprendre que la représentation et l'organisation des fichiers sur un disque dur est quelque chose qui s'étudie et s'optimise.

2. En réalité, c'est faux. L'inode d'un dossier indique un fichier spécial qui lui-même redirige vers les inodes. Je simplifie à outrance.

Dans un inode de dossier, on trouve des informations supplémentaires sur le contenu du dossier, et notamment... les noms de fichiers/sous-dossiers ! En effet, le nom d'un fichier ne fait pas partie de ses méta-données mais des données de son dossier parent.

Remarque.

- Un dossier contient des fichiers ou des dossiers qui contiennent eux-même... etc. On parle de l'arborescence des fichiers : l'organisation des dossiers/fichiers est un arbre. D'ailleurs, le tout premier dossier est appelé la *racine* du système. Sur Unix, il s'agit du dossier `/`.
- Puisque cet arbre correspond aux successions d'inodes qui indiquent des inodes des sous-dossiers, etc, il s'agit d'un exemple d'arbre implémenté comme une succession de « pointeurs »³. C'est une implémentation courante pour des arbres.

0.5.2 Liens physiques et symboliques

Un lien est un raccourci. Il y a deux types de liens :

- Un lien physique (« hard link ») : un lien physique `l` vers un fichier `f` correspond à un inode qui indique le fichier `f`. Autrement dit, c'est vraiment le même fichier ! En particulier, il a le même poids que `f` !
En bref, un lien physique permet de faire en sorte que le même fichier soit accessible depuis plusieurs endroits.
⚠ On ne peut pas faire un lien physique vers un dossier, uniquement vers un fichier. Cela garantit que l'arborescence reste un arbre (on pourrait avoir un cycle sinon).
- Un lien symbolique (« soft link ») : c'est un lien vers (donc un inode qui redirige vers⁴) un élément de l'arborescence. Il est donc très léger, et a l'avantage de pouvoir pointer vers un dossier.

Remarque.

- Un lien physique correspond au même inode que l'origine. Pourtant, un lien physique peut avoir un nom différent de l'original : c'est pour cela que le nom n'est pas stocké dans l'inode du fichier !
- Corollaire intéressant des liens physiques : l'espace utilisé sur votre disque dur n'est pas la somme des poids des éléments présents dans votre arborescence, puisque certains de ces éléments peuvent être un lien physique vers un autre élément de l'arborescence.
- Autre corollaire intéressant : déplacer le fichier `f` à un autre endroit de l'arborescence ne modifiera pas son hard link `l` : en effet, `l` pointe vers les blocs du disque dur, et non vers un endroit de l'arborescence. Plus fort encore, supprimer `f` ne modifiera pas `l` : en effet, supprimer `f` revient simplement à supprimer son inode. Comme les blocs du disque dur sont encore pointé par `l`, eux ne sont pas effacés.

On peut utiliser `ls -l` pour lister le contenu d'un dossier et afficher la nature de ses éléments (ainsi que leur date de dernière modification). En ajoutant `-sh`, on peut obtenir les poids des éléments :

```
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Bureau
lrwxrwxrwx 1 antoine antoine 28 oct. 10 11:35 CPGE -> /home/antoine/Documents/CPGE
drwxr-xr-x 17 antoine antoine 4,0K déc. 16 16:58 Documents
drwxrwxr-x 6 antoine antoine 4,0K déc. 29 22:35 foo
-rw-rw-r-- 1 antoine antoine 99K déc. 28 15:15 guidelines.pdf
drwxr-xr-x 12 antoine antoine 4,0K nov. 12 14:57 Images
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Modèles
lrwxrwxrwx 1 antoine antoine 33 oct. 10 11:35 MP2I-tex -> /home/antoine/Documents/CPGE/MP2I
lrwxrwxrwx 1 antoine antoine 41 oct. 10 11:35 MPI-tex -> /home/antoine/Documents/CPGE/MPI-Poitiers
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Musique
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Public
drwx----- 6 antoine antoine 4,0K févr. 5 09:00 snap
drwxr-xr-x 12 antoine antoine 20K févr. 4 11:28 Téléchargements
drwxr-xr-x 2 antoine antoine 4,0K oct. 9 20:59 Vidéos
```

FIGURE 9.10 – Contenu de mon dossier home obtenu via `ls -sh`.

3. Je pense que tout-e spécialiste du sujet s'étranglerait en m'entendant en parler comme des pointeurs, mais c'est l'idée. À outrance j'ai dit !

4. Vers un fichier spécial qui redirige vers

Sur la capture d'écran de la figure 9.10, notez que :

- MP2I-tex et MPI-tex sont des liens symboliques pointant vers un autre endroit de mon arborescence. Ils sont *extrêmement* légers.
- Les dossiers en eux-même sont très légers : en effet, un dossier est un nom ainsi qu'une redirection vers son contenu. En particulier, le poids d'un dossier ne comprend pas le poids de son contenu ! Notez que la raison pour laquelle ils sont malgré tout aussi lourd (4K) est parce qu'il y a une taille minimale. Je soupçonne que la raison pour laquelle Téléchargements est aussi lourd est parce qu'il ne tient pas dans 4K (il doit pointer vers beaucoup d'éléments, je n'ai pas fait le ménage depuis longtemps), et qu'il a donc pris la « prochaine » taille minimale.
- guidelines.pdf est lourd : c'est normal, c'est un vrai fichier, sa taille est donc la taille de son contenu (de ses blocs).

On peut également demander à `ls` d'afficher le numéro des inodes : il s'agit de l'option `-li` :

```
22154963 100K -rw-rw-r-- 2 antoine antoine 99K déc. 28 15:15 guide-hard
22154963 100K -rw-rw-r-- 2 antoine antoine 99K déc. 28 15:15 guidelines.pdf
22160938 0 lrwxrwxrwx 1 antoine antoine 14 févr. 5 10:49 guide-symb -> guidelines.pdf
```

FIGURE 9.11 – Un fichier `guidelines.pdf`, un lien physique `guide-hard` et un lien symbolique `guide-symb` vers lui. Affichage obtenu via `ls -lshi`.

Sur la capture d'écran de la figure 9.11, une première coordonnée s'est ajoutée : il s'agit du numéro de l'inode des éléments concernés. Notez que :

- Le lien symbolique et sa cible n'ont pas le même inode. Ils n'ont pas non plus le même poids, ni la même dernière date de modification (la dernière modification du lien est la création du lien). Le lien symbolique est indiqué comme étant un lien.
- Le lien physique et sa cible sont indiscernables. Et c'est normal, ils sont rigoureusement identiques ! Notez que la dernière modification du lien physique correspond à la dernière modification du fichier d'origine, puisque ce sont les mêmes ; et en particulier la dernière modification du lien physique n'est pas « sa création ». Le lien physique n'est pas indiqué comme étant un lien ; et d'ailleurs ce n'en est pas vraiment un... c'est juste le fichier d'origine.

0.6 Vrac de vocabulaire

Définition 2 (Vrac de vocabulaire).

- Chaque noeud a un certain nombre (éventuellement nul) d'**enfants** qui sont les racines des sous-arbres.
- L'**arité** d'un noeud est son nombre d'enfants.
- Un noeud d'arité 0 (sans enfants) est appelé **feuille**, un noeud d'arité non-nulle est appelé **noeud interne**.
- Si x est un enfant de y , on dit que y est le **parent** de x . La **racine** est le seul noeud qui n'a pas de père.
- Les enfants d'un même parent sont des **adelphe**s.
- Les **ancêtres** d'un noeud sont ses parents successifs jusqu'à la racine (parent, parent du parent, etc).
- La **profondeur** d'un noeud est la longueur du chemin (en nombre d'arêtes) qui relie la racine à ce noeud. La racine a profondeur 0, ses enfants 1, etc.
- L'étage p **étage** d'un arbre est l'ensemble des noeuds à profondeur p .
- La **hauteur** de l'arbre est la profondeur maximale d'un noeud.

FIGURE 9.12 – Illustration du vocabulaire

Remarque. Les termes « père / frère / fils » au lieu de « parent / adelphe / enfant » sont aussi utilisés. Ils ont tendance à l'être de moins en moins, notamment en anglais où l'on préfère très clairement « parent/sibling/child ».

0.7 Construction inductive

Théorème 3 (Induction structurelle).

Pour raisonner sur les arbres, on peut utiliser des inductions structurelles. Autrement dit, on procède généralement ainsi pour prouver une propriété vraie sur un arbre :

- Initialisation : la prouver vraie sur les feuilles.
- Hérédité : prouver que si elle est vraie pour les sous-arbres, alors elle est vraie pour l'arbre

Démonstration. Les arbres sont définis inductivement. □

Remarque. On peut aussi à la place faire des récurrences sur la hauteur ou sur le nombre de noeuds.

Théorème 4 (Fonctions inductives).

Pour définir une fonction sur les arbres, on peut procéder inductivement. Autrement dit, on procède généralement pour définir une fonction f qui prend en entrée un arbre :

- Initialisation : définir f sur les feuilles.
- Hérédité : définir f sur l'arbre à l'aide de f sur les sous-arbres.

Démonstration. Idem. □

1 Arbres binaires et implémentation

Dans toute cette partie, on considère des arbres dont les noeuds contiennent une valeur appelée « étiquette ». On note \mathcal{E} l'ensemble des étiquettes possibles.

1.0 Arbres binaires

1.0.0 Définition

Définition 5 (Arbres binaires, informel).

Un arbre binaire est un arbre où l'arité de tous les noeuds vaut au plus 2.

Définition 6 (Arbres binaires, formel).

L'ensemble $\mathcal{A}(\mathcal{E})$ des arbres binaires (non-stricts) étiquetés par \mathcal{E} est défini inductivement comme suit :

- L'arbre vide \perp (aussi appelé Nil) est un arbre de $\mathcal{A}(\mathcal{E})$.
- Relier deux arbres binaires par une racine donne un arbre binaire : pour tout $g, d \in \mathcal{A}(\mathcal{E})$ et $x \in \mathcal{E}$, le noeud $N(g, x, d)$ est un arbre de $\mathcal{A}(\mathcal{E})$.
 g est appelé l'**enfant gauche** de ce noeud et d l'**enfant droit**. x est appelé l'**étiquette** du noeud.

Remarque.

- J'abrégérai parfois $N(g, x, d)$ en (g, x, d) .
- Quand un noeud n'a que des enfants \perp , on dit souvent qu'il n'a pas d'enfants. De même, s'il a un seul enfant non-Nil, on dit souvent qu'il a un seul enfant. Ainsi, une feuille est un noeud de la forme $N(\perp, x, \perp)$; et les autres noeuds non-Nil sont les noeuds internes.
- Un arbre binaire est un arbre dans lequel chaque noeud a au plus deux enfants.
- Quand on représente un arbre binaire, on omet souvent les Nil pour alléger le tout :

(a) Un arbre binaire

(b) Le même arbre binaire

FIGURE 9.13 – Deux façons de représenter un arbre binaire.

- Le fait d'être enfant gauche ou droit à son importance. Ainsi, les deux arbres ci-dessous ne sont pas les mêmes :

(a) $N(N(2, Nil), 1, Nil)$ (b) $N(Nil, 1, N(2, Nil, Nil))$

FIGURE 9.14 – Deux arbres binaires distincts.

Définition 7 (Hauteur, formel).

La **hauteur** h d'un arbre se formalise ainsi :

- $h(\perp) = -1$
- Pour tout $(g, d) \in \mathcal{A}(\mathcal{E})$ et $x \in \mathcal{E}$, on pose $h(N(g, x, d)) = 1 + \max(h(g), h(d))$

Pourquoi cette définition ? Parce que la hauteur est la longueur du plus long chemin descendant de la racine à une feuille. D'où le cas récursif. Mais pourquoi ce cas de base ? Parce que l'on a envie de dire que dans l'arbre qui est réduit à une feuille, le plus long chemin descendant est de longueur 0. Il faut pour cela poser $h(\perp) = -1$. Au fond, tout cela provient du fait que l'on veut ignorer les \perp lorsque l'on raisonne sur des dessins.

(a) Un arbre binaire de hauteur 2.

(b) Un arbre binaire de hauteur 4.

FIGURE 9.15 – Hauteur dans un arbre binaire

Définition 8 (Étage entièrement rempli).

On dit que l'étage p d'un arbre binaire est entièrement rempli s'il contient 2^p noeuds.

Proposition 9.

L'étage p d'un arbre binaire est au plus entièrement rempli.

Démonstration. Soit A un arbre binaire. Montrons qu'un étage p de A a au plus 2^p noeud. Procédons par récurrence⁵ sur $p \in (\mathbb{N}, \leq)$.

5. Pour une (rare) fois, ce n'est pas une induction structurelle ! En effet, le cas de base de la profondeur est la racine et non les feuilles.

- Initialisation : par définition, un noeud à profondeur 0 a un chemin vide depuis la racine : autrement dit, c'est la racine. Il y a dans A au plus une racine, or $1 = 2^0$ d'où l'initialisation.
- Hérédité : soit $p \in \mathbb{N}^*$ tel que la propriété est vraie au rang $p - 1$, montrons-la vraie au rang p . Tout noeud de A à profondeur p est l'enfant d'un (et d'un seul) noeud de A à profondeur $p - 1$. Réciproquement, chaque noeud à profondeur $p - 1$ a au plus deux enfants car l'arbre est binaire. Il s'ensuit que le nombre de noeuds à profondeur p est au plus le double du nombre de noeuds à profondeur $p - 1$. En appliquant l'hypothèse de récurrence et en multipliant par deux, on obtient l'hérédité.
- Conclusion : On a prouvé la propriété vraie pour toute profondeur p dans A . Puisque A était un arbre binaire quelconque, on a montré la propriété vraie pour toute profondeur de tout arbre binaire.

□

Remarque. Pour que cette définition soit pertinente, il faudrait aussi prouver que la borne peut-être atteinte. Les arbres parfaits fourniront un tel exemple.

Proposition 10 (Nombre de noeud et hauteur).

Un arbre binaire (quelconque) de hauteur h à n noeuds. On a :

$$n \leq 2^{h+1} - 1$$

Démonstration. Le cas d'égalité sera prouvé dans la suite du cours. Montrons ue $n \leq 2^{h+1} - 1$, par induction structurelle sur l'arbre.

- Initialisation : \perp est de hauteur -1 et a $n = 0$ noeuds. On a bien $0 \leq 2^{-1+1} - 1 = 1 - 1 = 0$, d'où l'initialisation.
- Hérédité : Soit $N(g, x, d)$ un arbre tel que g et d vérifient la propriété, montrons que l'arbre la vérifie aussi. On a :

$$h(g, x, d) = 1 + \max(h(g), h(d))$$

$$n(g, x, d) = 1 + n(g) + n(d)$$

D'où $h(g) \leq h(g, x, d) - 1$ et de même pour $h(d)$, et donc :

$$\begin{aligned} n(g, x, d) &= 1 + n(g) + n(d) \\ &\leq 1 + (2^{h(g)+1} - 1) + (2^{h(d)+1} - 1) && \text{par hypothèse d'induction} \\ &\leq -1 + 2^{h(g,x,d)} + 2^{h(g,x,d)} && \text{car } h(g) \leq h(g, x, d) - 1 \\ &\leq -1 + 2^{h(g,x,d)+1} \end{aligned}$$


D'où l'hérédité.

- Conclusion : On a prouvé que pour tout arbre binaire, $n \leq 2^{h+1} - 1$.

□

1.0.1 Implémentation


Voici des types pour implémenter de tels arbres en C et OCaml. On veut pour chaque noeud stocker son sous-arbre gauche, son étiquette (« key » en anglais) et son sous-arbre droit.



```

1 struct tree_s {
2     struct tree_s *left;
3     int key;
4     struct tree_s *right;
5 };
6 typedef struct tree_s tree;

```



```

1 type 'a tree =
2     Nil
3   | Node of 'a tree * 'a * 'a tree

```

Exemple. Les légendes deux arbres de la figure 9.14 sont des `int tree` avec le type ci-dessus (avec \perp pour Nil et N pour Node).

Lorsque les étiquettes de l'arbre sont en bijection avec $\llbracket 0; n \rrbracket$ (où n est le nombre de noeuds), on peut aussi utiliser une représentation par tableau : dans `arr[i]`, on stocke l'étiquette du parent du noeud d'étiquette i .

(a) Un arbre binaire aux étiquettes en bijection avec $\llbracket 0; 8 \rrbracket$.

(b) Le tableau correspondant.

FIGURE 9.16 – Représentation par tableau de parenté

Remarque. Cette représentation servira surtout en MPI, pour implémenter l'algorithme Union-Find.

1.1 Arbres binaires stricts

1.1.0 Définitions

Définition 11 (Arbres binaires stricts, V0).

Un arbre binaire strict est un arbre binaire non-vide où l'arité de tous les noeuds internes vaut exactement 2.

Définition 12 (Arbres binaires stricts, V1).

L'ensemble $\mathcal{S}(\mathcal{E})$ des arbres binaires stricts est défini inductivement par :

- Les feuilles sont des arbres : pour tout $x \in \mathcal{E}$, $F(x)$ est un arbre de $\mathcal{S}(\mathcal{E})$.
- Relier deux arbres stricts par une racine donne un arbre strict : pour tout $g, d \in \mathcal{S}(\mathcal{E})$ et $x \in \mathcal{E}$, le noeud $N(g, x, d)$ est un arbre de $\mathcal{S}(\mathcal{E})$.

Démonstration. Prouvons rapidement qu'un arbre défini par la déf V1 vérifie le fait que tout noeud interne est d'arité 2, par induction structurelle :

- Une feuille n'est pas un noeud interne.
- Un noeud $N(g, x, d)$ vérifie le fait que ni g ni d ne sont vides. Il a donc pour arité 2, et comme ce n'est pas une feuille c'est un noeud interne. Par hypothèse d'induction, les noeuds de ses sous-arbres vérifient également V1.

Ainsi, cette définition construit bien des arbres binaires stricts correspondant à la première définition.

Prouvons maintenant que tout arbre binaire de définition V0 correspond à la définition V1. On procède par récurrence forte sur la hauteur $h \in \llbracket 0; +\infty \rrbracket$, de A un arbre binaire non-vide dont tous les noeuds internes sont d'arité 2 :

- **Initialisation** : Si $h(A) = 0$, alors A est une feuille : c'est bien un cas de la définition V1.
- **Hérédité** : Soit $h > 0$ tel que la propriété est vraie pour tout $h' < h$ et A un arbre de hauteur h . Comme $h > 0$, A est de la forme $N(g, x, d)$ avec g et d de hauteur inférieure. Par hypothèse A est d'arité 2 et pas une feuille, g et d sont non-vides. On peut donc appliquer l'hypothèse de récurrence à g et d : g et d correspondent à la définition v1. $N(g, x, d)$ correspond alors à la définition V1. D'où l'hérédité.

□

Remarque.

- On peut aussi prendre la définition des arbres binaires et y imposer dans le cas inductif que si un des enfant est \perp , alors l'autre doit également l'être.
- On peut aussi les définir en prenant deux ensembles d'étiquettes : un pour les feuilles et un pour les noeuds internes. Encore une fois, plein de variantes existent !
- Dans certains cours, « arbre binaire » signifie « arbre binaire strict ». Ce n'est pas le cas ici, notamment parce que j'ai envie de dire qu'un arbre binaire de recherche est un arbre binaire.

(a) Un arbre binaire **pas** strict.

(b) Un arbre binaire strict.

FIGURE 9.17 – Illustration des arbres binaires stricts

Proposition 13 (Noeud et feuilles dans un arbre binaire strict).

Dans un arbre binaire strict à n_i noeuds internes et f feuilles, on a :

$$f = n_i + 1$$

Démonstration. Nous l'avons déjà prouvé sur les mobiles de Calder, qui sont des arbres binaires stricts déguisés. □

1.1.1 Implémentation

Pour implémenter de tels arbres en C, on reprend le type des arbres binaires et on code de sorte à ce que si l'un des deux enfants est vide alors l'autre aussi. En OCaml, on peut aussi procéder ainsi mais on peut également traduire la définition récursive :

```

1 type 'a strict =
2   Leaf of 'a
3   | Node of 'a strict * 'a * 'a strict

```



1.2 Arbres parfaits et complets

1.2.0 Définitions

Définition 14 (Arbre parfait, informel).

Un arbre parfait est un arbre dont tous les étages non-vides sont entièrement remplis.

Proposition 15 (Nombre de noeuds d'un arbre parfait).

Un arbre binaire de hauteur h est parfait si et seulement si il a $2^{h+1} - 1$ noeuds.

Démonstration. L'implication directe se traite à peu près comme dans la preuve de la propriété 10 ; sauf qu'ici on ne majore pas mais on montre l'égalité : on prouve par récurrence sur $p \in (\llbracket 0; h \rrbracket)$ que l'étage p contient exactement 2^p noeuds (tout noeud de l'étage $p + 1$ est enfant d'un noeud de l'étage p , et tout noeud de l'étage p a 2 enfants afin de remplir le plus possible l'étage $p + 1$; on conclut par récurrence). On termine l'implication directe en sommant les 2^p .

Le sens réciproque est lui aussi très similaire : un arbre de hauteur h a h étages qui contiennent chacun au plus 2^p noeuds. Si la somme vaut $2^{h+1} - 1$, alors toutes ces majorations sont atteintes : chaque étage p a 2^p noeuds, donc est entièrement plein. \square

Définition 16 (Arbre parfait, formel).

L'ensemble $\mathcal{P}(\mathcal{E})$ des arbres parfaits est défini inductivement par :

- L'arbre vide \perp est un arbre parfait de $\mathcal{P}(\mathcal{E})$.
- Relier deux arbres parfaits *de même hauteur* par une racine donne un arbre binaire : pour tout $g, d \in \mathcal{P}(\mathcal{E})$ tels que $h(g) = h(d)$ et $x \in \mathcal{E}$, le noeud $N(g, x, d)$ est un arbre parfait de $\mathcal{P}(\mathcal{E})$.

Démonstration. Montrons que cette seconde définition implique la première : prouvons qu'un arbre ainsi défini a bien tous ses étages entièrement remplis. Procédons par induction structurelle :

- Initialisation : Immédiat puisqu'il n'y a aucun étage non-vide.
- Hérédité : Soit $N(g, x, d) \in \mathcal{P}(\mathcal{E})$ tel que g et d vérifient la propriété.

L'étage de profondeur 0 de p est entièrement rempli puisqu'il contient 1 noeud : la racine.

Par définition de la profondeur, un noeud de profondeur p dans g est de profondeur $p + 1$ dans $N(g, x, d)$ (et de même pour d). Donc l'étage de profondeur $p > 0$ est constitué des noeuds de profondeur $p - 1$ dans g et dans d . Or par hypothèse d'induction, il y en a 2^{p-1} dans g et de même dans d . D'où l'hérédité.

FIGURE 9.18 – Schéma de la preuve

- Conclusion : on a prouvé vraie la propriété par induction structurelle.

Le sens réciproque est un jeu d'écriture qui est technique (prouver qu'un objet peut correspondre à une définition inductive est souvent compliqué à rédiger de manière claire, concise et rigoureuse). Je vais simplement montrer un morceau de la preuve : si un arbre non-vide est parfait (définition informelle), alors ses deux sous-arbres ont la même hauteur. Notons $N(g, x, d)$ un tel arbre et h sa hauteur. Par définition, $h(g) \leq h - 1$ et de même pour d . L'étage de profondeur h de $N(g, x, d)$ est plein par hypothèse, et composé de l'étage $h - 1$ de g et de l'étage $h - 1$ de d . Pour qu'il soit plein, il faut que ces deux étages des sous-arbres soient plein, et en particulier que ces deux sous-arbres soient de même hauteur $h - 1$. \square

Exemple.

FIGURE 9.19 – Un arbre parfait à 15 noeuds

Proposition 17 (Strict vs parfait).

Un arbre binaire strict A est parfait si et seulement si toutes ses feuilles sont à même profondeur $h(A)$

Démonstration. Prouvons d'abord le sens direct, par induction structurelle sur A un arbre binaire strict (V1) qui est également parfait.

- Initialisation : c'est immédiat pour la feuille.
- Hérédité : Soit A un arbre binaire strict parfait de la forme $N(g, x, d)$ avec g et d binaires stricts vérifiant la propriété. Or, les feuilles de A sont les feuilles de g et de d . Celles-ci sont à profondeur respectives $h(g)$ dans g et $h(d)$ dans d , donc $h(g) + 1$ et $h(d) + 1$ dans A . Or, par construction des arbres parfaits, $h(g) = h(d) = h(A) - 1$. D'où l'hérédité.
- Conclusion : on a prouvé par induction structurelle que le sens direct est de l'implication est vrai. Pour le sens réciproque, on procède de même. \square

Définition 18 (Arbre binaire complet).

Un arbre binaire est dit complet si tous es étages non-vides sont entièrement remplis sauf éventuellement le dernier, et que celui-ci est rempli de gauche à droite.

Remarque. Ce dernier point n'est pas présent dans toutes les définitions. Je le mets ici pour simplifier la définition des tas.

Exemple.

FIGURE 9.20 – Un arbre binaire complet de hauteur 3

Proposition 19.

Un arbre binaire parfait est complet.

Démonstration. Immédiat. □

1.2.1 Implémentation des arbres binaires complets

On peut représenter les arbres binaires complets inductivement, comme des arbres binaires. Mais on peut également les stocker dans un tableau : l'idée est d'écrire dans le tableau les étiquettes des noeuds de l'arbre dans le sens de lecture (de gauche à droite et de haut en bas).

(a) Sens de lecture.

(b) Représentation dans un tableau.

FIGURE 9.21 – Représentation dans un tableau de l'arbre binaire complet précédent.

Proposition 20 (Indexation d'un arbre binaire complet dans un tableau).

Avec cet représentation :

- L'étiquette de la racine est stockée dans la case d'indice 0.
- Les enfants gauches et droits du noeud d'indice i correspondent respectivement aux indices $2i + 1$ et $2i + 2$.
- Le parent du noeud d'indice i correspond à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$

Démonstration. Admis (pour l'instant). □

Remarque.

- D'expérience, vous avez du mal à retenir cette formule. Deux possibilités : faire un effort particulier et régulier pour l'apprendre par coeur (comme les formules trigos en maths), ou être capable de la retrouver très rapidement sur des dessins. Dans tous les cas, comme tout le cours, il faut la connaître !
- Certaines implémentations proposent de plutôt ne pas utiliser la case d'indice 0 et commencer à l'indice 1. Cela fait que les enfants sont $2i$ et $2i + 1$, et le parent $\left\lfloor \frac{i}{2} \right\rfloor$. C'est en général avec ces formules-ci que vous confondez.

Pro-tip : Si vous utilisez cette représentation dans un code, commencez par vous définir des fonctions `gauche`, `droite` et `parent` qui renvoie les bons indices. Sinon, vous ferez tout ou tard une erreur d'inattention.

1.3 Peigne

Définition 21.

Un **peigne gauche** (resp. **peigne droit**) est un arbre binaire dont tous les noeuds internes sont de la forme $N(g, x, \perp)$ (resp. $N(\perp, x, d)$).

Exemple.

(a) Peigne gauche.

(b) peigne droit.

FIGURE 9.22 – Les deux peignes

Remarque.

- Un peigne est, fondamentalement, une liste chaînée.
- Ces arbres correspondent à un déséquilibre maximal entre le sous-arbre gauche et le sous-arbre droit. Ils nous donnent ainsi des pires cas de beaucoup d’algorithmes sur les arbres.

1.4 Arbres quelconques

1.4.0 Transformation LCRS

Définition 22 (Transformation LCRS).

La transformation LCRS (« **L**eft **C**hild **R**ight **S**ibling ») permet de transformer un arbre quelconque en arbre binaire. Elle fonctionne ainsi :

- L’arbre binaire a les mêmes noeuds que l’arbre d’origine, mais pas les mêmes arêtes.
- La racine de l’arbre binaire est la racine de l’arbre d’origine.
- Pour les autres noeuds, on leur donne comme enfant gauche la racine de leur premier sous-arbre et comme enfant droit leur prochain adelphe (« le noeud qui est à droite d’eux dans l’arbre d’origine »).

Exemple.

(a) Un arbre et les liens utiles à LCRS

(b) Transformation LCRS de l’arbre

FIGURE 9.23 – Transformation LCRS

Proposition 23.

La transformation LCRS est injective, et on peut également coder la transformation inverse.

Démonstration. Injection : admis. Inverse : cf TP. □

Remarque.

- À la main sur des exemples, la transformation et la dé-transformation se calculent assez bien. Les programmer est un bon exercice étoilé de programmation.
- Prouver l'injection se fait assez bien, mais il faudrait pour cela formaliser la transformation par un programme. Nous le ferons peut-être en TD étoilé après le TP.

1.4.1 Type général

Voici un type OCaml pour des arbres binaires quelconques. L'idée est que pour chaque noeud on stocke la liste de ses sous-arbres.

```
1 type 'a tree =
2   Nil
3   | Node of 'a * 'a tree list
```



Exemple. Le code OCaml ci-dessous correspond à l'arbre de la figure 9.24 :

```
17 let leaf x = Node (x, [])
18
19 let general =
20   let t0 = Node (1, [Node (2, [leaf 3; leaf 4]);
21                       Node (5, [leaf 6])
22                     ]) in
23   let t1 = Node (7, [leaf 8; leaf 9; leaf 10; leaf 11]) in
24   let t2 = Node (12, [leaf 13;
25                      Node (14, [leaf 15; leaf 16]);
26                      leaf 17;
27                      Node (18, [leaf 19])
28                    ]) in
29   Node (0, [t0; t1; t2])
```

dfs.ml

FIGURE 9.24 – L'arbre general

2 Parcours d'arbres

Définition 24.

Un **parcours d'arbre** est la visite des noeuds d'un graphe dans un ordre particulier, pour leur appliquer un certain « traitement ».

Remarque.

- Par exemple, on peut vouloir afficher les noeuds, calculer l'expression associée à un arbre, reconstituer les appels récursifs qui ont lieu, etc. En fait, la plupart des algorithmes qui utilisent l'entiereté d'un arbre sont des parcours.
- Dans ce cours, on traitera les enfants « de gauche à droite » pour simplifier. On peut bien sûr généraliser.

2.0 Parcours en profondeur

Définition 25 (Parcours en profondeur).

Le **parcours en profondeur** (« Depth First Search ») consiste à se déplacer dans le graphe en descendant le plus possible jusqu'à une feuille, puis en remontant jusqu'à pouvoir descendre à nouveau, et ainsi de suite.

Exemple.

FIGURE 9.25 – Parcours en profondeur d'un arbre A_{dfs}

Remarque.

- C'est un parcours naturellement récursif : on descend dans les enfants les uns après les autres.
- L'exploration exhaustive consiste à parcourir en profondeur l'arbre des extensions des solutions partielles jusqu'à trouver une solution valide (et s'arrêter alors).

FIGURE 9.26 – Exploration exhaustive pour SUBSET-SUM avec $E = \{2; 8; 3\}$ et $t = 5$

- Comme dans le retour sur trace, la « remontée » est correspond par le fait de quitter les appels récurifs : il n’y a pas à la coder.

Voici une façon d’implémenter un tel parcours sur un arbre d’arité quelconque⁶ (cf fin de la section précédente). On va y appliquer la fonction `visite` à chacun des noeuds. On y utilise la fonction `List.iter` qui permet d’évaluer une fonction sur tous les éléments d’une liste :

```

8  (** Applique la fonction f à chacune des étiquettes *)
9  let rec dfs visite tree =
10     match tree with
11     | Nil -> ()
12     | Node (etq, children) ->
13         let _ = visite etq in
14         List.iter (dfs visite) children

```



Exemple. Avec `general` l’arbre de la fin de la section précédente, l’expression `dfs (Printf.printf "%d ") general` a pour effet secondaire d’afficher 0 ... 19.

Remarque. Dans un arbre binaire :

- Le parcours en profondeur est plus simple à écrire (on peut éviter le `List.iter`).
- On passe trois fois par chaque noeud : avant le parcours de son enfant gauche, entre le parcours de ses deux enfants, et après le parcours de son enfant droit. Cela mène à trois notions : parcours préfixe, infix et postfixe.

2.0.0 Ordre préfixe

Définition 26 (Ordre préfixe pour un DFS).

Dans un parcours en profondeur d’un arbre, l’**ordre préfixe** consiste à effectuer le traitement d’un noeud *avant* d’explorer ses enfants.

Exemple.

- **Cet ordre est défini peu importe l’arité, pas uniquement pour les arbres binaires !**
- La fonction `dfs` précédente affiche l’étiquette d’un noeud avant d’explorer récursivement les enfants ; c’est donc un parcours en profondeur préfixe.
- Le parcours préfixe de l’arbre A_{dfs} traite les noeuds dans l’ordre 1, 2, 4, 5, 8, 9, 3, 6, 7. Notez que l’on visite un noeud « lorsque le trait du chemin passe à gauche ».

Dans un arbre binaire, un parcours en profondeur préfixe qui applique f à tous les éléments peut s’écrire ainsi :

```

44 let rec prefixe visite tree =
45     match tree with
46     | Nil -> ()
47     | Node (gauche, x, droite) ->
48         let _ = visite x in
49         let _ = prefixe visite gauche in
50         prefixe visite droite

```



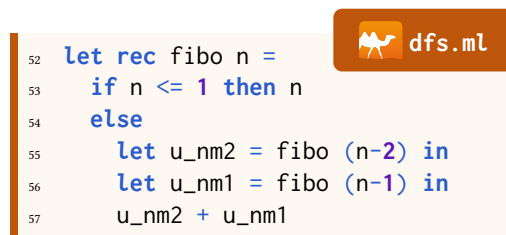
6. Bien sûr, les valeurs renvoyées et d’autres points peuvent changer selon le problème à résoudre. Typiquement, ignorer la sortie de `visite` n’est pas toujours pertinent.

Proposition 27 (Dates d'ouverture dans une fonction récursive).

Lors de l'exécution d'une fonction récursive, l'*empilement* des blocs d'activation des appels sur la pile mémoire correspond à un parcours en profondeur préfixe de l'arbre d'appels.

Rappelons que le bloc d'activation d'une fonction est créé (et donc empilé) lorsque l'appel débute, et supprimé (donc dépilé) lorsqu'il se termine.

Exemple.



```

52 let rec fibo n =
53   if n <= 1 then n
54   else
55     let u_nm2 = fibo (n-2) in
56     let u_nm1 = fibo (n-1) in
57     u_nm2 + u_nm1

```

(a) Une fonction récursive `fibo`

(b) Arbre d'appels de `fibo 3`

(c) Les états successifs de la pile mémoire

FIGURE 9.27 – Arbre d'appel et pile mémoire

Remarque. Dans utop, `#trace fibo;;` permet d'avoir un affichage des ouvertures et fermetures des appels récursifs.

2.0.1 Parcours infixe

Définition 28 (Ordre infixe pour un DFS).

Dans un parcours en profondeur d'un arbre binaire, l'**ordre infixe** consiste à effectuer le traitement d'un noeud *entre* les traitements de ses deux enfants.

Exemple.

- On peut essayer de généraliser aux arbres quelconques en disant que le traitement a lieu entre le traitement de deux enfants (avec un cas particulier pour les noeuds d'arité 1).
- Le parcours infixe de A_{dfs} traite les noeuds dans l'ordre 4, 2, 8, 5, 9, 1, 6, 3, 7. Notez que l'on visite un noeud « lorsque le trait du chemin passe *en dessous* ».
- Les mouvements de la résolution récursive des tours de Hanoï ont lieu dans l'ordre infixe : d'abord on fait un premier appel récursif, puis on effectue un mouvement, puis un second appel récursif.
- Nous verrons que le parcours infixe tri un arbre binaire de recherche.

Dans un arbre binaire, un parcours infixe ressemble à ceci :

```

60 let infixe visite tree =
61   match tree with
62   | Nil -> ()
63   | Node (gauche, x, droite) ->
64     let _ = prefixe visite gauche in
65     let _ = visite x in
66     prefixe visite droite

```



2.0.2 Parcours postfixe

Définition 29 (Ordre postfixe pour un DFS).

Dans un parcours en profondeur d'un arbre, l'**ordre postfixe** consiste à effectuer le traitement d'un noeud *après* les traitements de deux enfants.

Exemple.

- Le parcours postfixe sur A_{dfs} traite les noeuds dans l'ordre 4, 8, 9, 5, 2, 6, 7, 3, 1. Notez que l'on visite un noeud « lorsque le trait du chemin passe à droite ».
- La fonction ci-dessous qui évalue une expression arithmétique (cf section 1) effectue les calculs dans l'ordre postfixe :

```

27 let rec eval e =
28   match e with
29   | Int n -> n
30   | Var _ -> failwith "Je ne connais pas la valeur des variables."
31   | Fun (f, e0, e1) -> f (eval e0) (eval e1)

```



Proposition 30 (Dates de fermeture dans une fonction récursive).

Lors de l'exécution d'une fonction récursive, le *dépilement* des blocs d'activation des appels sur la pile mémoire correspond à un parcours en profondeur postfixe de l'arbre d'appels.

Exemple. Cf figure 9.27

2.0.3 Écriture itérative

Il est possible d'écrire un parcours en profondeur d'arbre de manière itérative. Cela peut permettre un contrôle un peu plus fin⁷, mais est plus difficile d'écrire sans bug. L'idée est de remplacer la récursivité par une pile :

Algorithme 12 : Parcours en profondeur d'arbre, itératif

Entrées : A un arbre; et $VISITE$ une fonction de traitement

```

1   $P \leftarrow$  pile vide
2  EMPILER la racine de  $A$  sur  $P$ 
3  tant que  $P$  est non-vide faire
4     $n \leftarrow$  DÉPILER( $P$ )
5     $VISITE(n)$ 
6    pour chaque enfant  $e \neq \perp$  de  $n$  faire
7      EMPILER  $e$  sur  $P$ 

```

7. Et encore, l'utilisation d'exceptions dans une fonction récursive permet d'implémenter des comportements exceptionnels.

Remarque. Cette version-ci correspond à un DFS préfixe, on peut bien sûr l'adapter aux autres ordres.

Exemple.

(a) L'arbre parcouru

(b) États de la pile P en débuts d'itération

FIGURE 9.28 – Algorithme de parcours en profondeur itératif

2.1 Parcours en largeur

Définition 31 (Parcours en largeur).

Le **parcours en largeur** (« Breadth First Search ») consiste à parcourir les sommets par profondeur croissante.

Exemple.

FIGURE 9.29 – Parcours en largeur d'un arbre A_{bfs}

2.1.0 Écriture itérative

Le parcours en largeur n'a pas, contrairement au DFS, une écriture récursive simple. En fait, il n'est pas basé sur un ordre LIFO (comme la récursivité) mais FIFO :

Algorithme 13 : Parcours en largeur d'arbre

Entrées : A un arbre; et $VISITE$ une fonction de traitement

```

1  $F \leftarrow$  file vide
2 ENFILER la racine de  $A$  dans  $F$ 
3 tant que  $F$  est non-vide faire
4    $n \leftarrow$  DÉFILER( $F$ )
5    $VISITE(n)$ 
6   pour chaque enfant  $e \neq \perp$  de  $n$  faire
7     ENFILER  $e$  dans  $F$ 
```

Exemple.

(a) L'arbre parcouru

(b) États de la file F en débuts d'itération

FIGURE 9.30 – Algorithme de parcours en largeur

2.1.1 (Mi-HP) Preuve du pseudo-code

Prouvons que ce pseudo-code est bien un parcours en largeur. C'est un bon entraînement⁸.

Théorème 32.

Le pseudo-code proposé est bel et bien un parcours en largeur, c'est à dire qu'il parcourt les sommets par profondeur croissante.

Démonstration. Le pseudo-code utilise une file et n'effectue que 4 opérations qui appartiennent au type abstrait : obtenir une file vide, tester si une file est vide, ENFILER et DÉFILER. Nous allons la considérer implémentée à l'aide de deux piles : cette implémentation était totalement correcte, faire ce choix ne modifiera pas le déroulé de l'algorithme. De plus, nous allons introduire une variable p qui stocke la profondeur en cours. Celle-ci n'interagira pas avec les autres variables, aussi l'ajouter ne modifie pas non plus le déroulé. On obtient le pseudo-code suivant :

```

1  $P_{in} \leftarrow$  pile vide
2  $P_{out} \leftarrow$  pile vide
3  $p \leftarrow -1$ 
4 EMPILER la racine de  $A$  sur  $P_{in}$ 
5 tant que  $P_{out}$  ou  $P_{in}$  est non-vide faire
6   si  $P_{out}$  est vide alors
7     RENVERSER  $P_{in}$  sur  $P_{out}$ 
8      $p \leftarrow p + 1$ 
9    $n \leftarrow$  DÉPILER( $P_{out}$ )
10  VISITE( $n$ )
11  pour chaque enfant  $e \neq \perp$  de  $n$  faire
12    EMPILER  $e$  sur  $P_{in}$ 

```

Pour prouver la correction partielle de l'algorithme, nous allons prouver les invariants suivants :

- I_{out} : « Tous les noeuds de P_{out} sont à profondeur p . »
- I_{in} : « Tous les noeuds de P_{in} sont à profondeur $p+1$. »

Expliquons tout d'abord à quoi servent ces invariants. À chaque itération, le sommet visité est un sommet de P_{out} donc d'après I_{in} à profondeur p ; or comme p est croissant on en déduit la correction de l'algorithme⁹. I_{in} sert à prouver I_{out} .

⁸. La somme des entiers ça va bien 5 minutes.

⁹. On pourrait formaliser cela avec des invariants supplémentaires... je préfère alléger le tout. Le coeur de la preuve porte sur I_{in} et I_{out} .

Avant la 1ère itération : I_{out} est trivial. De plus, P_{in} contient un seul noeud : la racine, qui est à profondeur 0. Or, $p = -1$: d'où l'initialisation de I_{in} .

Conservation : Supposons les deux invariants vrais au début d'une itération, montrons-les vrais à la fin. On utilisera les notations prime.

On va distinguer deux cas :

- Si P_{out} n'est pas vide. Alors on ne rentre pas dans le Si des lignes 6-8. Donc $p' = p$.
La seule modification faite à P_{out} est un DEPILE : ainsi, tous les éléments de P_{out}' étaient dans P_{out} . Comme $p = p'$, I_{out} entraîne I_{out}' .
Enfin, les seules modifications faites à P_{in} sont les EMPILE des enfants de noeud n . Or, noeud est à profondeur p donc ses enfants à $p+1$. Ainsi, les éléments de P_{in}' sont soit ces enfants à profondeur $p+1$, soient des éléments de P_{in} qui sont à profondeur $p+1$ d'après I_{in} . Comme $p = p'$, on conclut de même que I_{in}' est vrai.
- Si P_{out} est vide. Alors on rentre dans le Si, et donc :
 - $p' = p+1$
 - Après la ligne 7, le contenu de P_{out} est l'ancien contenu de P_{in} (en ordre renversé). En particulier, d'après I_{in} ce sont des noeuds de profondeur $p+1 = p'$. De même que dans le cas précédent, on en déduit I_{out}' .
 - Après la ligne 7, P_{in} est vidée (et ne sera re-remplie qu'à la 12).
 Enfin, P_{in}' contient uniquement les enfants de du noeud n . Or, n est à profondeur p' , donc ses enfants à $p'+1$: d'où I_{in}' .

Conclusion : On a prouvé que les quatre invariants étaient vrais à tout moment de l'exécution de la boucle Tant Que. Comme annoncé précédemment, on déduit de I_{out} et de la monotonie de p que les noeuds sont visités par profondeur croissante. □

2.1.2 (Mi-HP) Preuve, suite et fin

Je vous ai eu ! Nous n'avons pas prouvé *tout* ce qu'il fallait prouver. Nous avons prouvé que les sommets visités le sont par profondeur croissante... mais pas qu'ils sont tous visités. Prouvons cela :

Démonstration. Les visites correspondent aux défilement : on veut en fait prouver que tous les sommets sont défilés. Mais puisque la boucle termine lorsque la file est vide, cela revient à prouver que tous les sommets sont enfilés.

Montrons par récurrence sur la profondeur p d'un noeud quelconque qu'il est bien enfilé.

Initialisation : le seul noeud à profondeur $p = 0$ est la racine. Elle est enfilée à la ligne 2 (ou 4 dans la version modifiée).

Hérédité : supposons que tous les noeuds à profondeur p sont enfilés (peu importe quand), et montrons que tous ceux à profondeur $p + 1$ le sont. Un noeud à profondeur $p + 1$ a un parent à profondeur p , qui a été enfilé. En particulier, ce parent sera défilé. Or, quand on défile un noeud, on enfile tous ses enfants (ligne 11-12). Ainsi, les noeuds à profondeur $p + 1$ seront bien enfilés, d'où l'hérédité.

Conclusion : tous les noeuds sont bien enfilés à un moment ou un autre. On en déduit comme annoncé qu'ils seront bien tous visités. □

3 (HP) Propriétés avancées

3.0 Lemme de lecture unique

Dans le TP sur la notation polonaise inversée (RPN), j'ai affirmé que toute expression pouvait s'écrire ainsi, et sous-entendu que l'écriture d'une notation était unique. Or, la RPN correspond au parcours postfixe de l'arbre d'une expression : autrement dit, le parcours postfixe d'une expression définit.

On appelle cette propriété un lemme de lecture unique :

Théorème 33 (Lemme de lecture unique).

Un arbre (quelconque) est entièrement défini par son parcours préfixe, si celui-ci contient les arités des noeuds en plus des étiquettes.

De même avec le parcours postfixe.

Démonstration. Admis. □

Pour comprendre un peu mieux cela, essayons de reconstruire un arbre à partir d'un tel parcours. On note le résultat du parcours ainsi :

- $prefix(N(x, []))$ est $(x, 0)$
- $prefix(N(x, [a_1; \dots; a_k]))$ est $(x, k) \ prefix(a_1) \ \dots \ prefix(a_k)$

Remarque. J'ai ajouté des parenthèses et des virgules dans (x, k) ; mais on peut les enlever : dans la notation finale alternent étiquette et arité. En particulier, on ne va pas « tricher » en utilisant des parenthèses imbriquées pour reconstruire l'arbre.

Exemple.

FIGURE 9.31 – Arbre correspondant à « 1 3 2 2 5 0 6 0 3 0 4 1 7 0 »

Remarque. La preuve complète repose sur un lemme : « en tout point de la notation débute un unique facteur qui correspond à un arbre ». Mais comme toutes les preuves de lemme de lecture unique, c'est plus lourd que pédagogique... le TP RPN et la figure 9.31 ci-dessus nous suffiront largement.

3.1 Dénombrement des arbres binaires stricts

Dénombrons les arbres binaires stricts. On ignore le contenu des étiquettes : on s'intéresse uniquement à la « forme » de l'arbre binaire strict. On va montrer que leur nombre est lié à la suite de Catalan :

Définition 34 (Nombre de Catalan).

La suite $(c_n)_{n \in \mathbb{N}}$ des nombres de Catalan est définie par récurrence par :

- $c_0 = 1$
- $c_{n+1} = \sum_{k=0}^n c_k c_{n-k}$

Remarque. Les nombres de Catalan sont aussi connus pour être le nombre de mots de Dyck, c'est à dire le nombre de façon de placer n parenthèses ouvrantes et n fermantes de sorte à ce que le tout soit bien parenthésé.¹⁰

¹⁰. C'est un exercice classique de dénombrement dans des concours difficiles ; je vous laisse vous renseigner si cela vous intéresse.

(a) Les $c_3 = 5$ arbres binaires stricts à 3 noeuds internes

(b) Les $c_4 = 14$ arbres binaires stricts à 4 noeuds internes

FIGURE 9.32 – Dénombrement pour $n = 3$ et $n = 4$

Théorème 35.

Il y a c_n arbres binaires stricts non-étiquetés à n noeuds internes.

Remarque. J'insiste, *internes* !

Démonstration. Procédons par récurrence sur le nombre de noeuds internes.

- Il y a exactement 1 arbre binaire strict à 0 noeuds internes : la feuille.
- Supposons la propriété vraie jusqu'au rang n , montrons-la vraie au rang $n + 1$. Un arbre binaire strict à $n + 1$ noeuds internes est constitué de :
 - une racine (qui est un noeud interne).
 - un sous-arbre gauche possédant k noeuds internes avec $0 \leq k \leq n$ (il ne peut pas y en avoir $n + 1$ car la racine est un noeud interne).
 - un sous-arbre droit qui a donc $n + 1 - 1 - k = n_k$ noeuds internes.

Par hypothèse de récurrence, pour chaque k il y a c_k façons de construire un sous-arbre gauche, c_{n-k} de construire un sous-arbre droit, et donc $c_k c_{n-k}$ arbres binaires stricts à $n + 1$ noeuds internes dont le sous-arbre gauche a k noeuds internes. En sommant sur k , on obtient l'hérédité. \square

La suites des nombres de Catalan croit vite, aussi ce théorème nous apprend qu'il y a *beaucoup* binaires stricts, et donc encore plus d'arbres binaires.

Une autre application intéressante de ce théorème est qu'elle peut aider à calculer une expression plus agréable pour c_n :

Lemme 36.

Pour tout $n \in \mathbb{N}$:

$$(n + 2)c_{n+1} = 2(2n + 1)c_n$$

Démonstration. Nous allons interpréter les deux membres de l'égalité comme le cardinal de deux ensembles en bijection :

- À gauche : $(n + 2)c_{n+1}$ est le nombre de paires (A, f) avec A un arbre binaire strict à $n + 1$ noeuds internes et f une feuille de A . En effet, il y a c_{n+1} possibilités pour A , et comme A a $(n + 1) + 1$ feuilles, il reste $n + 2$ possibilités pour f .
- À droite : $2(2n + 1)c_n$ est le nombre de triplets (B, x, ϵ) où B est un arbre strict à n noeuds internes (c_n possibilités), x un noeud quelconque de B ($n + (n + 1)$ possibilités), et $\epsilon \in \{\leftarrow, \rightarrow\}$ (2 possibilités).

Montrons que ces deux ensembles sont en bijection en définissant une fonction bijective φ qui à (A, f) associe (B, x, ϵ) :

- B est obtenu à partir de A en supprimant la feuille f ainsi que son parent. L'adelphe de f est « remonté » à la place de son parent.
- x est le noeud qui a été remonté.
- ϵ vaut \leftarrow si f était enfant gauche, et \rightarrow si elle était enfant droit.

FIGURE 9.33 – Définition de φ

Pour prouver qu'il s'agit d'une bijection, exhibons sa réciproque φ^{-1} . À un triplet (B, x, ϵ) elle associe :

- A est obtenu en insérant un noeud y entre x et son parent. Si $\epsilon = \leftarrow$, on crée une nouvelle feuille enfant gauche de y et le sous-arbre de x devient enfant droit de y ; sinon symétrique.
- f est la feuille qui a été créée.

On peut vérifier qu'il s'agit bien d'une réciproque, d'où la bijection¹¹. □

Ce lemme, obtenu grâce aux arbres, permet d'obtenir une expression non-récursive des nombres de Catalan :

Proposition 37.

Pour tout $n \in \mathbb{N}$:

$$c_n = \frac{1}{n+1} \binom{2n}{n}$$

Démonstration. Procédons par récurrence sur \mathbb{N} :

- Pour $n = 0$, on a $c_0 = 1 = \frac{1}{0+1} \binom{0}{0}$.
- Supposons la propriété vraie au rang n , montrons-la vraie au rang $n + 1$. On a :

$$\begin{aligned} (n+2)c_{n+1} &= 2(2n+1)c_n && \text{d'après le lemme} \\ &= \frac{2(2n+1)}{n+1} \binom{2n}{n} && \text{par H.R.} \end{aligned}$$

Or :

$$\begin{aligned} \binom{2n+2}{n+1} &= \frac{(2n+2)!}{(n+1)!(n+1)!} \\ &= \frac{(2n+2)(2n+1)(2n)!}{(n+1)^2 n!} \\ &= \frac{(2n+2)(2n+1)}{(n+1)^2} \binom{2n}{n} \end{aligned}$$

Donc :

$$\begin{aligned} (n+2)c_{n+1} &= \frac{2(2n+1)}{n+1} \binom{2n}{n} \\ &= \frac{2(2n+1)}{n+1} \frac{(n+1)^2}{(2n+2)(2n+1)} \binom{2n+2}{n+1} \\ &= \binom{2n+2}{n+1} \end{aligned}$$

□

11. C'est très simple de s'en convaincre; et plus embêtant qu'autre chose d'en donner une preuve formelle (on montre que $\varphi \circ \varphi^{-1} = \varphi^{-1} \circ \varphi = Id$). Cette partie étant déjà du bonus hors-programme, je me limite au plus intéressant.

4 Arbres binaires de recherche

Dans toute cette section, on considère des arbres binaires dont les étiquettes appartiennent à un ensemble *totale*ment ordonné (\mathcal{E}, \leq) . On rappelle que g, x, d est un raccourci pour $N(g, x, d)$.

Les arbres binaires de recherche sont une structure de donnée qui vise à stocker des éléments et pouvoir efficacement :

- Rechercher si un élément est dedans.
- Insérer un nouvel élément dedans.
- Supprimer un élément dedans.

En bonus, on va également obtenir une recherche efficace du minimum et du maximum d'un ensemble.

Remarque.

- Précisons tout de suite : les arbres binaires de recherche en eux-même ne sont pas efficaces. Pour garantir leur efficacité, il faut garantir qu'ils ne soient pas trop déséquilibrés : c'est à cela que serviront les arbres rouge-noir (ARN).
- Les ARN sont assez efficace pour être utilisés dans des sections critiques du noyau Linux ! L'ordonnanceur (le programme qui décide quelle tâche faire avancer en priorité) est codé à l'aide d'un ARN.

4.0 Définition, caractérisation

Pour les preuves, il sera pratique d'avoir une notation pour l'ensemble des étiquettes des noeuds d'un arbre.

Définition 38 (Ensemble d'étiquettes).

On définit l'ensemble $S(A)$ des étiquettes des noeuds d'un arbre binaire A par :

- $S(\perp) = \emptyset$
- $S(g, x, d) = \{x\} \cup S(g) \cup S(d)$

Définition 39 (Arbres binaires de recherche).

L'ensemble $ABR(\mathcal{E})$ des **arbres binaires de recherche** sur \mathcal{E} est défini inductivement par :

- $\perp \in ABR(\mathcal{E})$
- $N(g, x, d) \in ABR(\mathcal{E})$ lorsque :
 - $g \in ABR(\mathcal{E})$ et $d \in ABR(\mathcal{E})$
 - et $\max(S(g)) \leq x \leq \min(S(d))$

Remarque.

- Pour simplifier, dans la suite de ce cours on supposera les étiquettes deux à deux distinctes. En particulier, les inégalités larges seront strictes. De plus, cela permet de confondre sans ambiguïté un noeud et son étiquette, ce qui allège des notations.
- En maths, on pose $\max \emptyset = -\infty$ et $\min \emptyset = +\infty$. En informatique, dans les ABR on considèrera similairement que pour tout $x \in \mathcal{E}$, $\max \emptyset < x < \min \emptyset$.
- La définition d'un ABR n'est **pas** locale : il ne suffit pas que chaque noeud ait les bonnes comparaisons avec les racines de ses enfants !! La figure 9.34 ci-dessous donne un contre-exemple.

Exemple.

- (a) Un ABR pour {3; 5; 6; 7; 8; 9; 13; 20}
- (b) **Pas** un ABR, malgré le fait que chaque noeud est supérieur à la racine de gauche et inférieur à celle de droite

FIGURE 9.34 – Définition d'un arbre binaire de recherche

Théorème 40 (Caractérisation des ABR).

Soit A un arbre. On a :

A est un ABR si et seulement si son parcours infixe est trié.

Démonstration. On notera $infixe(A)$ la suite des noeuds visités par le parcours infixe d'un arbre A . Ainsi $infixe(\perp) = \emptyset$ et $infixe(g, x, d)$ est la concaténation de $infixe(g)$, x et $infixe(d)$; que l'on notera $infixe(g) @ x @ infixe(d)$ pour simplifier.

Montrons d'abord l'implication directe. On montre par induction structurale sur A un ABR que $infixe(A)$ est trié.

- Initialisation : Si A est vide, c'est immédiat.
- Hérédité : Si $A = N(g, x, d)$ avec g et d vérifiant l'implication, montrons que $infixe(g, x, d)$ est trié. On a :

$$infixe(A) = \underbrace{infixe(g)}_{\text{trié par H.I.}} @ x @ \underbrace{infixe(d)}_{\text{trié par H.I.}}$$

Par H.I., les deux sous-parcours infixes sont triés par ordre croissant. Or, $\max S(g) < x < \min S(d)$; donc x est bien placé. Il s'ensuit que le tout est trié, d'où l'hérédité.

- Conclusion : Le parcours infixe d'un ABR est trié.

Montrons maintenant l'implication réciproque, par induction structurale sur A un arbre binaire.

- Initialisation : Si A est vide, c'est immédiat.
- Hérédité : Si $A = N(g, x, d)$ est un arbre binaire avec g et d vérifiant l'implication réciproque, montrons que A est un ABR. Comme $infixe(A)$ est trié, ses sous-suites $infixe(g)$ et $infixe(d)$ le sont aussi : donc par H.I., g et d sont des ABR. Mais comme $infixe(A)$ est trié, $\max S(g) < x < \min S(d)$. Donc A est un ABR, d'où l'hérédité.
- Conclusion : Un arbre binaire trié par parcours infixe est un ABR.

□

Remarque. Une preuve de l'équivalence par induction structurale aurait aussi été possible.

4.1 Opérations

4.1.0 Recherche

Les arbres binaires de recherche sont pensés pour faire des dichotomies dedans :

Exemple.

FIGURE 9.35 – Recherche dans un ABR

Cela donne le OCaml suivant

```

40  (** Renvoie [true] ssi [x] est présent dans [arbre] *)
41  let rec search x arbre =
42    match arbre with
43    | Nil -> false
44    | Node (g, etq, d) ->
45      if x = etq then true
46      else if x < etq then search x g
47      else search x d

```



Proposition 41 (Complexité de la recherche dans un ABR).

La recherche dans un ABR de hauteur h s'effectue en $\Theta(h)$ comparaisons.

Démonstration. La recherche se déplace dans un chemin de la racine à une feuille, en effectuant 2 comparaisons¹² par noeuds. D'où une complexité en $O(h)$ comparaisons.

Chercher x dans le peigne gauche ayant pour étiquettes $x + h \dots x + 1$ (de la racine à la feuille) prouve le besoin de $\Omega(h)$ comparaisons.

FIGURE 9.36 – Un pire cas de la recherche

□

12. En fait, en codant bien, c'est une seule : une fonction de comparaison totale (comme `Stdlib.compare` en OCaml) a trois issues possibles : « strictement inférieur », « égal » et « strictement supérieur ». On peut avec cela fusionner le test d'égalité et de strictement inférieur en un seul.

Remarque.

- On peut formaliser cette preuve ; mais elle est si simple que le gain n'est pas clair.
- Il est très important que l'ordre utilisé soit *total* : sinon, on ne sait pas dans quel sous-arbre aller !

4.1.1 Insertion

Pour insérer dans un ABR, on insère le nouvel élément au niveau des feuilles. Autrement dit, on recherche l'élément dans l'arbre : s'il est déjà présent, rien à faire¹³ ; sinon on l'insère à la place du \perp où la recherche termine.

Exemple.

FIGURE 9.37 – Insertions successives dans un ABR

Cela donne le code OCaml ci-dessous. On propose ici une implémentation fonctionnelle des ABR, donc l'insertion ne modifie pas l'arbre mais en renvoie un nouveau où l'insertion a eu lieu.

```

50 (** Renvoie l'abr obtenu en insérant [x] dans [arbre] *)
51 let rec insere x arbre =
52   match arbre with
53   | Nil ->
54     (* Créer la feuille contenant x *)
55     Node(Nil, x, Nil)
56   | Node (g, etq, d) ->
57     if x = etq then
58       (* Rien à faire, x est déjà là *)
59       arbre
60     else if x < etq then
61       (* Insérer à gauche *)
62       Node (insere x g, etq, d)
63     else
64       (* Insérer à droite *)
65       Node (g, etq, insere x d)

```



Proposition 42 (Complexité de l'insertion dans un ABR).

L'insertion dans un ABR de hauteur h s'effectue en $\Theta(h)$ comparaisons.

Démonstration. C'est la même que pour la recherche. □

13. Rappel : on s'est placé dans le cadre simplificateur des ABR avec éléments deux à deux distincts.

Proposition 43 (Variation de la hauteur lors de l'insertion dans un ABR).

Lors de l'insertion dans un ABR, la hauteur augmente au plus de 1.

Démonstration. L'insertion rajoute une feuille, qui rallonge donc un chemin de la racine aux feuilles de 1. Si ce chemin définissait la hauteur elle augmente de 1 ; sinon elle reste inchangée. \square

Remarque. On peut aussi prouver par induction structurelle sur A que $h(\text{insere}(x, A)) \leq h(A) + 1$

4.1.2 Suppression

Cette opération est plus délicate. On distingue quatre cas pour supprimer x de A :

- (i) Si $x \notin A$: rien à faire.
- (ii) Si x est une feuille : il suffit de la supprimer.
- (iii) Si x est un noeud interne avec un seul enfant : il suffit de le supprimer et de remonter son enfant à sa place.

FIGURE 9.38 – Suppression dans le cas où x a un seul enfant

- (iv) Sinon : x est un noeud interne avec deux enfants : c'est plus compliqué. On va se ramener au point précédent. Pour cela, on va écraser l'étiquette x avec l'étiquette du maximum de l'enfant gauche de x , puis aller ce dernier.

Proposition 44 (Maximum dans un ABR).

Le maximum d'un ABR A :

- n'a pas d'enfant droit.
- peut être trouvé en se déplaçant toujours à droite dans l'arbre.

Démonstration. • Considérons $N(g, m, d)$ le noeud qui contient le maximum. Si d est non-vide, tout noeud qu'il contient est strictement supérieur à m .

- Procédons par induction structurelle sur un abr A non-vide :
 - Si $A = N(g, x, \perp)$, par définition des ABR $\max S(g) < x$ donc x est le maximum de A .
 - Sinon, $A = N(g, x, d)$ avec $d \neq \perp$, par définition des ABR $\max S(g) < x < \min S(d)$. Il s'ensuit que $\max S(A) = \max S(d)$.

\square

Remarque.

- On a une propriété similaire pour le minimum.
- En corollaire, il est toujours simple de supprimer le maximum : on applique le cas

On peut en déduire une (mauvaise) façon de supprimer un noeud $N(g, x, d)$: le supprimer, remonter à sa place g et brancher d sous le maximum de g .

FIGURE 9.39 – Suppression dans un ABR, version peu efficace

Le problème de cette façon de faire est qu'elle déséquilibre fortement l'arbre, et risque de faire exploser la hauteur. Or, la hauteur est le paramètre déterminant dans la complexité des opérations ! À la place, nous allons plutôt procéder ainsi : remplacer x par le maximum du sous-arbre gauche, puis supprimer celui-ci.

FIGURE 9.40 – Suppression dans un ABR, version efficace !

Cela correspond au code OCaml ci-dessous, où `maximum` calcule le maximum d'un ABR :

```

85 (** Renvoie l'abr obtenu en supprimant [x] de [arbre] *)
86 let rec supprime x arbre =
87   match arbre with
88   | Nil -> Nil
89   | Node (g, etq, Nil) ->
90     if x = etq then g else Node (supprime x g, etq, Nil)
91   | Node (Nil, etq, d) ->
92     if x = etq then d else Node (Nil, etq, supprime x d)
93   | Node (g, etq, d) ->
94     if x = etq then
95       let maxi_g = maximum g in
96       Node (supprime maxi_g g, maxi_g, d)
97     else if x < etq then
98       Node (supprime x g, etq, d)
99     else
100      Node (g, etq, supprime x d)

```



Remarque. On pourrait rendre ce code plus efficace en faisant une fonction qui simultanément trouve et le maximum d'un ABR. Cela permettrait de parcourir une seule fois le sous-arbre gauche au lieu de deux dans les lignes 95-96.

Proposition 45 (Complexité de la suppression dans un ABR).

La suppression dans un ABR s'effectue en $\Theta(h)$ comparaisons.

Démonstration. Informellement, l'algorithme est en $O(h)$ car la suppression consiste en trois phases :

- Trouver x (complexité d'une recherche)
- Puis trouver le maximum de son sous-arbre gauche (linéaire en sa hauteur)
- Puis supprimer celui-ci (cas (iii) de la suppression, linéaire en la hauteur)

Pour prouver $\Omega(h)$, on considère exactement le même cas que pour la recherche ou l'insertion. \square

Exercice. Formaliser la preuve du $O(h)$. Pour cela, commencer par prouver que dans le cas (iii) de la suppression, la complexité est bel et bien linéaire en la hauteur. Ensuite, écrire l'équation de récurrence vérifiée par la complexité, et conclure (on peut admettre que la complexité est croissante en h pour simplifier).

4.1.3 Rotation

Les rotations gauches et droites sont des transformations d'un arbre binaire qui sont parfois utilisés dans certains algorithmes sur les ABR (notamment dans les ARN). J'ai réussi à écrire ce cours sans les utiliser, mais il est bon de savoir ce dont il s'agit :

Définition 46 (Rotation gauche et droite).

Les rotations gauche et droite d'un arbre sont les transformations suivantes :

FIGURE 9.41 – Rotation gauche et droite

Remarque.

- La rotation gauche ne peut pas s'appliquer à n'importe quel arbre (il faut que le sous-arbre gauche ait la bonne forme), et de même pour la rotation droite.
- La rotation est définie sur tout arbre binaire, pas uniquement sur les ABR.

Proposition 47.

Une rotation s'effectue en temps constant.

Démonstration. Implémentation-dépendant. \square

Proposition 48.

A est un ABR si et seulement si sa rotation (gauche ou droite) est un ABR.

Démonstration. Assez agréable est d'utiliser le parcours infixe. Laissé en exercice. \square

4.1.4 Tri par ABR

On déduit de ce qui précède un algorithme pour trier un tableau ou une liste via un ABR :

Algorithme 14 : Tri par ABR

Entrées : x_0, \dots, x_{n-1} à trier

Sorties : Une version triée de l'entrée

```

1  $A \leftarrow \perp$ 
2 pour chaque  $x_i$  faire
3    $A \leftarrow \text{INSERE}(x_i, A)$ 
4 renvoyer  $\text{infixe}(A)$ 
```

Proposition 49 (Complexité du tri par ABR).

En nombre de comparaisons, le tri par ABR a pour complexité :

- $\Theta(n^2)$ dans le pire des cas.
- $O(n \log_2 n)$ dans le cas moyen

Démonstration. Le parcours infixe d'un arbre à n peut-être écrit en complexité $\Theta(n)$ (cf question bonus TP). La question est donc la complexité de la boucle des lignes 2-3.

- Dans le pire des cas : chaque insertion coûte $O(h(A))$. Or, la hauteur augmente au plus de 1 à chaque insertion, donc la boucle coûte $O\left(\sum_{i=0}^{n-1} (-1 + i)\right) = O(n^2)$. De plus, dans le cas où les x_i sont déjà triés par ordre croissant ou décroissant, la hauteur augmente d'exactly 1 à chaque insertion : cette borne est atteinte, d'où le $\Theta(n^2)$.
- La complexité moyenne est admise. Rappelons simplement que la complexité moyenne pour n est la moyenne sur tous les x_0, \dots, x_{n-1} des complexités. Ici, comme la complexité ne dépend pas des valeurs précises x_i mais des comparaisons entre eux, on moyenne sur les permutations de \mathcal{S}_n .

□

Ce pire des cas fournit un bon exemple de la limitation de nos ABR actuels : la complexité des opérations dépend de la *hauteur*, or rien ne garantit que celle-ci soit faible. On voudrait garantir que les ABR ne soient pas « trop » déséquilibrés pour que la hauteur reste logarithmique.

4.2 Arbres Rouge-Noir

Les arbres Rouge-Noir (abrégés ARN), aussi appelés arbres bicolores, sont une façon de garantir l'équilibrage d'un arbre binaire de recherche.

4.2.0 Définition

Définition 50 (Arbre Rouge-Noir).

Un **arbre Rouge-Noir** est un arbre binaire de recherche où chaque noeud est colorié, en Rouge ou bien en Noir. On impose de plus que :

- (1) Tout noeud Rouge a un parent Noir.
- (2) Tous les chemins de la racine à un \perp contiennent autant de noeuds noirs.

Remarque.

- La propriété (1) équivaut à « Un noeud rouge n'a que des enfants noirs, et la racine est noire ».
- Rien n'interdit d'avoir plusieurs noeuds noirs à la suite !

Exemple.

(a) Un arbre Rouge-Noir à 8 noeuds

(b) Un arbre Rouge-Noir à 12 noeuds

FIGURE 9.42 – Deux arbres Rouge-Noir

Proposition 51.

Soit A est un arbre binaire colorié (qu'il soit ou non un ARN). S'équivalent :

- A vérifie le point (2) de la définition des ARN.
- A vérifie (2') : « Pour tout noeud x d'un arbre, tous les chemins de ce noeud x à un \perp contiennent autant de noeuds noirs ».

Démonstration. Procédons par double implication.

\Leftarrow) L'implication réciproque est immédiate puisque la racine est un noeud.

\Rightarrow) Pour l'implication directe, procédons par induction structurelle sur un arbre bicolorié A :

- Initialisation : Si $A = \perp$, c'est immédiat.
- Hérédité : Soit $A = N(g, x, d)$ tel que g et d vérifient l'implication, et montrons-la pour A . Supposons donc que A vérifie (2), et montrons qu'il vérifie (2'). Notons que (2') est immédiatement vrai pour la racine d'après (2). Pour les autres noeuds, remarquons que les chemins dans A de la racine à un \perp sont de la forme $x, \text{racine}(g), \dots, \perp$ ou $x, \text{racine}(d), \dots, \perp$

FIGURE 9.43 – Les chemins de la racine aux feuilles

D'après (2), ces chemins contiennent tous autant de noeuds noirs. En particulier, tous les chemins $racine(g), \dots, \perp$ et $racine(d), \dots, \perp$ ont le même nombre de noeuds noirs : en appliquant l'hypothèse d'induction à g et d , on en déduit que (2') est valide sur tous les noeuds de g et d . D'où l'hérédité.

- Conclusion : On a prouvé par induction structurelle que (2) \implies (2').

□

Définition 52 (Hauteur noire).

Dans un ARN, la **hauteur noire** d'un sous-arbre A_x enraciné en x , notée $bh(A_x)$ ou $bh(x)$, est le nombre d'arêtes qui vont vers un noeud noir sur un chemin de x à un \perp . Autrement dit, c'est le nombre de noeuds noirs sur un tel chemin, x exclu.

Démonstration. Pour que la hauteur noire soit bien définie, il faut qu'elle soit indépendante du chemin (la définition n'impose aucun chemin particulier). C'est le cas d'après (2').

□

Lemme 53.

Soit A un ARN, A_x un sous-arbre non-vide de A dont on note x la racine. Alors A_x a au moins $2^{bh(x)} - 1$ noeuds internes.

Démonstration. Soit A un ARN. Montrons par récurrence forte une hauteur h que « tout sous-arbre A_x de A non-vide de hauteur h vérifie le lemme » :

- Initialisation : $h = -1$ est trivial (il n'y a pas de sous-arbre non-vide). Traitons aussi $h = 0$: A_x est une feuille (0 noeuds internes) et $bh = 0$ d'où l'initialisation.
- Hérédité : Soit $h > 0$ tel que la propriété est vraie pour tous $h' < h$, et A_x sous-arbre non-vide de hauteur h . Donc $A_x = N(g, x, d)$ avec g et d non-vides (impossible de respecter les conditions d'ARN sinon).

Par définition de bh , on a $bh(g) \in \{bh(A_x) - 1, bh(A_x)\}$ et de même pour $bh(d)$. Par H.R., on sait que g a au moins $2^{bh(g)} - 1$ noeuds internes, et de même pour d . Donc en notant n_i le nombre de noeuds internes de A_x :

$$\begin{aligned} n_i &\geq (2^{bh(g)} - 1) + (2^{bh(d)} - 1) + 1 \\ &\geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 && \geq 2^{bh(x)} - 1 \end{aligned}$$

□

Théorème 54 (Un ARN est à peu près équilibré).

Soit A un arbre Rouge-Noir à n noeuds. Alors :

$$h(A) = O(\log_2 n)$$

Plus précisément, avec n_i le nombre de noeuds internes :

$$h(A) \leq 2 \log_2(n_i + 1)$$

Démonstration. La majoration de n déduit de l'inégalité puisque $n \geq n_i + 1$ et que \log est croissant. Montrons donc l'inégalité.

D'après la propriété (1) des ARN, dans tout chemin de la racine à \perp au moins la moitié des noeuds sont noirs (un rouge ne peut venir qu'après un noir donc il y a au moins autant de noirs que de rouges). Donc :

$$bh(A) \geq \frac{h}{2}$$

Or d'après le lemme, $2^{bh(A)} - 1 \leq n_i$, donc $2^{\frac{h}{2}} \leq n_i + 1$. On en déduit le résultat par croissance de \log_2 . \square

On vient de prouver qu'un ABR est à peu près équilibré ! Une autre façon de comprendre la preuve de ce théorème est la suivante : un chemin de la racine à \perp contenant bh noeuds noirs a au moins bh noeuds (s'ils sont tous noirs), et au plus $2bh + 1$ (alternance noir-rouge). Ainsi, il y a au plus un facteur 2 d'écart sur la longueur des différents chemins de la racine à un \perp , et l'arbre ne peut donc pas être trop déséquilibré !

FIGURE 9.44 – Interprétation de la preuve du théorème en termes de hauteur vs hauteur noire

Remarque. D'après la propriété 10, $h = \Omega(\log_2 n)$. Donc l'ordre de grandeur atteint par les ARN est optimal !

Proposition 55 (Complexité de la recherche dans un ARN).

La recherche dans un ARN s'effectue en $\Theta(\log_2 n)$ comparaisons.

Démonstration. Il suffit d'appliquer le même algorithme que dans un ABR, puisqu'un ARN est un ABR. Le théorème 54 conclut. \square

4.2.1 Insertion

Insérer dans un ABR est facile... mais dans un ARN, on doit garantir que les propriétés (1) et (2) restent respectées !

On va procéder ainsi :

- On insère comme dans un ABR, en coloriant la feuille créée en Rouge.
- Si cette insertion a rompu la règle (1) (tout Rouge a un parent Noir), on « fait remonter » l'erreur jusqu'à la racine, où elle est simple à réparer.

En particulier, (2) est toujours vérifiée et il n'y a pas à la réparer ! De plus, en faisant « remonter » la violation de (1), on garantit qu'il y a au plus une violation (1).

Plus précisément, une violation de (1) correspond à :

- Soit la racine n'est pas Noire : il suffit de la recolorier
- Soit il existe un noeud Rouge qui a un parent Rouge. Si ce parent est la racine, cf cas précédent. Sinon, comme il y a au plus une violation de (1), le grand-parent est Noir. En notant $x \leq y \leq z$ les 3 noeuds en questions, cela correspond aux quatre cas ci-dessous qui peuvent tous « remonter » de la même façon :

(a) Les 4 enchainements Noir-Rouge-Rouge possibles

(b) Une façon unique de tous les réparer

Cette façon de réparer a recolorié la racine du sous-arbre en rouge (alors qu'elle était noire). Si son parent était rouge, il y a toujours une violation de (1), mais moins profonde ! Ainsi, on fait « remonter » la violation jusqu'à la racine, où il suffit de recolorier la racine en Noir pour terminer.

Exercice. Vérifier que cette façon de réparer ne rompt pas (2), c'est à dire qu'elle ne casse pas la hauteur noire.

Exemple.

FIGURE 9.46 – Des insertions successives dans un ARN

Proposition 56 (Complexité de l'insertion dans un ARN).

L'insertion dans un ARN s'effectue en temps $\Theta(\log_2 n)$.

Démonstration. Sans code sous les yeux, je me contenterais d'une preuve très schématique. L'insertion est composée de deux phases :

- Insertion comme dans un ABR, en $O(h)$
- Puis remontée de la violation jusqu'à la racine. Une étape de la remontée se fait en temps constant (c'est un nombre fini de rotation), et on remonte vers la racine donc on fait au plus autant d'étapes que la hauteur.

Comme la hauteur est logarithmique en n , on conclut. \square

Remarque. En particulier, avec des ARN, le tri par ABR/ARN est en temps $O(n \log n)$!

4.2.2 Suppression

La suppression dans un ARN est plus compliquée.

Proposition 57 (Complexité de la suppression dans un ARN).

La suppression dans un ARN s'effectue en temps $\Theta(\log_2 n)$.

Démonstration. Admis. \square

La suppression fonctionne ainsi :

- Supprimer comme dans un ABR, en supprimant le maximum d'un sous-arbre gauche.
- Si le noeud supprimé était rouge, tout va bien.
- Sinon, on « ajoute » la couleur noir du noeud supprimé à celle de son parent. Cela crée éventuellement un noeud « doublé noir », qu'il faut ensuite faire remonter jusqu'à la racine.

4.3 Pour aller plus loin

Si vous voulez en savoir plus, vous pouvez aller voir :

- À propos de la suppression dans un ARN : <https://www.irif.fr/~carton/Enseignement/Algorithmique/Programmation/RedBlackTree/> . La définition des ARN qui y est utilisée est légèrement différente, mais cela ne change pas fondamentalement l'algo. Vous trouverez également sur cette page une façon un peu plus efficace de coder l'insertion (car elle arrive parfois à réparer le Rouge-Rouge localement, sans le faire remonter jusqu'à la racine).
- Un résumé très clair des grandes idées de ce cours¹⁴ : deuxième leçon de <https://www.college-de-france.fr/fr/agenda/cours/structures-de-donnees-persistantes> . Les ARN (et leur suppression) y sont traités, mais vous y trouverez aussi une autre façon d'équilibrer les ABR : les AVL !

14. Leçon de Xavier Leroy, le fondateur d'OCaml!

Chapitre 10

DÉCOMPOSITION EN SOUS-PROBLÈMES

Notions	Commentaires
Diviser pour régner. Rencontre au milieu. Dichotomie.	<p>On peut traiter un ou plusieurs exemples comme : tri par partition-fusion, comptage du nombre d'inversions dans une liste, calcul des deux points les plus proches dans une ensemble de points ; recherche d'un sous-ensemble d'un ensemble d'entiers dont la somme des éléments est donnée ; recherche dichotomique dans un tableau trié.</p> <p>On présente un exemple de dichotomie où son recours n'est pas évident : par exemple, la couverture de n points de la droite par k segments égaux de plus petite longueur.</p>
Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes. Calcul de bas en haut ou par mémorisation. Reconstruction d'une solution optimale à partir de l'information calculée.	On souligne les enjeux de complexité en mémoire. On peut traiter un ou plusieurs exemples comme : problème de la somme d'un sous-ensemble, ordonnancement de tâches pondérées, plus longue sous-suite commune, distance d'édition (Levenshtein).

Extrait de la section 4.3 du programme officiel de MP2I : « Décomposition en sous-problèmes ».

SOMMAIRE

0. Diviser pour Régner.....	209
0. Exemple introductif : le tri fusion	209
<i>Borne inférieure en pire des cas pour un tri par comparaisons (p. 209). Tri fusion (p. 209). Implémentation OCaml (p. 210). Implémentation en C (p. 211). Analyse de complexité (p. 213).</i>	
1. Principe général d'un algorithme Diviser pour Régner	214
2. Analyses de complexité	214
3. Exemple : recherche dichotomique	215
<i>La recherche dichotomique (p. 215). Une variante : la recherche d'un pic (p. 216).</i>	
4. Un autre exemple : algorithme de Karatsuba	217
<i>Présentation du problème (p. 217). Un Diviser pour Régner inefficace (p. 217). Algorithme de Karatsuba (p. 218).</i>	
5. Pire cas et cas moyen : le tri rapide	219
<i>Complexité en moyenne (p. 219). Borne inférieure en moyenne pour un tri par comparaisons (p. 219). Tri rapide (p. 220). Drapeau hollandais (p. 220). Implémentation du tri rapide en C (p. 222). Analyse de complexité (p. 223).</i>	
1. Rencontre au milieu	223
0. Problème d'optimisation associé à SUBSETSUM	223
1. Principe général d'un algorithme Rencontre au milieu	224
2. Programmation dynamique	225
0. Exemple fil rouge : plus lourd chemin dans une pyramide	225
<i>Présentation du problème (p. 225). Solution naïve en OCaml (p. 225).</i>	
1. Principe général de la Programmation Dynamique	227
2. Méthode de haut en bas	227
<i>Pour les pyramides (p. 227). Calcul de la complexité (p. 229). Pseudo-code générique (p. 230).</i>	

3. Méthode de bas en haut	231
<i>Pour les pyramides (p. 231). Calcul de la complexité (p. 231). Pseudo-code générique (p. 232). Optimisation de la mémoire (p. 232).</i>	
4. Comparaison des deux méthodes	234
5. Message d'utilité publique...	234
6. (Re)construction de la solution optimale	234
<i>Fonctionnement générique (p. 234). Pour les pyramides : mémorisation des solutions optimales (p. 235). Pour les pyramides : reconstruction de la solution optimale (p. 235).</i>	

0 Diviser pour Régner

0.0 Exemple introductif : le tri fusion

Remarque. En plus de présenter des méthodes algorithmiques **extrêmement** puissantes, ce cours présente aussi des algorithmes de tri usuels à savoir refaire.

On s'intéresse dans cet exemple introductif au problème du TRI, qui consiste à prendre en entrée x_0, \dots, x_{n-1} une liste linéaire d'éléments munis d'un ordre total, et à ordonner par ordre croissant ces x_i .

0.0.0 Borne inférieure en pire des cas pour un tri par comparaisons

Un théorème très important concerne les solutions à ce problème :

Théorème 1 (Borne inférieure sur les tris).

Tout algorithme qui résout le problème de TRI en ne distinguant les éléments que par des comparaisons entre eux s'exécute en $\Omega(n \log_2 n)$ comparaisons.

Démonstration. Je donne ici une version vulgarisée de la preuve. J'en connais deux versions rigoureuses que vous verrez peut-être en école (ou en MPI?) : par l'exhibition d'un adversaire, ou par un calcul d'entropie.

Il y a $n!$ façons possibles d'ordonner x_0, \dots, x_{n-1} . Une comparaison consiste à poser la question $x_i < ? x_j$ et élimine donc toutes les façons d'ordonner qui ne respectent pas cette comparaison. Ainsi, dans le pire des cas, chaque comparaison élimine au plus la moitié d'ordonnements possibles restants. Il faut donc $\Omega(\log_2(n!))$ comparaisons. Or, $\log_2(n!) \sim n \log_2 n$ donc il faut $\Omega(n \log_2 n)$ comparaisons. \square

Remarque.

- La plupart des algorithmes de tri sont en $O(n^2)$.
- Une autre façon de formuler ce théorème est « Sans hypothèse supplémentaire sur les données en entrée, tout algorithme de tri s'exécute en $\Omega(n \log_2 n)$ ».
- Il existe des algorithmes¹ qui passent sous cette borne inférieure : généralement, ils se basent sur le fait que les données sont des entiers/flottants pour pouvoir raisonner autrement que par comparaisons et échapper au théorème.

0.0.1 Tri fusion

Définition 2 (Tri fusion).

Pour trier une liste linéaire, le tri fusion fonctionne ainsi :

- (i) Diviser (scinder) la liste en deux sous-listes de longueurs égales (ou presque).
- (ii) Trier récursivement les deux sous-listes.
- (iii) Effectuer la fusion triée des deux sous-listes triées.

Les cas de base sont la liste linéaire à 0 ou à 1 élément.

Remarque. \triangle Attention, oublier le cas de base à 1 élément est une erreur classique, qui mène à une boucle infinie.

Exemple. La figure ci-dessous résume l'évolution des divisions et des fusions pour $[38; 27; 43; 3; 9; 82; 10]$. La même de division utilisée consiste à mettre les $\lceil \frac{n}{2} \rceil$ premiers éléments dans une liste, et les $\lfloor \frac{n}{2} \rfloor$ suivants dans l'autre :

1. Par exemple le tri par dénombrement ou le tri par base.

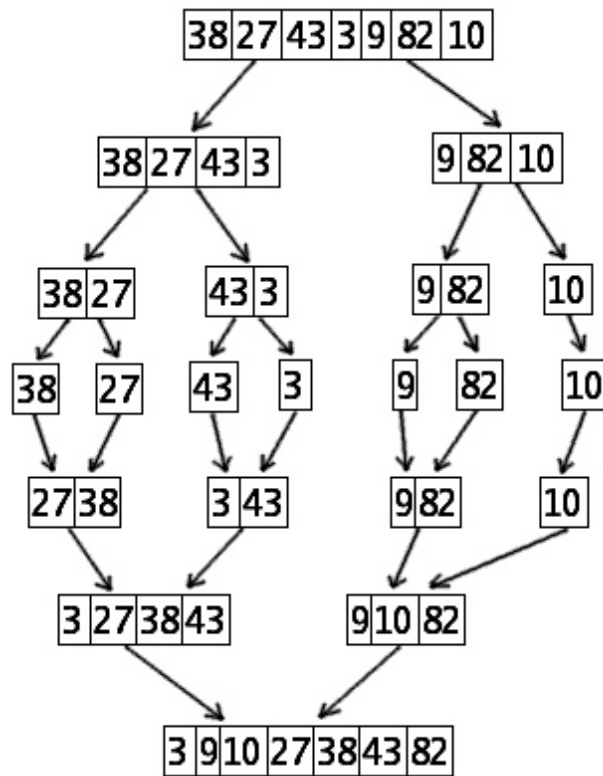


FIGURE 10.1 – Tri fusion de [38; 27; 43; 3; 9; 82; 10]. Src : Wikipédia

0.0.2 Implémentation OCaml

En OCaml, c'est la fonction de division `divide` qui est la plus compliquée à écrire. L'idée est de découper la liste selon « la parité de l'indice » : le premier élément va dans la première sous-liste et le second dans la deuxième, le troisième élément dans la première et le quatrième dans la deuxième, le cinquième dans la première et le sixième dans la deuxième, etc etc.

```

16 let rec divide lst =
17   match lst with
18   | []
19   | [_] -> lst, []
20   | t0::t1::q ->
21     let (d0, d1) = divide q in
22     t0::d0, t1::d1

```



La fusion est la fonction `merge` déjà étudiée :

```

24 let rec merge lst0 lst1 =
25   match (lst0, lst1) with
26   | [], l
27   | l, [] -> l
28   | t0::q0, t1::q1 ->
29     if t0 < t1 then
30       t0 :: (merge q0 lst1)
31     else
32       t1 :: (merge lst0 q1)

```



L'algorithme s'écrit alors ainsi :

```

34 let rec mergesort lst =
35   match lst with
36   | []
37   | [_] -> lst
38   | _ -> let lst0, lst1 = divide lst in
39           merge (mergesort lst0) (mergesort lst1)

```



0.0.3 Implémentation en C

La division en deux sous-tableaux est élémentaire en C si l'on se souvient qu'un tableau est affaibli en pointeur et est donc en fait l'adresse de sa première case. Ainsi, on va diviser en deux moitiés :

- La première moitié débute au même endroit que le tableau initial mais est deux fois plus courte.
- La seconde moitié débute à la case d'après la première moitié et prend les éléments restants.

FIGURE 10.2 – Pointeurs pour la division pour 38; 27; 43; 3; 9; 82; 10

Toutefois, la fonction de fusion est moins agréable à écrire en C qu'en OCaml, à cause des cas où l'on a fini de parcourir un des deux tableaux à fusionner mais pas encore fini l'autre :

```

10 int* merge(int const* arr, int len_arr, int const* brr, int len_brr) {
11     int len = len_arr + len_brr;
12     int* merged = (int*) malloc(len*sizeof(int));
13
14     int j_arr = 0; // prochain élément de arr
15     int j_brr = 0; // prochain élément de brr
16     for (int i = 0; i < len; i += 1) {
17         // On fait merged[i] = min(arr[j_arr], brr[j_brr])
18         if (j_arr >= len_arr) {
19             merged[i] = brr[j_brr];
20             j_brr += 1;
21         }
22         else if (j_brr >= len_brr
23                 || arr[j_arr] <= brr[j_brr]) {
24             merged[i] = arr[j_arr];
25             j_arr += 1;
26         }
27         else {
28             merged[i] = brr[j_brr];
29             j_brr += 1;
30         }
31     }
32
33     return merged;
34 }

```



FIGURE 10.3 – Premières et dernières étapes de la fusion triée de [0; 1; 3; 5; 6] et [2; 8; 9]

On en déduit le tri fusion :

```

40 int* mergesort(int const* arr, int len) {
41     // Cas de base
42     if (len == 0) { return NULL; }
43     else if (len == 1) {
44         int* sorted = malloc(sizeof(int));
45         sorted[0] = arr[0];
46         return sorted;
47     }
48
49     // Diviser ET trier
50     int len_left = len - len/2;
51     int len_right = len/2;
52     int* left = mergesort(arr, len_left);
53     int* right = mergesort(&arr[len_left], len_right);
54
55     // Fusionner
56     int* sorted = merge(left, len_left, right, len_right);
57
58     // Free et return
59     free(left);
60     free(right);
61     return sorted;
62 }

```



Remarque. Ce tri **n'est pas** en place, c'est à dire qu'il crée une nouvelle version triée du tableau au lieu de se limiter à modifier le tableau donné en entrée. Écrire un tri fusion en place efficace est plus compliqué (à cause de l'étape de fusion, qui si elle est mal écrite est trop coûteuse).

0.0.4 Analyse de complexité

On va ici prouver la complexité de la version OCaml en fonction de n la longueur de la liste, la version C se traitant de manière similaire.

- divide : Chaque appel effectue localement un nombre constant d'opérations, ainsi que 1 appel récursif. À chaque appel récursif, la longueur de la liste diminue de 2 et est minorée par 0 : donc divide est en $O(n)$.
- merge : Notons n_0 la longueur de `lst0` et n_1 de `lst1`. Chaque appel effectue localement un nombre, ainsi que 1 appel récursif dans lequel la longueur de l'une des deux listes diminue de 1 et l'autre est inchangée : donc merge est en $O(n_0 + n_1)$.
- mergesort : Chaque appel effectue localement un nombre d'opérations dominé par les appels à divide et merge. Le premier est en $O(n)$, et le second fusionne les deux listes renvoyées par divide (qui ont donc n élément au total) donc est en $O(n)$ aussi. De plus, chaque appel effectue 2 appels récursifs sur des listes de longueur $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$. Donc l'équation de complexité est :

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n)$$

On peut enlever les parties entières et le $O()$ sans changer l'ordre de grandeur asymptotique du résultat². On résout donc :

$$T(n) = 2T(\frac{n}{2}) + n$$

Par la méthode par arbre :

FIGURE 10.4 – Arbre de complexité de $T(n) = 2T(\frac{n}{2}) + n$

Ici chaque ligne de l'arbre coûte n , il y a $1 + \log_2 n$ lignes, donc la complexité $T(n)$ vérifie :

$$T(n) = O(n \log_2 n)$$

Remarque. D'après le théorème 1, l'ordre de grandeur de la complexité du tri fusion est optimal !

2. *Trust me I'm a computer scientist.*

0.1 Principe général d'un algorithme Diviser pour Régner

Définition 3 (Diviser pour Régner).

La méthode Diviser pour Régner (*Divide and Conquer*) permet de résoudre certains problèmes algorithmiques. Pour résoudre une instance I de taille n d'un problème, elle propose de procéder ainsi :

- (i) Diviser l'instance I en r instances indépendantes (du même problème) de taille n/c .
- (ii) Résoudre récursivement ces r instances.
- (iii) Fusionner (« Régner ») les solutions de ces r instances en une solution de I .

Remarque.

- Cette méthode est naturellement récursive.
- Il faut bien sûr traiter les cas de base à part.
- Le tri fusion est un exemple de méthode Diviser Pour Régner.
- Cette méthode n'est pas une recette à infallible : pour l'appliquer, il faut (et suffit) qu'une solution de l'instance puisse être exprimée comme la fusion de solutions d'une division de l'instance !!
- Rien n'impose que $r = c$: le nombre de sous-problèmes auxquels on se réduit et la taille de ces sous-problèmes sont, a priori, non-liés. Cependant, en pratique, chaque « élément » de l'instance en entrée va dans au plus une sous-instance et on a donc $r \leq c$.

FIGURE 10.5 – Division d'une instance en sous-instances

0.2 Analyses de complexité

En ignorant^{3 4} les parties entières et les $O()$, l'équation de récurrence vérifiée par la complexité d'un algorithme Diviser pour Régner est de la forme :

$$T(n) = rT(n/c) + f(n)$$

Avec r le nombre de sous-problèmes auxquels on se ramène, n/c la taille des sous-problèmes, et $f(n)$ le coût local (somme du coût de la division et de la fusion).

Nous avons déjà étudié dans le cours sur la complexité comment résoudre de telles équations. Pour rappel, on trace l'arbre d'appels qui est de la forme, on calcule le coût de chaque étage de l'arbre d'appel et on somme les coûts de ces étages. Cette somme est facile à calculer si l'on peut remarquer que :

- Le coût des étages croît (au moins) géométriquement : en exploitant cela, on montre que le coût total est dominé par le coût de l'étage du bas.
- Le coût des étages est constant.
- Le coût des étages décroît (au moins) géométriquement : en exploitant cela, on montra que le coût total est dominé par le coût de la racine.

3. *TRUST ME!*

4. Si vous voulez vérifier ma source : <https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf>

Remarque.

- Ces trois cas de figure sont des « astuces » à savoir appliquer, pas une propriété invocable. Autrement dit, une copie disant "d'après le cours, le coût des feuilles domine" n'aura pas les points liés au calcul.
- Pour rappel, l'arbre de $T(n) = rT(n/c) + f(n)$ ressemble à ceci :

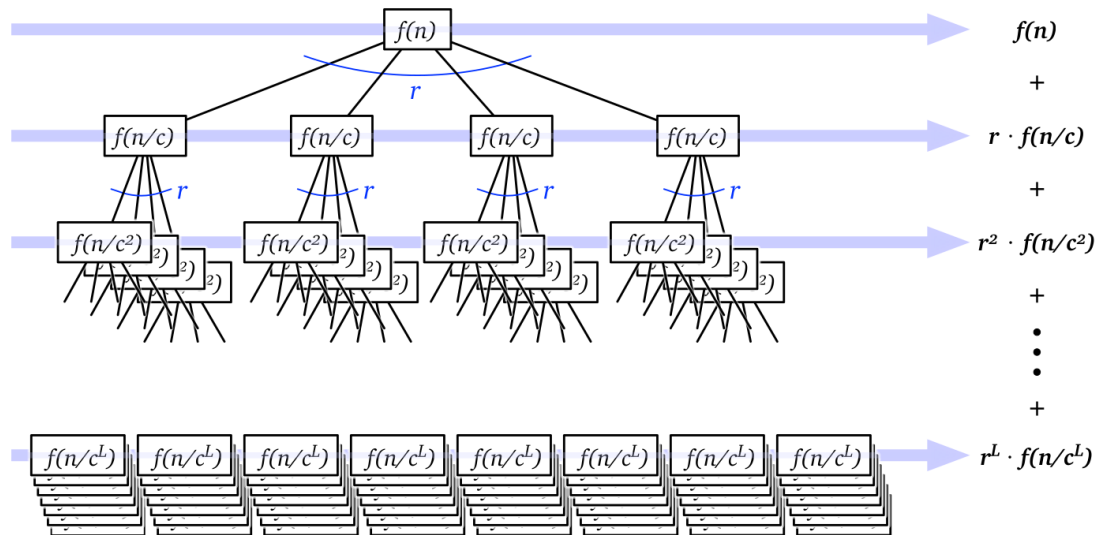


FIGURE 10.6 – Arbre de $T(n) = rT(n/c) + f(n)$. Src : J. Erikson

0.3 Exemple : recherche dichotomique

0.3.0 La recherche dichotomique

La recherche dichotomique d'un élément x dans un tableau `arr` est un exemple de méthode Diviser pour Régner, où la Division consiste à se réduire à 1 instance de taille $\text{len}/2$. L'étape de Fusion est vide, puisque l'on renvoie immédiatement ce que l'appel récursif renvoie. La voici ici écrite (récursivement) en C, avec des petites manipulations de pointeurs comme dans le tri fusion :

```

9  /* Recherche x dans arr (à len éléments) */
10 bool recherche(int x, int const* arr, int len) {
11     // Cas de base
12     if (len == 0) { return false; }
13     if (len == 1) { return arr[0] == x; }
14
15     // Cas récursif
16     int mid = len/2;
17     if (arr[mid] == x) { // On a trouvé x
18         return true;
19     }
20     else if (x < arr[mid]) { // On cherche dans la première moitié
21         return recherche(x, arr, mid);
22     }
23     else { // On cherche dans la seconde moitié
24         return recherche(x, &arr[mid+1], len-mid-1);
25     }
26 }

```

 dichotomie.c

Son équation de récurrence est $T(\text{len}) = T(\text{len}/2) + O(1)$ qui a pour solution $T(\text{len}) = O(\log_2(\text{len}))$.

FIGURE 10.7 – Pointeurs et dichotomie

0.3.1 Une variante : la recherche d'un pic

Dans un tableau arr d'entiers distincts (de longueur len), un **pic** est une case plus grande que ses deux voisins, c'est à dire un indice $i \in \llbracket 0; len \rrbracket$ tel que $arr[i-1] < arr[i]$ et $arr[i] > arr[i+1]$. Pour gérer le cas particulier des bords, on pose $arr[-1] = -\infty$ et $arr[len] = +\infty$: ainsi la définition de **pic** fonctionne aussi sur les bords :

| 5 | 3 | **14** | 6 | 2 | 10 | **11** | 9 |

FIGURE 10.8 – Pics (en gras) dans un tableau

Le problème PEAK FINDING est le problème de trouver *un* pic (n'importe lequel) dans un tableau donné en entrée.

Il existe une solution dichotomique très élégante à ce problème. Elle consiste à remarquer que si $arr[lo-1] < arr[lo]$ et que $arr[hi] > arr[hi+1]$, alors tout pic de $arr[lo : hi]$ est un pic de arr (avec $arr[i : j]$ le sous-tableau de arr allant des indices i inclus à j exclu).

FIGURE 10.9 – Invariant de la recherche de pic

On en déduit le code récursif suivant :

```

3  /** Recherche un pic dans arr[lo; hi[ */
4  int peak_finding_rec(int const arr[], int lo, int hi) {
5      int mid = (lo+hi)/2;
6
7      if (lo < mid && arr[mid-1] > arr[mid]) {
8          return peak_finding_rec(arr, lo, mid);
9      }
10     else if (mid + 1 < hi && arr[mid] < arr[mid+1]) {
11         return peak_finding_rec(arr, mid+1, hi);
12     }
13     else {
14         return mid;
15     }
16 }
```

 peak-finding.c

En notant $n = hi - lo$, son équation de complexité est $T(n) = T(n/2) + O(n)$ donc $T(n) = O(\log_2 n)$.

Remarque. \triangle Pour faire fonctionner ce code C, il faut préparer le tableau de sorte à ce que $arr[-1]$ et $arr[len]$ soient valides...

Exercice. Dédurre de la fonction précédente une fonction `peak_finding` qui prend en entrée un tableau d'entiers et renvoie l'indice de l'un de ses pics

0.4 Un autre exemple : algorithme de Karatsuba

0.4.0 Présentation du problème

On s'intéresse ici au problème de savoir multiplier deux entiers x et y écrits en binaire, tous les deux sur n bits (quitte à rajouter des 0 à droite au plus court des deux).

Les algorithmes classiques pour la multiplication (par exemple celui vu à l'école primaire) s'exécute en $O(n^2)$. Nous allons ici faire un Diviser pour Régner qui bat cette complexité.

0.4.1 Un Diviser pour Régner inefficace

L'idée de l'étape de division est de couper chacun des entiers en 2 moitiés (bits de poids faible et bits de poids forts). Par exemple :

$$44 = \overline{20010\ 1100} = \overline{20010} \cdot 2^4 + \overline{21100}$$

Pour simplifier, on suppose que n est une puissance de 2. Ainsi, x se décompose en $x = x_{hi} \cdot 2^{n/2} + x_{lo}$ avec x_{hi} et x_{lo} tous deux sur $n/2$ bits. De même pour y avec y_{hi} et y_{lo} .

Reste maintenant le plus difficile : quelles mutiplicationps d'entiers de $n/2$ bits faire pour retrouver $x \cdot y$? On veut faire le moins de multiplication possibles. Notons tout d'abord que les multiplication par une puissance de 2 sont peu couteuses : il s'agit d'un simple décalage des bits⁵.

Ensuite, développons le produit :

$$\begin{aligned} x \cdot y &= (x_{hi}2^{n/2} + x_{lo}) \cdot (y_{hi}2^{n/2} + y_{lo}) \\ &= x_{hi} \cdot y_{hi}2^n + x_{lo} \cdot y_{lo} + (x_{hi} \cdot y_{lo} + x_{lo} \cdot y_{hi})2^{n/2} \end{aligned}$$

Si on applique cette formule, en admettant que l'addition et la multiplication par une puissance de 2 se fait en $O(n)$, on obtient une complexité vérifiant $T(n) = 4T(n/2) + O(n)$ (le cas de base est la multiplication de deux entiers à 1 bit, qui est triviale). Traçons son arbre :

FIGURE 10.10 – Arbre de complexité de $T(n) = 4T(n/2) + O(n)$

Cet arbre a $1 + \log_2 n$ étage, et l'étage i coute $e_i = 4^i \frac{n}{2^i} = 2^i n$. Donc⁶ le cout total vérifie :

5. Pour la même raison que multiplier par 10^k est simple en base 10.

6. On peut remarquer que le coût des étages croît géométriquement : le reste de notre calcul doit donc être de l'ordre du coût de la dernière ligne, c'est à dire de $4^{\log_2 n} \cdot 1 = n^2$.

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n} e_i \\
&= n(2^{\log_2 n + 1} - 1) \\
&= O(n^2)
\end{aligned}$$

C'est à dire que l'on a rien gagné par rapport à l'algorithme naïf. Le problème ici est soit que le coût de la division/fusion est trop élevé - mais on ne voit pas trop comment l'améliorer - soit que la division n'est pas assez efficace : on a divisé en trop de sous-problèmes, ou en des sous-problèmes trop grands.

0.4.2 Algorithme de Karatsuba

L'algorithme de Karatsuba consiste à procéder comme précédemment, mais à remarquer que le terme $x_{hi} \cdot y_{lo} + x_{lo} \cdot y_{hi}$ peut-être exprimé à l'aide des autres multiplications que l'on effectue et d'une nouvelle multiplication :

$$\begin{aligned}
x_{hi} \cdot y_{lo} + x_{lo} \cdot y_{hi} &= \underbrace{x_{hi} \cdot y_{hi}}_{\text{déjà calculé}} + \underbrace{x_{lo} \cdot y_{lo}}_{\text{déjà calculé}} \\
&\quad - (x_{hi} - x_{lo}) \cdot (y_{hi} - y_{lo})
\end{aligned}$$

Ainsi, pour calculer $x \cdot y$, on se ramène aux trois instances suivantes :

- Calculer $x_{hi} \cdot y_{hi}$
- Calculer $x_{lo} \cdot y_{lo}$
- Calculer $(x_{hi} - x_{lo}) \cdot (y_{hi} - y_{lo})$

Le coût de la division est donc $O(n)$ (calcul des lo et hi ainsi que deux soustractions) et celui de la fusion $O(n)$ (des additions et des multiplications par une puissance de 2). Donc l'équation de complexité est $T(n) = 3T(n/2) + O(n)$. Traçons son arbre :

FIGURE 10.11 – Arbre de complexité de $T(n) = 3T(n/2) + O(n)$

Il y a $1 + \log_2 n$ étage, et l'étage i coûte $e_i = 3^i \frac{n}{2^i} = \left(\frac{3}{2}\right)^i n$. En notant $h = \log_2 n$, on constate que $e_{i+1} = \frac{3}{2}e_i$ donc que $e_i = \left(\frac{2}{3}\right)^{h-i} e_h$. Il s'ensuit :

7. Croissance géométrique à nouveau !

$$\begin{aligned}
T(n) &= \sum_{i=0}^h e_i \\
&= e_h \underbrace{\sum_{i=0}^h \left(\frac{2}{3}\right)^{h-i}}_{\text{converge}} \\
&= O(e_h)
\end{aligned}$$

Or :

$$\begin{aligned}
e_h &= 3^{\log_2 n} \cdot 1 \\
&= e^{\ln(3) \frac{\ln(n)}{\ln(2)}} \\
&= n^{\log_2(3)} \\
&\simeq n^{1,585}
\end{aligned}$$

Où l'arrondi est fait à la valeur supérieure : d'où $T(n) = O(n^{\log_2(3)}) = O(n^{1,585})$. On a battu les algorithmes naïfs !

Remarque. L'algorithme de Karatsuba est un excellent exemple de la difficulté d'optimiser des Diviser pour Régner : il ne faut pas juste trouver une idée réursive, il faut aussi se demander si on ne peut pas *mieux* diviser ou *mieux* régner. Tout gain sur l'une de ces étapes est significativement répercuté sur la complexité du tout.

0.5 Pire cas et cas moyen : le tri rapide

0.5.0 Complexité en moyenne

Définition 4 (Complexité moyenne).

La complexité en moyenne d'un algorithme est la fonction T_{moy} qui à n associe la moyenne des complexités des instances de taille n . Autrement dit :

$$T_{moy}(n) = \mathbb{E}(\text{complexité d'une instance de taille } n \text{ prise uniformément au hasard})$$

Proposition 5.

Pour tout algorithme, avec T_{best} sa complexité meilleur cas, T_{moy} sa complexité moyenne, et T_{pire} sa complexité pire des cas, on a :

$$T_{best}(n) \leq T_{moy}(n) \leq T_{pire}(n)$$

Démonstration. Immédiatement en développant l'espérance comme une somme et en minorant/majorant ses termes par le meilleur et le pire cas. \square

0.5.1 Borne inférieure en moyenne pour un tri par comparaisons

Le théorème 1 est formulé comme une complexité pire des cas. On peut montrer qu'il est aussi valable dans le cas moyen⁸. C'est à dire que si on prend une liste de longueur n uniformément au hasard, et qu'on la trie par comparaisons, l'espérance du nombre de comparaisons requises sera un $\Omega(n \log_2 n)$.

Autrement dit « le théorème 1 n'est pas seulement valable dans le pire des cas, mais aussi "en pratique", quand on prend une liste à peu près quelconque⁹ ».

8. Après formalisation, il s'agit de montrer que pour un arbre binaire à $n!$ feuille, la profondeur moyenne des feuilles est un $\Omega(n \log_2 n)$.

9. Entendre par là « ui n'est ni un meilleur ni un pire cas ».

Bien entendu, comme le tri fusion est en $O(n \log 2n)$ dans le pire des cas, il l'est aussi dans le cas moyen. Cependant, en pratique, il existe un autre algorithme qui s'exécute plus rapidement : le tri rapide.

0.5.2 Tri rapide

Définition 6 (Tri rapide).

Pour trier une liste linéaire, le tri rapide fonctionne ainsi :

- (i) Choisir un élément p dans la liste et l'appeler *pivot*
- (ii) Diviser la liste en les éléments inférieurs au pivot et les éléments supérieurs au pivot
- (iii) Trier ces deux sous-listes
- (iii) Concaténer le tout

Le cas de base est celui de la liste à 0 ou 1 élément.

Lorsqu'il est bien implémenté, le tri rapide est... très rapide¹⁰. Pour sa rapidité, il y a deux facteurs cruciaux :

- Bien choisir le pivot : on veut un pivot qui mène à une division de la liste en deux sous-listes à peu près de longueur égale.
- La division doit s'exécuter rapidement.

On voudrait de plus le réaliser *en place*, c'est à dire que l'on veut modifier le tableau pris en entrée et non créer de nouveaux tableaux.

Pour simplifier, on prendra comme pivot le premier élément du tableau¹¹. Pour faire un tri rapide, il faut donc pouvoir résoudre le problème suivant (qui correspond à la phase de Division) :

- Entrée : *arr* un tableau de *len* éléments et p un pivot
- Tâche : réordonner *arr* de sorte à ce que tous les éléments strictement inférieurs à p soit au début, puis que suivent tous les éléments égaux à p , et enfin que terminent tous les éléments strictement supérieurs à p

FIGURE 10.12 – La forme attendue pour *arr* en sortie

0.5.3 Drapeau hollandais

Ce problème peut-être simplifié et abstrait comme le problème du drapeau hollandais¹² :

- Entrée : *arr* un tableau de *len* éléments valant 1 2 ou 3.
- Tâche : réordonner *arr* de sorte à ce qu'il contienne d'abord tous les 1, puis tous les 2, puis tous les 3. La seule modification du tableau autorisée est l'échange du contenu de 2 cases.

On voudrait de plus faire cela efficacement, si possible en temps linéaire en *len* ! Il existe une solution qui fait cela, et qui plus est qui fait cela en un seul parcours du tableau. L'idée est la suivante : la zone du tableau déjà parcourue est découpée en 3 zones comme attendu. Quand on croise un nouvel élément, on fait des échanges au niveau des fins/débuts des zones pour l'insérer efficacement.

10. J'aime les algorithmes bien nommés.

11. Ce n'est même pas un si mauvais choix : en espérance, cela est équivalent au fait de prendre un élément au hasard dans le tableau.

12. Pourquoi hollandais ? Parce que le drapeau hollandais est tricolore, et que cette abstraction a été popularisée par E. Dijkstra qui est... hollandais.

Montrons les trois cas de figure sur des schémas. On nomme i le nombre d'éléments du tableau déjà lus : ainsi, $arr[0 : i[$ est découpé en 3 zones comme indiqué. Notons nb_1 le nombre de 1 dans ce début de arr , nb_2 le nombre de 2 dans ce début et nb_3 le nombre dans ce début.

(a) Situation initiale : on doit insérer $arr[i]$ dans une des trois zones

(b) Insertion d'un 1 en deux échanges

(c) Insertion d'un 2 en un échange

(d) Insertion d'un 3 en un échange

FIGURE 10.13 – Résolution efficace du drapeau hollandais

Avec cette méthode, on maintient les invariants suivants :

- $arr[0 : nb_1[$ ne contient que des 1.
- $arr[nb_1 : nb_1+nb_2[$ ne contient que des 2
- $arr[nb_1+nb_2 : nb_1+nb_2+nb_3[$ ne contient que des 3
- Avec i l'indice de la case actuellement lue, nb_1 est le nombre de 1 entre les indices 0 inclus et i exclu dans le tableau initial (de même pour nb_2 et nb_3)

À la fin du processus, les 3 premiers invariants prouvent que le tableau est bien découpé en 3 zones comme attendu, et le quatrième prouve que le nombre de 1/2/3 est bien le bon : gagné !

Cette méthode correspond au pseudo-code suivant :

Algorithme 15 : Résolution du drapeau hollandais

Entrées : arr un tableau de len éléments valant 1 2 ou 3.

```

1   $nb\_1 \leftarrow 0$ 
2   $nb\_2 \leftarrow 0$ 
3   $nb\_3 \leftarrow 0$ 
4  pour  $i$  de 0 inclus à  $len$  exclu faire
5      si  $arr[i] = 1$  alors
6          Échanger  $arr[nb\_1]$  et  $arr[i]$ 
7          Échanger  $arr[nb\_1 + nb\_2]$  et  $arr[i]$ 
8           $nb\_1 \leftarrow nb\_1 + 1$ 
9      sinon si  $arr[i] = 2$  alors
10         Échanger  $arr[nb\_1 + nb\_2]$  et  $arr[i]$ 
11          $nb\_2 \leftarrow nb\_2 + 1$ 
12     sinon
13          $nb\_3 \leftarrow nb\_3 + 1$ 

```

0.5.4 Implémentation du tri rapide en C

Pour l'étape de division, on va prendre le premier élément comme pivot puis appliquer l'algorithme du drapeau hollandais. En termes savants, cette façon de faire s'appelle le **partitionnement de Lomuto**^{13 14}.

Voici une fonction qui permet d'échanger deux valeurs :

```

9  /** Échange le contenu des cases pointées par a et b */
10 void swap(int* a, int* b) {
11     int tmp = *a;
12     *a = *b;
13     *b = tmp;
14     return;
15 }
```

 quicksort.c

Et voici le tri rapide. On utilise les mêmes astuces de pointeur que précédemment. Notez que cette fois, on n'alloue pas de mémoire supplémentaire : tout est fait dans le arr d'origine ! Le tri est en place.

```

18 /** Trie arr (qui a len éléments) */
19 void quicksort(int* arr, int len) {
20     // Cas de base
21     if (len <= 1) { return; }
22
23     // Diviser
24     int pivot = arr[0];
25     int nb_pt = 0; // nb d'elem < pivot
26     int nb_eq = 0; //      ==
27     int nb_gd = 0; //      >
28     for (int i = 0; i < len; i += 1) {
29         if (arr[i] < pivot) {
30             swap(&arr[nb_pt], &arr[i]);
31             swap(&arr[nb_pt+nb_eq], &arr[i]);
32             nb_pt += 1;
33         }
34         else if (arr[i] == pivot) {
35             swap(&arr[nb_pt + nb_eq], &arr[i]);
36             nb_eq += 1;
37         }
38         else {
39             nb_gd += 1;
40         }
41     }
42
43     // Régner
44     quicksort(arr, nb_pt);
45     quicksort(&arr[nb_pt+nb_eq], nb_gd);
46
47     return;
48 }
```

 quicksort.c

13. Il existe un autre partitionnement plus efficace : le partitionnement de Hoare.

14. En réalité le partitionnement de Lomuto est plus simple : il divise en deux zones, une inférieure ou égale au pivot et l'autre strictement supérieure au pivot.

0.5.5 Analyse de complexité

L'étape de division est linéaire en n la longueur, et la fusion est instantanée. Dans le pire des cas, le pivot divise très mal (en deux parties de taille 0 et $n - 1$) : on obtient l'équation de récurrence pire des cas $T(n) = T(n - 1) + O(n)$, qui a pour solution... $O(n^2)$

Pourtant, et je maintiens, ce tri est rapide !! En effet, il n'y a à peu près *que* dans le pire des cas que la division est déséquilibrée. En pratique (en *moyenne*), les deux sous-listes sont à peu près équilibrées.

Proposition 7 (Complexités du tri rapide et du tri fusion).

En notant n le nombre d'éléments :

Complexité \ Algorithme	Tri fusion	Tri rapide
Pire des cas	$O(n \log_2 n)$	$O(n^2)$ rare
Cas moyen	$O(n \log_2 n)$	$O(n \log_2 n)$ avec faible constante

Démonstration. Les complexités pire cas ont été prouvées. Le cas moyen du pire des cas est compris entre le pire cas et le théorème $\Omega(n \log_2 n)$ en complexité moyenne... d'où sa valeur.

Reste la case en bas à droite. C'est un bon exercice de proba de MP2I, que nous ferons peut-être lorsque vous aurez eu le cours de proba. En attendant, admettons-le. \square

Pour accélérer l'algorithme du tri rapide, il faut soit accélérer la division, soit garantir un bon équilibre des deux sous-listes. Pour ce second point, deux façons de faire :

- Utiliser la médiane de *arr* comme pivot : c'est possible (on peut la calculer en temps linéaire, cf https://fr.wikipedia.org/wiki/M%C3%A9diane_des_m%C3%A9diannes - cela devrait vous rappeler SLOWSELECT vu en DS), mais en pratique le temps de calcul de la médiane coûte plus cher que le gain apportée.
- Utiliser l'élément médian parmi $arr[0]$, $arr[len/2]$, $arr[len - 1]$: c'est en pratique incroyablement efficace et fait que le pire des cas n'arrive presque jamais !

Enfin, pour accélérer encore plus, on peut changer le cas de base : il s'avère en effet que le tri rapide n'est pas le plus rapide pour les petits tableaux. Lorsque la longueur du tableau passe en-dessous d'un certain seuil (par exemple $n \leq 4$), on termine alors non pas récursivement mais en utilisant un autre tri (le tri par insertion est populaire à cette fin).

Remarque.

- Vous devriez maintenant avoir tous les éléments théoriques pour comprendre le code source de la fonction `qsort` de C : <https://github.com/lattera/glibc/blob/master/stdlib/qsort.c>. Il vous manque un ou deux éléments pratiques (notamment ce que signifie ajouter/soustraire à un pointeur), mais avec un peu d'acharnement vous devriez finir par comprendre le fonctionnement.
- Le tri rapide est un excellent exemple de Diviser pour Régner qui n'est pas efficace dans le pire des cas mais très efficace dans le cas moyen, donc tout à fait utilisable en pratique.

1 Rencontre au milieu

1.0 Problème d'optimisation associé à SUBSETSUM

Considérons le problème d'optimisation associé à SUBSET-SUM

- Entrée : $E \subset \mathbb{N}$ un ensemble fini d'entiers positifs et $t \in \mathbb{N}$ une « cible » (target).
- Tâche : Trouver la plus grande somme d'éléments de E inférieure ou égale à t ?

Nous pouvons utiliser une solution en retour sur trace¹⁵. Mais considérons ainsi une solution tout à fait naïve : calculer toutes les sommes possibles de E , et en déduire la plus grande inférieure ou égale à t . Cette solution naïve s'exécute en $O(2^n)$.

Nous allons mélanger cette solution naïve exponentielle avec une étape de division et de fusion pour obtenir une meilleure solution :

- (i) Diviser arbitrairement E en F et G deux moitiés de tailles égales.
- (ii) Calculer toutes les sommes possibles de F et G à l'aide de l'algorithme naïf.
- (iii) Fusionner ces sommes de F et G en toutes les sommes possibles de E et conclure.

Remarquez que l'on ne fait *pas* un Diviser pour Régner, puisque l'on ne continue pas les Divisions/-Fusions récursivement ! Calculons la complexité, avec n le nombre d'éléments de E

- La division se fait en $O(n)$ ou $O(1)$ selon comment l'ensemble est codé. Mettons que l'on utilise des tableaux, donc $O(1)$ (en vrai, on s'en fiche vu l'ordre de grandeur des complexités suivantes).
- Les deux appels récursifs se font en temps $2^{n/2}$.
- Il y a $2^{n/2}$ sommes possibles pour F et G , donc en triant puis faisant une fusion triée l'étape de fusion se fait en temps $O(n2^{n/2})$.

Donc la complexité du tout est un $O(n2^{n/2})$. On a divisée la complexité du naïf par un facteur $\frac{2^{n/2}}{n}$: c'est énorme !

Remarque. Ni la solution en retour sur trace, ni la solution ci-dessus ne sont en temps polynomial. Mais peut-être y a-t-il une solution intelligente qui nous échappe et réussit à atteindre la complexité polynomiale ? C'est en réalité (sans doute) voué à l'échec : en MPI, vous verrez que l'on pense qu'il n'existe pas de solution polynomiale à ce problème^{16 17}.

1.1 Principe général d'un algorithme Rencontre au milieu

L'idée est de s'inspirer de la méthode Diviser pour Régner pour gagner en complexité, mais en ne divisant/fusionnant qu'une seule fois (au lieu de le faire récursivement) :

Définition 8 (Rencontre au milieu).

La **rencontre au milieu** (*meet in the middle*) s'applique à des problèmes dont on dispose déjà d'une solution très lente. Elle propose de procéder ainsi pour résoudre une instance I de taille n :

- (i) Diviser l'instance I en r sous-instances de taille n/c
- (ii) Résoudre les sous-instances à l'aide de la solution très lente
- (iii) Fusionner les solutions des sous-instances en une solution de I

Comme la solution naïve est très lente, l'appliquer sur des sous-instances significativement plus petite est un gain de temps qui compense le coût de la division/fusion.

Remarque.

- Si la solution naïve est polynomiale, il n'y a aucun gain d'ordre de grandeur de complexité car $\left(\frac{n}{c}\right)^k = O(n^k)$.
- Les c peuvent être différentes entre les sous-instances.
- À vrai dire, la rencontre au milieu n'est pas, à ma connaissance, une méthode générale comme le Diviser pour Régner. C'est plus une astuce qui sert parfois, notamment et surtout pour SUBSETSUM.

15. Et l'élaguer efficacement par la méthode de Séparation&Évaluation, cf MPI.

16. Il en existe une si et seulement $P = NP$; ce qui est certes une question ouverte mais la communauté des chercheurs penche plutôt vers $P \neq NP$.

17. Si vous n'avez pas compris la note de bas de page précédente, c'est normal : vous êtes en MP2I alors que les classes de complexité P et NP sont au programme de MPI.

2 Programmation dynamique

2.0 Exemple fil rouge : plus lourd chemin dans une pyramide

2.0.0 Présentation du problème

On considère une pyramide d'entiers comme sur la figure ci-dessous, dans laquelle on veut trouver un chemin descendant de poids maximal (en partant du sommet) :

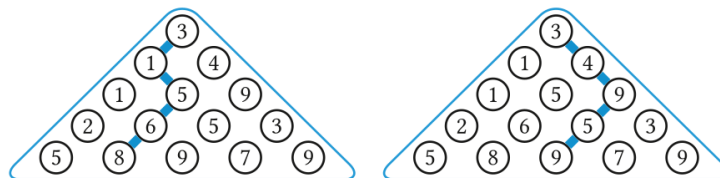


FIGURE 10.14 – Une pyramide d'entiers et deux chemins descendants dans celle-ci

Donnons-nous un peu de vocabulaire :

- Les *enfants* d'un noeud de la pyramide sont les deux noeuds qui sont respectivement juste en dessous à gauche et juste en dessous à droite.
- Un *chemin* descendant est une succession d'enfants depuis un point de départ donné et jusqu'à la ligne du bas.
- Le *poids* d'un chemin est la somme des valeurs des noeuds du chemin.
- Pour un noeud, on définit ses sous-pyramides gauche et droite :

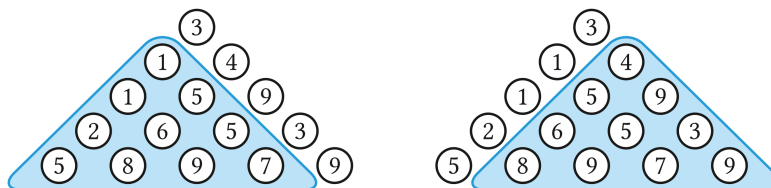


FIGURE 10.15 – Les deux sous-pyramides (ici du sommet). On remarque qu'elles s'intersectent (beau-coup).

Remarque. Les pyramides ne sont *pas* des arbres, car les sous-arbres ne sont pas disjoints. Par exemple, le noeud 5 de la troisième ligne est à la fois l'enfant droit du 1 et l'enfant gauche du 4 de la ligne d'au-dessus.

Dans un premier temps, on va uniquement chercher $s_{i,j}$, le poids d'un plus long chemin descendant depuis le noeud de coordonnées (i, j) .

2.0.1 Solution naïve en OCaml

On va utiliser un tableau de tableaux pour représenter une pyramide en OCaml. Par exemple, la pyramide ci-dessous correspond à :

```
8 let pyramide_cours =
9   [| [| 3 |];
10      [| 1; 4 |];
11      [| 1; 5; 9 |];
12      [| 2; 6; 5; 3 |];
13      [| 5; 8; 9; 7; 9 |]
14   |]
```

 pyramide.ml

Ainsi, la première ligne est `pyramide_cours.(0)`, la seconde ligne `pyramide_cours.(1)`, etc. Remarquez que les enfants du noeud de coordonnée (i, j) sont les noeuds de coordonnées $(i + 1, j)$ et $(i + 1, j + 1)$. D'où :

```
44 let gauche j = j
45 let droite j = j+1
```



Débutons par une solution naïve au problème : comme pour le calcul de la hauteur d'un arbre binaire, on a $s_{i,j} = \text{pyra}_{i,j} + \max(s_{i+1,\text{gauche}(j)}, s_{i+1,\text{droite}(j)})$ (avec *pyra* la pyramide).

FIGURE 10.16 – Solution récursive pour le plus lourd chemin

```
48 (** Renvoie le poids d'un plus lourd chemin descendant
49     dans la pyramide [pyra] qui débute en [(lgn,col)] *)
50 let rec poids_naif pyra lgn col =
51   let nb_lgn = Array.length pyra in
52   if lgn >= nb_lgn then
53     0
54   else
55     let p_gauche = poids_naif pyra (lgn+1) (gauche col) in
56     let p_droite = poids_naif pyra (lgn+1) (droite col) in
57     pyra.(lgn).(col) + max p_gauche p_droite
```



Cependant, si l'on calcule la complexité temporelle de cette fonction avec n le nombre de ligne, l'équation associée est $T(n) = 2T(n - 1) + O(1) = O(2^n)$.

Pourquoi est-ce autant ? C'est pourtant un beau diviser pour régner, non ? *NON!!* Les sous-problèmes auquel on se réduit ne sont pas indépendants : on cherche un chemin dans la « sous-pyramide gauche », et un chemin dans la « sous-pyramide droite », mais celles-ci s'intersectent ! Cela mène à des appels récursifs qui ont lieu plusieurs fois :

FIGURE 10.17 – (Début de) l'arbre d'appel de `poids_naif pyramide_cours 0 0`

Exercice. Dans une exécution de `poids_naif` $p \neq 0$, déterminer combien de fois est calculée $s_{i,j}$

Pour corriger ce défaut, nous allons utiliser de la programmation dynamique !

Remarque. De son côté, la complexité spatiale se calcule comme suit : chaque appel est localement en espace constant, l'arbre d'appel est de hauteur n , d'où une complexité spatiale linéaire en n le nombre de lignes.

2.1 Principe général de la Programmation Dynamique

Définition 9 (Cadre d'application de la programmation dynamique).

La **programmation dynamique** s'applique lorsque l'on résout un problème à l'aide d'une idée récursive ; mais que cette résolution mène à recalculer plusieurs fois les mêmes valeurs. La programmation dynamique consiste alors à mémoriser les valeurs déjà calculées pour ne pas les recalculer.

Remarque. Le fait que le code soit basé sur une idée récursive ne signifie pas que la fonction en elle-même est récursive. Vous avez par exemple déjà codée la dichotomie en itératif alors qu'elle est basée sur une idée récursive.

2.2 Méthode de haut en bas

2.2.0 Pour les pyramides

Nous allons modifier la fonction `poids_naif` pour qu'elle mémorise les valeurs déjà calculées. Plus précisément, on va maintenir un tableau de tableaux `s` tel que `s[i][j]` vaut `None` si $s_{i,j}$ n'a pas déjà été calculée, et `Some s` avec `s` le poids associé sinon :

```

60  (** Renvoie le poids d'un plus lourd chemin descendant
61      dans la pyramide [pyra] qui débute en [(lgn,col)].
62
63      [s] est la matrice des valeurs déjà calculées
64  *)
65  let rec poids_memo s pyra lgn col =
66      (* Si s.(i).(j) n'est pas déjà calculée : la calculer *)
67      if s.(lgn).(col) = None then begin
68          let nb_lgn = Array.length pyra in
69          if lgn = nb_lgn-1 then (* la ligne du bas *)
70              s.(lgn).(col) <- Some pyra.(lgn).(col)
71          else (* pas sur la ligne du bas *)
72              let g = poids_memo s pyra (lgn+1) (gauche col) in
73              let d = poids_memo s pyra (lgn+1) (droite col) in
74              s.(lgn).(col) <- Some (pyra.(lgn).(col) + max g d)
75      end
76      ;
77      (* Ensuite, dans tous les cas : renvoyer la valeur mémorisée *)
78      Option.get s.(lgn).(col)
    
```

 pyramide.ml

Remarque.

- `s` est un tableau, donc son contenu est passé (comme) par référence : c'est pour cela que ça marche ! Quand un appel récursif modifie `s.(lgn).(col)`, cette modification reste là pour les appels suivants !
- À la fin, j'utilise `Option.get` : à une valeur de type `'a option` de la forme `Some p`, elle associe `p` (et sinon lève une exception).

- Pourquoi utiliser des Option ? Une alternative est d'utiliser une valeur particulière qui signifie « la réponse n'a pas encore été calculée ». Par exemple, si j'admets que tous les noeuds contiennent une valeur positive, -1 est une valeur possible pour signifier « pas encore calculé ». Cela a cependant plusieurs inconvénients majeurs :
 - Il faut trouver une telle valeur particulière. En effet, se tromper sur elle plante tout l'algorithme (l'algo croit qu'une valeur n'est pas calculée alors qu'elle l'est...).
 - Utiliser des options permet d'obtenir des erreurs *de type*, à la *compilation* si jamais on les manipule mal et que l'on confond valeur déjà calculée et valeur non encore calculée. C'est *extrêmement* appréciable : le débogage se fait à la compilation (avec un message qui indique la ligne précise de l'erreur) et non à l'exécution¹⁸ !!

Exemple. Déroulons cette algorithme sur la pyramide précédente, en partant de $(0, 1)$:

FIGURE 10.18 – Arbre d'appels poids_memo en partant de $(0, 1)$. La mémorisation revient à élaguer l'arbre.

18. De manière générale c'est, et à raison, un argument majeur des langages fortement typés : le compilateur détecte énormément d'erreurs qui seraient détectés à l'exécution dans d'autres langages.

2.2.1 Calcul de la complexité

Calculer la complexité de `poids_memo`. L'analyse usuelle des fonctions récursives n'est pas très pratique ici, car l'idée de cette fonction est juste de faire en sorte que le pire des cas n'arrive qu'une fois.

À la place, comptons combien de fois chaque valeur est appelée. Remarquons pour cela un lemme central :

Lemme 10.

Dans un algorithme de programmation dynamique mémorisé (de haut en bas), le calcul de la solution d'une instance se fait récursivement au plus une fois.

Démonstration. Cf le code : si la valeur est déjà calculée, on se contente de la renvoyer (et donc on ne continue pas récursivement). Si elle n'est pas calculée, on la calcule récursivement... et on la stocke pour les appels suivants qui ne vont donc pas la recalculer récursivement ! \square

Or, pour `pyra_memo`, l'appel sur (i, j) peut-être provenir d'un appel sur $(i - 1, j - 1)$ (le parent a appelé son enfant droit) ou $(i - 1, j)$ (le parent a appelé son enfant gauche). Chacune de ces deux façons d'appeler (i, j) a lieu au plus une fois (cf le lemme). Ainsi, le nombre d'appels qui a lieu est au plus le double du nombre de cases de s .

En notant n le nombre de lignes, il y a $O(n^2)$ cases. Or, chaque appel effectivement localement un nombre constant d'opérations. Donc la complexité temporelle est de la forme $T(n) = O(n^2)$; c'est à dire linéaire en la taille de la pyramide !!

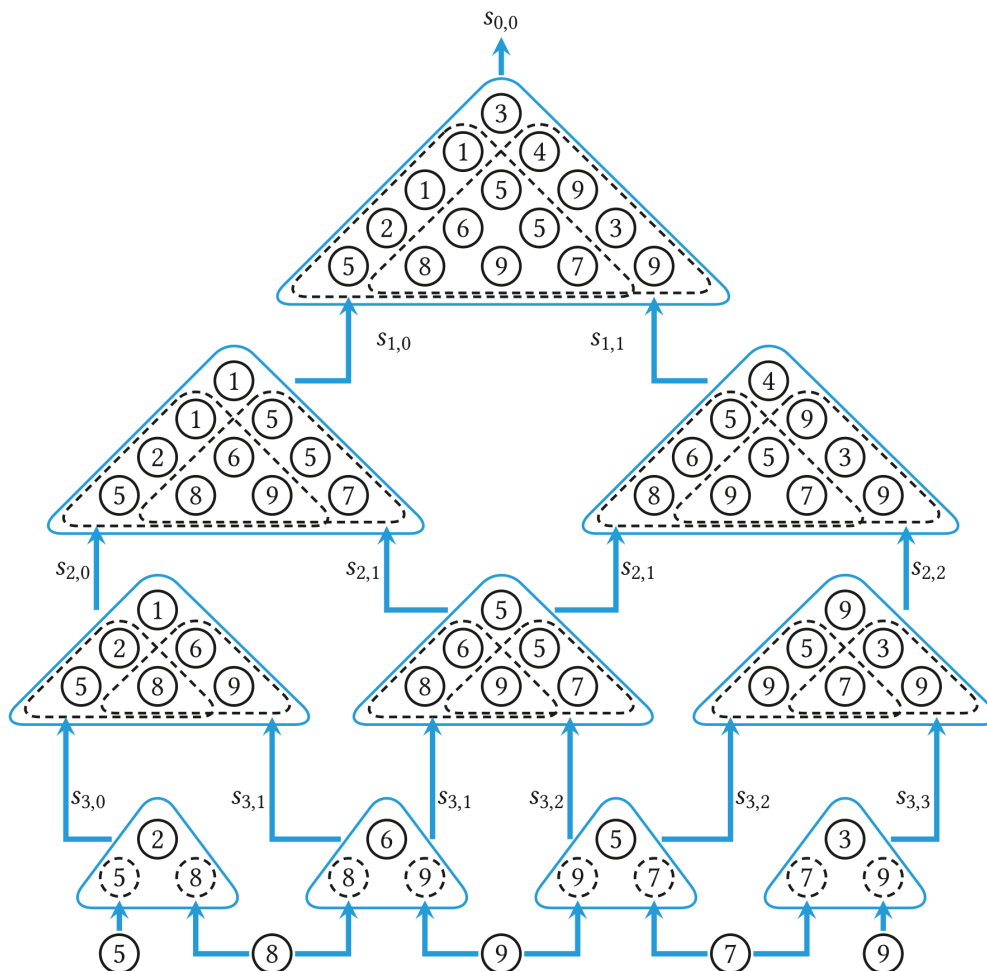


FIGURE 10.19 – Arbre des dépendances entre les instances.

Passons maintenant à la complexité spatiale : il y a les mêmes appels récursifs que dans la solution naïve (qui coûte $O(n)$ en mémoire), mais il faut en plus disposer de la mémoire s qui a les mêmes dimensions que la pyramide : d'où un coût spatial en $O(n^2)$.

La programmation dynamique est un *tradeoff* : on gagne (beaucoup) de temps en payant (un peu) de mémoire.

2.2.2 Pseudo-code générique

Définition 11 (Haut en bas).

En programmation dynamique, la méthode **de haut en bas** consiste à modifier une solution récursive pour qu'elle mémorise les valeurs déjà calculées afin de ne pas les recalculer. Cette mémorisation est aussi appelée la **mémoïsation**.

Voici un pseudo-code générique d'une solution mémoïsée :

Fonction HautEnBas

Entrées : *mem* une mémoire, *x* une entrée

Sorties : *y*, la solution à l'entrée *x*

```

1 si mem[x] n'est pas déjà calculé alors
2   si x est un cas de base alors
3     mem[x] ← solution y du cas de base
4   sinon
5     y ← solution pour x calculée récursivement par HAUTENBAS(mem, ...)
6     mem[x] ← y
7 renvoyer mem[x]
```

Remarque.

- Il faut déjà disposer d'une solution récursive au problème pour pouvoir construire une fonction comme ci-dessus.
En fait, on ne s'attaque pas à un problème en se disant « je vais faire de la programmation dynamique ! ». On s'attaque à un problème en se disant « Je cherche une formule de récurrence, et si (lorsque) j'en trouve une je me demande si elle est accélérable par programmation dynamique. »
- Il est *capital* que le calcul récursif de la solution *y* soit fait avec la fonction mémoïsée : si on utilise la solution naïve... bah elle est très lente, donc notre code aussi¹⁹.
- En pratique, il faut commencer par initialiser la mémoire *mem* (souvent en marquant toutes les valeurs comme n'étant pas encore calculées). Ainsi, une solution de haut en bas est composée d'une fonction mémoïsée comme ci-dessus ; et d'une fonction principale qui initialise la mémoire puis appelle la précédente pour qu'elle calcule la solution.

Exemple. Voici une telle « fonction principale » pour `poids_memo` :

```

83 let poids pyramide =
84   let nb_lgn = Array.length pyramide in
85
86   (* La mémoire est un triangle, comme la pyramide *)
87   let create_lgn lgn =
88     Array.make (lgn+1) None in
89   let s = Array.init nb_lgn create_lgn in
90
91   (* On peut maintenant utiliser [poids_memo s] *)
92   poids_memo s pyramide 0 0
```



pyramide.ml

19. J'aime quand une subtilité s'explique aussi trivialement.

2.3 Méthode de bas en haut

2.3.0 Pour les pyramides

Repartons de la figure 10.19 : on note que les instances qui correspondent à la ligne du bas n'ont aucun pré-requis : on peut les calculer directement. Une fois celles-ci calculées, on peut calculer celles de la ligne d'au-dessus. Et ainsi de suite.

On en déduit la solution non-réursive suivante. Cette fois-ci, je n'utilise pas des Option.²⁰ À la place, vous me voyez utiliser 0 pour remplir les cases non encore calculées. Notez également qu'une ligne numéro lgn a lgn+1 colonnes, indexées de 0 à lgn.

```

97 let poids_bashaut pyramide =
98   let nb_lgn = Array.length pyramide in
99
100  (* On crée la mémoire *)
101  let create_lgn lgn =
102    Array.make (lgn+1) 0 in
103  let s = Array.init nb_lgn create_lgn in
104
105  (* On peut remplir la ligne du bas. *)
106  s.(nb_lgn-1) <- Array.copy pyramide.(nb_lgn-1);
107
108  (* Ensuite on remplir les lignes supérieurs
109   les unes après les autres *)
110  for lgn = nb_lgn - 2 downto 0 do
111    for col = 0 to lgn do
112      let g = s.(lgn+1).(gauche col) in
113      let d = s.(lgn+1).(droite col) in
114      s.(lgn).(col) <- pyramide.(lgn).(col) + max g d
115    done
116  done;
117
118  (* Plus qu'à renvoyer le contenu de s.(0).(0) *)
119  s.(0).(0)
    
```



Remarque.

- On peut créer une ligne fictive en dessous de la ligne du bas, qui stocke les plus lourds chemins vides. Cela simplifie le code en évitant les lignes 107 à 111.
- Cet exemple des pyramides est tiré de *Informatique MP2I/MPI* de Balabonski, Conchon, Filiatre, Nguyen et Sarte. Dans le livre, vous trouverez des codes plus concis que les miens - mais les miens ont des noms plus explicites pour les variables et les quantités intermédiaires.²¹

2.3.1 Calcul de la complexité

Cette analyse de complexité-ci est bien plus standard. Notons $n = nb_lgn$ pour abrégier.

Les lignes 101-103 effectuent n fois la création d'une ligne, donc $O(n^2)$ opérations. La double boucle (110-116) a un corps en $O(1)$ donc la boucle interne est en $O(lgn)$ donc le tout en $O(\sum_{l=0}^{n-2} l) = O(n^2)$. Les autres lignes de code sont de complexité inférieure. D'où une complexité temporelle $T(n) = O(n^2)$, comme pour la version de haut en bas...

Sauf que dans ce $O(n^2)$, la composante linéaire liée au coût de la récursivité a disparu ! Or, comme la pile mémoire (où sont empilés les appels) est généralement plus limitée que le tas mémoire (où sont généralement allouées les mémoires de la programmation dynamique), ce n'est pas négligeable.

²⁰. Je devrais - encore une fois, cela me protège efficacement des erreurs de programmation - ; mais cela rend le code un peu plus intimidant à lire.

²¹. Disons que les leurs ont une longueur moins intimidante ; mais j'espère que les miens sont plus simples à comprendre.

2.3.2 Pseudo-code générique

Définition 12 (Bas en haut).

En programmation dynamique, la méthode **de bas en haut** consiste à calculer les instances les unes après les autres dans un ordre fixé qui soit compatible avec les dépendances de la formule récursive.

Ainsi, il n'y a pas besoin de récursivité puisque l'ordonnancement assure que les solutions des sous-instances dont on a besoin ont déjà été calculées.

Voici un pseudo-code récursif d'une solution de bas en haut. On se donne un ordre (bien fondé) \preceq sur les instances qui soit compatible avec l'ordre de calcul :

Fonction BasEnHaut

Entrées : x une instance

Sorties : La solution y de cette instance

```

1  $mem \leftarrow$  mémoire qui stockera les résultats
2 pour chaque instance  $x' \preceq x$ , par ordre croissant faire
3    $y' \leftarrow$  solution de  $x'$  (calculée à partir de  $mem[x'']$  pour  $x'' \preceq x'$ )
4    $mem[x'] \leftarrow y'$ 
5 renvoyer  $mem[x]$ 
```

Remarque.

- Pour cette façon de faire, il faut à la fois disposer d'une formule de récurrence (pour la ligne 7 du pseudo-code); et d'un ordre de calcul compatible avec cette formule.
- Pour certain-es auteur-ices, le terme « programmation dynamique » désigne spécifiquement la méthode de bas en haut, l'autre étant la « mémoïsation ». À titre personnel²², je considère qu'il est plus pertinent d'appeler « programmation dynamique » toute solution basée sur une idée récursive et sur la mémorisation des résultats intermédiaires pour ne pas les recalculer. Mais sachez que certains sujets de concours pourront être en désaccord avec moi.

2.3.3 Optimisation de la mémoire

La méthode de bas en haut permet d'optimiser la mémoire : on peut « oublier » les solutions des instances qui ne serviront plus. On peut parfois réussir à gagner ainsi un ou plusieurs ordres de grandeur de complexité spatiale !

Exemple. Pour les pyramides, dans la version de bas en haut, pour calculer une ligne lgn , on utilise la ligne $lgn + 1$ du dessous. Mais une fois lgn calculée, la ligne $lgn + 1$ ne servira plus !

En d'autres termes, on peut modifier le calcul de bas en haut pour qu'il n'utilise que deux lignes comme mémoire : la ligne du dessous (calculée), et la ligne actuelle que l'on est en train de calculée.

FIGURE 10.20 – Les deux lignes de la mémoire dont on a besoin

22. Mais pas que, beaucoup de collègues de MP2I/MPI partagent mon avis.

On obtient le code ci-dessous :

```

122 let poids_2lgn pyramide =
123   let nb_lgn = Array.length pyramide in
124
125   let ligne_dessous = Array.copy (pyramide.(nb_lgn - 1)) in
126   let ligne = Array.make (nb_lgn + 1) 0 in
127
128   (* Invariant : ligne_dessous.(col) est le poids
129      d'un plus lourd chemin descendant (lgn,col) *)
130   for lgn = nb_lgn - 2 downto 0 do
131     for col = 0 to lgn do
132       let g = ligne_dessous.(gauche col) in
133       let d = ligne_dessous.(droite col) in
134       ligne.(col) <- pyramide.(lgn).(col) + max g d
135     done;
136
137     (* lgn va augmenter : ligne devient la ligne du dessous *)
138     for col = 0 to nb_lgn do
139       ligne_dessous.(col) <- ligne.(col)
140     done
141   done;
142
143   ligne.(0)
    
```

 pyramide.ml

Mais on peut même faire encore mieux ! Remarquons que *ligne_dessous[col]* est utilisé pour *ligne[col - 1]* et *ligne[col]*. Autrement dit, on peut fusionner les deux lignes en une seule ; dont le début vaut *ligne[col]* et la fin vaut *ligne_dessous[col]* :

```

148 let poids_opt pyramide =
149   let nb_lgn = Array.length pyramide in
150   let s = Array.copy pyramide.(nb_lgn - 1) in
151   for lgn = nb_lgn - 2 downto 0 do
152     for col = 0 to lgn do
153       s.(col) <- pyramide.(lgn).(col) + max s.(gauche col) s.(droite col)
154     done
155   done;
156   s.(0)
    
```

 pyramide.ml

Exercice. Formaliser l'explication précédente en un (ou des) invariant(s) qui prouve la correction de `poids_opt` .

La fonction `poids_opt` (ainsi que `poids_2lgn`) a la même complexité temporelle que les programmations dynamiques précédentes... mais est en espace $O(nb_lgn)$! On a gagné un ordre de grandeur ! Notez que c'est le même espace que la solution naïve : nous avons donc amélioré significativement le temps sans plus rien payer en espace !

2.4 Comparaison des deux méthodes

Résumons les avantages et inconvénients :

Haut en Bas (mémoïsation)	Bas en Haut
Simple à écrire à partir d'une version récursive naïve	Demande de déterminer un ordre de calcul (et de ne pas se tromper dans la manipulation des indices des boucles).
Simple à prouver par induction	Simple à prouver avec des invariants ; en tous cas jusqu'à un certain niveau d'optimisation de la complexité spatiale...
Ne calcule que les valeurs nécessaires	Selon la qualité de l'ordre de calcul, peut calculer des valeurs inutiles
	Peut optimiser son coût en espace

Remarque. Mentionnons tout de même que la version naïve récursive a l'avantage d'être la plus simple à coder et de pouvoir aider à déboguer une programmation dynamique.

2.5 Message d'utilité publique...

LA PROGRAMMATION DYNAMIQUE, CE N'EST PAS REMPLIR DES TABLEAUX. C'EST OPTIMISER UNE SOLUTION RÉCURSIVE.

Pourquoi est-ce que j'insiste ? Parce que :

- La mémoire utilisée n'est pas obligatoirement un tableau.
- Et surtout... pour concevoir une solution en programmation dynamique, il faut d'abord et avant tout concevoir une solution récursive. Le tableau, s'il apparaît, vient seulement ensuite et uniquement comme réponse à « dans quoi vais-je stocker mes résultats intermédiaires ? ».

2.6 (Re)construction de la solution optimale

2.6.0 Fonctionnement générique

Jusqu'à présent, nous avons vu comment calculer la *valeur* d'une solution optimale. Mais quid de la solution optimale elle-même ?

Exemple. On sait calculer le *poids* d'un plus lourd chemin descendant dans une pyramide, mais quel est le plus lourd chemin associé ?

Définition 13 ((Re)construction de la solution optimale en programmation dynamique).

Il y a deux grandes façons de calculer une solution optimale par programmation dynamique :

- En plus de stocker les valeurs des solutions optimales des sous-problèmes, stocker une solution optimale associée. On peut par exemple stocker dans la mémoire des paires (*valeur*, *solution*) ; ou utiliser deux mémoires.
- À la fin du calcul de la valeur de la solution optimale, reconstruire la solution optimale : on sait quel sous-problème de l'entrée est optimal donc on connaît le « premier choix » de la solution optimale, et ainsi de suite.

2.6.1 Pour les pyramides : mémorisation des solutions optimales

Le code que je présente ici repart de la solution de bas en haut (optimisée), mais on peut tout à fait le faire de haut en bas.

On va représenter un chemin comme la liste des indices (lgn, col) . On va utiliser deux tableaux :

- s qui stocke $s_{i,j}$ comme précédemment
- sol qui stocke les solutions partielles associées aux valeurs de s .

```

161 let chemin_opt pyramide =
162   let nb_lgn = Array.length pyramide in
163   let s = Array.copy pyramide.(nb_lgn-1) in
164   let sol = Array.init (nb_lgn+1) (fun col -> [nb_lgn-1, col]) in
165   for lgn = nb_lgn-2 downto 0 do
166     for col = 0 to lgn do
167       let g, d = s.(gauche col), s.(droite col) in
168       s.(col) <- pyramide.(lgn).(col) + max g d;
169       let sol' = if g >= d then sol.(gauche col) else sol.(droite col) in
170       sol.(col) <- (lgn,col) :: sol'
171     done
172   done;
173   sol.(0)
    
```

 pyramide.ml

La complexité temporelle de cette méthode est en $O(n^2)$ comme précédemment. La complexité spatiale, elle, change : en plus du coût spatial $O(n)$ que l'on avait précédemment, il y a désormais le coût des différentes listes. Dans le pire des cas, celle-ci sont un $O(n^2)$ donc le coût spatial redevient $O(n^2)$.

2.6.2 Pour les pyramides : reconstruction de la solution optimale

Le code que je présente ici prend en argument une mémoire s telle que $s.(i).(j) = s_{i,j}$ et renvoie un chemin optimal (représenté par une liste comme précédemment). Il peut donc être effectué à la fin de la méthode de haut en bas, ou de bas en haut non-optimisé. Il n'est par contre pas compatible avec les méthodes de bas en haut optimisée en espace (puisque l'optimisation consiste à « oublier » des solutions des sous-problèmes, dont on a besoin ici).

```

179 let rec reconstruit s lgn col =
180   let nb_lgn = Array.length s in
181   if lgn = nb_lgn - 1 then
182     [(lgn, col)]
183   else
184     if s.(lgn+1).(gauche col) >= s.(lgn+1).(droite col) then
185       (lgn, col) :: reconstruit s (lgn+1) (gauche col)
186     else
187       (lgn, col) :: reconstruit s (lgn+1) (droite col)
    
```

 pyramide.ml

La complexité temporelle et spatiale de cette fonction est $O(n)$, mais elle suit l'exécution d'une fonction qui remplit s en temps et espace $O(n^2)$.

Chapitre 11

GRAPHES NON-PONDÉRÉS

Notions	Commentaires
Graphe orienté, graphe non orienté. Sommet (ou noeud); arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité, forte connexité. Graphe orienté acyclique. Arbre en tant que graphe connexe acyclique. Forêt. Graphe biparti.	Notation : graphe $G = (S, A)$, degrés $d_+(s)$ et $d_-(s)$ dans le cas orienté. On n'évoque pas les multi-arcs. On représente un graphe orienté par une matrice d'adjacence ou par des listes d'adjacence.

Extrait de la section 3.4 du programme officiel de MP2I : « Structures de données relationnelles ».

Notions	Commentaires
Notion de parcours (sans contrainte). Notion de parcours en largeur, en profondeur. Notion d'arborescence d'un parcours.	On peut évoquer la recherche de cycle, la bicolorabilité d'un graphe, la recherche de plus courts chemins dans un graphe à distance unitaire.
Accessibilité. Tri topologique d'un graphe orienté acyclique à partir de parcours en profondeur. Recherche des composantes connexes d'un graphe non orienté.	On fait le lien entre accessibilité dans un graphe orienté acyclique et ordre.

Extrait de la section 4.5 du programme officiel de MP2I : « Algorithmique des graphes ».

SOMMAIRE

0. Graphes non-orientés	239
0. Introduction	239
1. Définitions	240
<i>Vocabulaire (p. 240). Chemins et connexité (p. 241).</i>	
2. Graphes non-orientés remarquables	245
3. Propriétés combinatoires	246
<i>Liens entre S et A (p. 246). Bornes de connexité et d'acyclicité (p. 247). Liens avec les arbres (p. 249).</i>	
4. (Mi-HP) Coloration de graphes	251
<i>Degré chromatique (p. 251). (HP) Cas des graphes planaires (p. 252). 2-coloration de graphe (p. 253). (MPI) 3-coloration de graphe : NP-difficile (p. 254).</i>	
1. Graphes orientés	255
0. Définitions	255
<i>Vocabulaire (p. 255). Chemins orientés (p. 256).</i>	
1. Études des composantes fortement connexes	257
2. Implémentation	260
0. Matrice d'adjacence	260
1. Listes d'adjacence	261
2. Comparatif	263

3. Parcours de graphes.....	264
0. Parcours générique	264
<i>Parcours depuis un sommet (p. 264). Parcours complet (p. 266). Complexité (p. 266). Arbre de parcours (p. 267).</i>	
1. Parcours en largeur	270
<i>Définition et propriété (p. 270). Une optimisation : le marquage anticipé (p. 270). Implémentation en OCaml (p. 271).</i>	
2. Parcours en profondeur	272
<i>Définition et propriétés (p. 272). Pas de marquage anticipé!! (p. 274). Implémentation en OCaml (p. 275).</i>	
3. Application : calcul des composantes connexes	276
4. Application : calcul d'un ordre topologique	277
<i>Définition (p. 277). Construction d'un ordre topologique (p. 277). Détection de cycles (p. 278). Implémentation en OCaml (p. 279).</i>	
4. Graphes bipartis.....	281
0. Définition	281
1. Couplage (dans un graphe biparti ou non)	282

0 Graphes non-orientés

0.0 Introduction

Voici un graphe :

FIGURE 11.1 – Un graphe G_{ex}

Il est composé de **sommets** (0, 1, 2,...) qui peuvent être liés par des **arêtes**. Un enchainement d'arête forme un **chemin** ou un **cycle**.

Remarque.

- C'est une structure très naturelle pour représenter des réseaux ! Par exemple des cartes routières, des connexions entre des serveurs, etc.
- Les graphes ont des applications *très* vastes : une arête entre deux sommets correspond à un lien, une *relation* entre ces deux sommets. On dit que les graphes sont une **structure de données relationnelles**.
- Le problème originel de la théorie des graphes est un problème de réseaux. Il s'agit du problème du pont de Königsberg ; qui demande à déterminer s'il existe un cycle qui passe une et une seule fois par chaque pont¹ dans ce schéma du centre-ville de Königsberg (circa 1735) :

(a) Schéma du centre-ville

(b) Représentation en graphe

FIGURE 11.2 – Le problème des ponts de Königsberg. Chaque rive est un sommet, et chaque pont une arête.

- Un autre problème se modélisant bien comme des graphes est le problème d'attribution de créneau : on considère des activités qui demandent à réserver une même salle, chacune sur un créneau fixé. Deux activités ne peuvent pas réserver la même salle en même temps. On peut représenter cela comme un graphe : chaque activité est un sommet, et deux activités incompatibles sont reliées. On veut donc trouver un ensemble de sommet sans « arête interne » le plus grand possible : on dit que l'on recherche un indépendant maximal.

1. On appelle cela un chemin eulérien. C'est un problème plutôt simple à résoudre.

(a) Des créneaux d'activité

(b) Représentation en graphe

FIGURE 11.3 – Indépendant maximal sur un graphe d'intervalle

0.1 Définitions

0.1.0 Vocabulaire

Convention 1.

Soit E un ensemble :

- On note $\mathcal{P}_2(S)$ l'ensemble des parties d'ordre 2 de S , c'est à dire des $\{x, y\} \subseteq S$ (avec $x \neq y$).
- On note $|E|$ le cardinal de E .

Définition 2 (Graphe non-orienté).

Un **graphe non-orienté** est un couple $G = (S, A)$ où :

- S est un ensemble fini d'éléments appelés des **sommets** (ou **noeuds**).
- $A \subseteq \mathcal{P}_2(S)$ est un ensemble d'**arêtes**.

Remarque.

- En anglais, un *sommet* est un *vertex* et une arête une *edge*. Aussi on note parfois les graphes $G = (V, E)$.
- Les arêtes sont symétriques : $\{u, v\} = \{v, u\}$.
- Cette définition interdit les **boucles**, c'est à dire les arêtes d'un sommet vers lui-même. Elle interdit aussi les **multi-arêtes**, c'est à dire une arête qui relie plus de 2 sommets.
- Graphiquement, on représente les sommets avec des ronds et les arêtes avec des traits.
- Les valeurs de S sont une façon d'identifier les sommets, mais aussi une façon de les étiquetter.
- On peut aussi vouloir étiquetter les arêtes : c'est par exemple ce que l'on fera dans les graphes pondérés. Un formalisme agréable pour cela est d'ajouter au graphe une fonction qui à une arête associe son étiquette.

Convention 3.

On note uv l'arête $\{u, v\}$.

Définition 4 (Vrac de vocabulaire).

Dans $G = (S, A)$ un graphe :

- Pour $a = uv$ une arête de A , on dit que u et v sont les **extrémités** de l'arête. Réciproquement, on dit que l'arête est **incidente** à ses extrémités.
- Pour u et v deux sommets de S , on dit que u et v sont **adjacents** si $uv \in A$.
- Le nombre de sommets $|S|$ est appelé l'**ordre** du graphe.
- Pour u un sommet de S , le nombre $\deg(u)$ d'arêtes incidentes à u est appelé le **degré** de u .

Définition 5 (Sous-graphe et graphe induit).

Soit $G = (S, A)$ un graphe.

Pour $S' \subseteq S$ et $A' \subseteq A$, on dit que $G' = (S', A')$ est un **sous-graphe** de G .

Pour $S' \subseteq S$, en posant $A_{|S'} = A \cap \mathcal{P}_2(S')$, on dit que $G_{|S'} = (S', A_{|S'})$ est le **sous-graphe induit** par S' de G .

Exemple.

(a) Le graphe K_6 (b) Un sous-graphe de K_6 (c) Le sous-graphe K_6 induit par $\{0; 2; 4; 5\}$

FIGURE 11.4 – Sous-graphes et sous-graphe induit

Remarque. Une autre transformation usuelle est l'obtention d'un mineur : un mineur d'un graphe est obtenu par répétition d'opérations parmi les 3 suivantes :

- Supprimer un sommet.
- Supprimer une arête.
- Contracter une arête, c'est à dire fusionner ses deux extrémités en un seul sommet et supprimer la boucle créée.

Elle est notamment utile pour caractériser les graphes planaires : un graphe peut-être représenté dans le plan sans que 2 de ses arêtes ne s'intersectent si et seulement si il ne contient ni K_5 ni $K_{3,3}$ comme mineurs (ces deux graphes sont définis plus loin).

0.1.1 Chemins et connexité

Définition 6 (Chemins, cycles).

Dans $G = (S, A)$ un graphe :

- Un **chemin** de longueur p est une suite finie v_0, v_1, \dots, v_p de $p + 1$ sommets tels que pour tout $i \in \llbracket 0; p \rrbracket$, $v_i v_{i+1} \in A$.
Si de plus toutes ces arêtes $v_i v_{i+1}$ sont deux à deux distinctes, on dit que le chemin est **simple**.
Si de plus tous les v_i sont distincts, on dit que le chemin est **élémentaire**.
- Un **cycle** est un chemin simple v_0, \dots, v_p tel que $v_0 = v_p$.
Si de plus v_0, \dots, v_{p-1} est élémentaire, on dit que le cycle est **élémentaire**.

Exemple.

FIGURE 11.5 – Un graphe contenant des chemins et des cycles

Remarque.

- La longueur d'un chemin ou d'un cycle est son nombre d'arêtes. Ainsi, la hauteur d'un arbre correspond à cette définition.
- Un cycle est de longueur au moins 3. En effet, un cycle de longueur 2 serait de la forme x, y, x mais ce chemin n'est pas simple.
- Un chemin élémentaire est simple puisque si les sommets sont distincts les arêtes le sont aussi.
- Les terminologies peuvent changer. Notamment, il arrive que « chemin simple » désigne « chemin élémentaire »... Toujours penser à vérifier les définitions utilisées !

Lemme 7 (Facteur et concaténation de chemins).

Soit $G = (S, A)$ un graphe non-orienté et s_0, \dots, s_p un chemin. Alors :

- Pour tous $0 \leq i \leq j \leq p$, s_i, \dots, s_j est un chemin. Si le chemin initial était simple (resp. élémentaire), celui-ci est également simple (resp. élémentaire).
- Si s_p, \dots, s_q est un chemin, alors $s_0, \dots, s_p, \dots, s_q$ est un chemin.

Démonstration. Brièvement :

- Si un \forall est vrai sur un ensemble, alors il est vrai sur tout sous-ensemble. À appliquer à $s_k s_{k+1} \in A$; ainsi qu'à simple et élémentaire.
- Si un \forall est vrai sur deux ensembles, alors il est vrai sur leur union. De même.

□

Définition 8 (Distance).

Soit G un graphe non-orienté et u et v deux sommets du graphe. On appelle **distance** de u à v , notée $d(u, v)$, le minimum des longueurs des chemins de u à v .

Exemple. Cf exemple précédent.

Remarque. En particulier :

- $d(u, v) = 0$ si et seulement si $u = v$
- $d(u, v) = 1$ si et seulement si $uv \in A$
- $d(u, v) = +\infty$ si et seulement si v n'est pas accessible depuis u

Lemme 9 (Éléментарité d'un chemin minimal).

Soit $G = (S, A)$ un graphe orienté et u et v deux sommets du graphe. Alors tout chemin de u à v de longueur $d(u, v)$ est élémentaire.

Démonstration. Si $d(u, v) = +\infty$, il n'existe aucun tel chemin, donc le résultat est immédiat. On suppose donc $d(u, v) < +\infty$.

Considérons un chemin de u à v de longueur $d = d(u, v)$ non-élémentaire, c'est à dire de la forme $\underbrace{s_0, \dots, s_{i-1}, s_i}_{=u}, s_{i+1}, \dots, s_{j-1}, s_j, s_{j+1}, \dots, \underbrace{s_d}_{=v}$ avec $s_i = s_j$ (et $i < j$).

FIGURE 11.6 – Un chemin non-élémentaire

En particulier, s_0, \dots, s_i est un chemin de u à s_i , et comme $s_i = s_j$, s_i, s_{j+1}, \dots, s_d est un chemin de s_i à v . Donc $s_0, \dots, s_i, s_{j+1}, \dots, s_d$ est un chemin de u à v de longueur $d(u, v) - (j - i) < d(u, v)$. Absurde. \square

Remarque.

- Il est très important de savoir formaliser un chemin comme une suite de sommets. Une bonne preuve de graphe, c'est un dessin convainquant accompagné d'une preuve qui formalise rigoureusement ce dessin !
- L'absurde n'est pas nécessaire : on prouve que dans le chemin $s_i = s_j$ implique $i = j$ et l'élémentarité s'en déduit. Il me permet uniquement de gagner quelques mots.

Théorème 10 (Lemme d'extraction de chemin).

Soit $G = (S, A)$ un graphe non-orienté et u et v deux sommets du graphe. S'équivalent :

- (0) Il existe un chemin de u à v .
- (1) Il existe un chemin simple de u à v .
- (2) Il existe un chemin élémentaire de u à v .

Démonstration. (2) \implies (1) et (1) \implies (0) sont immédiats par définition.

Montrons que (0) \implies (2) : s'il existe un chemin de u à v , alors $d(u, v) < +\infty$. Considérons donc un chemin de longueur minimale de u à v : d'après le lemme précédent, il est élémentaire. \square

Remarque.

- Le nom de ce lemme est inventé par moi. Je ne lui connais pas de nom officiel ; mais comme ce résultat très souvent le nommer est confortable.
- En pratique, on peut extraire un chemin simple d'un chemin, et un chemin élémentaire d'un chemin simple (d'où le nom que je donne au lemme). Notez toutefois que cette preuve, basée sur le lemme 9, montre (par récurrence) comment extraire un chemin élémentaire d'un chemin, mais pas comment extraire un chemin simple non-élémentaire.
C'est rarement un besoin que l'on rencontre, donc je ne m'attarde pas dessus. L'idée est la même : si l'on passe deux fois par une même arête, on peut enlever tout ce qu'il y a entre les deux passages (ainsi que le second passage). Continuer récursivement. Servir chaud.

- L'extraction de cycle est plus embêtante à formaliser : on aurait envie de dire « s'il existe un chemin non-vide de u à u alors il existe un cycle »... mais c'est faux :

FIGURE 11.7 – Cas problématique de l'extraction de cycle

Ce cas correspond au cas où le chemin simple que l'on extrait du chemin est... vide, donc pas un cycle. Il faut interdire ce cas pour que l'extraction de cycle fonctionne.

Théorème 11 (Lemme d'extraction de cycle).

Soit G un graphe non-orienté. S'équivalent :

- (0) Il existe un chemin d'un sommet u à lui-même dans lequel il existe une arête empruntée une seule fois.
- (1) G contient un cycle.
- (2) G contient un cycle élémentaire.

Démonstration. Laissée en TD. Vous en prouvez aussi une variante en MPI. □

Définition 12 (Acyclicité).

Un graphe est dit **acyclique** s'il ne contient pas de cycle. On parle aussi de **forêt**.

Définition 13 (Composantes connexes).

Dans un graphe G , la relation d'accessibilité, c'est à dire la relation binaire « v est accessible depuis u », est une relation d'équivalence. Ses classes d'équivalence sont appelées les **composantes connexes**.

En particulier, les composantes connexes partitionnent les sommets du graphe, et toute arête relie deux sommets de la même composante.

Si u est un sommet, on note C_u la composante connexe de u .

Démonstration. Pourvons brièvement que la relation d'accessibilité est bien d'équivalence :

- Tout sommet est accessible depuis lui-même par un chemin de longueur 0. D'où la réflexivité.
- Si v est accessible depuis u , alors il existe s_0, \dots, s_p un chemin de longueur p de $u = s_0$ à $v = s_p$. Mais comme pour tout arête xy , $xy \in A \iff yx \in A$, s_p, \dots, s_0 est aussi un chemin de v à u . D'où la symétrie.
- Si v est accessible depuis u et w depuis v , alors il existe s_0, \dots, s_p un chemin de $u = s_0$ à $v = s_p$ et t_0, \dots, t_q de $v = t_0$ à $w = t_q$. Mais comme $s_p = t_0$, $s_0, \dots, s_p, t_1, \dots, t_q$ est un chemin de u à w . D'où la transitivité.

FIGURE 11.8 – Transitivité de l'accessibilité



Exemple. Le graphe initial de ce cours, G_{ex} , est partitionné en deux composantes connexes.

Définition 14 (Graphe connexe).

Un graphe est dit **connexe** s'il possède une seule composante connexe.

0.2 Graphes non-orientés remarquables

Certains graphes ou familles de graphes sont des exemples classiques. On nomme :

- P_n le chemin élémentaire de longueur n .

(a) P_1

(b) P_3

(c) P_6

FIGURE 11.9 – Chemin élémentaire P_n

- C_n le cycle élémentaire de longueur n :

(a) C_3

(b) C_5

(c) C_8

FIGURE 11.10 – Cycle C_n

- K_n la **clique** (aussi appelé **graphe complet**) à n sommets, c'est à dire le graphe d'ordre n où toutes les arêtes xy possibles existent :

(a) K_3

(b) K_4

(c) K_5

FIGURE 11.11 – Clique K_n

- $K_{p,q}$ la **clique bipartie** (aussi appelé **graphe complet**) à p et q sommets, c'est à dire le graphe d'ordre où les sommets sont partitionnés en deux parties de p et q avec toutes les arêtes possibles entre les parties et aucune au sein d'un groupe :

(a) $K_{1,3}$ (b) $K_{3,3}$ (c) $K_{3,4}$ FIGURE 11.12 – Clique $K_{p,q}$

- Le **graphe de Petersen** est un contre-exemple à de *très* nombreuses conjectures sur les graphes. Il est donc très utile à avoir en tête :

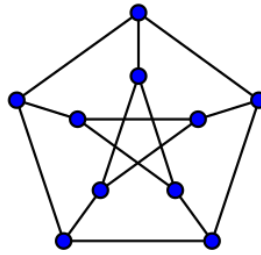


FIGURE 11.13 – Graphe de Petersen (src : Wikipédia)

0.3 Propriétés combinatoires

Dans toute cette sous-section, n désigne le nombre de sommets du graphe étudié.

0.3.0 Liens entre $|S|$ et $|A|$

Proposition 15 (Nombre maximal d'arêtes).

Dans $G = (S, A)$ un graphe non-orienté :

$$|A| \leq \frac{n(n-1)}{2}$$

Démonstration. $|\mathcal{P}_2(S)| = \binom{n}{2} = \frac{n(n-1)}{2}$

□

Proposition 16 (Nombre d'arêtes et degrés).

Dans $G = (S, A)$ un graphe non-orienté :

$$|A| = \frac{\sum_{s \in S} \deg(s)}{2}$$

Démonstration. Informellement : quand on somme les degrés, on compte chaque arête exactement deux fois (une fois par sommet incident).

Formellement : posons $1_{a,s}$ l'indicatrice de « s est incident à a ». Alors :

$$\begin{aligned}
\sum_{s \in S} \deg(s) &= \sum_{s \in S} \sum_{a \in A} \mathbb{1}_{a,s} \\
&= \sum_{a \in A} \sum_{s \in S} \mathbb{1}_{a,s} && \text{car les sommes sont finies} \\
&= \sum_{a \in A} 2 && \text{car chaque arête a exactement 2 sommets incidents} \\
&= 2|A|
\end{aligned}$$

□

0.3.1 Bornes de connexité et d'acyclicité

Proposition 17 (Connexité et nombre d'arêtes).

Un graphe non-orienté connexe d'ordre n possède au moins $n - 1$ arêtes.

Démonstration. Par récurrence sur n .

- Initialisation : C'est immédiat pour le graphe vide ainsi que pour les graphes à 1 sommet.
- Hérédité : Supposons la propriété vraie jusqu'au rang $n > 1$ exclu, montrons-la vraie au rang n .

Soit $G = (S, A)$ un graphe non-orienté connexe avec $|S| = n$.

Comme G est connexe, pour tout $s \in S$, $\deg(s) > 0$ (un sommet de degré 0 est isolé donc non connecté au reste du graphe). Distinguons maintenant deux cas :

- Si il existe $s_0 \in S$ de degré 1 : on va se ramener à $G_{|S \setminus \{s_0\}}$. En effet, si celui-ci est connexe, il a $n - 1$ sommets et $|A| - 1$ arêtes donc lui appliquer l'HR donne $|A| - 1 \geq (n - 1) - 1$ qui permet de conclure.

FIGURE 11.14 – $G_{|S \setminus \{s_0\}}$

Montrons donc que $G_{|S \setminus \{s_0\}}$ est connexe (pour lui appliquer l'HR). Soient u et v deux sommets de G distincts de s_0 . Comme G est connexe (et par lemme d'extraction), considérons un chemin élémentaire de u à v et montrons que ce chemin n'utilise pas s_0 :

- Il n'en est pas une extrémité car u et v sont distincts de s_0 .
- Il n'en est pas un sommet intermédiaire : notons t l'unique voisin de s_0 . Si s_0 est un sommet intermédiaire du chemin élémentaire de u à v , alors l'arête menant à s_0 est ts_0 et l'arête suivante est s_0t donc le chemin n'est pas élémentaire. Contradiction.

Aussi, tout chemin de u à v dans G est un chemin de u à v dans $G_{|S \setminus \{s_0\}}$. Donc la connexité de G implique celle de $G_{|S \setminus \{s_0\}}$. On conclut comme indiqué au début de ce point.

- Sinon, pour tout $s \in S$ on a $\deg(s) \geq 2$. Donc :

$$n = \frac{1}{2} \sum_{s \in S} \deg(s) \leq \frac{1}{2} \sum_{s \in S} \deg(s) = |A|$$

□

Lemme 18.

Soit $G = (S, A)$ un graphe non-orienté non-vide dont tous les sommets sont de degré au moins 2. Alors il contient un cycle.

Démonstration. Idée de la preuve : on part d'un sommet, et on avance dans le graphe (il n'y a pas de cul de sac par hypothèse). Au bout d'un moment, on repasse sur un sommet déjà visité et on en déduit un cycle.

FIGURE 11.15 – La suite (s_i) construite et le cycle atteint

On définit la suite (s_i) comme suit : s_0 est un sommet de S fixé, s_1 un voisin de s_0 , et pour $i > 1$, s_{i+1} est un voisin de s_i distinct de s_{i-1} (possible car $\deg(s_i) \geq 2$). Comme S est fini, il existe des sommets qui se répètent dans cette suite. Notons j l'indice de la première répétition, c'est à dire le premier indice de la suite tel qu'il existe $i < j$ tel que $s_i = s_j$. Alors s_i, \dots, s_j est un chemin de s_i à s_i par construction.

Montrons que ce chemin est simple^{2,3}. S'il ne l'est pas, une arête $s_k s_{k+1}$ est empruntée deux fois, avec $k+1 \leq j$. Mais alors on passe deux fois par s_k ... alors que s_j correspond par définition à la première répétition. Contradiction.

FIGURE 11.16 – La contradiction

On a donc trouvé un chemin simple de longueur > 1 (car $i < j$) de s_i à s_i , c'est à dire un cycle. \square

Proposition 19 (Borne supérieure d'acyclicité).

Un graphe non-orienté acyclique non-vide d'ordre n possède au plus $n - 1$ arêtes.

Démonstration. Procédons par récurrence sur n :

- Initialisation : Pour $n = 1$, c'est immédiat.
- Hérédité : Supposons la propriété vraie jusqu'au rang $n > 1$ exclu, montrons-la vraie au rang n . Soit G un graphe non-orienté acyclique d'ordre n .

D'après le lemme précédent, G possède un sommet s_0 de degré au plus 1. Considérons à nouveau $G_{[S \setminus \{s_0\}]}$: tout cycle du graphe induit est un cycle de G , donc $G_{[S \setminus \{s_0\}]}$ est acyclique. Donc par HR ce graphe induit a au plus $n - 2$ arêtes. Or, $\deg(s_0) \leq 1$ donc G a au plus une arête de plus que $G_{[S \setminus \{s_0\}]}$, d'où $|A| \leq n - 1$.

2. On ne peut pas juste en extraire un chemin simple : l'extraction pourrait extraire le chemin vide qui est simple mais n'est pas un cycle...

3. Je n'ai pas formalisée l'extraction de cycle, donc on doit faire sans !

FIGURE 11.17 – $G_{[S \setminus \{s_0\}]}$

□

0.3.2 Liens avec les arbres

Les arbres au sens du cours sur les arbres ne sont pas stricto sensu des graphes non-orientés (ils ont une notion « de haut en bas »). Toutefois, il est très naturel de vouloir transformer considérer un arbre comme un graphe.

Définition 20 (Arbre non-enraciné).

On appelle **arbre non-enraciné** un graphe (non-orienté) acyclique connexe.

Remarque.

- Parfois, on appelle *arbre* un graphe (non-orienté) acyclique connexe et *arbre enraciné* un arbre au sens du cours sur les arbres. Encore une fois, vérifiez les définitions avec lesquelles vous travaillez !
- Un arbre non-enraciné est un arbre dont on a oublié qui est parent et enfant sur chaque arête, ainsi que l'ordre gauche-droite éventuel sur les enfants.
- Réciproquement, on peut enraciner un arbre non-enraciné en « attrapant un sommet et soulevant » (c'est à dire en faisant un parcours depuis ce sommet !).

Dans la suite de cours, tant qu'il n'y a pas de risque de confusion, j'appellerai « arbre » un graphe acyclique connexe.

Théorème 21 (Caractérisations d'un arbre non-enraciné).

Soit $G = (S, A)$ un graphe non-orienté d'ordre n . S'équivalent :

- (i) G est acyclique et connexe (c'est à dire est un arbre)
- (ii) G est acyclique et possède $n - 1$ arêtes.
- (iii) G est connexe et possède $n - 1$ arêtes.
- (iv) G est minimalement connexe, c'est à dire qu'on ne peut lui enlever une arête sans rompre la connexité.
- (v) G est maximalement acyclique, c'est à dire qu'on ne peut lui rajouter une arête sans rompre l'acyclicité.
- (vi) Pour tous u et v sommets, il existe un unique chemin élémentaire de u à v .

Démonstration. • (i) \implies (ii) et (iii) est immédiat avec les propriétés précédentes.

- Montrons (ii) \implies (i) : soit $G = (S, A)$ un graphe acyclique à $n - 1$ arêtes. Notons r son nombre de composantes connexes, et n_0, \dots, n_{r-1} les nombres de sommets de ces composantes. Chacune d'entre elles est connexe et acyclique, donc a $n_i - 1$ arêtes. D'où :

$$\begin{aligned}
n - 1 &= |A| \\
&= \sum_{i=0}^{r-1} (n_i - 1) && \text{les arêtes sont partitionnées entre les CC qui sont des arbres} \\
&= \left(\sum_{i=0}^{r-1} n_i \right) - r \\
&= n - r && \text{les sommets sont partitionnés entre les CC}
\end{aligned}$$

D'où $r = 1$: le graphe est connexe.

- Montrons (iii) \implies (i). Soit $G = (S, A)$ un graphe connexe à $n - 1$ arêtes. Supposons par l'absurde qu'il possède un cycle $s_0, \dots, s_{p-1}, \underbrace{s_p}_{=s_0}$. Alors supprimer l'arête $s_0 s_1$ ne déconnecte pas le graphe : on peut la remplacer par le chemin $s_0, s_{p-1}, s_{p-2}, \dots, s_1$; qui ne contient pas $s_0 s_1$ puisqu'un cycle est simple.

FIGURE 11.18 – Un cycle « propose » deux chemins

Donc le sous-graphe obtenu en supprimant $s_0 s_1$ est connexe à $n - 2$ arêtes : absurde.

Nous avons montré l'équivalence des trois premiers points. Les points (iv) et (v) ont également presque déjà été faits :

- (iii) \implies (iv) : c'est exactement la preuve fait pour (iii) \implies (i).
- (iv) \implies (v) : un graphe minimalement connexe est acyclique (sinon on peut enlever une arête du cycle, cf (iii) \implies (i)). Il est donc connexe est acyclique, donc a $n - 1$ arêtes : donc ajouter une arête lui donnerait $> n - 1$ arêtes ce qui est impossible pour un graphe acyclique.
- (v) \implies (i) : un graphe G maximalement acyclique est connexe car sinon on pourrait connecter deux composantes par une arête sans créer de cycle.

Terminons avec l'équivalence entre les 5 premiers points et (vi) :

- (i) \implies (vi) : comme un arbre est connexe, il existe un chemin entre toute paire de sommets. Supposons par l'absurde qu'il existe deux sommets u et v avec deux chemins élémentaires distincts : ces deux chemins sont distincts par au moins un sommet donc au moins une arête. En particulier, on en déduit un chemin de u à u dans lequel une arête est empruntée une seule fois, donc un cycle⁴ : absurde puisqu'un arbre est acyclique.
- (vi) \implies (v) : un tel graphe est connexe. Mais il est de plus minimalement connexe puisqu'ôter une arête uv déconnecte u de v (cette arête est un chemin élémentaire de u à v donc le seul).

□

Remarque. Ces six points sont tous utiles !

- Le point (v) mène à l'algorithme de Kruskal (MPI) et le point (iv) à l'algorithme reverse-delete (également de Kruskal, MPI).
- Le point (vi) assure que l'on peut enraciner un arbre depuis n'importe quel sommet.
- Les points (ii) et (iii) sont très pratiques pour analyser la complexité d'un algorithme sur des arbres.

4. Finalement, l'extraction de cycle me sert à rendre cette preuve plus agréable...

0.4 (Mi-HP) Coloration de graphes

0.4.0 Degré chromatique

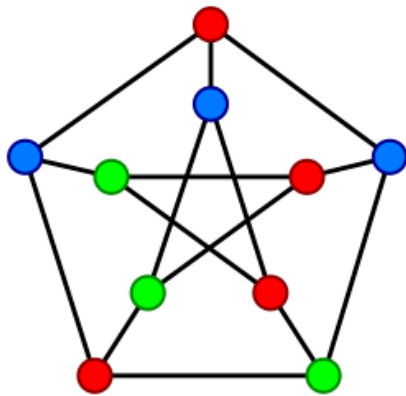
Définition 22 (k -coloration de graphe).

Soit $G = (S, A)$ un graphe. Une k -coloration de G est une application $c : S \mapsto \llbracket 0; k \rrbracket$ telle que pour tout $uv \in A$, $c(u) \neq c(v)$.

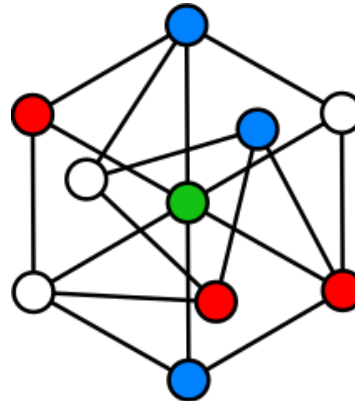
Un graphe qui admet une k -coloration est dit k -coloriable.

Remarque.

- Autrement dit, une k coloration donne une couleur à chaque sommet (avec k couleurs autorisées au maximum) de sorte à ce que deux sommets adjacents ne soient pas de la même couleur.
- On est pas obligé d'utiliser les k couleurs pour k colorier, on peut en utiliser moins.
- Les différents composantes connexes sont indépendantes pour le coloriage (colorier un sommet d'une composante n'aura aucun impact sur les couleurs possibles des sommets d'une autre). On suppose donc souvent qu'un graphe est connexe quand on le colorie.



(a) 3-coloration du graphe de Petersen



(b) 4-coloration d'un graphe (graphe de Golomb)

FIGURE 11.19 – Coloration de graphe

Définition 23 (Degré chromatique d'un graphe).

Le degré chromatique d'un graphe G , noté $\chi(G)$, est le plus petit k tel qu'il existe une k -coloration du graphe.

Exemple. Pour les deux graphes précédents, le coloriage donné correspondait au nombre chromatique.

Proposition 24.

Le nombre chromatique d'un graphe est supérieur ou égal à celui de tous ses sous-graphes.

Démonstration. Un k -coloriage d'un graphe peut-être appliqué à chacun des sous-graphes (en ignorant les éventuels sommets absents des sous-graphes) et reste valide puisque les arêtes des sous-graphes sont des arêtes du graphe. \square

Proposition 25 (Degré chromatique des graphes usuels).

Pour les graphes usuels, en ignorant les graphes à 1 sommet ou moins :

Clique : $\chi(K_n) = n$

Cycle élem. : $\chi(C_n) = \begin{cases} 2 & \text{si } n \text{ est pair} \\ 3 & \text{sinon} \end{cases}$

Chemin élem. : $\chi(P_n) = 2$

Étoile : $\chi(E_n) = 2$

Arbres : $\chi = 2$

Démonstration. • Dans une clique, tous les sommets sont adjacents donc doivent avoir des couleurs deux à deux distinctes.

- $\chi(C_n) \geq 2$ car il y a deux sommets adjacents. Dans le cas où le cycle est de longueur paire, le coloriage qui alterne est valide donc le $\chi = 2$. Dans le cas où la longueur est impaire, ce coloriage n'est pas valide mais c'est le seul 2-coloriage possible. On exhibe à la place un 3-coloriage et conclut.
- Cas particulier des arbres.
- Idem.
- On enracine l'arbre, et on donne une couleur aux étages pairs et une autre aux étages impairs.

□

Définition 26 (Degré maximal d'un graphe).

On note $\Delta(G)$ le degré maximal d'un graphe G .

Proposition 27.

Tout graphe G est $(\Delta(G) + 1)$ -coloriable.

Démonstration. On colorie les sommets les uns après les autres, dans un ordre arbitraire. Quand on colorie un sommet, il y a au plus Δ couleurs déjà utilisées pour ses voisins : il en reste donc une disponible pour le sommet lui-même.

□

Définition 28 (Clique number).

Pour G un graphe, on dit $\omega(G)$ l'ordre de la plus grande clique qui est un sous-graphe de G .

Proposition 29 (Bornes sur χ).

Pour un graphe G :

$$\omega(G) \leq \chi(G) \leq \Delta(G) + 1$$

Démonstration. On a déjà prouvé les deux inégalités.

□

0.4.1 (HP) Cas des graphes planaires**Définition 30 (Graphe planaire).**

Un graphe est dit planaire s'il peut être représenté dans le plan sans que deux de ses arêtes ne se croisent.

Exemple.

- Le graphe de Petersen est planaire.

- K_4 est planaire (il suffit de faire passer une des « diagonales » du carré par l'extérieur et l'autre par l'intérieur).
- K_5 n'est pas planaire.
- $K_{3,3}$ non plus (c'est l'énigme des trois maisons).

Avant de parler coloriage, mentionnons un théorème qui caractérise les graphes planaires à l'aide de la notion de *mineur* (que nous avons mentionnée avec les sous-graphes et les graphes induits) :

Théorème 31 ((très HP) Théorème de Kuratowski-Wagner).

Un graphe est planaire si et seulement si il ne contient ni $K_{3,3}$ ni K_5 comme mineurs.

Démonstration. Ahahahahaha. Non. □

Théorème 32 (Théorème des 4 couleurs).

Tout graphe planaire est 4-coloriable.

Démonstration. Non, mais voici l'idée :

- Dans tout graphe planaire il existe un sommet de degré au plus 5. On en déduit un algorithme pour 6-colorier un graphe planaire (enlever ce sommet, 6-colorier le graphe induit obtenu, remettre le sommet et lui donner une couleur que ses 5 voisins n'ont pas).
- À l'aide de la méthode des chaînes de Kempe⁵, modifier l'algorithme précédent pour que 5 couleurs suffisent.
- Écrire un programme qui trouve les situations critiques « minimales » où la méthode des chaînes de Kempe aboutit à 5 couleurs et non 4 : tous les autres cas sont 4-coloriables. Ensuite, 4-colorier chacune des situations minimales gênantes et conclure (on trouve de 600 à 1500 de situations critiques, selon l'efficacité du premier programme) : cette étape-ci est également faite par ordinateur. □

Remarque.

- Le théorème des 4 couleurs a une importance historique en informatique, car il s'agit de l'une des premières utilisations d'un ordinateur pour mener à terme une preuve. Cela a mené à des débats sur la recevabilité de la preuve.
- La preuve a depuis été formalisée en Rocq, un assistant de preuve (c'est à dire un programme qui vérifie si une preuve est correcte). Autrement dit, on a fourni les codes, une preuve des codes, et Rocq a validé que cette (longue) preuve de (longs) programmes est correcte.
- On présente généralement ce théorème en termes de cartes : on peut colorier des régions connexes d'une carte avec 4 couleurs sans que deux régions adjacentes n'aient la même couleur (l'adjacence ne peut pas être réduite à un point).

Remarque. Le théorème des 4 couleurs est une implication, pas une équivalence : $K_{3,3}$ est 2-coloriable mais n'est pas planaire.

0.4.2 2-coloration de graphe

Déterminer si un graphe est 1-coloriable est simple : il faut et suffit qu'il soit entièrement déconnecté. Tester si un graphe est 2-coloriable est également simple : il n'y a pas de choix à faire !

Proposition 33.

Soit G un graphe et s_0 un sommet du graphe. Alors G est 2-coloriable si et seulement si $v \mapsto d(s_0, v) \bmod 2$ est un 2-coloriage.

5. Pour plus d'informations, allez voir l'article Wikipédia sur le théorème des 5 couleurs.

Démonstration. La réciproque est immédiate. Montrons l'implication directe.

Considérons un 2-coloriage de G . Quitte à échanger les couleurs, supposons que s_0 soit colorié avec la couleur 0. Ses voisins ont donc la couleur 1. Les sommets à distance 2 sont voisins des sommets à distance 1 (qui sont de couleur 1) : ils sont donc de couleur 0. Et ainsi de suite : on montre par récurrence qu'un 2-coloriage de G est celui annoncé par la propriété. \square

Corollaire 34.

Tester si un graphe est 2-coloriable se fait en temps linéaire en la taille du graphe.

Démonstration. Corollaire du parcours en largeur (qui calcule les distances). \square

En fait, les graphes 2-coloriables correspondent à une famille de graphes particulières :

Proposition 35.

Un graphe est deux coloriable si et seulement il est biparti.

Démonstration. Cf fin du cours. \square

En particulier, on retrouve le fait que les arbres sont 2-coloriables.

0.4.3 (MPI) 3-coloration de graphe : NP-difficile

1 et 2 coloriables étaient très simples... mais savoir si un graphe est 3 coloriable est (très) compliqué ! On ne sait pas le faire en temps polynomial, et on pense que cela est impossible !!

En MPI, vous prouverez cela en prouvant que :

Théorème 36.

Le problème de savoir si un graphe est 3-coloriable est NP-Complet.

Démonstration. MPI. Réduction depuis, probablement, 3-SAT. \square

Remarque. En fait, ce que ce théorème dit, c'est que le problème de 3-coloriage a deux grandes qualités :

- On peut « encoder » dedans beaucoup d'autres problèmes. Par exemple, il existe une façon de transformer des instances de SUBSETSUM⁶ en des graphes tels que 3-colorier ces graphes corresponde à une solution de l'instance.
- Et malgré cette grande force d'encodage, il est simple de vérifier si un coloriage proposé est une solution valide ou non.

Remarque. La notion de NP-complétude est au programme de MPI, pas du tout de MP2I. Ce petit morceau de cours a juste pour but de vous donner un avant goût !

6. Étant donné S ensemble d'entiers positifs et t , trouver s'il existe une partie de S qui se somme à t .

1 Graphes orientés

1.0 Définitions

1.0.0 Vocabulaire

Dans un graphe orienté, les arêtes ne sont plus à double sens. C'est assez classique pour un problème de réseau ou de dépendances.

Définition 37 (Graphe orienté).

Un **graphe orienté** est un couple $G = (S, A)$ où :

- S est un ensemble fini d'éléments appelés des **sommets** (ou **noeuds**).
- $A \subseteq \{(x, y) \in S \times S \mid x \neq y\}$ est un ensemble d'**arcs**.

Remarque.

- Cette fois-ci, le sens des arêtes compte ! En effet, l'arc (la paire) (x, y) est distincte de l'arc (la paire) (y, x) .
- Les boucles sont toujours interdites : la définition interdit explicitement des (x, x) .
- Graphes induits et sous-graphes sont définis comme précédemment.

Proposition 38.

Un graphe orienté d'ordre n a au plus $n(n - 1)$ arcs.

Démonstration. Pour définir un arc, il faut et suffit de choisir le sommet de départ de l'arc (n choix) puis celui d'arrivée (qui doit être distinct, donc $n - 1$ choix). \square

Définition 39 (Degré orienté).

Dans $G = (S, A)$ un graphe orienté, pour $s \in S$ un sommet, on appelle :

- **degré entrant** de s , noté $\deg_-(s)$, le nombre d'arcs qui vont *vers* s ; autrement dit le nombre d'arcs us du graphe.
- **degré sortant** de s , noté $\deg_+(s)$, le nombre d'arcs qui *partent* de s ; autrement dit le nombre d'arcs sv du graphe.
- **degré total** de s , noté $\deg(s)$, est $\deg(s) = \deg_-(s) + \deg_+(s)$.

Remarque. Les $+$ et $-$ des degrés orientés sont à comprendre en termes de « contribution au reste du graphe » : le degré sortant se note avec un plus car il « donne » au reste du graphe en y menant, alors que le degré entrant « prend » au reste du graphe.

Proposition 40.

Dans $G = (S, A)$ un graphe orienté :

$$\sum_{s \in S} \deg_+(s) = \sum_{s \in S} \deg_-(s) = |A|$$

Démonstration. Cela revient à dénombrer les arêtes en comptant leur source ou bien leur destination. \square

1.0.1 Chemins orientés

Définition 41 (Chemin orienté, circuit).

On définit un **chemin orienté** comme précédemment : v_0, \dots, v_p est un chemin de longueur p si et seulement si pour tous $i \in \llbracket 0; p \rrbracket$, $v_i v_{i+1} \in A$.

Un chemin orienté simple ou élémentaire est défini comme précédemment. Un **circuit** est un cycle orienté et est défini comme précédemment (idem pour les circuit élémentaires).

Remarque.

- On peut avoir des circuits de longueur 2, puisque l'arc aller et l'arc retour sont distincts :

FIGURE 11.20 – Un circuit de longueur 2

- La relation d'accessibilité n'est pas symétrique : il peut y avoir un chemin de u à v mais pas de v à u :

FIGURE 11.21 – Accessibilité à sens unique

Exemple.

- Le graphe orienté ci-dessous n'a pas de circuit :

FIGURE 11.22 – Pas un cycle orienté

- On dit parfois qu'un graphe sans circuit est **acyclique**⁷ ; mais cela peut mener à confusion (cf exemple précédent). Aussi je vais éviter d'utiliser ce terme dans le cas orienté.
- Le graphe des dépendances d'un mode d'emploi (ou d'une compilation ;)) doit être sans circuit ; puisqu'un circuit correspondrait à une étape e_0 qui dépend d'une étape e_1 qui... qui dépend de e_p qui dépend de e_0 , donc e_0 se requiert elle-même.
- Plus généralement, le graphe des successeurs immédiats dans une relation d'ordre est sans circuit.

Définition 42 (Distance orienté).

On peut définir la distance $d(u, v)$ comme précédemment.

7. C'est ce que l'on a fait dans les rappels sur les relations d'ordre !

Remarque. Notez que cette fois-ci, elle n'est plus symétrique !

Théorème 43 (Extraction de chemins orientés).

Soit $G = (S, A)$ un graphe orienté et u et v deux sommets du graphe. S'équivalent :

- (0) Il existe un chemin orienté de u à v .
- (1) Il existe un chemin orienté simple de u à v .
- (2) Il existe un chemin orienté élémentaire de u à v .

Démonstration. Comme en non-orienté : un chemin orienté de longueur minimale est élémentaire. \square

Théorème 44 (Extraction de cycle orienté).

Soit $G = (S, A)$ un graphe orienté. S'équivalent :

- (0) Il existe un chemin orienté d'un sommet u vers lui-même.
- (1) Il existe un circuit.
- (2) Il existe un circuit élémentaire.

Démonstration. Laissée en exercice. \square

Remarque.

- Ici, il n'y a plus besoin de l'hypothèse « chemin de u vers lui-même qui n'emprunte pas deux fois la même arête » : cette hypothèse servait à interdire « l'aller-retour » (qui n'est pas un cycle non-orienté), mais cette situation est un cycle orienté.
- En fait, dans l'extraction de circuit, la seule situation à éviter est de « faire un tour de plus ».

1.1 Études des composantes fortement connexes

Définition 45 (Bi-accessibilité).

Dans un graphe orienté G , on définit la relation \mathcal{R}' de bi-accessibilité par $u\mathcal{R}'v$ lorsqu'il existe un chemin orienté de u à v et un de v à u .

C'est une relation d'équivalence dont les classes sont appelées les **composantes fortement connexes** (abrégié CFC).

Si u est un sommet, on note C_u sa composante fortement connexe.

Démonstration. Prouvons brièvement que c'est bien une relation d'équivalence :

- Le chemin orienté vide mène de u à u .
- La symétrie est par définition.
- La transitivité se fait comme dans le cas non-orienté.

\square

Exemple.

FIGURE 11.23 – Un graphe orienté et ses composantes fortement connexes

Proposition 46.

Les composantes fortement connexes d'un graphe orienté sans circuit sont des singletons.

Démonstration. Par contraposée : si $u \neq v$ sont dans la même CFC, il existe un chemin orienté (non-vide) de u à v et un de v à u . En les concaténant, on obtient un chemin orienté non-vide de u à u dont on peut extraire un circuit. \square

Définition 47 (Graphe des CFC).

Soit $G = (S, A)$ un graphe orienté. On note \mathcal{C} l'ensemble de ses CFC.

On pose $G^{CFC} = (\mathcal{C}, \mathcal{E})$ où $C_i C_j \in \mathcal{E}$ lorsqu'il existe dans G un arc d'un sommet de C_i vers un sommet de C_j . Ce graphe est appelé le **graphes des composantes fortement connexes** de G .

Exemple. Pour le graphe orienté de l'exemple précédent :

FIGURE 11.24 – Graphe des CFC du graphe de la figure 11.23

Théorème 48 (Acyclicité du graphe des CFC).

Pour tout G graphe orienté, G^{CFC} est sans circuit.

Démonstration. Procédons par l'absurde. Soit $G = (S, A)$ un graphe orienté et G^{CFC} son graphe des CFC. On suppose qu'il contient un circuit, que l'on nomme $C_0, C_1, \dots, C_{r-1}, C_r$. Autrement dit, dans G , il existe un arc $v_0 u_1$ d'un sommet de C_0 à un sommet de C_1 , $v_1 u_2$ de C_1 à C_2 , etc, jusqu'à $v_{r-1} u_0$:

FIGURE 11.25 – Impossibilité d'un cycle dans G^{CFC}

Mais comme chacun des C_i est fortement connexe, pour chaque i il existe dans G un chemin orienté de u_i à v_i . En concaténant tous ces chemins, on obtient un (long) chemin dans G :

$$u_0 \rightarrow \dots \rightarrow v_0 \rightarrow u_1 \rightarrow \dots \rightarrow v_1 \rightarrow u_2 \rightarrow \dots \dots \dots \rightarrow v_{r-1} \rightarrow u_0$$

En particulier, ce cycle donne un chemin de u_1 à v_0 . Or, $v_0 u_1 \in A$: donc les deux sommets devraient être dans la même CFC, absurde. \square

Remarque. Une façon de comprendre ce théorème est que l'on peut ordonner les composantes fortement connexes. Ainsi :

- Deux sommets d'un graphe orienté « sont au même niveau » s'ils sont dans la même CFC.
- Les sommets dans des composantes distinctes sont comparables s'il existe un chemin de la composante de l'un vers la composante de l'autre dans le graphe des CFC

Exemple. Voici un graphe de flot de controle :

FIGURE 11.26 – Un graphe de flot de controle

L'analyse de ce graphe des CFC consiste à savoir quelle(s) opération(s) peuvent avoir lieu avant quelles autres (et est donc utile pour compiler). Trouver les CFC permet également de trouver quelles sont les boucles, c'est à dire les emplacements du code susceptibles de ne pas terminer !

Exercice. Retrouver et spécifier le code d'origine.

2 Implémentation

Dans toute cette partie, on suppose que l'ensemble des sommets est $\llbracket 0; n \rrbracket$: autrement dit, les sommets sont des indices.

Remarque. Quand on voudra échapper à cette hypothèse, on utilisera un dictionnaire (que l'on verra bientôt) !

2.0 Matrice d'adjacence

Définition 49.

La **matrice d'adjacence** d'un graphe (orienté ou non) $G = (\llbracket 0; n \rrbracket, A)$ est la matrice $M = (m_{i,j})_{0 \leq i,j < n}$ définie par :

$$m_{i,j} = \begin{cases} 1 & \text{si } ij \in A \\ 0 & \text{sinon} \end{cases}$$

Exemple.

(a) Un graphe (orienté)

(b) Matrice d'adjacence associée

FIGURE 11.27 – Matrice d'adjacence d'un graphe

Remarque.

- La matrice d'adjacence d'un graphe non-orienté est symétrique.
- L'absence de boucles (arêtes uu) dans nos graphes impose que la diagonale est nulle.
- Le degré sortant d_+ d'un sommet est la somme de sa ligne.
- Le degré entrant d_- d'un sommet est la somme de sa colonne.
- On pourrait^{8 9} utiliser des booléens à la place des entiers.

8. On devrait.

9. Mais les 0/1 sont assez standards... et plus rapide à écrire.

- Si l'on veut étiquetter les arcs, on peut mettre None dans la matrice pour un arc absent et Some etq pour un arc présent.

Proposition 50 (Complexités avec une matrice d'adjacence).

Avec une matrice d'adjacence, on a les complexités suivantes :

- Tester si une arête est présente : $O(1)$
- Trouver tous les voisins d'un sommet : $\Theta(|S|)$
- Ajouter un noeud : $O(|S|^2)$
- Ajouter une arête : $\Theta(1)$
- Enlever un noeud : $O(|S|^2)$
- Enlever une arête : $\Theta(1)$

Démonstration.

- Pour savoir si $ij \in A$, il suffit de lire $M[i][j]$
- Pour trouver tous les voisins de i , il faut parcourir toute la ligne $M[i]$ pour y trouver tous les 1 (et ce même si le degré du sommet est très faible!).
- Pour ajouter un nouveau noeud, il faut recréer toute la matrice avec le nouveau noeud en plus.
- Il suffit de modifier la bonne case de la matrice.
- Les deux derniers points sont comme l'ajout de sommet/arête.

□

Remarque.

- Cette représentation est particulièrement adaptée lorsque le graphe est dense, c'est à dire lorsqu'il possède « beaucoup » d'arêtes.
- Si on utilise des tableaux dynamiques, ajouter/enlever le sommet d'indice n peut se faire en $\Theta(|S|)$ amorti : il suffit d'ajouter une nouvelle case à la fin de chaque ligne, et une nouvelle ligne en bas de la matrice.

Proposition 51.

Une matrice d'adjacence coûte $\Theta(|S|^2)$ en mémoire.

Démonstration. Djur à faire sans implémentation précise... surtout qu'avec une implémentation volontairement catastrophique des tableaux vous pouvez me donner tort... mais dans l'idée, on stocke chaque coefficient une seule fois et c'est tout. □

2.1 Listes d'adjacence

Définition 52 (Listes d'adjacence).

Soit $G = (\llbracket 0; n \rrbracket, A)$ un graphe et $s \in S$. On appelle **liste d'adjacence** de s la liste de ses voisins. La **représentation par listes d'adjacence** de G consiste à calculer et stocker toutes les listes d'adjacence. On utilise généralement pour cela un tableau (ou un tableau associatif) qui à un sommet associe sa liste d'adjacence.

Exemple.

FIGURE 11.28 – Liste d’adjacence du graphe de la figure 11.27

Remarque.

- Sauf mention contraire, on ne suppose pas les listes d’adjacences ordonnées : les voisins peuvent être dans n’importe quel ordre.
- Je n’ai volontairement pas précisé comment sont implémentées les listes : on veut une façon de faire qui permet d’ajouter ou d’enlever le premier élément efficacement ; peu importe l’implémentation précise (liste chaînée, tableau borné, tableau dynamique).

Proposition 53 (Complexités avec des listes d’adjacence).

Avec des listes d’adjacence, on a les complexités suivantes :

- Tester si une arête uv est présente : $O(deg_+(u))$
- Trouver tous les voisins d’un noeud u : $\Theta(1)$
- Ajouter un noeud : $O(|S|)$
- Ajouter une arête uv : $\Theta(1)$
- Enlever un noeud : $O(|S| + |A|)$
- Enlever une arête uv : $O(deg_+(u))$

Démonstration.

- Il faut parcourir toute la liste d’adjacence de u pour y chercher v . Cette liste est de longueur $deg_+(u)$.
- Les voisins sont exactement la liste d’adjacence, qui est déjà calculée !
- Pour ajouter un nouveau noeud, il faut ajouter une case au tableau des listes.¹⁰
- Pour ajouter une arête, il suffit de l’ajouter à la liste d’adjacence.
- Les deux derniers points sont comme l’ajout de sommet/arête.
- Pour enlever un noeud, il faut enlever sa case du tableau des listes en $O(|S|)$; mais aussi parcourir toutes les autres listes pour y enlever le noeud s’il y est présent. Cela revient à parcourir toutes les arêtes en $O(|A|)$.
- Pour enlever une arête, il faut parcourir la liste d’adjacence

□

Remarque.

- Cette représentation est adaptée quand le graphe est creux, c’est à dire quand la matrice d’adjacence a « beaucoup » de zéros.
- C’est la représentation la plus utilisée en MP2I/MPI, car c’est la plus adaptée au parcours du graphe.
- Des implémentations spécifiques pourraient obtenir des complexités différentes. Par exemple, en imposant que les listes d’adjacence sont des tableaux (dynamiques) triées, la recherche d’une arête se fait en temps $\log(deg_+)$... mais l’insertion devient linéaire en deg_+ .

¹⁰. Là encore, avec des tableaux dynamiques cela se fait en temps constant.

Proposition 54.

Les listes d'adjacence contiennent $\Theta(|S| + |A|)$ en mémoire.

Démonstration. Là encore, dur à faire sans implémentation précise... Mais dans l'idée, on stocke $|S|$ listes. La liste d'un sommet s contient $\deg_+(s)$ valeurs, donc elles contiennent au total $\sum_{s \in S} \deg_+(s) = |A|$. \square

Remarque. En comparant avec le coût d'une matrice d'adjacence, on comprend pourquoi cette représentation est adaptée aux graphes creux.

2.2 Comparatif

Opération \ Structure	Matrice d'adjacence	Listes d'adjacence
Tester si $uv \in A$	1	$\deg_+(u)$
Parcourir les voisins de u	$ S $	$\deg_+(u)$
Ajouter un sommet	$ S ^2$	$ S $
Ajouter une arête	1	1
Enlever un sommet	$ S ^2$	$ S + A $
Enlever une arête uv	1	$\deg_+(u)$

(a) Complexités temporelles des opérations usuelles

	Matrice d'adjacence	Listes d'adjacence
Complexité spatiale	$ S ^2$	$ S + A $

(b) Complexités spatiale

FIGURE 11.29 – Ordre de grandeur des complexités pour des matrices d'adjacence et des listes d'adjacence

3 Parcours de graphes

3.0 Parcours générique

3.0.0 Parcours depuis un sommet

Le parcours de graphe sert à se déplacer dans un graphe en appliquant un certain traitement lorsque l'on visite un sommet. On ne peut cependant pas simplement appliquer tel quel l'algorithme de parcours des arbres :

FIGURE 11.30 – Boucle infinie d'un DFS à cause d'un cycle

Il faut « réparer » le parcours en imposant que l'on ne peut pas repasser deux fois par le même sommet. Afin de garder le code du parcours ci-dessous, on se donne un « sac » qui permet de stocker les arêtes empruntées (elle joue le rôle de la pile/file que l'on a utilisée dans le parcours d'arbre).

Algorithme 16 : Parcours de graphe

Entrées : G un graphe et s_0 un sommet du graphe

```

1   $sac \leftarrow$  sac vide
2  Insérer  $s_0$  dans  $sac$ 
3  tant que  $sac$  est non-vide faire
4       $u \leftarrow$  extraire un sommet du  $sac$ 
5      si  $u$  n'est pas déjà marqué alors
6          Marquer  $u$ 
7          Visiter  $u$  (çad lui appliquer le traitement que l'on veut)
8          pour chaque  $v$  voisin de  $u$  faire
9              Insérer  $v$  dans  $sac$ 
```

Remarque.

- En pratique, le marquage (qui permet donc de savoir si un sommet a déjà été visité ou non) se fait avec un tableau qui à un sommet associe s'il est marqué (true) ou non (false).
- Le sac peut contenir des doublons. Cela sera même très important pour les parcours en profondeur.
- On peut optimiser en ne réinsérant pas dans le sac des sommets déjà marqués. Je ne le fais pas encore dans ce pseudo-code afin de simplifier l'analyse de complexité, mais une fois celle-ci terminée je le ferai systématiquement.

Exemple.

(a) Un graphe

(b) Un parcours du graphe, sans réinsérer des sommets déjà marqués

Proposition 55.

Un parcours de G depuis s_0 visite exactement les sommets accessibles depuis s_0 .

Démonstration.

- Tout sommet visité est accessible depuis s_0 . En effet, tout sommet visité est d'abord extrait du sac : on montre l'invariant que tous les sommets du sac sont accessibles depuis s_0 :
 - Avant la première itération de la boucle `while`, le sac contient uniquement s_0 .
 - Si l'invariant est vrai au début d'une itération : on extrait u du sac, qui est donc accessible depuis s_0 . Ensuite si l'on rentre dans le `Si`, on ajoute dans le sac les voisins de u : ils sont donc accessibles depuis u donc par transitivité depuis s_0 . D'où la conservation.
- Tout sommet u accessible est visité. Pour montrer cela, remarquons que tout sommet ajouté dans le sac fini par être visité et montrons par récurrence sur $d(s_0, u)$ que u est inséré dans le sac :
 - Si $d(s_0, u) = 0$, alors $u = s_0$ et ce sommet est bien inséré dans le sac (avant la boucle).
 - Si $d(s_0, u) > 0$ et que la propriété est vraie pour tout sommet plus proche : considérons un plus court chemin de s_0 à u et nommons p_u le sommet juste avant u dans ce chemin. Comme un sous-chemin d'un plus court chemin est un plus court chemin, $d(s_0, p_u) = d(s_0, u) - 1$. Donc par HR, p_u a été ajouté dans le sac. Mais d'après la condition de fin du `while`, p_u finit par sortir du sac. La première fois qu'il en sort, on entre dans le `Si` et ajoute au sac tous ses voisins, dont u .

□

3.0.1 Parcours complet

Il arrive souvent que l'on veuille visiter *tous* les sommets. On appelle cela faire un **parcours complet** du graphe. C'est en fait très simple :

Algorithme 16 : Parcours complet

Entrées : G un graphe

```

1 pour chaque sommet  $u$  de  $G$  faire
2   | si  $u$  n'a pas déjà été marqué lors d'un parcours alors
3   |   | Lancer un parcours de  $G$  depuis  $u$ 
```

Remarque. En pratique, cela demande de faire en sorte que le tableau qui mémorise si un sommet est marqué ou non soit partagé entre les différents parcours. Cela se met typiquement en oeuvre en en faisant une entrée de la fonction de parcours.

Proposition 56.

Lors d'un parcours complet du graphe, chaque sommet est visité exactement une fois.

Démonstration. Chaque sommet est visité au plus une fois grâce au marquage partagé, et le parcours complet lance un parcours depuis chaque sommet non-encore marqué (donc chaque sommet est visité au moins une fois). \square

3.0.2 Complexité

Théorème 57 (Complexité d'un parcours).

En notant c_{visite} la complexité de la visite d'un sommet, c_{insere} le coût de l'insertion dans le sac et $c_{extraite}$ le coût de l'extraction du sac, la complexité d'un parcours d'un graphe G est :

- $\Theta(|S|c_{visite} + |A|(c_{insere} + c_{extraite}))$ pour un graphe implémenté par liste d'adjacence.
- $\Theta(|S|c_{visite} + |S|(|S| + c_{insere} + c_{extraite}))$ pour un graphe implémenté par matrice d'adjacence.

Démonstration. Comptons les opérations qui ont lieu. Tout d'abord, la boucle Pour du parcours de graphe complet, sans compter les parcours qu'elle lance, est en $\Theta(|S|)$. Étudions les opérations qui ont lieu lors des parcours :

- Il y a les opérations hors de la boucle. Elles sont en $O(1)$, comme on lance un parcours depuis au plus chaque sommet cela fait $O(|S|)$ au total.
- Il y a chaque début d'itération : tester si le sac est vide en $O(1)$, extraire un sommet du sac en $\Theta(c_{extraite})$ et tester si le sommet est marqué en $O(1)$: le tout est donc en $\Theta(c_{extraite})$. Mais comme une extraction n'a lieu qu'après une insertion, on va compter ces coûts-ci lorsque l'on compte les insertions.
- Et le corps du Si de la boucle TantQue. Chaque exécution de ce corps effectue :
 - un marque en $O(1)$
 - une visite en $\Theta(c_{visite})$
 - Puis le parcours de tous les voisins de u pour les insérer. Sur une liste d'adjacence, cela se fait en $\Theta(deg_+(u)c_{insere})$, sur une matrice en $\Theta(|S| + deg_+(u)c_{insere})$; la différence venant du fait que dans une matrice d'adjacence on doit parcourir tous les sommets pour savoir s'ils sont voisins ou non.

Terminons les calculs dans le cas des listes d'adjacence. Comme chaque sommet est visité exactement une fois, la complexité des boucles while des parcours successifs est un Θ de :

$$\begin{aligned}
 \sum_{u \in S} c_{visite} + deg_+(u)(c_{extraite} + c_{insere}) &= |S|c_{visite} + (c_{extraite} + c_{insere}) \sum_{u \in S} deg_+(u) \\
 &= |S|c_{visite} + (c_{extraite} + c_{insere})|A|
 \end{aligned}$$

La complexité totale étant la complexité de toutes les boucles while ci-dessous, plus le $O(|S|)$ total des lignes 1-2 du parcours (avant la boucle) plus le $O(|S|)$ total de la boucle for du parcours complet, on obtient bien le résultat attendu. \square

Remarque. Dans cette preuve, j'ai supposée que créer un sac vide et tester si un sac était vide sont en $O(1)$... c'est très raisonnable.

Corollaire 58.

Quand le coût de la visite, de l'insertion et de l'extraction sont constant, on obtient les complexités suivantes pour un parcours de G depuis s_0 :

- $O(|S| + |A|)$ pour des listes d'adjacence
- $O(|S|^2)$ pour une matrice d'adjacence.

Démonstration. C'est un corollaire du théorème précédent, avec un $O()$ et non un Θ car on ne fait ici pas un parcours complet. \square

Cela dit, la preuve du théorème était un peu longue. On la présente souvent de manière plus courte comme suit :

Démonstration.

- Avec des listes d'adjacence : Chaque sommet est visité au plus une fois. Lors de la visite d'un sommet, on le visite puis on parcourt sa liste d'adjacence pour insérer ses voisins. Chaque extraction correspond à une insertion. Donc la complexité totale est :

$$O\left(\sum_{s \in S} \underbrace{1}_{\text{visite}} + \underbrace{\deg_+(s)}_{\text{extraction et insertion des voisins}}\right) = O(|S| + |A|)$$

- Avec une matrice d'adjacence : Chaque sommet est visité au plus une fois. Lors de la visite d'un sommet, on le visite puis on parcourt sa ligne de la matrice pour insérer ses voisins. Chaque extraction correspond à une insertion. Donc la complexité totale est :

$$O\left(\sum_{s \in S} \underbrace{1}_{\text{visite}} + \underbrace{|S|}_{\text{parcours ligne}} + \underbrace{\deg_+(s)}_{\text{extraction et insertion des voisins}}\right) = O(|S|^2)$$

\square

Remarque.

- J'accepte tout à fait la preuve ci-dessus en DS (elle provient presque d'un rapport de jury après tout!). On peut très simplement la modifier pour gérer c_{visite} , c_{insere} et c_{extrait} .
- Les parcours de graphe, et l'analyse de leur complexité, sont le B-A-BA de l'algorithmique des graphes qui constitue un *gros* morceau de MP2I/MPI! Il est donc *très* important de savoir écrire un parcours et de savoir réaliser la preuve abrégée ci dessus!!

3.0.3 Arbre de parcours

Définition 59.

L'**arbre de parcours** associé à un parcours d'un graphe G depuis s_0 est l'arbre constitué des arcs $p_u u$ où p_u est le sommet qui a inséré l'exemplaire de u dont l'extraction a causé la visite de u . Autrement dit, on insère des arcs $p_u u$ dans le sac. Quand on extrait un arc $p_u u$ du sac, si u n'a pas encore été marqué on note que p_u est son parent.

Remarque. Cette définition est rendue subtile car un sommet peut apparaître en doublon dans le sac; et qu'il y aurait alors ambiguïté sur qui est le parent.

Exemple.

FIGURE 11.32 – Ambiguïté possible sur la parenté si l'on ne fait pas attention

Proposition 60.

Pour représenter un arbre de parcours, le plus agréable est généralement un tableau de parenté : pour chaque sommet, on mémorise son parent dans le parcours.

Remarque. Cela fonctionne car on peut identifier uniquement un sommet (par exemple, par son étiquette.)

Exemple.

FIGURE 11.33 – Arbre de parcours pour l'exemple de parcours générique précédent et tableau de parenté associé

Cela donne le pseudo-code suivant :

Algorithme 17 : Calcul de l'arbre du parcours

Entrées : G un graphe et s_0 un sommet du graphe

Sorties : *parent* un tableau tel que *parent*[u] est le parent de u lors du parcours (vide si u n'est pas visité ou est s_0)

```

1  sac ← sac vide
2  parent ← tableau dont toutes les cases sont vides
3  Insérer ( $s_0, s_0$ ) dans sac
4  tant que sac est non-vide faire
5      ( $p_u, u$ ) ← extraire un sommet du sac
6      si  $u$  n'est pas déjà marqué alors
7          Marquer  $u$ 
8          parent[ $u$ ] ←  $p_u$ 
9          pour chaque  $v$  voisin de  $u$  non-encore marqué faire
10             Insérer  $v$  dans sac
11 parent[ $s_0$ ] ← vide
12 renvoyer parent
```

Remarque. Notez que :

- On insère désormais des *arcs* dans le sac et non plus des sommets!!
- Il y a une légère difficulté : quel est l'arc qui insère s_0 le sommet initial? Aucun... Donc on fait comme si s_0 était son propre parent; et on corrige cela à la fin (ligne 11) en marquant que s_0 n'a en fait pas de parent.
- Les éléments qui n'ont pas de parent dans l'arbre sont s_0 et les sommets non-accessibles depuis s_0 .
- En OCaml, un type option est très adapté pour stocker les parentés.

Proposition 61.

Un chemin depuis s_0 dans l'arbre de parcours de G depuis s_0 est un chemin depuis s_0 dans G . Autrement dit, l'arbre de parcours est un sous-graphe de G .

Démonstration. C'est presque exactement la preuve du fait que le parcours depuis s_0 visite les sommets accessibles depuis s_0 . □

Proposition 62.

Soit G un graphe et s_0 un sommet de G . On suppose que tous les sommets sont accessibles depuis s_0 .

Si l'on dé-enracine (et désoriente) l'arbre du parcours de G depuis s_0 , il est bien connexe acyclique : c'est bien un arbre.

Démonstration. Il est connexe car tout le monde est relié à s_0 le sommet de départ du parcours (encore une fois presque exactement la preuve du fait que le parcours visite les sommets accessibles).

De plus, chaque sommet a exactement un prédecesseur sauf s_0 qui n'en a pas : avec n le nombre de sommets de l'arbre, il y a donc $n - 1$ liens de parentés ($n - 1$ arêtes) et il s'agit donc d'un arbre d'après le théorème de caractérisation des arbres. □

Remarque. Les sommets non-accessibles depuis s_0 ne sont pas dans l'arbre, donc il est toujours acyclique. Ils sont juste embêtants pour la connexité... Mais dans l'idée, on a même pas envie de dire qu'ils sont dans l'arbre de parcours : c'est donc assez naturel de les exclure comme le fait cette propriété.

Définition 63.

Les parentés calculées par un parcours complet forme une **forêt de parcours**, c'est à dire qu'il s'agit de plusieurs arbres disjoints.

Démonstration. Dans un parcours complet, chaque parcours complet visite exactement les sommets non encore marqués accessibles depuis le point de départ. Donc les arbres de parcours sont disjoints; aussi leur union crée une forêt. □

Exemple.

FIGURE 11.34 – Une forêt de parcours possible pour un graphe

3.1 Parcours en largeur

3.1.0 Définition et propriété

Définition 64.

Le **parcours en largeur** d'un graphe consiste à parcourir les sommets par distance croissante au sommet initial.

Il est obtenu **en utilisant une file dans le parcours**.

Exemple.

FIGURE 11.35 – Parcours en largeur d'un graphe (sans réinsérer les sommets déjà marqués)

Remarque. Il n'y a pas de règle sur l'ordre d'enfilage des voisins d'un sommet. Cela n'a pas d'importance¹¹.

Théorème 65 (Arbre d'un parcours en largeur).

Dans un graphe G (non-pondéré), l'arbre du parcours en largeur depuis s_0 est un arbre de plus courts chemins depuis s_0 .

Démonstration. PREUVE FAUSSE À REPRENDRE. Admis en attendant.

□

3.1.1 Une optimisation : le marquage anticipé

Comme une file est FIFO, l'extraction d'un sommet u de la file correspond à sa *première* insertion. Aussi, au lieu de marquer un sommet lorsqu'il sort de la file, on peut le marquer lors de son *entrée* dans

¹¹. Si vous avez besoin d'un ordre précis, vous ne faites pas un parcours en largeur ; vous faites un parcours en largeur particulier.

la file sans changer l'ordre de parcours. On parle de **marquage anticipé**. Cela permet de réduire la taille maximale de la file en y éliminant les doublons.

3.1.2 Implémentation en OCaml

On se donne les types suivants pour manipuler un graphe par liste d'adjacence, et un type d'arbres représentés par tableau de parenté :

```

3 type sommet = int
4 type graphe = sommet list array (* listes d'adjacence *)
5
6 type arbre = sommet option array (* tableau de parenté *)

```

 `parcours.ml`

On propose ci-dessous une implémentation en OCaml du parcours en largeur, qui utilise le module Queue pour manipuler des files. Il s'agit de files impératives : les opérations que l'on fait dessus *modifient* la file.

```

12 let bfs (g : graphe) (s0 : sommet) : arbre =
13   let n = Array.length g in
14   let parent = Array.make n None in
15   let marque = Array.make n false in
16   let file = Queue.create () in
17   (* La file contient des arcs [(pu,u)] avec
18      [pu] le sommet qui enfile [u] dans la file *)
19   Queue.add (s0, s0) file;
20   marque.(s0) <- true; (* marquage anticipé !*)
21
22   while not (Queue.is_empty file) do
23     let (pu, u) = Queue.take file in
24     parent.(u) <- Some pu;
25
26     let insere v =
27       if not marque.(v) then begin
28         Queue.add (u,v) file;
29         marque.(v) <- true (* marquage anticipé ! *)
30       end
31     in
32     List.iter insere g.(u)
33   done;
34
35   parent.(s0) <- None;
36   parent

```

 `parcours.ml`

3.2 Parcours en profondeur

3.2.0 Définition et propriétés

Définition 66.

Le **parcours** en profondeur d'un graphe consiste à parcourir les sommets en « avançant » le plus possible dans le graphe, et en revenant sur ses pas (jusqu'à pouvoir avancer à nouveau) lorsque tous les embranchements possibles ont déjà été visités.

Il s'écrit très naturellement **récursivement**, ou bien **en parcours avec une pile**.

Lors d'un parcours en profondeur récursif, on appelle **ouverture** d'un sommet le moment où ce sommet est marqué/visité, et **fermeture** le moment où les appels récursifs sur ses voisins termine. On numérote souvent ces sommets à partir de 0 : la première ouverture a lieu au temps 0, l'ouverture/fermeture qui suit au temps 1, celle qui suit encore au temps 2, etc.

Voici l'écriture récursive du parcours en profondeur (remarquez qu'il faut que le marquage soit partagé entre les appels) :

Algorithme 18 : DFS

Entrées : G un graphe, u un sommet

```

1 si  $u$  n'est pas déjà marqué alors
  // Ouverture
2   Marquer  $u$ 
3   Visiter  $u$ 
4   pour chaque voisin  $v$  de  $u$  non-encore marqué faire
5     DFS( $G, v$ )
  // Fermeture
```

Exemple.

FIGURE 11.36 – Parcours en profondeur récursif d'un graphe

Remarque.

- Le parcours récursif empile des appels sur la... pile d'appels. Donc il utilise aussi une pile !
- Les notions d'ouverture et de fermeture sont compliquées à définir en itératif : c'est une raison de préférer le parcours récursif.
- L'ouverture d'un sommet a lieu avant sa fermeture.

L'arbre d'appels d'un parcours en profondeur a d'excellentes propriétés, qui font de ce parcours un *must-have*. Avant de les étudier, formalisons la construction de l'arbre :

Algorithme 19 : DFS, mais avec construction de l'arbre de parcours en profondeur

Entrées : G un graphe, p_u un sommet et u un sommet

Pré-conditions : p_u est le sommet qui a lancé l'appel actuel

```

1 si  $u$  n'est pas déjà marqué alors
    // Ouverture
2 Marquer  $u$ 
3 Noter que  $p_u$  est le parent de  $u$  dans l'arbre
4 pour chaque voisin  $v$  de  $u$  non-encore marqué faire
5     DFS( $G, u, v$ )
    // Fermeture
```

Exemple.

FIGURE 11.37 – Un arbre de parcours en profondeur avec les dates d'ouverture/fermeture

Lemme 67.

On considère un parcours en profondeur complet d'un graphe G . On note $d(\cdot)$ et $f(\cdot)$ les temps d'ouverture et fermeture (qui ont une numérotation commune à partir de 0).

Alors v est descendant de u dans la forêt de parcours si et seulement si $d(u) < d(v) < f(v) < f(u)$.

Démonstration. C'est dur à formaliser ; et quand on le fait c'est embêtant pour pas grand chose¹². Je vais simplement donner l'idée :

- Sens direct : v est descendant de u signifie exactement que u a lancé un appel qui a lancé un appel qui... qui a lancé l'appel qui a visité v . Ainsi, $d(u) < d(v)$. Mais de plus, avant que u ne se ferme ses appels récursifs (et les appels de ses appels, etc) doivent se fermer : donc $f(v) < f(u)$. D'où le sens direct de l'implication.
- Sens réciproque : Si $d(u) < d(v) < f(v) < f(u)$, cet encadrement signifie exactement que l'appel de la visite de v a été au-dessus de l'appel de la visite de u dans la pile d'appel (avec éventuellement d'autres appels entre les deux). Or, d'après le fonctionnement de la pile d'appel tous les appels au-dessus de l'appel de u sont des appels récursifs de u , ou des appels de leurs appels, etc. En particulier, l'ouverture de v est un appel d'un appel d'un appel ... de l'appel de u : v descend donc de u dans l'arbre d'appels.

□

12. Et je déteste voir ce lemme demandé en concours ; car je ne sais pas ce qui est attendu.

Lemme 68.

On considère un parcours en profondeur complet d'un graphe G . On note $d(\cdot)$ et $f(\cdot)$ les temps d'ouverture et fermeture (qui ont une numérotation commune à partir de 0). Alors il ne peut pas exister de u et v tels que $d(u) < d(v) < f(u) < f(v)$.

Démonstration. À peu près comme la preuve précédente : $d(u) < d(v) < f(u)$ signifie que l'appel sur u a fait un appel qui ... qui a fait l'appel sur v . En particulier, l'appel sur v doit terminer avant de rendre la main à l'appel sur u , donc $f(v) < f(u)$. \square

Théorème 69 (Théorème des parenthèses).

On considère un parcours en profondeur complet d'un graphe G . On note $d(\cdot)$ et $f(\cdot)$ les temps d'ouverture et fermeture (qui ont une numérotation commune à partir de 0).

Pour deux sommets u et v , une seule des trois conditions suivantes est vérifiée :

- Les intervalles $[d(u); f(u)]$ et $[d(v); f(v)]$ sont disjoints, et ni u ni v ne sont descendant l'un de l'autre dans la forêt du parcours complet.
- l'intervalle $[d(u); f(u)]$ est entièrement inclus dans l'intervalle $[d(v); f(v)]$ et u est un descendant de v dans la forêt de parcours.
- l'intervalle $[d(v); f(v)]$ est entièrement inclus dans l'intervalle $[d(u); f(u)]$ et v est un descendant de u dans la forêt de parcours.

Démonstration. D'après le second lemme précédent, les intervalles sont soit inclus soit disjoints. Le premier lemme conclut. \square

Remarque.

- Ce que ce théorème dit, c'est que les ouvertures/fermetures sont « bien parenthésées », et que la forêt de parcours indique les imbrications de ces parenthèses.
- En MPI, vous approfondirez ce théorème en classifiant les arcs du graphe G de en 4 catégories par rapport à la forêt de parcours G_{Π} ¹³. Ce sera le théorème de classification des arcs.

3.2.1 Pas de marquage anticipé !!

Considérons le parcours en profondeur itératif avec une pile. Si l'on effectue un marquage anticipé (c'est à dire que l'on interdit les doublons dans la pile), on peut obtenir l'arbre de parcours suivant :

FIGURE 11.38 – Le marquage anticipé ne donne PAS un parcours en profondeur

Cet arbre n'est pas un arbre de parcours en profondeur... En fait :

Proposition 70.

Quand on écrit un parcours en profondeur itératif :

- avec marquage anticipé, on obtient pas un DFS.
- **SANS marquage anticipé**, on obtient bien un DFS.

13. Les arcs de liaison (les arcs G qui sont dans G_{Π}), les arcs avant (les arcs de G « raccourcissent » un chemin de G_{Π}), les arcs arrière (les arcs de G qui « remontent » un chemin de G_{Π}), et les arcs transverses (les autres, çad ceux qui relient deux sommets qui ne sont pas ancêtres l'un de l'autre dans G_{Π}).

Démonstration. Le premier point est Figure 11.38, le second est admis. □

Remarque.

- Le parcours en profondeur s'écrit très très bien en récursif, où cette subtilité n'apparaît pas.
- On appelle parfois « parcours pile » le parcours itératif avec une pile et marquage anticipé.

3.2.2 Implémentation en OCaml

Voici le code de l'implémentation récursive, pour construire l'arbre de parcours.

```
67 let dfs (g : graphe) (s0 : sommet) : arbre =
68   let n = Array.length g in
69   let marque = Array.make n false in
70   let parent = Array.make n None in
71
72   let rec dfs_rec pu u =
73     if not marque.(u) then begin
74       marque.(u) <- true;
75       parent.(u) <- Some pu;
76       List.iter (fun v -> if not marque.(v) then dfs_rec u v) g.(u)
77     end
78   in
79
80   dfs_rec s0 s0;
81   parent.(s0) <- None;
82   parent
```

 parcours.ml

Remarque. Dans le code ci-dessus, les tests de non-marquage ligne 72 et 75 sont redondants... mais c'est spécifique à l'écriture récursive. Il vaut mieux mettre trop de tests de non-marquage que pas assez.

Et voici une autre version, qui cette fois calcule les dates d'ouverture et de fermeture lors d'un parcours complet. On y utilise la fonction `incr` qui augmente de 1 le contenu d'une référence.

```
89 let dates (g : graphe) : int array * int array =
90   let n = Array.length g in
91   let ouverture = Array.make n (-1) in
92   let fermeture = Array.make n (-1) in
93   let time = ref 0 in
94   let marque = Array.make n false in
95
96   let rec dfs_rec u =
97     if not marque.(u) then begin
98       marque.(u) <- true;
99       ouverture.(u) <- !time; incr time;
100
101       List.iter (fun v -> if not marque.(v) then dfs_rec v) g.(u);
102
103       fermeture.(u) <- !time; incr time
104     end
105   in
106
107   for s0 = 0 to n-1 do
108     if not marque.(s0) then dfs_rec s0
109   done;
110   ouverture, fermeture
```

 parcours.ml

3.3 Application : calcul des composantes connexes

On peut utiliser des parcours pour calculer les composantes connexes d'un graphe non-orienté. En effet, on a dit qu'un parcours visite exactement les sommets accessibles depuis son point de départ. Donc dans un graphe non-orienté, un parcours visite exactement une composante connexe.

On va donc faire un parcours complet : il numérote tous les sommets du programme parcours par 0 (et ils forment exactement une composante connexe), tous les sommets du second parcours par 1, etc. Ainsi, on construit les composantes connexes C_0, C_1, \dots en numérotant tous les sommets de la composante C_i par i .

Cela donne le code suivant en OCaml, toujours avec les mêmes types que précédemment :

```

120 let composantes (g : graphe) : int array =
121   let n = Array.length g in
122   let comp = Array.make n (-1) in
123   let num_comp = ref 0 in
124   let marque = Array.make n false in
125
126   let bfs s0 =
127     let file = Queue.create () in
128     Queue.add s0 file;
129     marque.(s0) <- true; (* marquage anticipé ! *)
130     while not (Queue.is_empty file) do
131       let u = Queue.take file in
132       marque.(u) <- true;
133       comp.(u) <- !num_comp;
134       let insere v =
135         if not marque.(v) then begin
136           Queue.add v file;
137           marque.(v) <- true (* marquage anticipé ! *)
138         end
139       in
140       List.iter insere g.(u)
141     done
142   in
143
144   for s0 = 0 to n-1 do
145     if not marque.(s0) then begin
146       bfs s0;
147       incr num_comp
148     end
149   done;
150   comp

```



En particulier :

Proposition 71.

Un parcours complet d'un graphe non-orienté permet de calculer les composantes connexes en temps linéaire en la taille du graphe.

Remarque. Pour un graphe orienté, on veut calculer les composantes fortement connexes. C'est un problème plus compliqué, que vous résoudrez en MPI à l'aide de l'algorithme de Kosaraju¹⁴.

14. Il s'agit d'exploiter à leur plein potentiel le théorème des parenthèses et le théorème de classification des arcs; c'est un très bel algorithme.

3.4 Application : calcul d'un ordre topologique

3.4.0 Définition

Définition 72 (Ordre topologique).

Soit $G = (S, A)$ un graphe orienté. Un **ordre topologique** sur S est un ordre \preccurlyeq tel que $uv \in A$ implique $u \prec v$.

Exemple.

FIGURE 11.39 – Un graphe orienté et des sommets numérotés selon un ordre topologique

Remarque.

- Un ordre topologique sert à trouver un ordre dans lequel réaliser les tâches d'un graphe de dépendances.
- Il n'y a pas unicité d'un ordre topologique. Par exemple, le graphe ci-dessous admet deux ordres topologiques :

FIGURE 11.40 – Multiples ordres topologiques

Il s'avère même que trouver le nombre d'ordres topologiques est un problème difficile¹⁵.

Proposition 73.

Soit G un graphe orienté qui contient un circuit. Alors G n'a pas d'ordre topologique.

Démonstration. Soit u_0, \dots, u_p un tel circuit. Alors un ordre topologique donnerait $u_0 \prec \dots \prec u_p = u_0$ donc $u_0 \prec u_0$. Contradiction. \square

3.4.1 Construction d'un ordre topologique

Définition 74 (DAG).

Un **DAG (Direct Acyclic Graph)** est un graphe orienté sans cycle.

Théorème 75 (Calcul d'un ordre topologique).

Soit $G = (S, A)$ un DAG.

Ordonner les sommets de S par date de fin décroissante dans un DFS complet donne un ordre topologique.

Démonstration. Soit $G = (S, A)$ un DAG et $d()$ et $f()$ les dates d'ouverture et de fermeture lors d'un parcours en profondeur complet.

Soit $uv \in A$. Distinguons deux cas :

- Si le parcours visite u avant v : alors comme $uv \in A$, la visite de u lance un appel sur v . Ainsi, v n'était pas visité à l'ouverture de u et est garanti de l'être à la fermeture de u : donc $d(u) < d(v) < f(u)$. D'après le théorème des parenthèses, on a en fait $d(u) < d(v) < f(v) < f(u)$ et on a donc bien $u < v$.
- Sinon il visite v avant u : alors montrons que $d(v) < f(v) < d(u) < f(u)$.
D'après le théorème des parenthèses, il n'y a que deux possibilités avec $d(v) < d(u)$:

- Celle que l'on veut prouver.
- Et l'imbrication des intervalles, c'est à dire $d(v) < d(u) < f(u) < f(v)$. Or, dans ce cas, d'après le théorème des parenthèses il y a un chemin dans l'arbre de parcours de v à u . Comme l'arbre de parcours est un sous-graphe, il y aurait un chemin dans G de v à u . Mais puisque $uv \in A$ cela donne un circuit : absurde.
Donc la seule possibilité était la précédente, qui donne bien $u < v$.

□

En particulier, on a prouvé que :

Corollaire 76.

Un graphe orienté G admet un ordre topologique si et seulement si c'est un DAG.

Si c'est le cas, un ordre peut-être calculé en temps linéaire en la taille du graphe par un parcours en profondeur.

3.4.2 Détection de cycles**Lemme 77 (Théorème du chemin blanc).**

Dans G un graphe orienté (pas forcément un DAG), un sommet v est descendant d'un sommet u dans la forêt de parcours si et seulement si lors de la visite de u (le moment $d(u)$) il existe un chemin de u à v composé uniquement de sommets qui non-encore ouverts.

Démonstration. \Rightarrow) Immédiat d'après le théorème des parenthèses.

\Leftarrow) Supposons que lors de la visite de u (donc au temps $d(u)$), il existe un tel chemin vers un sommet v mais que v ne devienne pas descendant de u dans l'arbre.

Sans perte de généralité, supposons que tous les autres sommets du chemin deviennent descendants de u (sinon, remplacer v par le premier sommet du chemin qui ne devient pas descendant). Notons w le sommet juste avant v dans le chemin :

FIGURE 11.41 – Preuve du chemin blanc

D'après le théorème des parenthèses, on a donc $d(u) < d(w) < f(w) < f(u)$.

Reste à se demander quand est-ce que v doit être ouvert/fermé par rapport à w . Pour que v ne soit pas descendant de w , il ne doit pas avoir été ouvert par w (théorème des parenthèses); mais puisqu'il est voisin de w il doit avoir été ouvert avant.

Donc :

$$d(u) < d(v) < d(w) < f(w) < f(u)$$

On conclut en appliquant encore le théorème des parenthèses qui donne $f(v) < f(u)$ et donc v descendant de u : absurde. □

Proposition 78.

Soit $G = (S, A)$ un graphe orienté. Alors G admet un circuit si et seulement si lors d'un DFS complet de G il existe un sommet x qui a un voisin y ouvert non-encore fermé.

Démonstration.

\Rightarrow) C'est à peu près comme la preuve du théorème de calcul d'ordre topologique : si lors de la visite de x , son voisin y est déjà ouvert et pas encore fermé, on a $d(y) < d(x) < f(y)$. D'après le théorème des parenthèses, cela implique $d(y) < d(x) < f(x) < f(y)$ et donc il y a un chemin de y à x . Mais y est voisin de x c'est à dire $\forall u \in A$: d'où un cycle.

\Leftarrow) Par contraposée. Considérons $s_0, \dots, \underbrace{s_p}_{=s_0}$ un circuit dans G . Quitte à renuméroter, supposons que s_0 soit le premier sommet visité par le parcours.

Par théorème du chemin blanc, s_{p-1} est descendant de s_0 et donc (théorème des parenthèses) : $d(s_0) < d(s_{p-1}) < f(s_{p-1}) < f(s_0)$. Autrement dit, lors du parcours de s_{p-1} , celui voit son voisin s_0 qui est ouvert non encore fermé. □

On en déduit un critère pour détecter si un graphe orienté est un DAG. Ainsi, on peut faire un code qui :

- Renvoie une erreur si le graphe donné en entrée contient un circuit.
- Renvoie un ordre topologique sinon.

3.4.3 Implémentation en OCaml

On utilise toujours les mêmes types pour implémenter. Afin d'alléger le code, remarquons qu'ici le tableau des dates d'ouverture ne sert pas : pour tester si un sommet v a déjà été ouvert, il suffit de tester si $\text{marque}[v]$ est vrai. En particulier, on va ne compter *que* les dates de fermeture, de 0 à $n - 1$.

De plus, l'étape finale consiste grosso-modo à « échanger » les indices et les valeurs du tableau fermeture. Plus précisément, le sommet fermé au temps $n - 1$ est le premier sommet de l'ordre donc va à l'indice 0 du tableau des sommets trié, celui fermé au temps $n - 2$ va à l'indice 1, etc : le sommet fermé au temps f va à l'indice $n - 1 - f$.

FIGURE 11.42 – Construction du tableau trié des sommets à l'aide du tableau des temps de fermeture


```

158 exception Cycle
159
160 let ordre_topo (g : graphe) : sommet array =
161   let n = Array.length g in
162   let fermeture = Array.make n (-1) in
163   let time = ref 0 in
164   let marque = Array.make n false in
165
166   let rec dfs_rec u =
167     if not marque.(u) then begin
168       marque.(u) <- true;
169
170       let check_voisin v =
171         if marque.(v) && fermeture.(v) = -1
172         then raise Cycle
173         else if not marque.(v) then dfs_rec v
174       in
175       List.iter check_voisin g.(u);
176
177       fermeture.(u) <- !time; incr time
178     end
179   in
180
181   (* Calculer toutes les fermetures *)
182   for u = 0 to n-1 do
183     if not marque.(u) then dfs_rec u
184   done;
185
186   (* Construire l'ordre à partir des fermetures *)
187   let ordre = Array.make n (-1) in
188   for u = 0 to n-1 do ordre.(n-1 - fermeture.(u)) <- u done;
189   ordre

```

4 Graphes bipartis

4.0 Définition

Définition 79 (Graphe biparti).

Un graphe $G = (S, A)$ (orienté ou non) est dit **biparti** s'il existe une bipartition $S = T \sqcup U$ tel que toute arête/arc de A ait une extrémité dans T et l'autre dans U .

Exemple.

(a) Un graphe biparti

(b) Un graphe biparti

FIGURE 11.43 – Des graphes bipartis

Exemple. On a défini au début du cours $K_{p,q}$ la clique bipartie.

Proposition 80.

Les arbres (graphe non-orienté acyclique connexe) sont bipartis.

Démonstration. C'est... surprenant embêtant avec la définition « graphe » des arbres (alors que ce serait très simple dans des arbres orientés du cours sur les arbres). Vous le prouverez en MPI. \square

Proposition 81.

Un graphe non-orienté G est biparti si et seulement si il est deux coloriable.

Démonstration. Par double implication.

\Rightarrow) Soit $S = T \sqcup U$ une bipartition des sommets qui vérifie la définition de graphe biparti. Colorions avec la couleur 0 les sommets de T et 1 ceux de U . Alors toute arête a une extrémité 0 et une extrémité 1, donc le coloriage est valide.

\Leftarrow) Soit $c : S \rightarrow \{0; 1\}$ une 2-coloration du graphe. Posons $T = c^{-1}(\{0\})$ et $U = c^{-1}(\{1\})$. Alors T et U sont bien une bipartition de S , et comme c est un coloriage valide il n'y a pas d'arête entre deux sommets de T ou entre deux de U .

\square

4.1 Couplage (dans un graphe biparti ou non)

Définition 82.

Soit $G = (S, A)$ un graphe non-orienté (pas forcément biparti). Un **couplage** est un ensemble $B \subseteq A$ d'arêtes dont les extrémités sont deux à deux distinctes.

Un couplage B est dit **parfait** si chaque sommet est incident à une arête du graphe.

Exemple. Pour choisir les duos de chambres à l'internat, on considère le graphe suivant :

- Chaque élève est un sommet
- Chaque arête est une possibilité de mettre deux élèves dans la même chambre. On peut imposer des contraintes en enlevant des arêtes, comme « pas de chambre mixte », ou « horaires de matin compatibles », etc.

Choisir les duos de chambre revient alors à choisir un couplage parmi ce graphe ! Notez que le graphe n'est pas biparti.

Exemple. Le couplage ci-dessous n'est pas parfait :

FIGURE 11.44 – Un couplage non-parfait dans un graphe (biparti)

Proposition 83.

Dans un graphe biparti, on peut trouver efficacement un couplage de cardinal maximal.

Démonstration. MPI! Mais si vous êtes impatient·es, il y a une résolution très bien expliquée dans cette leçon de Claire Mathieu au Collège de France : <https://www.college-de-france.fr/fr/agenda/lecon-inaugurale/algorithmes/algorithmes> (à partir de 11min35). \square

Remarque. Si je reprends mon exemple de l'internat, un couplage maximal est le remplissage du plus de chambres possibles (en respectant les contraintes)... c'est un objectif raisonnable.

Remarque. Une variante célèbre de ces problèmes de couplage dans des graphes bipartis consiste à donner des *préférences* à chaque noeud. C'est typiquement ce qui a lieu sur Parcoursup ! Les candidat·es ont des préférences vis à vis des formations, et les formations vis-à-vis des candidat·es. On cherche alors un couplage où deux candidat·es ne peuvent pas mutuellement améliorer leur satisfaction en échangeant leur formation ; on appelle cela un *couplage stable*. Pour en savoir plus : https://fr.wikipedia.org/wiki/Probl%C3%A8me_des_mariages_stables

Chapitre 12

ALGORITHMES GLOUTONS

Notions	Commentaires
Algorithme glouton fournissant une solution exacte.	On peut traiter comme exemples d'algorithmes exacts : codage de Huffman, sélection d'activité, ordonnancement de tâches unitaires avec pénalités de retard sur une machine unique.

Extrait de la section 4.3 du programme officiel de MP2I : « Décomposition en sous-problèmes ».

SOMMAIRE

0. Généralités	284
0. Principe général	284
<i>Définition (p. 284). Intérêt (p. 285).</i>	
1. Exemple : ordonnancement d'intervalle	285
2. Preuve d'optimalité	287
1. Plus d'exemples	288
0. Stockage de fichiers	288
1. Rendu de monnaie	288
<i>En euros (p. 289). Dans un système monétaire quelconque (p. 289).</i>	
2. Stay tuned !	289
<i>Algorithme de Huffman (p. 289). Algorithme de Kruskal (p. 290). Algorithme ID3 (p. 290).</i>	
3. Message d'utilité publique	290
2. (HP) Théorie des matroïdes	291
0. Matroïde	291
1. Algorithme glouton	291

0 Généralités

Contexte : On s'intéresse dans ce cours à la résolution de problèmes dont la construction d'une solution peut-être décomposée comme une succession de choix (comme en retour sur trace!). Le problème que l'on étudie peut-être un problème d'existence, ou bien (plus fréquemment!) un problème d'optimisation.

0.0 Principe général

0.0.0 Définition

Définition 1.

La technique algorithmique dite **gloutonne** consiste à résoudre un problème de la manière suivante :

- On construit une solution graduellement, choix après choix. On ne reviendra jamais en arrière.
- À chaque étape, on fait le choix qui est localement optimal.

Exemple. Considérons SUBSETSUM, version problème d'optimisation : étant donné une liste ℓ d'entiers et $t \in \mathbb{N}$, quelle est la somme maximale d'éléments de ℓ qui soit inférieure à t ?

Une solution gloutonne est de parcourir la liste et de prendre chaque élément si on peut l'ajouter aux éléments déjà pris sans que cela ne dépasse t .

FIGURE 12.1 – Algorithme glouton pour SUBSETSUM avec $\ell = [3; 2; 2; 10; 4; 5]$ et $t = 20$

Remarque.

- L'exemple précédent montre qu'un algorithme glouton peut ne pas renvoyer une solution optimale (il y a une solution qui se somme à 19). En fait, c'est même souvent le cas : une succession de choix localement optimaux mène rarement à une solution globalement optimale !
- En fait, un glouton ne permet presque jamais de calculer une solution optimale, mais souvent une approximation correcte d'une solution optimale (cf MPI).
- La différence avec le retour sur trace est que ce dernier peut revenir en arrière pour remettre en question un choix précédent¹.

1. Cela signifie aussi que quand vous écrivez un "retour sur trace" récursif qui ne vérifie pas le résultat de l'appel récursif... vous ne remettez pas en cause le choix actuel s'il a été invalidé par les appels récursifs, et vous faites donc en fait un algorithme glouton.

0.0.1 Intérêt

Avantages	Inconvénients
Simple à concevoir Faible complexité temporelle Permet parfois d'approximer décemment une solution optimale. Les preuves de correction sont souvent (très) similaires.	Renvoie généralement une solution non optimale.

TABLE 12.1 – Avantages et inconvénients d'un algorithme glouton

0.1 Exemple : ordonnancement d'intervalle

On considère un gymnase qui organise une journée sportive dans sa grande salle. De nombreuses personnes veulent organiser une activité : elles requièrent la salle sur un intervalle de temps donné. Deux activités ne peuvent pas avoir lieu en même temps. Comment organiser le plus d'activités possibles ?

FIGURE 12.2 – Problème du gymnase (ordonnancement d'intervalles)

Définition 2 (Ordonnancement d'intervalles).

Le problème d'ordonnancement d'intervalles est le suivant :

Entrée : un ensemble $S = \{r_0, \dots, r_{n-1}\}$ de n requêtes. Chaque requête r_i est un intervalle ouvert non-vide $]d_i; f_i[$ (avec $d_i < f_i$).

Tâche : Trouver la plus grande partie de S constituée d'ensembles deux à deux disjoints

Remarque.

- En anglais, ce problème s'appelle *Interval Scheduling*.
- On peut représenter le problème comme un graphe : les requêtes sont des noeuds, et deux noeuds sont reliés si leurs intervalles s'intersectent. Comme l'intersection est symétrique, le graphe est non-orienté. On veut alors trouver un ensemble de noeuds deux à deux non voisins : on appelle cela un *indépendant* du graphe.

Il s'avère que trouver un plus grand indépendant dans un graphe quelconque correspond à un problème NP-Complet (donc a priori impossible à résoudre en temps polynomial²)... sauf que dans notre gymnase, on ne se base pas sur n'importe quel graphe : c'est un graphe (d'intersection d')intervalles ! Peut-être existe-t-il un glouton ?

2. Sauf si $P = NP$. Mais je m'avance beaucoup sur la MPI là !

Idée gloutonne	Contre-exemple à l'optimalité

TABLE 12.2 – Des idées gloutonnes pour Interval Scheduling et des contre-exemples

0.2 Preuve d'optimalité

Proposition 3 (Preuve par échange).

Pour prouver qu'un algorithme glouton est correct (c'est à dire qu'il renvoie bien une solution optimale), on procède souvent ainsi :

1. Lemme d'échange : Montrer qu'il existe une solution optimale où le premier choix effectué est celui de l'algorithme glouton.
2. Récurrence : Montrer qu'on peut combiner ce premier choix glouton avec une solution optimale du sous-problème qu'il reste à résoudre en une solution optimale du problème. En déduire la correction de l'algorithme par récurrence.

Remarque.

- Le schéma de preuve proposé ci-dessus marche pour *beaucoup* d'algorithmes gloutons.
- La difficulté est généralement le lemme d'échange ; mais il ne faut pas oublier de conclure par récurrence pour prouver que la solution gloutonne est optimale !

Exemple. Prouvons la correction de l'algorithme glouton « prendre en priorité les requêtes se terminant en premier » pour l'ordonnancement d'intervalles. Considérons donc $\mathcal{R} = \{r_0, \dots, r_{n-1}\}$ les requêtes, avec $r_i =]d_i; f_i[$; on les suppose triées par f_i croissant.

- Lemme d'échange : Montrons qu'il existe une solution optimale contenant la requête r_0 . Soit $S_{opt} = \{r_{i_0}, \dots, r_{i_{k-1}}\}$ une solution optimale (triée par date de fin croissante). Si $r_{i_0} = r_0$, c'est bon ; sinon montrons qu'on peut échanger r_{i_0} avec r_0 .

FIGURE 12.3 – Notations de la preuve du lemme d'échange

Comme S_{opt} est trié, pour $j > 0$ on a $f_{i_0} \leq f_{i_j}$. De plus, comme S_{opt} est une solution, ses intervalles ne s'intersectent pas : donc l'inégalité précédente implique aussi $f_{i_0} \leq d_{i_j}$: les intervalles suivants de S_{opt} débutent après la fin de r_{i_0} . Comme $f_0 \leq r_{i_0}$ (r_0 se termine avant r_{i_0}), on peut bien remplacer r_{i_0} par r_0 sans créer d'intersections.

Le nouvel ensemble obtenu contient toujours autant d'intervalles, il est donc également optimal : il existe une solution optimale contenant r_0 le premier choix glouton.

- Récurrence : On montre par récurrence forte sur le nombre d'événements que l'algorithme glouton renvoie le bon résultat.

— Initialisation : Trivial pour $n=0$.

— Hérédité : Soit $n > 0$. L'algorithme glouton choisit tout d'abord r_0 . Les intervalles suivants qu'il prend ne s'intersectent pas avec r_0 : ce sont des intervalles de $\mathcal{R}' = \{r_j \text{ t.q. } r_j \cap r_0 = \emptyset\}$. L'algorithme glouton renvoie donc $\{r_0\} \cup S'$ avec S' une solution de \mathcal{R}' ; et par hypothèse de récurrence S' est solution optimale de \mathcal{R}' .

Reste à prouver que S est une solution optimale pour \mathcal{R} . D'après le lemme d'échange, il existe une solution optimale de la forme $\{r_0\} \cup E'$ avec E' des intervalles. Or, tous les intervalles de E' n'intersectent pas r_0 : ce sont donc des intervalles de \mathcal{R}' . En particulier, comme \mathcal{R}' est optimal, $|\mathcal{R}'| \geq |E|$ et donc $|\{r_0\} \cup S'| \geq |\{r_0\} \cup E|$.

On a montré que la solution gloutonne choisit au moins autant d'intervalles qu'une solution optimale : elle est donc optimale.

Remarque. Une implémentation de cette algorithme glouton se fait en $O(n \log_2 n)$: d'abord on trie les intervalles (en $O(n \log n)$). Ensuite, on prend un intervalle à la condition qu'il n'intersecte pas le dernier intervalle que l'on a pris. Comme les intervalles sont triés par date de fin, cela suffit à garantir que

l'intervalle que l'on veut prendre n'intersecte *aucun* intervalle précédent : le test de non-intersection avec les intervalles déjà pris se fait ainsi en temps constant³ ! Soit une complexité totale en $O(n \log_2 n)$.

1 Plus d'exemples

1.0 Stockage de fichiers

On dispose de n de poids respectifs p_0, \dots, p_{n-1} . On veut ranger ces fichiers possibles dans un disque dur de capacité maximale P . L'objectif est de ranger le plus de fichiers possibles⁴.

On propose l'algorithme glouton suivant : « prendre en priorité les fichiers de plus faible poids ».

Prouvons-le correct. Quitte à trier, supposons les p_i croissants.

- Lemme d'échange : Soit $S_{opt} = \{p_{i_0}, \dots, p_{i_k}\}$ une solution optimale non-vide⁵. On a donc $\sum_{j=0}^k p_{i_j} \leq P$. Comme $p_0 \leq p_{i_0}$, cette majoration reste vraie en échangeant p_{i_0} avec p_0 . On obtient ainsi une solution optimale qui contient p_0 le premier choix glouton.
- Récurrence : Prouvons par récurrence forte sur le nombre n de fichiers que l'algorithme glouton crée une solution optimale sur une entrée à n fichiers et avec poids maximal P quelconque.
 - Notons que c'est trivial pour tous les cas où $P < p_0$ puisque tout fichier est alors trop lourd et l'algorithme glouton renvoie \emptyset qui est optimal car seule solution possible. On ignore donc ces cas par la suite.
 - Initialisation : Trivial pour $n = 0$.
 - Hérédité : S'il y a $n > 0$ fichiers, alors l'algorithme glouton choisit d'abord p_0 puis un ensemble S' d'autres fichiers. Le poids de ces autres fichiers ne peut dépasser $P - p_0$. Notons F' l'ensemble des fichiers plus lourds que p_0 ; par H.R. S' est une solution optimale de $(F', P - p_0)$.

Or, par lemme d'échange, il existe une solution optimale de la forme $\{p_0\} \cup E'$. Or, dans cette solution, par définition de p_0 tous les éléments de E' sont plus lourds que p_0 donc sont dans F' . De plus, comme c'est une solution, on a $p_0 + \sum_{p \in E'} p \leq P$ donc les éléments de E' ont un poids total inférieur à $P - p_0$. Ainsi, E' est une solution de l'instance $(F', P - p_0)$.

Par optimalité de S' dans F' , $|S'| \geq |E|$. En particulier, la solution renvoyée par l'algorithme glouton a au moins autant de fichiers que $\{p_0\} \cup E$ donc est optimale.

Remarque.

- L'algorithme ci-dessus est en complexité $O(n \log n)$, parce qu'il faut commencer par trier.
- La comparaison avec le problème du sac à dos est frappante : on ne connaît pas de solution polynomiale au sac à dos (et on pense qu'il n'y en a pas car on pense que $P \neq NP$) ; pourtant si on fixe toutes les valeurs du sac à dos à 1 on obtient le problème que l'on vient de résoudre avec un *glouton*.

1.1 Rendu de monnaie

Le problème du **rendu de monnaie** est le suivant : étant donnée une somme s , comme rendre⁶ la somme s en utilisant le moins de pièces/billets⁷ possibles ?

Exemple. en euros, pour rendre $s = 48$, le nombre minimal de pièces à utiliser est 5 : $48 = 20 + 20 + 5 + 2 + 1$.

3. En particulier, le tri nous permet aussi de gagner en efficacité sur les tests d'intersection. On dit que c'est un prétraitement qui permet d'accélérer l'algorithme (et de le rendre correct - c'est toujours sympa).

4. Autrement dit, c'est un problème du sac à dos où tous les poids valent 1

5. En toute rigueur, il faudrait traiter à part le cas trivial où toute solution optimale est vide.

6. « Rendre » signifie ici « donner ».

7. Pour simplifier, je parlerai uniquement de pièces dans la suite

1.1.0 En euros

On propose l'algorithme glouton suivant : « Utiliser la plus grosse pièce possible jusqu'à avoir tout rendu ». Remarquez que c'est l'algorithme qui a été utilisé pour rendre la monnaie dans l'exemple précédent.

Montrons qu'il est optimal... mais cette fois-ci ; la preuve ne suivra pas exactement le schéma habituel. Pour simplifier la preuve, je vais me limiter aux pièces suivantes : 1, 2, 5, 10.

Notons que dans toute solution optimale, il y a au plus :

- Une pièce de 5 : si on en utilise deux, on peut les remplacer par une de 10.
- Deux de 2 : si on en utilise 3, on peut les remplacer par une de 5 et une de 1.
- Une de 1 : idem que 5.

Aussi, dans toute solution optimale, il y a au plus quatre pièces qui ne sont pas des 10. Mais de plus, on n'utilise pas ces quatre pièces à la fois car $5 + 2 + 2 + 1 = 10$: donc les pièces qui ne sont pas des dix se somment à au plus 9. Donc le nombre de 10 utilisés est $\lfloor \frac{s}{10} \rfloor$, c'est à dire le nombre maximal de fois où l'on peut prendre un 10 : c'est exactement ce que fait l'algorithme glouton.

On conclut ensuite par récurrence : si $s < 10$, on vérifie que l'algorithme glouton est optimal. Sinon, d'après ce qui précède⁸, l'algorithme glouton, comme toute solution optimale, rend $\lfloor \frac{s}{10} \rfloor$ pièces de 10 puis rend $s - \lfloor \frac{s}{10} \rfloor$. Or cette seconde étape est faite optimalement d'après l'initialisation, et on conclut.

1.1.1 Dans un système monétaire quelconque

La preuve précédente fonctionnait car on pouvait proposer des remplacements efficaces de plusieurs petites pièces par une grosse : par exemple, trois 2 remplacés par un 5 et un 1. Mais cela n'est pas le cas dans tout système monétaire ! Considérez par exemple un système monétaire qui a des pièces de 6, de 4 et de 1. Alors pour rendre 8, l'algorithme glouton utilise $8 = 6 + 1 + 1$ alors que $8 = 4 + 4$ est mieux !

Autrement dit, dans un système monétaire quelconque l'algorithme glouton ne fonctionne pas. Heureusement, on peut trouver une formule de récurrence : le nombre minimal $n(s)$ de pièces à utiliser pour rendre s peut se définir ainsi :

$$n(s) = \begin{cases} 0 & \text{si } s = 0 \\ 1 + \min\{s - p \mid p \text{ est une pièce tq } p \leq s\} & \text{sinon} \end{cases}$$

Cette formule fonctionne peu importe le système monétaire : on peut donc l'utiliser. Comme elle risque d'appeler plusieurs fois les mêmes valeurs intermédiaires, on a tout intérêt à utiliser de la programmation dynamique !

Exercice. Quelle complexité obtient-on avec de la programmation dynamique bien écrite ?

1.2 Stay tuned!

Plusieurs algorithmes autres gloutons seront vus en MP2I ou en MPI :

1.2.0 Algorithme de Huffman

L'algorithme de Huffman est un algorithme glouton qui prend en entrée un texte et renvoie un encodage binaire des lettres du texte de sorte que à ce que l'encodage de tout le texte (concaténation des encodages des lettres) soit le plus léger possible (tout en restant décodable).

Nous le verrons le mois prochain, dans le chapitre sur l'algorithmique du texte !

8. Qui ressemble donc beaucoup à un lemme d'échange...

1.2.1 Algorithme de Kruskall

En MPI, vous étudierez le problème de trouver un arbre couvrant de poids minimal : étant donné un graphe non-orienté aux arêtes pondérées, comment trouver un sous-graphe qui est un arbre dont la somme des arêtes est le plus faible possible ? L'algorithme de Kruskall est un glouton qui procède ainsi : « prendre l'arête de poids le plus faible qui ne crée pas de cycle ; répéter jusqu'à avoir pris $n-1$ arêtes ».

FIGURE 12.4 – Un arbre couvrant de poids minimal

1.2.2 Algorithme ID3

En MPI, vous étudierez le problème de construire un arbre de décision. C'est un arbre qui permet de classer des données dans différentes catégories en fonction de critères binaires.

FIGURE 12.5 – Un arbre de décision

L'algorithme ID3 est un algorithme qui essaye de construire un arbre à partir d'un ensemble de données déjà classées, de manière gloutonne : il met en racine la question binaire qui maximise le gain d'information⁹, et continue récursivement.

Cependant, cet algorithme glouton-ci n'est pas optimal au sens où il ne crée pas un arbre de décision de hauteur minimale - mais il est rapide et ses arbres ne sont tout de même pas trop mauvais !

1.3 Message d'utilité publique

LES GLOUTONS QUI RENVOIENT UNE SOLUTION OPTIMALE SONT L'EXCEPTION ET NON LA RÈGLE.

Au lieu de chercher un glouton, on peut plutôt chercher une expression d'une solution optimale, souvent récursive¹⁰.

9. Notion compliquée que vous verrez en MPI.

10. Et on peut souvent optimiser l'implémentation de cette écriture récursive par programmation dynamique.

2 (HP) Théorie des matroïdes

Il y a un cas où l'on *sait* que l'algorithme glouton fonctionne : les matroïdes. C'est peut-être un peu compliqué (et tout à fait hors-programme).

2.0 Matroïde

Définition 4 (Matroïde).

Soit E un ensemble fini non-vidé et $\mathcal{I} \subseteq \mathcal{P}(E)$ également non-vidé. On dit que \mathcal{I} est un **matroïde** de E lorsque :

- Échange : Pour tous $X, Y \in \mathcal{I}$, si $|X| < |Y|$ alors il existe $e \in Y \setminus X$ tel que $X \cup \{e\} \in \mathcal{I}$.
- Hérédité : Pour tout $X \in \mathcal{I}$, tout $Y \subseteq X$ vérifie $Y \in \mathcal{I}$.

Les éléments de \mathcal{I} sont appelés des **indépendants**.

Remarque. Dans l'idée, les indépendants sont des solutions d'un problème. Les deux conditions se comprennent alors ainsi :

- Échange : si une solution Y est plus grande que X , alors on peut agrandir la solution X en lui un élément bien choisi de Y .
- Hérédité : une « sous-solution » est une solution.

Définition 5 (Indépendant maximal).

Soit \mathcal{I} un matroïde de E et $Y \in \mathcal{I}$. On dit que X est **maximal** s'il est maximal pour l'inclusion parmi les indépendants.

Remarque. Autrement dit, un indépendant est maximal si on ne peut plus lui rajouter d'éléments.

Lemme 6 (Indépendants maximaux d'un matroïde).

Tous les indépendants maximaux d'un matroïde ont le même cardinal.

Démonstration. Soit X et Y deux indépendants. Par échange, si $|X| < |Y|$, alors il existe $e \in Y \setminus X$ tel que $\{e\} \cup X$ soit aussi indépendant : en particulier, X n'était pas maximal. En particulier, c'est absurde si X et Y sont maximaux mais de cardinaux distincts. \square

Définition 7 (Matroïde pondéré).

Un **matroïde pondéré** \mathcal{I} de E est un matroïde accompagné d'une fonction de poids $w : E \rightarrow \mathbb{R}_+$. On définit le poids de $X \in \mathcal{P}(E)$, noté $w(X)$, comme $\sum_{x \in X} w(x)$.

2.1 Algorithme glouton

On s'intéresse au problème de trouver un indépendant de poids maximal dans un matroïde pondéré.

On propose l'idée gloutonne suivant : trier les éléments de $E = \{e_0, \dots, e_{n-1}\}$ par poids décroissant. Partir de l'ensemble vide, et pour chaque e_i l'ajouter aux éléments déjà pris si ce que l'on obtient est un indépendant. Autrement dit :

Algorithme 20 : Glouton pour un indépendant de poids maximal

Entrées : (\mathcal{I}, E, w) un matroïde pondéré
Sorties : M un indépendant de poids maximal

```

1 Trier  $E = \{e_0, \dots, e_{n-1}\}$  par poids décroissant.
2  $M \leftarrow \emptyset$ 
3 pour  $i$  de 0 à  $n - 1$  faire
4   | si  $A \cup \{e_i\} \in \mathcal{I}$  alors
5   |   |  $A \leftarrow A \cup \{e_i\}$ 
6 renvoyer  $A$ 
```

Théorème 8 ((HP) Optimalité de glouton dans un matroïde pondéré).

L'algorithme précédent renvoie un indépendant de poids maximal.

Démonstration.

- Lemme d'échange : Notons e_k le premier élément choisi par glouton, autrement dit le premier élément ajouté à A . Soit S_{opt} un indépendant de poids maximal. Si $e_k \in S_{opt}$, on a prouvé le lemme d'échange. Sinon, par propriété d'échange des matroïdes, on peut ajouter à $\{e_k\}$ un élément x de S_{opt} . En réappliquant l'échange à $\{e_A, x\}$ et S_{opt} , on peut en ajouter un second : etc etc, par $|S_{opt}| - 1$ applications de la propriété d'échange on construit X un indépendant qui contient e_A et $|S_{opt}| - 1$ éléments de S_{opt} . Notons e_S l'élément de S_{opt} qui n'est pas dans X .

Come $\{e_S\} \subseteq S_{opt}$, $\{e_S\}$ est un indépendant : en particulier, comme e_A est le premier élément de E dont le singleton forme un indépendant on a $w(e_A) \geq w(e_S)$. Mais on peut alors conclure :

$$\begin{aligned}
 w(X) - w(S_{opt}) &= \sum_{x \in X} w(x) - \sum_{y \in S_{opt}} w(y) \\
 &= w(e_A) - w(e_S) && \text{car ce sont les seuls termes différents de } X \text{ et } S_{opt} \\
 &\geq 0 && \text{d'après ce qui précède}
 \end{aligned}$$

Ainsi, X est un indépendant de poids supérieur à S_{opt} (qui est de poids maximal), donc est de poids maximal.

- Récurrence :
 - Immédiat si $|E| = 0$
 - Sinon, en notant e_k le premier choix glouton, on pose $E' = \{e_0, \dots, e_{k-1}\}$ et $\mathcal{I}' = \{X \subseteq E' \text{ tq } X \cup \{e_k\} \in \mathcal{I}\}$. Par propriété d'hérédité, tous les indépendants de \mathcal{I}' sont des indépendants de \mathcal{I} et on en déduit que \mathcal{I}' est un indépendant de \mathcal{I}' : l'algorithme glouton renvoie donc une solution optimale sur \mathcal{I}' .
 On conclut alors à l'aide du lemme d'échange comme dans les preuves gloutonnes précédentes.

□

Remarque.

- Les matroïdes sont hors-programme, mais ils permettent de comprendre pourquoi « lemme d'échange + récurrence » fonctionne souvent pour les gloutons : c'est parce que beaucoup de problèmes qui ont une solution gloutonne sont des matroïdes cachés.
- Attention, tous les problèmes admettant une solution gloutonne ne sont pas des matroïdes ! Premièrement parce que « algorithme glouton » est défini assez vaguement, et deuxièmement parce que l'informatique n'est pas triviale¹¹.

11. Ce serait trop beau d'avoir une méthode magique pour savoir si un problème admet ou non une solution gloutonne...

Exercice. Soit $G = (S, A)$ un graphe non-orienté pondéré par $w : A \rightarrow \mathbb{R}_+$. On dit que $X \subseteq A$ est une forêt si le graphe obtenu en ne gardant que les arêtes de X est une forêt. Alors l'ensemble des forêts est un matroïde pondéré de A . En particulier, en choisissant la bonne relation d'ordre, on en déduit un algorithme pour trouver une forêt de poids minimal : c'est l'algorithme de Kruskal (MPI)!

Chapitre 13

STRUCTURES DE DONNÉES (S2)

Notions	Commentaires
Structure de tableau associatif implémenté par une table de hachage.	La construction d'une fonction de hachage et les méthodes de gestion des collisions éventuelles ne sont pas des exigences du programme.

Extrait de la section 3.2 du programme officiel de MP2I : « Structures de données séquentielles ».

Notions	Commentaires
Implémentation d'un tableau associatif par un [...] arbre bicolore.	On note l'importance de munir l'ensemble des clés d'un ordre total.
Propriété de tas. Structure de file de priorité implémentée par un arbre binaire ayant la propriété de tas.	Tri par tas.

Extrait de la section 3.3 du programme officiel de MP2I : « Structures de données hiérarchiques ».

SOMMAIRE

0. Tableaux associatifs	296
0. Type abstrait	296
<i>Motivation (p. 296). Type abstrait (p. 296).</i>	
1. Implémentation avec des listes d'associations	297
2. Implémentation avec des ABR équilibrés	298
3. Implémentation avec des tables de hachage	299
<i>Fonction de hachage (p. 299). Table de hachage (p. 300). Attention aux objets mutables! (p. 301).</i>	
4. Complexités	301
5. Tri par dénombrement	302
6. Un type d'ensemble	303
1. Tas	304
0. Définition	304
<i>Définition (p. 304). Implémentation en OCaml (p. 306).</i>	
1. Opérations	306
<i>Percolation vers le haut (p. 306). Insertion (p. 308). Percolation vers le bas (p. 309). Extraction du minimum (p. 311).</i>	
2. Complexités	312
3. Tri par tas	312
2. Files de priorité	313
0. Type abstrait	313
1. Implémentation avec des ABR équilibrés	314
2. Implémentation avec des tas	315
3. Complexités	315
4. Une opération supplémentaire	316

0 Tableaux associatifs

0.0 Type abstrait

0.0.0 Motivation

Les tableaux que l'on a manipulés jusqu'à présent stockent une valeur par indice : autrement dit, à chaque indice ils *associent* une valeur.

Exemple. Dans les graphes, on a beaucoup manipulé des tableaux qui à un noeud associe son parent dans l'arbre de parcours, ou le fait que le noeud soit marqué ou non.

Cependant, quand il s'agit de stocker des associations, les tableaux ont deux limitations :

- Les indices d'un tableau sont obligatoirement des entiers.
- Les indices sont obligatoirement un segment $\llbracket 0; n \rrbracket$ avec n connu à l'avance.

On voudrait pouvoir échapper à ces deux limitations.

Définition 1 (Tableau associatif).

Un **tableau associatif**, aussi appelé **dictionnaire**, est une structure de données abstraite qui permet de stocker des associations de la forme

$$clef \mapsto valeur$$

Exemple.

FIGURE 13.1 – Un tableau associatif

Exemple.

- Un... dictionnaire (par exemple du français) est un tableau associatif qui à un mot (clef) associe sa définition (valeur).
- Un annuaire à un nom (clef) associe un numéro de téléphone (valeur).
- Si on se place dans un graphe (où les sommets ne sont pas forcément des entiers), la représentation par listes d'adjacences à un sommet (clef) associe la liste de ses voisins (valeur).

0.0.1 Type abstrait

Les opérations que l'on veut pouvoir faire sur un dictionnaire sont :

- Obtenir un dictionnaire vide.
- Tester si une clef est associée à quelque chose.
- Accéder à la valeur associée à une clef.
- Ajouter une association $clef \mapsto valeur$

- Enlève l'association d'une clef.
- Tester si un dictionnaire est vide.
- Obtenir une association d'un dictionnaire (n'importe laquelle).

Remarque. Il y a une subtilité : que se passe-t-il quand on ajoute une nouvelle association pour une clef déjà associée ? Différents fonctionnements sont possibles :

- On peut lever une erreur. C'est peu pratique, réécrire dans une « case » d'un dictionnaire est courant pour la même raison que réécrire dans une case d'un tableau est courant.
- La nouvelle association peut écraser la précédente (comme dans un tableau).
- La nouvelle association se rajoute « par dessus la précédente » : la nouvelle est l'association « active » (donc quand on accède on trouve la nouvelle valeur), mais si on supprime à l'avenir l'association de la clef est enlevée seule la nouvelle association sera enlevée.

FIGURE 13.2 – Plusieurs d'associations pour une même clef

Voici le type abstrait du type dictionnaire :

<u>Type</u> :	Dict
<u>Utilise</u> :	Clef, Valeur, Bool
<u>Opérations</u> :	
	<code>dict_vide</code> : Rien \rightarrow Dict
	<code>est_asso</code> : Dict \times Clef \rightarrow Bool
	<code>acces</code> : Dict \times Clef \rightarrow Valeur
	<code>ajoute</code> : Dict \times Clef \times Valeur \rightarrow Dict
	<code>enlève</code> : Dict \times Clef \rightarrow Dict
	<code>est_vide</code> : Dict \rightarrow Bool
	<code>choisir</code> : Dict \rightarrow Clef \times Valeur

TABLE 13.1 – Signature fonctionnelle du type abstrait dictionnaire

0.1 Implémentation avec des listes d'associations

Définition 2 (Liste d'associations).

Une liste d'associations est une liste linéaire de paires (*clef*, *valeur*).
C'est une façon naïve d'implémenter des tableaux associatifs.

Exemple. Ci-dessus une liste d'associations $\text{string} \mapsto \text{int}$, où la liste linéaire est implémentée par une `list` :



```

1 let dict = [
2   ("dragon", 6);
3   ("forêt", 5);
4   ("chateau", 7);
5   ("cave", 4);
6   ("abrac", 11);
7   ("réponse", 42);
8   ("souterrain", 10)
9 ]

```

En OCaml, plusieurs fonctions du module `List` sont pré-codées pour manipuler des listes d'associations : https://ocaml.org/manual/4.14/api/List.html#1_Associationlists

L'avantage principal de cette façon de faire est sa simplicité, l'inconvénient est sa complexité temporelle¹...

0.2 Implémentation avec des ABR équilibrés

Pour cette implémentation, il *faut* que les clefs soient munies d'un ordre total.

Définition 3 (Arbres équilibrés pour tableaux associatifs).

Étant donné un ensemble \mathcal{K} totalement ordonné (ensemble de clefs), et \mathcal{V} (ensemble de valeurs) de valeur, on peut représenter $A \subseteq \mathcal{K} \times \mathcal{V}$ dans un arbre binaire de recherche comme suit :

- Un noeud stocke une paire (une association) $(k, v) \in A$
- Pour comparer deux noeuds, on compare uniquement les clefs.

Exemple. L'arbre ci-dessous stocke les mêmes associations que la liste d'associations précédente :

FIGURE 13.3 – Implémentation d'un tableau associatif par un arbre binaire de recherche

Proposition 4.

Pour que cette implémentation soit efficace, il faut que l'arbre soit équilibré. On peut par exemple utiliser des arbres Rouge-Noir.

Remarque.

- On peut aussi utiliser des AVL pour équilibrer les ABR (hors-programme, mais vous avez un lien vers une très bonne explication dans le cours sur les arbres).
- Si les clefs sont des strings et ont beaucoup de préfixes en commun, utilisent un *trie* au lieu d'un ABR peut-être pertinent.

1. Donc pour des petits dictionnaires, pas de problèmes !

0.3 Implémentation avec des tables de hachage

0.3.0 Fonction de hachage

Le cas facile des tableaux associatifs est celui où l'ensemble des clefs est de la forme $\llbracket 0; n \rrbracket$, puisque l'on peut alors simplement utiliser des tableaux. L'objectif des tables de hachage est de se ramener à ce cas facile.

Définition 5 (Fonction de hachage).

Soit $M \in \mathbb{N}^*$ (plutôt petit) et \mathcal{K} un (grand) ensemble de clefs.

Une **fonction de hachage** pour \mathcal{K} est une fonction $h : \mathcal{K} \rightarrow \llbracket 0; M \rrbracket$.

Pour $k \in \mathcal{K}$, on dit que $h(k)$ est le **hash** (ou **empreinte**) de k .

Si $h(x) = h(y)$, on dit que x et y sont en **collision**.

Remarque. Une fonction de hachage **doit** être déterministe : on veut que hacher deux fois le même objet donne deux fois la même empreinte !

Exemple.

- Si $\mathcal{K} = \text{string}$, un hachage possible est de prendre la somme des lettres ($a = 1, b = 26, \dots$) modulo M . En pratique, cette façon de faire ne répartit pas assez uniformément (car certaines lettres sont bien plus probables que d'autres).
- Si $\mathcal{K} = \text{double}$, on peut prendre le code binaire de chaque flottant, l'interpréter comme un entier et le prendre modulo M .
- Plus généralement, une fonction de hachage hache souvent la clef en un entier, puis se ramène $\llbracket 0; m \rrbracket$ par un modulo.

FIGURE 13.4 – Fonctionnement de beaucoup de fonctions de hachage

Proposition 6 (Qualités d'une fonction de hachage).

Une « bonne » fonction de hachage est une fonction qui a peu de collision. Autrement dit, elle répartit « à peu près uniformément » les éléments de \mathcal{K} :

$$\mathbb{P}(h(x) = h(y)) \simeq \frac{1}{M}$$

Remarque.

- En pratique, les clefs ne sont pas toujours équiprobables, on ne prend donc pas toujours la loi uniforme (sur l'ensemble des (x, y)) pour \mathbb{P} .
- Une autre propriété désirable est la **pseudo-injectivité** : que deux éléments proches aient des empreintes éloignées. En effet, en pratique, quand on hache des éléments on hache souvent beaucoup d'éléments « les uns à la suite des autres ». La pseudo-injectivité permet alors d'à peu près garantir que tous ces éléments auront une empreinte différente.
En fait, la pseudo-injectivité peut même servir à remplacer le hachage uniforme : en pratique, c'est plus simple à obtenir et fonctionne au moins aussi bien.

0.3.1 Table de hachage

Définition 7 (Table de hachage).

Soit \mathcal{K} ensemble de clefs et \mathcal{V} de valeurs. On fixe $M \in \mathbb{N}^*$ et on se dote d'une fonction de hachage associée.

Une **table de hachage** est une façon d'implémenter un tableau associatif qui fonctionne ainsi :

- La table est un tableau à M cases, appelées **alvéoles**.
- Dans l'**alvéole** d'indice e se trouve la liste des paires (clef, valeur) dont l'empreinte de la clef est e .

FIGURE 13.5 – Schéma d'une table de hachage

Exemple. Pour les mêmes associations que précédemment, avec $M = 5$ et hachage la somme des lettres modulo M :

FIGURE 13.6 – Implémentation d'un tableau associatif par une table de hachage

Les tables de hachage sont fournies en OCaml par le module `Hashtbl` : <https://ocaml.org/manual/4.14/api/Hashtbl.html>

Remarque. Dans la définition donnée ici, les collisions sont gérées en stockant la liste des valeurs en collision pour chaque empreinte. Il existe d'autres façons de faire, par exemple en faisant un deuxième niveau de hachage pour encore diminuer le nombre de collisions :

FIGURE 13.7 – Table à double hachage

0.3.2 Attention aux objets mutables !

Les objets dont le contenu est mutable (comme les tableaux) sont assez embêtants quand on les hash. En effet, est-ce que l'on veut hacher :

- Leur contenu : dans ce cas, hacher deux fois le même objet mutable (par exemple le même tableau) pourra tout à fait ne pas donner deux fois la même empreinte ! C'est embêtant, ça veut dire qu'en modifiant l'objet mutable on « oublie » où se trouve toutes ses associations dans le dictionnaire...
- Leur adresse : dans ce cas, deux variables différentes mais ayant exactement le même contenu (par exemple deux tableaux distincts dont le contenu des cases est égal, ou pire encore deux pointeurs qui pointent vers la même mémoire) ne seront pas identifiés comme identiques...

Proposition 8.

En OCaml, le hachage d'un objet mutable hache son *contenu*. Ainsi, hacher deux fois le même tableau peut tout à fait ne pas donner la même empreinte.

(HP) En Python, hacher un objet mutable est interdit. On va alors généralement calculer la valeur du contenu et la hacher elle.

0.4 Complexités

En notant A l'ensembles des associations à stoker, et M la taille de la table de hachage (et on supposant que les comparaisons/hachage de clefs se font temps constant) :

Opération \ Implémentation	Liste d'associations	ARN	Table de hachage
dict_vide	1	1	1
est_asso	$ A $	$\log_2 A $	1
acces	$ A $	$\log_2 A $	dépend des collisions (gros 1 en pratique)
ajoute (avec doublons de clefs)	1	$\log_2 A $	dépend des collisions (gros 1 en pratique)
ajoute (sans doublons de clefs)	$ A $	$\log_2 A $	dépend des collisions (gros 1 en pratique)
enleve	$ A $	$\log_2 A $	dépend des collisions (gros 1 en pratique)
est_vide	1	1	M (ou moins selon l'implem)
choisir	1	1	M

TABLE 13.2 – Complexités des opérations usuelles sur les dictionnaires

Remarque. La complexité des tables de hachage est en pratique un (un peu gros) $O(1)$.

0.5 Tri par dénombrement

Les dictionnaires permettent d'écrire l'algorithme de tri ci-dessous, où je note $d[x]$ la valeur associée à x dans le dictionnaire d :

Algorithme 21 : Tri par dénombrement

Entrées : arr un tableau à trier
Sorties : Rien, mais modifie arr pour le trier

```

1  $d \leftarrow$  dictionnaire vide
2 pour chaque  $x \in arr$  faire
3   si  $x$  est une clef de  $d$  alors  $d[x] \leftarrow d[x] + 1$ 
4   sinon  $d[x] \leftarrow 1$ 
5
6  $i \leftarrow 0$ 
7 pour chaque clef  $x$  de  $d$  dans l'ordre croissant faire
8   pour  $k$  de 0 à  $d[x]$  exclu faire
9      $arr[i] \leftarrow x$ 
10     $i \leftarrow i + 1$ 
```

Remarquez que pour conclure, cet algorithme a besoin de parcourir les clefs dans l'ordre croissant... Autrement dit, il a besoin de trier les clefs. Sans hypothèse supplémentaire sur les données, il est donc en $\Omega(n \log_2 n)$.

Cependant, dans le cas où les données sont des entiers on peut procéder ainsi :

- Commencer par trouver le min m et le max M du tableau.
- Utiliser la fonction de hachage suivante : $x \mapsto x - m$. Notez que cette fonction est croissante !
- Créer une table à $M - m + 1$ cases, et appliquer l'algorithme précédent avec.
- Pour obtenir les clefs dans l'ordre, il suffit alors de parcourir la table dans l'ordre (le hachage est croissant) !

Exemple.

FIGURE 13.8 – Tri par dénombrement

Proposition 9 (Tri par dénombrement).

Le tri par dénombrement d'un tableau de len entiers dont le min est m et le max M s'effectue en $\Theta(len + m + M)$.

Démonstration. Analyse du pseudo-code précédemment expliqué. □

Remarque. On avait pourtant prouvé qu'un tri se fait en $\Omega(n \log_2 n)$, non ? Non ! On avait prouvé qu'un tri *par comparaisons*² est en $\Omega(n \log_2 n)$ comparaisons.

Or, ce tri-ci ne fonctionne pas par comparaisons : on utilise le fait que les entrées sont des entiers pour fonctionner autrement.

0.6 Un type d'ensemble

Les tableaux associatifs permettent de facilement représenter des ensembles : les éléments qui sont dans l'ensemble sont associés à quelque chose, et ceux qui n'y sont pas ne sont pas associés.

Remarque.

- Cela permet notamment de faire des « ensemble de sommets marqués » dans un parcours ! Attention toutefois : lorsque l'on peut utiliser un tableau de booléens, cela sera (presque) toujours plus efficace.
- Dans le cas d'une implémentation par arbre... autant utiliser directement un arbre équilibré qui stocke les éléments de l'ensemble, et oublier les « quelque chose » qui jouent le rôle de valeur. Le module `Set` fournit de tels ensembles en OCaml, mais il est très hors-programme³.

2. Un tri où la seule façon de distinguer deux données est de les comparer

3. Car il est défini à l'aide d'un foncteur, c'est à dire d'une « fonction » qui prend en argument un module et renvoie un module.

1 Tas

Dans toute cette section, on se donne (\mathcal{E}, \leq) un ensemble totalement ordonné d'étiquettes.

1.0 Définition

1.0.0 Définition

Définition 10 (Tas min).

Un tas binaire min (anglais : *binaray min heap*) (resp. tas binaire max / *binary max heap*), abrégé **tas min** (resp. **tas max**), est un arbre binaire complet étiqueté par \mathcal{E} dont l'étiquette de chaque noeud est inférieure (resp. supérieure) à celle des racines de ses enfants.

Exemple.

FIGURE 13.9 – Un tas min contenant $\llbracket 0; 9 \llbracket$

Remarque.

- On peut aussi définir des tas min (resp. max) ternaires, ou k-aires en général. Dans ce cours (et dans le programme) on se limite aux tas binaires.
- **Tas et ABR ne sont pas la même notion!** Quelques différences :
 - La propriété d'ABR n'est *pas* locale (on ne peut pas juste la vérifier entre chaque noeud et les racines de ses enfants); celle de tas ci (elle est définie comme étant locale).
 - Dans un ABR, le minimum est obtenu en allant toujours à gauche, dans un tas min il est à la racine.
 - Rechercher un élément dans un ABR est rapide, pas dans un tas (doit-on aller à gauche ou à droite?).
 - Un ABR est trié par parcours infixe, un tas pas du tout (cf exemple ci-dessus).

Proposition 11 (Définition non-locale d'un tas).

Un arbre binaire A est un tas min si et seulement si pour tout noeud, l'étiquette du noeud est inférieure à l'étiquette de tous les noeuds de ses sous-arbres.

Démonstration. La réciproque est immédiate. L'implication s'obtient par transitivité de \leq le long de tout chemin descendant. \square

Remarque. La définition non-locale ne sert pas en pratique, puisque la version locale est plus simple à tester. Toutefois, c'est bon de l'avoir en tête pour prouver certaines propriétés théoriques.

Définition 12 (Rappel : arbre binaire complet dans un tableau).

On peut représenter un arbre binaire complet dans un tableau en stockant ses éléments dans l'ordre d'un parcours en largeur (de gauche à droite). Cela fait que :

- La racine est à l'indice 0.
- L'étage $p > 0$ correspond aux noeuds d'indice $\llbracket 2^p - 1; 2^{p+1} - 1 \rrbracket$
- Si i est l'indice d'un noeud, son noeud parent est à l'indice $\lfloor \frac{i-1}{2} \rfloor$.
- Si i est l'indice d'un noeud, ses (éventuels) enfants gauche et droit sont respectivement à l'indice $2 * i + 1$ et $2 * i + 2$.

Démonstration. On peut prouver ces propriétés :

- Immédiat pour la racine.
- Soit i un étage. Comme l'arbre est complet, tous les étages d'avant sont entièrement remplis : d'après le cours sur les arbres, ils contiennent donc au total :

$$\sum_{p=0}^{i-1} 2^p = 2^i - 1$$

Il y a donc au total exactement $2^i - 1$ noeuds dans les étages 0, ..., $i - 1$, qui sont ceux rangés avec les noeuds de l'étage i dans le tableau (BFS). Comme on indice à partir de 0, on obtient bien que le premier élément de l'étage i est d'indice $2^i - 1$. Comme de plus l'étage i contient au plus 2^i noeuds, ils vont au plus jusqu'à l'indice $2^{i+1} - 2$ inclus.

- Corollaire du point suivant.
- Écrivons i sous la forme $2^p - 1 + k$ (avec p la profondeur du noeud i), et supposons que le noeud correspondant a des enfants. Ce noeud a donc k adelphe avant lui sur son étage. Par BFS, les enfants de ces adelphe sont exactement les noeuds avant les enfants de i sur la rangée $p + 1$.

FIGURE 13.10 – Notations de la preuve

Ainsi, l'enfant gauche du noeud d'indice i a pour indice $2^{p+1} - 1 + 2k$. Or :

$$2 * (2^p - 1 + k) + 1 = 2^{p+1} - 1 + 2k$$

De même pour le noeud droit.

□

Proposition 13 (Tas dans un tableau).

Pour implémenter un tas, on peut stocker ses valeurs dans un tableau (comme ci-dessus). Il est utile d'anticiper en faisant un tableau trop grand, aussi on stocke à part la taille (le nombre de noeuds) du tas.

1.0.1 Implémentation en OCaml

Comme en C, on peut faire des « struct » en OCaml. On appelle cela des **enregistrements**. Par défaut les champs d'un enregistrement sont non-mutables; le mot-clef `mutable` permet de les rendre mutables^{4 5}.

Pour les tas, on peut donc procéder ainsi :

```

5 type 'a minHeap = {
6   mutable len : int;      (* Taille du tas *)
7   mutable tab : 'a array (* Tableau qui contient les éléments *)
8 }
```

 minHeap.ml

Remarque.

- Le champ `len` est mutable car on va ajouter et enlever des éléments, donc la taille du tas va évoluer.
- Le champ `tab` est mutable car on va parfois remplacer le tableau par un autre! On va en effet utiliser le même redimensionnement que les tableaux dynamiques pour « modifier » efficacement la longueur du tableau⁶.

Remarque. Une variante fréquente de la représentation des tas est de stocker la racine dans la case d'indice 1, et d'utiliser la case d'indice 0 pour stocker la taille (le nombre de noeuds) du tas.

Exercice. Quels sont les liens entre indices des enfants et des parents dans cette variante ?

1.1 Opérations

Remarque. Dans toute la suite de ce cours, on considère uniquement des tas min. Les tas max se construisent de manière symétrique.

On se donne les fonctions suivantes :

```

11 (* Renvoie l'indice du parent du noeud d'indice i *)
12 let parent i = (i-1) / 2
13
14 (* Renvoie l'indice de l'enfant gauche du noeud d'indice i *)
15 let gauche i = 2*i+1
16
17 (* Renvoie l'indice de l'enfant droite du noeud d'indice i *)
18 let droite i = 2*i+2
```

 minHeap.ml

1.1.0 Percolation vers le haut

Définition 14 (Pseudo-tas).

On appelle **pseudo-tas** (min) un arbre binaire complet dont la propriété de tas est rompue sur au plus une arête. À reprendre.

Exemple.

4. Il ne marche qu'en définissant un type enregistrement, pas sur tout et n'importe quoi...

5. Fun fact : une référence est en fait un enregistrement constituée d'un seul champ mutable!

6. Pour rappel : quand le tableau est plein, le remplacer par un deux fois plus long, quand il est à 75% vide le remplacer par un deux fois plus court.

FIGURE 13.11 – Un pseudo-tas min avec en exergue l'arête qui rompt la propriété de tas

Définition 15 (Percolation vers le haut).

La **percolation vers le haut** est une opération qui permet de « réparer » un pseudo-tas (min) en tas (min). Elle procède comme suit : tant qu'il existe un noeud plus petit que son parent, l'échanger avec son parent.

Remarque. La percolation vers le haut consiste à « pousser » un élément vers le haut, « à travers » les autres : d'où le terme de *percolation*.

On se donne `swap t i j` une fonction qui échange les cases `i` et `j` d'un tableau. Avec elle, on écrit la percolation :

```

27 (* Percolation vers le haut dans le pseudo-tas
28    [t] depuis l'indice [i] *)
29 let rec percoleHaut t i =
30   if i > 0 && t.tab.(parent i) > t.tab.(i) then begin
31     swap t.tab i (parent i);
32     percoleHaut t (parent i)
33   end

```

 minHeap.ml

Remarque. On aurait aussi pu implémenter avec une boucle `while`.

Proposition 16 (Correction de percoleHaut).

Si `t` correspond à un pseudo-tas, et que `i` est l'indice de l'enfant éventuellement plus petit que son parent, alors `percoleHaut t i` est un tas. Faux. À reprendre.

Démonstration. La correction totale de `swap` est admise⁷. Pour la correction totale de `percoleHaut` :

- La fonction termine car `i` est un variant. En effet il entier, minoré strictement par 0 (ligne 29) et décroît strictement en cas d'appel récursif (ligne 31 ; comme `i > 0` on a bien `parent i < i`).
- On montre la correction partielle par récurrence forte sur $i \in \mathbb{N}$:
 - Initialisation : Si $i = 0$, alors `i` n'a pas de parent. Il ne peut donc pas être plus petit que son parent, et le pseudo-tas est en fait un tas. De même si `t.tab.(parent i) <= t.tab.(i)`.
 - Hérédité : Soit $i > 0$ tel que la correction soit vraie jusqu'au rang i exclu, montrons-la vraie au rang i . Par hypothèse, la seule rupture possible de la propriété de tas est entre i et `parent(i)`. On suppose que `t.tab.(parent i) > t.tab.(i)` (l'autre cas a été traité en initialisation). On va alors faire un échange. Les arêtes potentiellement impactées par l'échange sont toutes celles dont une des extrémités est d'indice i ou `parent(i)` :

7. Et triviale.

(a) Avant swap

(b) Après swap

FIGURE 13.12 – Arêtes impactées par l'échange entre i et $\text{parent}(i)$

On utilise les conventions de nommage des noeuds et des arêtes de la figure ci-dessus. Avant l'échange, comme par hypothèse seule C enfrait la propriété de tas, on a :

Arête	Extrémités	Analyse
A	$pp \rightarrow p$	Valide par hypothèse, donc $pp \leq p$
B	$p \rightarrow a$	Idem, donc $p \leq a$
C	$p \rightarrow x$	Invalide : $p > x$ par hypothèse
D	$x \rightarrow y$	Idem que A, donc $x \leq y$
E	$x \rightarrow z$	Idem, donc $x \leq z$

Après l'échange :

Arête	Extrémités	Analyse
A	$pp \rightarrow x$	Peut-être invalide
B	$x \rightarrow a$	Valide car $x < p \leq a$
C	$x \rightarrow p$	Valide car $x < p$
D	$p \rightarrow y$	Prblm !!
E	$p \rightarrow z$	Prblm !!

Ainsi, après l'échange, la seule possible violation de la propriété de tas est sur l'arête A (entre l'indice $\text{parent}(i)$ et son parent). En appliquant l'hypothèse de récurrence à l'appel récursif on obtient qu'après l'appel récursif t vérifie bien la propriété de tas.

□

1.1.1 Insertion

Grâce à la percolation, on obtient un algorithme très simple pour insérer :

Définition 17 (Insertion dans un tas min).

Pour insérer dans un tas min, on ajoute le nouvel élément au premier emplacement disponible, puis on le fait percoler vers le haut pour le mettre au bon endroit.

En plus de l'implémenter, on va redimensionner le tableau du tas lorsque nécessaire. À cette fin, on se donne `Array.redim tab new_length` qui crée une copie de `tab` de longueur `new_length` (quitte à enlever la fin de `tab` ou à remplir la fin avec des valeurs en plus).

Voici donc l'insertion :

```

42  (** Insertion dans un tas binaire *)
43  let add t x =
44    (* Si besoin, doubler le tableau *)
45    if t.len = Array.length t.tab then
46      t.tab <- redim t.tab (2 * Array.length t.tab);
47
48    (* Puis insérer en percolant vers le haut *)
49    t.tab.(t.len) <- x;
50    t.len <- t.len + 1;
51    percoleHaut t (t.len-1)

```



Exemple.

(a) Début de l'insertion

(b) Percolation vers le haut

(c) Fin de l'insertion

FIGURE 13.13 – Insertion d'un noeud dans un tas

1.1.2 Percolation vers le bas

Définition 18 (Percolation vers le bas).

La **percolation vers le bas** est une opération qui permet de « réparer » un pseudo-tas (min) en tas (min). Elle procède comme suit : tant qu'il existe un noeud plus grand que son enfant, l'échanger avec le plus petit de ses enfants.

Remarque.

- La percolation vers le bas consiste à « pousser » un élément vers le bas.
- Il est important de bien échanger avec le plus *petit* des enfants :

FIGURE 13.14 – La percolation vers le bas ne répare pas si on échange avec le plus grand enfant

Voici une implémentation. C'est un peu plus long que la percolation vers le haut car il faut trouver quel est l'enfant minimal, tout en gérant les cas où il y a 0 ou 1 enfant :

```

54 (* Percolation vers le bas dans le tableau
55    [tab] depuis l'indice [i] *)
56 let rec percoleBas t i =
57   if gauche i < t.len then
58
59     (* Calculer l'indice du plus petit enfant*)
60     let i_enf_min =
61       if droite i < t.len && t.tab.(gauche i) > t.tab.(droite i)
62       then droite i
63       else gauche i
64     in
65
66     (* Si besoin, échanger et continuer de percoler*)
67     if t.tab.(i) > t.tab.(i_enf_min) then begin
68       swap t.tab i (i_enf_min);
69       percoleBas t i_enf_min
70     end

```



minHeap.ml

Proposition 19 (Correction de percoleBas).

Si t correspond est un « pseudo-pseudo-tas », c'est à dire que c'est un tas partout sauf éventuellement entre un noeud i et ses enfants, et que i est l'indice de ce noeud, alors `percoleBas t i` est un tas. Faux. À reprendre.

Remarque. Ici la notion de pseudo-tas n'est pas la bonne : il peut y avoir 2 erreurs puisqu'un noeud est mal placé par rapport à ses enfants, qui peuvent être deux. D'où « pseudo-pseudo-tas ».

Démonstration. Pour la correction totale de `percoleBas` :

- La fonction termine car i est un variant. En effet il entier, majoré strictement par $t.len$ (ligne 57) et croit strictement en cas d'appel récursif (ligne 69; puisque `gauche i` et `droite i` sont strictements supérieurs à i).
- On montre la correction partielle par récurrence forte décroissante sur $i \in \llbracket 0; t.len \rrbracket$:
 - Initialisation : Si $i = t.len$, alors i n'a pas d'enfants. Il ne peut donc pas être plus grand que ses enfants, et le pseudo-pseudo-tas est en fait un tas.
Remarquons également que par définition `i_enf_min` est l'indice du plus petit des enfants (lorsqu'il y en a). Donc si on ne rentre pas dans le `if` ligne 67, le noeud i est plus petit que son plus petit enfant, donc est bien placé et le pseudo-pseudo-tas est en fait un tas.
 - Hérédité : Soit $0 \leq i < t.len$ tel que la correction soit vraie de $t.len$ jusqu'au rang i exclu, montrons-la vraie au rang i . Par hypothèse, la seule rupture possible de la propriété de tas est entre i et ses enfants.
On procède ensuite comme dans la preuve précédente : on liste les arêtes avant et après l'échange, on conclut que les problèmes éventuels sont descendus sur les arêtes D et E, et on conclut par hypothèse de récurrence.

(a) Avant swap

(b) Après swap

FIGURE 13.15 – Arêtes impactées par l'échange entre i et i_{enf_min}

□

1.1.3 Extraction du minimum

Les tas min sont optimisés pour lire et extraire le minimum. Notons immédiatement que le minimum est à l'indice 0 (d'après la version non-locale de la définition des tas), et que donc le trouver est immédiat.

Cependant, l'enlever n'est *a priori* pas si trivial. On va utiliser la même astuce que dans les ABR : échanger l'élément à supprimer avec un élément facile à supprimer.

Définition 20 (Extraction du minimum d'un tas).

Pour extraire le minimum d'un tas, on l'échange avec le dernier élément du tas et on le supprime. Puis, on fait percoler vers le bas cet ancien dernier élément du tas qui vient d'être placé à la racine.

Cela donne le code suivant (avec redimensionnement dynamique) :

```

73  (** Extracition dans un tas binaire *)
74  let take_min t =
75    (* Si besoin, contacter le tableau *)
76    if t.len < 3 * Array.length t.tab / 4 then
77      t.tab <- redim t.tab (Array.length t.tab / 2);
78
79    let mini = t.tab.(0) in
80    swap t.tab 0 (t.len-1);
81    t.len <- t.len - 1;
82    percoleBas t 0;
83    mini

```

 minHeap.ml

Exemple.

(a) Échange entre la racine et le dernier
 (b) Percolation vers le bas
 (c) Fin de l'extraction

FIGURE 13.16 – Extraction du minimum dans un tas

1.2 Complexités

Analysons pour l'instant la complexité sans prendre en compte le redimensionnement dynamique.

Proposition 21.

L'insertion et l'extraction dans un tas binaire à n éléments se fait en temps $\Theta(\log_2 n)$.

Démonstration. Dans l'insertion et l'extraction, il y a des opérations en temps constant (dont l'éventuel `swap`), et un appel à `percoleHaut` ou `percoleBas`. Analysons la première, la seconde se fera de même.

`percoleHaut` fait à chaque appel $O(1)$ opération plus un appel sur parent i . Ainsi, i est divisé par 2 à chaque fois. Or il vaut initialement au plus $t.\text{len}-1$ (puisque c'est un indice valide), d'où une complexité en $O(\log_2 t.\text{len})$.

Pour prouver que cette borne est atteinte, on peut considérer l'insertion d'un noeud étiqueté par 0 dans le tas où les noeuds sont étiquetés par leur profondeur : il faut percoler jusqu'en haut, donc faire un appel par étage donc $\Omega(\log_2 t.\text{len})$ appels. \square

Le redimensionnement dynamique a déjà été analysé au premier semestre : il coûte $O(1)$ en amorti. On obtient donc $\Theta(\log_2 n)$ en amorti pour insérer et extraire.

1.3 Tri par tas

Les tas donnent un algorithme de tri, qui « correspond » au tri par sélection : créer le tas contenant tous les éléments, puis extraire les minimums successifs et les placer dans l'ordre :

```

86 (** Trie en place [arr] *)
87 let heapSort arr =
88   let t = {len = 0; tab = Array.make 1 tab.(0)} in
89   for i = 0 to Array.length arr - 1 do
90     add t arr.(i)
91   done;
92   for i = 0 to Array.length arr - 1 do
93     arr.(i) <- take_min t
94   done

```

 minHeap.ml

Remarque.

- C'est le seul tri explicitement au programme, vous devez donc le maîtriser sur le bout des doigts !
- Cela étant dit, les autres tris que l'on a vus sont des immenses classiques qui seront probablement considérés comme acquis par les concours.

Proposition 22 (Complexité du tri par tas).

Le tri par tas trie un tableau à n éléments en temps $\Theta(n \log_2 n)$

Démonstration. Il s'agit d'additionner les $\log_2()$ successifs : la phase d'insertion (ligne 89-90) insère à chaque itération un noeud dans un tas à i éléments. Or :

$$\sum_{i=2}^{n-1} \log_2 i = \log_2 \left(\prod_{i=2}^{n-1} i \right) = \log_2((n-1)!) = \Theta(n \log_2 n)$$

D'où le fait que la phase d'insertion soit en temps $\Theta(n \log_2 n)$. De même pour les extractions successives. \square

Remarque.

- Il existe un algorithme pour construire un tas en temps *linéaire* ! Mais l'extraction, elle, se fait toujours en temps quasi-linéaire.
- Il existe des variantes des tas qui cherchent à proposer de nouvelles opérations, implémentées (très) efficacement. On peut par exemple citer les tas binomiaux⁸ ou les tas de Fibonacci⁹

2 Files de priorité

Remarque. On fait ici des files de priorité *min* ; les *max* se définissent de manière symétrique.

2.0 Type abstrait

Définition 23.

Soit (\mathcal{K}, \leq) un ensemble totalement ordonné d'éléments appelés *priorités* et \mathcal{V} un ensemble de *valeurs*.

Une **file de priorité min** sur $\mathcal{K} \times \mathcal{V}$ est une structure de données de paires (priorité, valeur) où l'on peut insérer une nouvelle paire, trouver la paire de priorité minimale et extraire la paire de priorité minimale.

Exemple.

8. Je vous renvoie à Wikipédia.

9. Idem.

FIGURE 13.17 – Insertions et extractions dans une file de priorité

Voici le type abstrait du type file de priorité :

<u>Type</u> :	FilePrio
<u>Utilise</u> :	Prio, Valeur, Bool, Entier
<u>Opérations</u> :	$\text{file_vide} : \text{Rien} \rightarrow \text{FilePrio}$ $\text{ajoute} : \text{File_Prio} \times \text{Prio} \times \text{Valeur} \rightarrow \text{File_Prio}$ $\text{trouve_min} : \text{File_Prio} \times \text{Prio} \times \text{Valeur}$ $\text{enlève_min} : \text{File_Prio} \rightarrow \text{Prio} \times \text{Valeur}$ $\text{est_vide} : \text{File_Prio} \rightarrow \text{Bool}$ $\text{taille} : \text{File_Prio} \rightarrow \text{Entier}$

TABLE 13.3 – Signature fonctionnelle du type abstrait file de priorité

2.1 Implémentation avec des ABR équilibrés

Proposition 24.

Un arbre binaire équilibré contenant des paires (priorité, valeur) ordonnées par priorité permet d'implémenter une file de priorité efficacement.

Démonstration. Si besoin, rappelons que l'on a vu comment extraire le minimum d'un ABR en temps linéaire en la hauteur : c'est plus simple que d'extraire un élément quelconque! \square

Exemple.

FIGURE 13.18 – Un arbre rouge-noir implémentant une file de priorité

2.2 Implémentation avec des tas

Proposition 25.

Un tas contenant des paires (priorité, valeur) ordonnées par priorité permet d'implémenter une file de priorité efficacement.

Exemple.

FIGURE 13.19 – Un tas implémentant la même file de priorité

2.3 Complexités

Pour F une file de priorité, en notant $|F|$ sa taille :

Opération \ Implémentation	ARN	Tas binaire
file_vide	1	1
ajoute	$\log_2 F $	$\log_2 F $
trouve_min	$\log_2 F $	1
enleve_min	$\log_2 F $	$\log_2 F $
est_vide	1	1
taille $ F $ ou 1 (selon l'implem)	1	

TABLE 13.4 – Complexités des opérations usuelles sur les files de priorité

2.4 Une opération supplémentaire

On veut parfois pouvoir *modifier* la priorité d'un élément (notamment dans l'algorithme de Dijkstra). Une solution simple pour cela est de mémoriser un dictionnaire qui à une valeur associe une information qui permet de retrouver la « position » la structure :

- Dans un ABR on associe la priorité, qui permet de faire des recherches dans l'ABR et donc de trouver l'élément.
- Dans un tas on associe l'indice dans le tableau.

Pour modifier la priorité, on peut alors :

- Dans un ABR, supprimer le noeud et l'insérer avec la nouvelle priorité.
- Dans un tas, modifier la priorité puis percoler vers le haut ou le bas le noeud modifié.

Dans les deux cas c'est en temps logarithmique... Mais très couteux en mémoire puisqu'il faut maintenir un dictionnaire !

Chapitre 14

LOGIQUE

Notions	Commentaires
Variables propositionnelles, connecteurs logiques, arité. Formules propositionnelles, définition par induction, représentation comme un arbre. Sous-formule. Taille et hauteur d'une formule.	Notations : $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$. Les formules sont des données informatiques. On fait le lien entre les écritures d'une formule comme mot et les parcours d'arbres.
Quantificateurs universel et existentiel. Variables liées, variables libres, portée. Substitution d'une variable.	On ne soulève aucune difficulté technique sur la substitution. L'unification est hors programme.

Extrait de la section 4.5 du programme officiel de MP2I : « Syntaxe des formules logiques ».

Notions	Commentaires
Valuations, valeurs de vérité d'une formule propositionnelle. Satisfiabilité, modèle, ensemble de modèles, tautologie, antilogie. Équivalence sur les formules. Conséquence logique entre deux formules.	Notations V pour la valeur vraie, F pour la valeur fausse. Une formule est satisfiable si elle admet un modèle, tautologique si toute valuation en est un modèle. On peut être amené à ajouter à la syntaxe une formule tautologique et une formule antilogique ; elles sont en ce cas notées \top et \perp . On présente les lois de De Morgan, le tiers exclu et la décomposition de l'implication. On étend la notion à celle de conséquence ϕ d'un ensemble de formules Γ : on note $\Gamma \models \phi$. La compacité est hors programme.
Forme normale conjonctive, forme normale disjonctive. Mise sous forme normale.	Lien entre forme normale disjonctive complète et table de vérité. On peut représenter les formes normales comme des listes de listes de littéraux. Exemple de formule dont la taille des formes normales est exponentiellement plus grande.
Problème SAT, n -SAT, algorithme de Quine.	

Extrait de la section 4.6 du programme officiel de MP2I : « Sémantique de vérité du calcul propositionnel ».

SOMMAIRE

0. Syntaxe des formules logiques	319
0. Formules propositionnelles	319
1. Notations et parcours	320
2. Égalités et substitutions	321

1. Sémantique des formules propositionnelles	323
0. Évaluation d'une proposition logique	323
1. Conséquence, équivalence logique	325
2. Satisfiabilité	327
2. Formes normales.....	329
0. Définitions	329
1. Construction de DNF/CNF	330
2. Représentation informatique d'une formule	332
3. SAT et algorithme de Quine	333
0. Problème SAT	333
1. Algorithme de Quine	334
2. Une petite parenthèse : les graphes de flots	336
4. Quantificateurs et introduction à la logique du premier ordre	337

Le but de l'étude de la logique est de formaliser ce qu'est un raisonnement, et ce qu'est une preuve (MPI). *Il faut donc bien distinguer le raisonnement "méta" que l'on utilise pour formaliser et manipuler la logique, de celui qu'on définit dans le cadre de la logique (et qui est censé décrire le premier.)*

Cela a au moins deux grands intérêts :

- Garantir que ce que l'on prouve est correct, autrement dit garantir que l'on peut correctement déduire la véracité de la conclusion à partir de la véracité du résultat.
- Automatiser la recherche de preuve, ou la vérification de la correction d'une preuve.

En MP2I, on se concentrera surtout sur la formalisation de ce qu'est une *formule propositionnelle*, et sur les liens logiques entre différentes propositions.

En MPI, vous verrez comment prouver forme qu'une proposition est vraie (d'une manière plus efficace et intuitive que le retour sur trace). Ce faisant, vous formaliserez en fait ce qu'est une *preuve* !

0 Syntaxe des formules logiques

0.0 Formules propositionnelles

Intuitivement, une formule propositionnelle est « une propriété mathématique sans quantificateurs ». Formalisons cette notion par induction :

Définition 1 (Formules propositionnelles).

Soit \mathcal{V} un ensemble infini dénombrable dont les éléments sont appelés variables propositionnelles. On notera souvent ces variables x, y, \dots ou encore x_0, x_1, \dots .

L'ensemble \mathcal{P} des **formules propositionnelles** est défini inductivement par :

- $\top, \perp \in \mathcal{P}$
- $\mathcal{V} \subseteq \mathcal{P}$
- si $P, Q \in \mathcal{P}$, alors :
 - $\neg P \in \mathcal{P}$
 - $P \wedge Q \in \mathcal{P}$
 - $P \vee Q \in \mathcal{P}$
 - $P \rightarrow Q \in \mathcal{P}$
 - $P \leftrightarrow Q \in \mathcal{P}$

On note souvent les propriétés logiques P, Q, \dots .

Les symboles utilisés ci-dessus sont :

- \top, \perp (**top, bot (ou bottom)**) : constantes.
- \neg : **non** - négation : connecteur unaire.
- \wedge : **et** - conjonction : connecteur binaire.
- \vee : **ou** - disjonction : connecteur binaire.
- \rightarrow : **implique** - implication : connecteur binaire.
- \leftrightarrow : **équivalent à** - équivalence : connecteur binaire.

Remarque.

- On peut aussi choisir de ne pas mettre les constantes \top et \perp dans la syntaxe des formules propositionnelles, car on pourra en retrouver des formulations équivalentes en utilisant uniquement les autres symboles.
- On appelle aussi une formule propositionnelle une **proposition**. Dans ce contexte, une proposition n'a pas de quantificateurs (ce ne sont pas les "propositions" que vous écrivez tout le temps dans vos cours de maths)!
- ⚠ On ne dit pas pour l'instant ce que *veut dire* $\neg P$, ou encore $P \vee Q$. Les connecteurs n'ont *pas de sens en soi*, ce sont juste des symboles. Nous avons uniquement donné les *règles d'écritures* qui permettent d'écrire une formule propositionnelle, mais n'avons rien dit de comment *comprendre* une telle formule.

Autrement dit, nous avons spécifiée la **syntaxe** mais pas la **sémantique**.

- Nous avons fait pareil avec les types abstraits. La signature d'un type abstrait définit uniquement sa syntaxe ; et nous avons vu qu'il fallait ensuite l'accompagner d'axiomes, de spécifications des fonctions, pour donner du sens.

Proposition 2.

De manière équivalente, on peut représenter une formule par un arbre dont les noeuds internes sont étiquetés par des connecteurs, et les feuilles par des variables propositionnelles ou des constantes. Un noeud étiqueté par un connecteur unaire est d'arité 1, binaire d'arité 2.

Exemple.

On pourrait formaliser la notion d'arbre associé à une formule et prouver une bijection entre les arbres et les formules¹. Toutefois, cela n'a que peu d'intérêt logique² puisqu'il s'agit simplement d'interpréter la définition inductive de proposition comme une définition d'arbre étiqueté par les connecteurs.

Type des formules propositionnelles (aussi appelées "propositions") en OCaml :

```
1 type prop =
2   | Const of bool (*top ou bot*)
3   | Var of int (*l'entier i représente la variable x_i*)
4   | Non of prop
5   | Et of prop * prop
6   | Ou of prop * prop
7   | Implique of prop * prop
8   | Equivaut of prop * prop
```



Remarque. on reconnaît une définition d'arbre étiquetés par les constructeurs, et c'est normal !

Exemple. avec ce type OCaml, l'arbre précédent s'écrirait :

Définition 3.

Soit $P \in \mathcal{P}$ une formule propositionnelle. On appelle :

- hauteur de P la hauteur de l'arbre associé à P .
- taille de P le nombre noeuds (internes ou non) de l'arbre.
- sous-formule de P un sous-arbre de P (vu comme une formule).

0.1 Notations et parcours

À quel parcours de l'arbre correspond la formule propositionnelle donnée en exemple ? **Réponse :**

Et dans le langage OCaml ? **Réponse :**

Écrire cette formule sous forme postfixe. **Réponse :**

La notation usuelle des formules propositionnelles est la notation infixe. Il faut donc des parenthèses pour lever les ambiguïtés. Sans parenthèses ni règle d'écriture, une écriture sous forme infixe pourrait correspondre à plusieurs propositions différentes.

1. Vous pouvez le faire en exercice si vous voulez.

2. Mais a des intérêts en tant qu'exercice : vous faire manipuler les définitions et les inductions.

Exemple. $x \wedge y \vee z \wedge a$ peut correspondre à :

Remarque.

- Si on connaît l'arité de chaque noeud (c'est le cas ici), les écritures préfixes et postfixes correspondent à un unique arbre possible (on a une bijection). Il n'y aurait donc pas d'ambiguïté, même sans parenthèses. Ce résultat est connu sous le nom de « lemme de lecture unique »³.
- Malgré cela, on privilégie généralement la notation infixe, munie de règles de priorité et éventuellement de parenthèses (comme en maths !). C'est la version la plus lisible, même si elle a besoin de parenthèses.

Définition 4 (Priorités logiques).

On utilise les règles de priorités (donc de parenthésages implicites) suivantes pour les connecteurs :

- \neg est prioritaire sur \wedge et \vee
- \wedge est prioritaire sur \vee .

Remarque. Il est fortement recommandé d'écrire quand même les parenthèses entre les \vee et les \wedge , par souci de clarté. **Et je vous demande de le faire.**

Exemple. Donner l'arbre associé à $(\neg\neg x \wedge y) \vee (\neg z \wedge x)$

Notations : On écrit parfois :

- \bar{x} pour $\neg x$
- $x.y$ pour $x \wedge y$
- $x + y$ pour $x \vee y$

Cependant, ces notations sont souvent réservées à des variables propositionnelles, et je ne vous les recommande vraiment pas, notamment pour des raisons de distributivité qu'on verra plus tard (le "+" sera distributif sur le "×" en plus de l'inverse... ce qui est très perturbant!).

0.2 Égalités et substitutions

Définition 5 (Égalité syntaxique).

On dit que deux propositions P et Q sont **syntactiquement égales** si leurs arbres associés sont égaux, i.e. si « elles s'écrivent pareil ».

Remarque.

- Le cas des parenthèses est fourbe, car il mène à des écritures « différentes » alors qu'il s'agit de la même formule et du même arbre. Par exemple : $((x \wedge y) \vee z)$ et $(x \wedge y) \vee z$ sont syntactiquement égaux, bien qu'ils semblent avoir une écriture « différente ».
- Ces parenthèses sont la seule difficulté de cette définition.
- Attention à ne pas aller trop vite : ceci est une égalité syntaxique et non sémantique. Autrement dit, $P \rightarrow Q \wedge Q \rightarrow P$ **n'est pas syntactiquement égal** à $P \leftrightarrow Q$. Notre intuition nous dit que ces formules ont la même *signification*, mais ce sont bien deux formules distinctes syntactiquement.

3. On l'a mentionné rapidement dans la partie hors-programme du cours sur les arbres.

Définition 6 (Extension de l'égalité syntaxique).

On étend l'égalité syntaxique en lui ajoutant les règles suivantes :

- \wedge est commutatif syntaxiquement : $P \wedge Q$ est syntaxiquement égal à $Q \wedge P$. De même pour \vee et \leftrightarrow .
- \wedge est associatif syntaxiquement : $P \wedge (Q \wedge R)$ est syntaxiquement égal à $(P \wedge Q) \wedge R$. De même pour \vee et \leftrightarrow .

Autrement dit, on considère que les enfants d'un noeud de l'arbre ne sont pas ordonnés, et on considère des noeuds \vee et \wedge d'arité quelconque. On notera $\bigvee_{i=0}^n P_i$ pour $P_0 \vee P_1 \vee \dots \vee P_n$. De même pour \wedge .

Exemple.

Remarque. L'égalité syntaxe non-étendue est simple à tester (on parcourt les deux arbres pour vérifier leur égalité). L'égalité syntaxique étendue par associativité et commutativité est plus difficile à tester⁴.

Définition 7 (Substitution).

Soient $P, Q \in \mathcal{P}$, soit $x \in \mathcal{V}$.

La **substitution de x par Q dans P** , notée $P[Q/x]$, est la formule obtenue en remplaçant dans P toutes les occurrences de x par Q .

Formellement, on la définit inductivement :

- $\perp[Q/x] = \perp$, $\top[Q/x] = \top$
- $x[Q/x] = Q$
- si $y \in \mathcal{V} \setminus \{x\}$, $y[Q/x] = y$
- $(\neg P)[Q/x] = \neg(P[Q/x])$
- $(P_1 \wedge P_2)[Q/x] = (P_1[Q/x]) \wedge (P_2[Q/x])$. De même pour \vee , \rightarrow et \leftrightarrow

Remarque.

- Vision arborescente : substituer x par Q revient à remplacer toutes les feuilles étiquetées par x par une copie de l'arbre de Q .
- En toute généralité, la substitution n'est pas commutative : $P[Q/x][R/y] \neq P[R/y][Q/x]$.

Exercice. Trouver une condition nécessaire pour qu'elle le soit. **Réponse :**

4. Et ne parlons pas d'une égalité sémantique : vérifier l'égalité sémantique de formules quantifiées est très littéralement un métier que l'on nomme mathématicien.

1 Sémantique des formules propositionnelles

Nous allons maintenant donner du *sens* aux règles d'écritures définies précédemment.

1.0 Évaluation d'une proposition logique


Dans cette partie, on note $\mathbb{B} = \{\text{Vrai}; \text{Faux}\}$ (abrégiés en V et F) ou de manière équivalente $\mathbb{B} = \{1; 0\}$, l'ensemble des booléens.

Définition 8 (Fonctions logiques).

On définit les fonctions *Non*, *Et*, *Ou*, *Implique*, *Equivaut* par leurs tables de vérité :

x	y	Non(x)	Et(x,y)	Ou(x,y)	Implique(x,y)	Equivaut(x,y)
V	V					
V	F					
F	V					
F	F					

Remarque.

-  Ne pas confondre ces **fonctions** avec les **symboles** de connecteurs logiques $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$.
- *Aide à l'intuition* Notez bien que $\text{Implique}(F, F) = V$, autrement dit "faux implique faux" est vrai. Si ça peut paraître contre-intuitif au départ, c'est pourtant bien le cas, et on veut que ce soit le cas. Voici un exemple pour vous en convaincre :

Il vous semblera tout à fait normal (et même évident) de dire que : si n est un multiple de 4, alors n est pair est **vrai** sur l'ensemble des entiers.

Mais alors, vous conviendrez que vous acceptez comme vraies les propositions suivantes :

- Si 4 est un multiple de 4, alors 4 est pair.
- Si 2 est un multiple de 4, alors 2 est pair.
- Si 1 est un multiple de 4, alors 1 est pair.

On voit ici grâce à ce "théorème" les 3 cas possibles de l'implication (qui rendent l'implication vraie, donc) :

- Vrai implique vrai.
- Faux implique vrai.
- Faux implique faux.

Définition 9 (Valuation).

Une **valuation** est une application de l'ensemble des variables propositionnelles \mathcal{V} dans \mathbb{B} . On la note souvent $v : \mathcal{V} \rightarrow \mathbb{B}$

Remarque. On parle aussi de distribution de vérité. Cela revient à choisir pour chaque variable si elle est vraie ou fausse.

Définition 10 (Modèle (MP2I)).

Une **interprétation**, aussi appelée un **modèle**, d'une formule propositionnelle est une valuation des variables propositionnelles de cette formule.

Remarque. Cette définition de modèle est un cas restreint d'une définition plus large que vous verrez en MPI.

Exemple. Si l'on se restreint à 3 variables x, y, z , alors la fonction v suivante est un modèle de la formule $x \vee y \wedge z$:

$$\begin{aligned} v : \mathcal{V} &\rightarrow \mathbf{B} \\ x &\mapsto F \\ y &\mapsto V \\ z &\mapsto V \end{aligned}$$

Définition 11 (Évaluation).

Soit v une valuation. On définit $eval_v : \mathcal{P} \rightarrow \mathbf{B}$ l'application qui à une formule P associe son évaluation par la valuation v ainsi :

- $eval_v(\top) = V, \quad eval_v(\perp) = F$
- Pour tout $x \in \mathcal{V}, eval_v(x) = v(x)$
- $eval_v(\neg P) = Non(eval_v(P))$
- $eval_v(P \wedge Q) = Et(eval_v(P), eval_v(Q)).$
- $eval_v(P \vee Q) = Ou(eval_v(P), eval_v(Q)).$
- $eval_v(P \rightarrow Q) = Implique(eval_v(P), eval_v(Q)).$
- $eval_v(P \leftrightarrow Q) = Equivaut(eval_v(P), eval_v(Q)).$

Exemple. l'évaluation de la proposition exemple par la valuation précédente donne :

Remarque.

- On note aussi e_v pour $eval_v$, et on parle d'évaluation de P **dans le contexte de v** .
- \mathcal{V} est infini : impossible à programmer, pas pratique pour les exemples. On se limite en général à définir une valuation pour les variables qui apparaissent dans les formules que l'on considère. C'est d'autant plus courant lorsque l'on décrit un modèle d'une formule (comme dans l'exemple précédent de modèle).
- On note $\mathcal{V}(P)$ l'ensemble des variables apparaissant dans P . C'est un ensemble fini puisqu'une formule l'est.
- Si *Vrai*, *Faux* = 1, 0 et plus généralement si $V > F$, alors l'évaluation d'un \wedge est un *min* et celle de \vee un *max*.

Exemple. Vous parlez à trois personnes A, B et C. On note x_A le fait que A dit la vérité, x_B B dit la vérité, x_C C dit la vérité. On peut alors écrire E , une formule représentant le fait qu'au moins l'un d'entre eux dit la vérité. **Réponse :**

Une valuation est ici une façon de répartir qui dit la vérité et qui ment. Si on veut satisfaire la proposition, i.e. assurer qu'au moins une personne dit la vérité, alors voici les valuations qui conviennent :

(Réponse :)

Définition 12 (Satisfaction).

On dit qu'une valuation v satisfait une formule P si $eval_v(P) = Vrai$.

Exemple. Les valuations indiquées à l'exemple précédent satisfont E . On dit aussi qu'elles **valident** E .

1.1 Conséquence, équivalence logique

Maintenant que l'on a de la sémantique, on peut créer des relations entre différentes formules, au delà de la seule égalité syntaxique.

Définition 13 (Conséquence, équivalence).

Soient $P, Q \in \mathcal{P}$ deux formules propositionnelles.

- On dit que Q est une **conséquence logique** de P , et on note $P \models Q$, si toute valuation satisfaisant P satisfait Q .
- Si $\Gamma = \{P_1, \dots, P_n\}$ est un ensemble de formules propositionnelles, on dit que $\Gamma \models Q$ si toute valuation satisfaisant toutes les formules de Γ satisfait également Q . On écrit $\models P$ pour $\emptyset \models P$.
- P et Q sont dits **sémantiquement équivalentes** si $P \models Q$ et $Q \models P$. On note $P \equiv Q$.

\models se prononce « satisfait ».

Exemple :

Remarque.

- Autrement dit, deux formules sont sémantiquement équivalentes si et seulement si elles sont satisfaites par exactement les mêmes valuations.
- Si $P \equiv Q$, on dit aussi que P et Q sont **logiquement équivalentes** ou **tautologiquement équivalentes**.
- ⚠ L'équivalence logique n'est **pas** l'égalité syntaxique. Contre-exemple :

$$P \models Q \text{ ssi } (P \rightarrow Q) \equiv \top \text{ ssi } \models P \rightarrow Q.$$

$$P \equiv Q \text{ ssi } (P \leftrightarrow Q) \equiv \top \text{ ssi } \models P \leftrightarrow Q.$$

Preuve :

Proposition 14.

Si $P \equiv Q$ alors pour toute variable $x \in \mathcal{V}$ et toute formule $R \in \mathcal{P}$, on a $P[R/x] \equiv Q[R/x]$.

Démonstration. Soient $P, Q \in \mathcal{P}$ telles que $P \equiv Q$. Soient $R \in \mathcal{P}$ et $x \in \mathcal{V}$. Soit v une valuation quelconque. Il faut montrer que $eval_v(P[R/x]) = eval_v(Q[R/x])$.

Considérons donc la valuation v' définie par :

$$\begin{cases} v'(x) = eval_v(R) \\ v'(y) = v(y) \text{ quand } y \neq x \end{cases}$$

Comme $P \equiv Q$, v' étant une valuation, on a $eval_{v'}(P) = eval_{v'}(Q)$.

Il suffit donc de montrer que $eval_{v'}(P) = eval_v(P[R/x])$ et de même pour Q . On montrera uniquement la propriété pour P , celle pour Q se fera de même. On procède par induction structurale sur $P \in \mathcal{P}$.

- Si $P = \top$ ou \perp , c'est immédiat.
- Si $P = y$ où $y \in \mathcal{V} \setminus \{x\}$: alors $eval_{v'}(P) = eval_v P$ et $P[R/x] = P$, d'où l'égalité.
- Si $P = x$, alors $eval_{v'}(P) = v'(x) = eval_v(R)$. Or, dans ce cas-ci $P[R/x] = R$ et donc $eval_v(P[R/x]) = eval_v(R)$ d'où l'égalité.
- Si $P = P_1 \wedge P_2$, alors :

$$\begin{aligned} eval_{v'}(P) &= Et(eval_{v'}(P_1), eval_{v'}(P_2)) \\ &= Et(eval_v(P_1[R/x]), eval_v(P_2[R/x])) && \text{par H.I.} \\ &= eval_v(P_1[R/x] \wedge P_2[R/x]) && \text{par def de } eval_v \\ &= eval_v(P[R/x]) && \text{par def de la substitution} \end{aligned}$$

- Les cas des autres connecteurs logiques se traitent de même.

La propriété pour Q se fait de même. Comme annoncé, on en déduit que $eval_v(P[R/x]) = eval_v(Q[R/x])$. \square

Proposition 15.

Si $P \equiv P'$ et $Q \equiv Q'$, alors on a les équivalences sémantiques suivantes :

$$P \wedge Q \equiv P' \wedge Q', \quad P \vee Q \equiv P' \vee Q', \quad \neg P \equiv \neg P', \quad P \rightarrow Q \equiv P' \rightarrow Q' \quad \text{et} \quad P \leftrightarrow Q \equiv P' \leftrightarrow Q'.$$

Démonstration. On prouvera uniquement la première équivalence sémantique, les autres se faisant de même.

Soient donc P, Q, P', Q' tels que $P \equiv P'$ et $Q \equiv Q'$ et prouvons $P \wedge Q \equiv P' \wedge Q'$. Soit donc v une valuation. On a :

$$\begin{aligned} eval_v P' \wedge Q' &= Et(eval_v(P'), eval_v(Q')) \\ &= Et(eval_v(P), eval_v(Q)) \\ &= eval_v(P \wedge Q) \end{aligned}$$

□

Exercice. prouvez les autres cas.

1.2 Satisfiabilité

Définition 16 (Tautologie).

Une **tautologie** est une proposition $P \in \mathcal{P}$ telle que pour toute valuation v , $eval_v(P) = Vrai$. Autrement dit, $\models P$.

Proposition 17.

Une proposition P est une tautologie ssi $P \equiv \top$

Démonstration. Il suffit d'utiliser que pour toute valuation v , $eval_v(\top) = Vrai$.

□

Définition 18 (Satisfiabilité).

Une proposition P est dite **satisfiable** s'il existe une valuation qui la satisfait. On appelle alors une telle valuation un **témoin de satisfiabilité** de P .

Remarque. une formule est donc satisfiable ssi (au moins) une des lignes de sa table de vérité donne Vrai ; et une tautologie ssi toutes ces lignes donnent Vrai.

Définition 19 (Antilogie).

Une **antilogie** (ou contradiction) est une proposition non satisfiable.

Remarque. autrement dit, P est une antilogie ssi il n'existe aucune valuation la satisfaisant ssi toutes les lignes de sa table de vérité donnent Faux.

Proposition 20.

P est une antilogie ssi $P \equiv \perp$

Démonstration. Il suffit d'utiliser que pour toute valuation v , $eval_v(\perp) = Faux$.

□

Proposition 21 (Équivalences sémantiques usuelles).

Soient P, Q, R des propositions. Alors :

- $P \wedge P \equiv P$ (idempotence de \wedge)
- $P \vee P \equiv P$ (idempotence de \vee)
- $\neg\neg P \equiv P$ (double négation)
- $P \vee \neg P \equiv \top$ (tiers exclu)
- $P \wedge \neg P \equiv \perp$ (contradiction)
- $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$ (loi de Morgan)
- $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ (loi de Morgan)
- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$ (distributivité de \wedge sur \vee)
- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ (distributivité de \vee sur \wedge)

Démonstration. Toutes se font de la même manière. Faisons la première loi de Morgan ensemble : \square

Exercice. : faire le tiers exclu et une des loi de distributivité

Remarque.

- Contrairement à ce à quoi vous êtes peut-être habitués sur les anneaux / espaces vectoriels, ici les deux lois sont distributives l'une sur l'autre.
- **Aucune de ces règles n'est une égalité syntaxique**
- \triangle La double négation et le tiers exclu ne sont valables qu'en "logique classique", dont on explore une petite facette dans ce chapitre (un cas restreint très particulier). Il n'est pas toujours automatiquement vérifié comme ici, et on ne le supposera pas toujours acquis. Dans des cadres logiques plus vaste et plus complets, comme la "logique intuitionniste", on peut très bien par exemple avoir des "propriétés" qui ne sont ni vraies ni fausses. Notamment, la logique intuitionniste interdit le raisonnement "par l'absurde".
- Astuce pratique : vous êtes assez mauvais pour appliquer des distributivités avec les symboles \wedge et \vee . Par contre, vous êtes *excellents* pour appliquer dans \mathbb{R} la distributivité de \times sur $+$. Donc pour appliquer la distributivité que vous voulez, **au brouillon**, remplacez le symbole que vous voulez par \times et l'autre par $+$.

2 Formes normales

2.0 Définitions

Définition 22 (Littéral).

Un **littéral** est une formule composée uniquement d'une variable ou de la négation d'une variable (« x », « $\neg y$ », etc).

Une **clause disjonctive** (resp. **clause conjonctive**) est une disjonction de littéraux (resp. une conjonction de littéraux).

Exemple.

- Voici une clause disjonctive :
- Voici une clause conjonctive :

Remarque. On peut former 2^k clauses disjonctives (resp conjonctives) contenant k littéraux et où chaque variable apparaît une et une seule fois. Pour chaque variable, il faut et il suffit de choisir si elle apparaît positivement ou négatif.

Exemple. avec deux variables a et b :

Définition 23 (Disjunctive / Conjunctive Normal Form).

On appelle :

- **forme normale disjonctive** d'une proposition P (*Disjunctive Normal Form*, ou DNF en anglais) une disjonction de clauses conjonctives logiquement équivalente à P .
- **forme normale conjonctive** d'une proposition P (*Conjunctive Normal Form*, ou CNF en anglais) une conjonction de clauses disjonctives logiquement équivalente à P .

Exemple. Soit $P = (a \vee (c \wedge d)) \wedge (b \vee (c \wedge d))$.

- Une DNF de P est :
- Une CNF de P est :

Remarque.

- Une DNF est « un gros OU (de ET) ».
- Une CNF est « un gros ET (de OU) ».
- On impose parfois⁵ que dans une clause chaque variable apparaisse au plus une fois. En effet, « $x \wedge \neg x$ » n'est pas une clause très pertinente... idem avec un \vee .

5. Et parfois non. Corollaire : bien lire les énoncés !

Convention 24 (Clause vide).

Une disjonction vide est égale à \perp (neutre de \vee pour les évaluations).

$$\bigvee_{x \in \emptyset} x = \perp$$

Une conjonction vide est égale à \top :

$$\bigwedge_{x \in \emptyset} x = \top$$

Ce sont les seules DNF/CNF composées de 0 clauses.

2.1 Construction de DNF/CNF

Théorème 25.

Toute formule propositionnelle admet une DNF et une CNF.

Démonstration. Soit $P \in \mathcal{P}$. Notons x_0, \dots, x_{n-1} ses variables propositionnelles.

- Construisons une DNF pour P .
On trace la table de vérité de P :

Chacune des 2^n lignes correspond exactement à une clause conjonctive contenant chacune des x_i exactement une fois, et donc exactement à une valuation sur les (x_i) . Par exemple, $\neg x_0 \wedge \neg x_1 \wedge \dots \wedge \neg x_{n-1}$ est la première ligne, $x_0 \wedge \neg x_1 \wedge \dots \wedge \neg x_{n-1}$ la seconde, etc etc.

On note C_i la clause correspondante à la i ème ligne. On sélectionne les clauses C_i sur lesquelles P s'évalue à Vrai selon la table :

$$P \equiv \bigvee_{i \text{ tq ligne } i \text{ met } P \text{ vrai}} C_i$$

On a bien trouvé une DNF pour P .

- Construisons maintenant une CNF.

L'idée est de nier une DNF et d'utiliser les lois de Morgan. D'après le point précédent, on sait que $\neg P$ possède une DNF à r clauses possédant chacune n littéraux nommés l_j :

$$\neg P \equiv \bigvee_{i=0}^{r-1} C_i \equiv \bigvee_{i=0}^{r-1} \bigwedge_{j=0}^{n-1} l_j$$

Mais alors, par double négation et lois de Morgan :

$$P \equiv \neg \neg P \equiv \bigwedge_{i=0}^{r-1} \bigvee_{j=0}^{n-1} \neg l_j$$

On a bien trouvé une CNF pour P .



Exemple. Trouver une DNF de la formule suivante :

$$\varphi = (x \wedge \neg z) \rightarrow (\neg x \wedge \neg(y \wedge \neg z))$$

Pour trouver une DNF, traçons la table de vérité de la formule :

x	y	z	$x \wedge \neg z$	$y \wedge \neg z$	$(\neg x \wedge \neg(y \wedge \neg z))$	φ
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Remarque.

- La lecture du tableau peut aussi permettre d'obtenir une CNF.
- Avec cette façon de construire, chaque clause contient *exactement* une fois chaque variable apparaissant dans P . On parle alors de DNF/CNF canonique et elle est unique à l'ordre des clauses près.
- L'arbre d'une telle DNF/CNF est de hauteur au plus trois... mais peut-être très gros car la racine peut avoir jusqu'à 2^n fils, et ses enfants peuvent avoir n fils (si chaque littéral apparaît au plus une fois).

- La taille d'une DNF/CNF peut-être exponentiellement grande par rapport à la formule initiale.
- Une formule peut avoir une DNF très courte et une CNF très longue ou inversement.

Exemple.

Mettre sous CNF la formule propositionnelle (en DNF) suivante :

$$\varphi = (x_1 \wedge x'_1) \vee (x_2 \wedge x'_2) \vee \cdots \vee (x_n \wedge x'_n)$$

2.2 Représentation informatique d'une formule

On a vu comment représenter une formule quelconque en OCaml (sous forme d'un arbre, avec un type somme). On peut faire plus efficace en exploitant le format d'une CNF ou d'une DNF. On adapte nos représentations comme suit :

- **Négation** : On représente la variable x_i par l'entier naturel i , et sa négation (littéral $\neg x_i$) par l'entier négatif $-i$.
- **Clause** : Dans une clause, on sait que les littéraux sont tous séparés par le même symbole \wedge (pour une clause conjonctive) ou \vee (pour une clause disjonctive). Il est donc inutile de les stocker !
On représente une clause $C_i = (l_1 \wedge l_2 \wedge \dots \wedge l_r)$ comme une liste de littéraux $[l_1; l_2; \dots; l_r]$.
- **formule** : Il faut indiquer si la formule est en CNF ou en DNF, et donner chacune de ses clauses (elles sont alors séparées par un même symbole \vee ou \wedge comme précédemment). On a (entre autres) deux possibilités : soit on stocke la formule et son format ensemble, soit on les stocke séparément.

```
1 type literal = int (* i si x_i, -i si not(x_i) *)
2 type clause = literal list
3 type formule = CNF of clause list | DNF of clause list
```



Remarque. On peut aussi vouloir retenir à part le nombre de variables (pour y accéder en $O(1)$), et/ou stocker à part les clauses et l'information DNF/CNF. Dans ce cas, on peut par exemple utiliser un type enregistrement :

```
1 type forme = CNF | DNF
2 type formule = {forme : forme; clauses : clause list; nbvars : int}
```



On pourrait aussi utiliser : `type formule = DNF of (int * clause list) | CNF of ...`

3 SAT et algorithme de Quine

3.0 Problème SAT

Définition 26 (Problème SAT).

On définit le problème de décision SAT :

- **Entrée** : $\varphi \in \mathcal{P}$ une formule propositionnelle.
- **Question** : φ est-elle satisfiable ?

Et ses variantes :

- CNF-SAT idem, mais φ est en CNF.
- k -SAT idem, mais φ est une CNF dont chaque clause contient au plus k littéraux.

Remarque.

- SAT est un problème **central** non seulement en théorie de la complexité mais aussi en pratique. Énormément de problèmes s'encodent assez facilement comme des problèmes SAT ; comme par exemple le Sudoku ou la coloration de graphes (cf TD).
- SAT est un problème a priori « difficile à résoudre », c'est à dire que l'on pense qu'il n'existe pas d'algorithme polynomial pour le résoudre⁶. Rendez-vous en MPI !
- Une autre partie de l'importance de SAT provient du fait qu'un programme n'est jamais qu'une suite d'opérations logiques sur des bits. Étudier SAT revient ainsi à étudier ce qu'un ordinateur peut calculer.

Voici des sous-problèmes classiques de SAT :

- CNF-SAT et les k -SAT sont des restrictions de SAT à des sous-ensembles de formules. Donc si on sait résoudre SAT, on sait automatiquement résoudre CNF-SAT et chacun des k -SAT avec la même complexité que SAT. (On dit que ces problèmes sont *plus faciles* à résoudre que SAT.)
- CNF-SAT est en réalité un problème aussi difficile à résoudre que SAT.
- Si $k \geq 3$, alors k -SAT est un problème aussi difficile que SAT (cf MPI).
- Si $k \leq 2$, on obtient un problème que l'on peut résoudre en temps linéaire (cf TD) à l'aide de théorie des graphes et de tris topologiques !

En revanche :

- DNF-SAT, le problème SAT restreint à des formules sous DNF est trivial à résoudre. En effet, une formule en DNF est satisfiable SSI une (au moins) de ses clauses l'est. On peut donc les traiter indépendamment. Pour chaque clause (conjonctive), on détermine linéairement si elle est satisfiable : il faut et il suffit qu'il existe une valuation qui mette chaque littéral à vrai. (On vérifie qu'on n'a pas un littéral et sa négation dans la même clause en temps linéaire...)

Ceci n'est pas incompatible avec le fait que SAT soit "difficile à résoudre". En effet, chaque formule peut effectivement être mise en DNF, mais cette DNF aura une taille exponentielle par rapport à la formule initiale en général. Cette façon de faire nous donne donc bien une solution exponentielle (en la taille de la formule).

- 1-SAT peut de même trivialement être résolu en temps linéaire (on obtient juste une conjonction de variables).
- 2-SAT peut être résolu en temps linéaire (cf TD et MPI), en se ramenant au problème de calculs de composantes fortement connexes.

6. Et si vous en trouvez un, vous avez prouvé $P \neq NP$ et gagné 1 million de dollars... mais plus probablement vous avez fait une erreur.

Définition 27 (MAX-SAT).

- *Entrée* : une formule propositionnelle φ en CNF.
- *Tâche* : Déterminer le nombre maximal de clauses que l'on peut satisfaire simultanément dans φ .

Remarque. MAX-SAT n'a aucun sens si φ n'est pas en CNF !

3.1 Algorithme de Quine

Remarque. Une solution serait d'explorer exhaustivement les 2^n valuations possibles. Ce n'est pas utilisable en pratique, par exemple pour $n \geq 100$.

En revanche, d'énormes efforts ont été fournis ces dernières décennies pour développer des SAT-solvers capables de résoudre en temps raisonnable des instances "typiques" à plusieurs centaines de milliers de variables. Bien qu'ils restent exponentiels en pire cas, ils ont donc de nombreuses applications pratiques.

On va voir un exemple (basique) de SAT-solver légèrement optimisé : l'algorithme de Quine.

Définition 28 (Arbre de décision).

Un **arbre de décision** est un arbre (parfois même un graphe) dont les arêtes sont étiquetées par des « choix » menant à différents noeuds.

Exemple.

Principe de l'algorithme de Quine : On construit progressivement un arbre de décision marqué par les choix de valeur logique (vrai/faux) donnée à chaque variable. La spécificité de cet algorithme est sa manière d'élaguer cet arbre de décision. Après chaque choix de valeur logique, on simplifie la formule en l'évaluant partiellement, de la façon suivante :

- Si on est ramenés à une formule contenant uniquement \top , on s'arrête : la formule est satisfiable. Notez que cela peut avoir lieu *bien* avant d'avoir donné une valeur à toutes les variables. Ex : $x_0 \vee (x_1 \wedge x_2 \wedge \dots \wedge x_{n-1})$.
- Si on est ramenés à \perp , on s'arrête : la branche en cours de l'arbre n'aboutira jamais à une satisfaction de la formule. L'algorithme va alors essayer d'autres choix.
- **On simplifie à chaque étape en appliquant les règles suivantes :**

Proposition 29 (Simplif. de Quine).

- $\varphi \wedge \perp \equiv \perp \wedge \varphi \equiv \perp$
- $\varphi \wedge \top \equiv \top \wedge \varphi \equiv \varphi$
- $\varphi \vee \perp \equiv \perp \vee \varphi \equiv \varphi$
- $\varphi \vee \top \equiv \top \vee \varphi \equiv \top$
- $\neg \perp \equiv \top$
- $\neg \top \equiv \perp$
- $\varphi \rightarrow \top \equiv \perp \rightarrow \varphi \equiv \top$
- $\top \rightarrow \varphi \equiv \varphi$
- $\varphi \rightarrow \perp \equiv \neg \varphi$

Exemple. Faire tourner l'algorithme de Quine sur la formule suivante :

$$\varphi = (x \wedge \neg z) \rightarrow (x \wedge \neg(y \wedge \neg z))$$

Tracer bien l'arbre de décision au fur et à mesure à côté !

FIGURE 14.1 – Élagage d'un arbre de décision par l'algorithme de Quine

L'algorithme correspond au pseudo-code suivant :

Algorithme 22 : QUINE

Entrées : φ une formule propositionnelle
Sorties : Vrai si φ est satisfiable ; Faux sinon

// Étape essentielle (qui accélère et élague!)

- 1 Simplifier φ
- 2 **si** $\varphi = \top$ **alors renvoyer** Vrai
- 3 **si** $\varphi = \perp$ **alors renvoyer** Faux

// Sinon : essayer une possibilité, puis si besoin l'autre

- 4 $x \leftarrow$ une variable présente dans φ
- 5 **renvoyer** QUINE($\varphi[\perp/x]$) || QUINE($\varphi[\top/x]$)

Remarque.

- Dans le pseudo-code ci-dessus, on suppose disposer du ou logique noté `||` de C/OCaml : il est paresseux, et donc évalue d'abord son opérande gauche et ensuite seulement si nécessaire son opérande droit.
- C'est un algorithme de retour sur trace !
- La notion d'arbre de décision sera revue en MPI, dans le cadre de l'IA.
- Δ Ce pseudo-code seul ne suffit pas, il faut aussi coder la simplification d'une formule⁷, ainsi que la substitution d'une variable par une proposition dans une formule. De plus, il faut un moyen de sélectionner une variable dans φ .

3.2 Une petite parenthèse : les graphe de flots

On peut généraliser la notion d'arbre de décision à celle de graphe de flot (« flowchart »). Il s'agit simplement d'autoriser certains choix à nous faire revenir en arrière.

Un graphe de flot est un graphe orienté étiqueté par les arêtes et les sommets. On essaye en général de se limiter aux arêtes de degré au plus 2 (choix binaires).

Exemple. Les graphes de flot de contrôle sont les graphes de flot représentant l'exécution d'algorithmes en fonction des tests logiques qui y ont lieu.

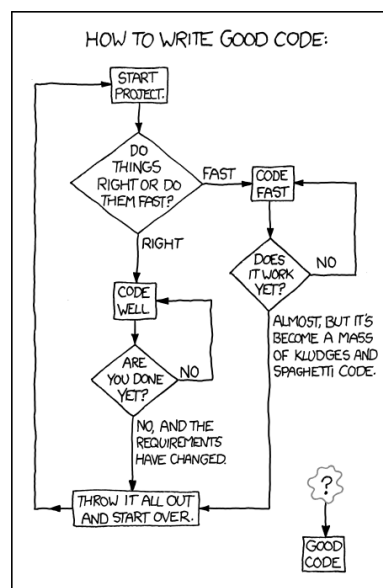


FIGURE 14.2 – Good code. Src : xkcd.com/844

7. C'est l'étape qui contient tout le travail ! cf TP

4 Quantificateurs et introduction à la logique du premier ordre

Jusqu'à présent, nous ne nous sommes intéressé·es qu'au **calcul propositionnel** (avec les formules propositionnelles). Il se trouve que ce cadre est très restreint et ne permet pas de faire grand chose en pratique. En particulier, vous noterez l'absence des quantificateurs \forall et \exists omniprésents dans les théorèmes mathématiques.

Passons maintenant à un formalisme plus général : celui de la logique du premier ordre, que l'on appelle aussi **calcul des prédicats**. On va ajouter deux notions majeures lors de ce changement :

- 1) On ajoute les quantificateurs \forall et \exists à la syntaxe de nos formules
- 2) Les "briques de bases" des formules ne sont plus les variables propositionnelles $x_i \in \mathbb{Z}$, mais n'importe quel «prédicat» (comme par exemple une égalité $x = y$). En particulier, **il n'y a plus aucune raison que les formules construites soient vraies soit fausses**, le tiers exclu et la double négation ne tiendront plus en toute généralité.

Remarque. La logique du premier ordre à proprement parler est hors programme, et on ne verra donc que sa syntaxe, la sémantique étant très explicitement hors programme. Cependant, les quantificateurs ainsi que tout le travail effectué sur les quantificateurs est au programme et à savoir maîtriser, et ce travail perd beaucoup de son sens en dehors du contexte de la logique du premier ordre.

Définition 30.

Un **langage** (du premier ordre) \mathcal{L} est la donnée d'une famille quelconque de symboles divisés en trois sortes :

- les symboles de **fonction**, dotés chacun d'une arité dans \mathbb{N}^* (son nombre d'arguments). Par exemple, une fonction peut être unaire, binaire, etc (n-aire).
- les symboles de **constante**. Il s'agit en quelque sorte de "symboles de fonction d'arité 0".
- les symboles de **relation**, aussi appelées **prédicats**, doté chacun d'une arité $\in \mathbb{N}^*$. On supposera toujours qu'on dispose au moins d'un symbole de relation « = » (égalité) d'arité 2, dans tout langage.

Remarque.

- On appelle aussi un langage \mathcal{L} une **signature**, ou un **vocabulaire**.
- Vous connaissez au moins un autre exemple de symbole de relation classique : \leq ou encore $<$.

Exemple.

Donnons la signature de la théorie des groupes :

Dans toute cette partie, on se dote d'un ensemble infini dénombrable χ de variables. \triangle Ce ne sont plus des variables propositionnelles, mais de simples variables (qui pourront "valoir" n'importe quoi, des entiers, des matrices, etc.).

Définition 31.

L'ensemble \mathcal{T} des termes sur un langage \mathcal{L} est défini inductivement par :

- $\chi \subseteq \mathcal{T}$ (autrement dit, les variables sont des termes).
- les constantes sont des termes : si c est un symbole de constante, alors $c \in \mathcal{T}$.
- Pour tout symbole f de fonction d'arité $n \in \mathbb{N}^*$ et pour tous termes $t_1, t_2, \dots, t_n \in \mathcal{T}$, on a $f(t_1, \dots, t_n) \in \mathcal{T}$

Remarques :

- Attention, nous sommes de nouveau dans le monde de la syntaxe ! $f(t_1, \dots, t_n)$ est une pure écriture de symboles, pas une "application" de fonction, même si c'est comme ça que ce serait **interprété** (dans le monde de la sémantique).
- Autre formulation des termes sur \mathcal{L} : l'ensemble \mathcal{T} est le plus petit ensemble contenant les variables et les constantes et qui est stable par "application" des symboles de fonction de \mathcal{L} .

Exemple.

- Donner quelques termes sur le langage de la théorie des groupes :

De manière équivalente, on peut encore définir les termes comme des arbres. Dessinez l'un des termes précédents (qui n'est pas un cas de base) sous forme d'un arbre :

Définition 32.

Une **formule atomique** sur un langage \mathcal{L} est de la forme $R(t_1, \dots, t_n)$ où R est un symbole de relation n -aire de \mathcal{L} et t_1, \dots, t_n sont des termes sur \mathcal{L} .

Exemple.

Écrire quelques formules atomiques du langage de la théorie des groupes :

Définition 33.

L'ensemble \mathcal{F} des **formules** de la logique du premier ordre est défini inductivement par :

- Les formules atomiques sont des formules.
- Si $\varphi, \varphi' \in \mathcal{F}$, et $x \in \chi$ alors on construit les formules suivantes :
 - $\neg \varphi \in \mathcal{F}$
 - $\varphi \wedge \varphi' \in \mathcal{P}$
 - $\varphi \vee \varphi' \in \mathcal{P}$
 - $\varphi \rightarrow \varphi' \in \mathcal{P}$
 - $\varphi \leftrightarrow \varphi' \in \mathcal{P}$

mais aussi :

- $\exists x, \varphi \in \mathcal{F}$
- $\forall x, \varphi \in \mathcal{F}$

Les symboles \exists et \forall ci-dessus sont appelés des **quantificateurs** :

- \exists (**il existe**) : quantificateur existentiel.
- \forall (**pour tout**) : quantificateur universel.

Remarques :

- Ne pas confondre une *formule* avec une *formule propositionnelle*. L'une est défini dans le cadre de la logique du premier ordre, l'autre dans le cadre du calcul propositionnel. Si vous voulez abréger le second, dites plutôt *proposition* que formule (même si l'abus de langage existe).
- Ces formules sont aussi (de manière équivalente) des arbres. Ex : représenter la formule suivante (du langage de la théorie des groupes) sous forme d'un arbre (où on rappelle que e est un symbole de constante) :

$$\forall x, ((\exists y, x * y = y) \rightarrow x = e)$$

- Les priorités usuelles s'appliquent pour se passer des parenthèses. En particulier, \forall et \exists attrapent tout ce qu'ils peuvent.

Définition 34 (Occurrences libres et liées).

Une **occurrence** d'une variable x dans une formule φ est une position de x dans φ (i.e. un noeud étiqueté par x dans l'arbre associé).

Une occurrence de x est dite **liée** si elle est contenue dans une sous-formule de la forme $\exists x, \varphi'$ ou $\forall x, \varphi'$. Dans ce cas, le dernier quantificateur $\forall x$ ou $\exists x$ rencontré avant x est appelé le **point de liaison** de x .

Si une occurrence de x n'est pas liée, elle est dite **libre**.

Une variable x est dite **libre** si elle possède au moins une occurrence libre, et liée si elle apparaît dans φ et n'est pas libre.

Une formule est dite **close** si elle ne contient aucune variable libre.

On note $V(\varphi)$ l'ensemble des variables qui apparaissent dans φ et $VL(\varphi)$ l'ensemble de ses variables libres.

Définition 35.

La **portée** d'un quantificateur est le sous-arbre enraciné en ce quantificateur, i.e. la portée contient toutes les occurrences de sa sous-formule. Mais Δ , ce n'en est pas forcément le point de liaison.

Exemple.

Remarque.

- C'est comme en programmation, quand on cherche à savoir quelle variable contient quoi et désigne quoi à tout moment du programme. Les notions de portée coïncident.
- Δ Comme en programmation, on évite de donner le même nom de variable à des variables liées par différents points. Ce sont des variables muettes, on peut les renommer.

Définition 36 (Substitution (avec quantificateurs)).

Soit t un terme, φ une formule et x une variable.

La **substitution** de x par t dans φ est la formule obtenue en remplaçant chaque occurrence *libre* de x par dans φ par le terme t .

Remarque.

- Δ Ça n'aurait aucun sens ici de substituer une variable par une formule (en quelque sorte, leurs "types" ne correspondent pas, contrairement au calcul propositionnel où les deux s'évaluent à un booléen Vrai ou Faux).

Et la sémantique alors ? Qu'est ce que ça veut dire tout ça ? C'est Hors-Programme...

Mais pour vous en donner une idée, il ne faut pas seulement interpréter les variables cette fois (comme dans le calcul propositionnel), mais aussi chacun des symboles du langage ! Lorsqu'on a dit ce que "veut dire" chacun de ces symboles, on peut évaluer une formule. Le résultat obtenu dépend fortement de l'**interprétation** (ou **modèle**) qu'on regarde, et n'est plus nécessairement Vrai ou Faux (pensez aux formules indécidables par exemple).

Exemple. Prenons un langage contenant un symbole de fonction $*$ (une loi "multiplication") et un symbole de relation $=$ (égalité). Considérons la formule $\forall x, \forall y, x * y = y * x$ (commutativité de la loi $*$).

Donner une interprétation⁸ du langage tel que cette formule s'évaluerait à Vrai, et une interprétation pour laquelle la formule s'évaluerait à Faux.

8. On ne l'a pas défini formellement, mais vous pouvez déjà le faire intuitivement.

Chapitre 15

ALGORITHMIQUE DU TEXTE

Notions	Commentaires
Recherche dans un texte. Algorithme de Boyer-Moore. Algorithme de Rabin-Karp.	On peut se restreindre à une version simplifiée de l'algorithme de Boyer-Moore, avec une seule fonction de décalage. L'étude précise de la complexité de ces algorithmes n'est pas exigible.
Compression. Algorithme de Huffman. Algorithme Lempel-Ziv-Welch.	On explicite les méthodes de décompression associées.

Extrait de la section 4.4 du programme officiel de MP2I : « Algorithmique des textes ».

SOMMAIRE

Chapitre non-encore rédigé.

Chapitre 16

GRAPHES PONDÉRÉS

Notions	Commentaires
Pondération d'un graphe. Étiquettes des arcs ou des arêtes d'un graphe.	On motive l'ajout d'information à un graphe par des exemples concrets : graphe de distance, automate fini, diagramme de décision binaire.

Extrait de la section 3.4 du programme officiel de MP2I : « Structures de données relationnelles ».

Notions	Commentaires
Notion de plus courts chemins dans un graphe pondéré. Algorithme de Dijkstra. Algorithme de Floyd-Warshall.	On présente l'algorithme de Dijkstra avec une file de priorité en lien avec la représentation de graphes par listes d'adjacences. On présente l'algorithme de Floyd-Warshall en lien avec la représentation de graphes par matrice d'adjacence.

Extrait de la section 4.5 du programme officiel de MP2I : « Algorithmique des graphes ».

SOMMAIRE

Chapitre non-encore rédigé.

Chapitre 17

BASES DE DONNÉES

Notions	Commentaires
Vocabulaire des bases de données : tables ou relations, attributs ou colonnes, domaine, schéma de tables, enregistrements ou lignes, types de données.	On présente ces concepts à travers de nombreux exemples. On s'en tient à une notion sommaire de domaine : entier, flottant, chaîne ; aucune considération quant aux types des moteurs SQL n'est au programme. Aucune notion relative à la représentation des dates n'est au programme ; en tant que de besoin on s'appuie sur des types numériques ou chaîne pour lesquels la relation d'ordre coïncide avec l'écoulement du temps. Toute notion relative aux collations est hors programme ; en tant que de besoin on se place dans l'hypothèse que la relation d'ordre correspond à l'ordre lexicographique usuel.
Clé primaire.	Une clé primaire n'est pas forcément associée à un unique attribut même si c'est le cas le plus fréquent. La notion d'index est hors programme.
Entités et associations, clé étrangère.	On s'intéresse au modèle entité–association au travers de cas concrets d'associations $1 - 1$, $1 - *$, $* - *$. Séparation d'une association $* - *$ en deux associations $1 - *$. L'utilisation de clés primaires et de clés étrangères permet de traduire en SQL les associations $1 - 1$ et $1 - *$.
Requêtes SELECT avec simple clause WHERE (sélection), projection, renommage AS. Utilisation des mots-clés DISTINCT, LIMIT, OFFSET, ORDER BY. Opérateurs ensemblistes UNION, INTERSECT et EXCEPT, produit cartésien.	Les opérateurs au programme sont +, -, *, / (on passe outre les subtilités liées à la division entière ou flottante), =, <>, <, <=, >, >=, AND, OR, NOT, IS NULL, IS NOT NULL.
Jointures internes $T_1 \text{ JOIN } T_2 \dots \text{ JOIN } T_n \text{ ON } \phi$, externes à gauche $T_1 \text{ LEFT JOIN } T_2 \text{ ON } \phi$.	On présente les jointures (internes) en lien avec la notion d'associations entre entités.
Agrégation avec les fonctions MIN, MAX, SUM, AVG et COUNT, y compris avec GROUP BY.	Pour la mise en oeuvre des agrégats, on s'en tient à la norme SQL99. On présente quelques exemples de requêtes imbriquées.
Filtrage des agrégats avec HAVING.	On marque la différence entre WHERE et HAVING sur des exemples.

Extrait de la section 7 du programme officiel de MP2I : « Bases de données ».

SOMMAIRE

Chapitre non-encore rédigé.