

Informatique commune

Olivier Reynet

16-03-2023

TABLE DES MATIÈRES

I	Introduction	3
1	Crayons et questions	5
A	Pourquoi travailler l'informatique?	5
B	L'ingénieur, la construction et le calcul	6
C	Programme d'informatique et philosophie de la construction	6
D	Comment travailler l'informatique?	7
2	Comment lire ce cours?	9
A	L'informatique et le langage : une histoire de mots	9
B	L'anglais et le français	9
C	À la frontière du programme officiel --> HORS PROGRAMME	10
★	D Option informatique	10
3	Algorithmes à maîtriser	11
A	Rechercher	11
B	Trier	11
C	Calculer	11
D	Calculs sur les types structurés	12
E	Concepts	12
4	Au commencement, des concepts	13
A	Algorithmes, programmes et processus	13
B	Langages, compilateurs et machines --> HORS PROGRAMME	15
C	Des paradigmes différents --> HORS PROGRAMME	18
D	Systèmes d'exploitations --> HORS PROGRAMME	20
E	Bibliothèques logicielles	22
II	Semestre 1	25
5	Types et opérateurs	27
A	Types	27
B	Opérateurs	29
C	Opérateurs sur les chaînes de caractères	32
D	Une variable Python est une référence	32

TABLE DES MATIÈRES

E	Mots-clefs du langage	33
6	Structurons	37
A	Anatomie d'un programme Python	37
B	Programmation structurée	38
C	Fonctions Python	43
D	Conventions de nommage	45
7	Listes Python	47
A	Constructeurs de listes	47
B	Opérations sur une liste	49
C	Concaténation et démultiplication de listes	50
D	Des listes indiçables	50
E	Des listes itérables	50
F	Des listes tronçonnables	51
G	Les algorithmes simples mais incontournables	51
H	Listes, copies et références	53
I	Listes, paramètres et fonctions	53
J	Tuples	55
8	Trier et rechercher	57
A	Comment caractériser un algorithme de tri?	58
B	Trier un tableau	58
C	Comparatif des tris	60
D	Recherche séquentielle	60
E	Recherche dichotomique	61
F	Compter les opérations	64
G	Trier avec les fonctions natives de Python	67
9	Récursivité	69
A	Principes	69
B	Types de récursivité	71
C	Du récursif à l'itératif	73
10	Introduction à Numpy	75
A	Importation	76
B	Créer des vecteurs, des matrices ou des tableaux	76
C	Accéder aux éléments d'un tableau	77
D	Opérations élément par élément	78
E	Opérations matricielles	79
F	Types de données	80
G	Autres fonctions	80

TABLE DES MATIÈRES

III Semestre 2	83
11 Terminaison et correction	85
A Un programme se termine-t-il?	85
B Variant de boucle	87
C Démontrer la terminaison d'un algorithme	87
D Le résultat est-il correct?	90
E Invariant de boucle	91
F Démontrer que le résultat d'une boucle est correct	91
12 Complexité	95
A Complexités algorithmiques	95
B Notation asymptotique	96
C Typologie de la complexité	99
D Calcul du coût d'une instruction	99
E Calculs classiques de complexité	100
F Exemple de la recherche dichotomique	102
G Exemple de l'exponentiation rapide	104
H Exemple du tri fusion	105
I Synthèse	105
13 Diviser pour régner	109
A Diviser pour régner	109
B Exemple de la recherche dichotomique	112
C Exemple de l'exponentiation rapide	113
14 Algorithmes gloutons	115
A Principe	115
B Modélisation	116
C Exemple du sac à dos	119
D Gloutonnerie et dynamisme	120
15 Les mots des graphes	121
A Typologie des graphes	121
B Implémentation des graphes	125
C Caractérisation structurelle des graphes	126
D Isomorphisme des graphes	128
E Chaînes, cycles et parcours	130
F Sous-graphes et connexité	133
G Coloration de graphes	134
H Distances	136
I Arbres	136

TABLE DES MATIÈRES

16 Propriétés des graphes	139
A Des degrés et des plans	139
B Caractérisation des chaînes, des cycles et des graphes	140
C Graphes acycliques et connexes	141
D Coloration, graphes planaires et nombre chromatique	141
E Principe d'optimalité et plus court chemin dans un graphe	142
17 Algorithmes et graphes	143
A Parcours d'un graphe	143
B Trouver un chemin dans un graphe	147
C Plus courts chemins dans les graphes pondérés	150
IV Semestre 3	161
18 Structures de données --- HORS PROGRAMME	163
A Type abstrait de données et structure de données	163
B TAD tableaux et listes	165
C Implémentation des tableaux	165
★ D Implémentations des listes	168
E Bilan de complexités des opérations sur les structures listes et tableaux	169
19 Dictionnaires	171
A Types abstraits de données (TAD)	171
B Constructeurs de dictionnaires	173
C Opérations sur un dictionnaire	174
D Fusionner des dictionnaires	176
E Des dictionnaires itérables	176
F Implémentation d'un TAD dictionnaire	176
G Tables de hachage	177
20 Programmation dynamique	183
A Motivations	183
B Principes de la programmation dynamique	185
C Exemples simples de programmation dynamique	186
D Le retour du sac à dos	189
E Programmation dynamique itérative	191
F Programmation dynamique récursive : mémoïsation	193
21 Bases de données relationnelles	195
A Pourquoi?	195
B Données et gestion des données	196
C De la conception à l'implémentation physique	197
D Le modèle relationnel	203
E Traduction des associations dans le modèle relationnel	205

TABLE DES MATIÈRES

F	Du modèle physique à la création des tables	---> HORS PROGRAMME	210
22	Modèle relationnel et langage SQL		213
A	Requêtes SQL		213
B	Projection : SELECT ...FROM		216
C	Sélection : WHERE		217
D	Jointure : JOIN ...ON		219
E	Agréger, grouper et filtrer des résultats		222
F	Opérations ensemblistes		223
G	Requêtes imbriquées ---> HORS PROGRAMME		224
H	De belles requêtes SQL		225
23	Et la machine apprit		227
A	Principes		227
B	Apprentissage supervisé : k plus proches voisins		230
C	Apprentissage non supervisé : k moyennes		236
24	Théorie des jeux		239
A	Introduction à la théorie des jeux		240
B	Jeux d'accessibilité, l'exemple des jeux de Nim		244
C	Modélisation d'un jeu d'accessibilité		247
D	Stratégies et positions		247
E	Attracteurs		248
F	Algorithme de calcul de l'attracteur		249
★ G	Solution des jeux de Nim et impartialaux ---> HORS PROGRAMME		250
H	Au-delà des jeux d'accessibilité, les heuristiques		251
I	Minimax et les heuristiques		252
J	Élagage $\alpha\beta$ sur un arbre Minimax ---> HORS PROGRAMME		255
K	A* pour trouver un chemin		257
V	Annexes		259
Bibliographie			261
Articles			261
Livres			262
Vidéos			262
Sites web			262
Tables de figures			263
Tables de tableaux			269
Tables de codes			271
Index			273

TABLE DES MATIÈRES

LISTE DES ALGORITHMES

1	Produit de deux nombres	14
2	Produit de deux nombres entiers avec boucle inconditionnelle	40
3	Produit de deux nombres entiers avec boucle conditionnelle	41
4	Tri par sélection	59
5	Tri par insertion	59
6	Tri par comptage	60
7	Recherche séquentielle d'un élément dans un tableau	61
8	Recherche d'un élément par dichotomie dans un tableau trié	62
9	Recherche d'un élément par dichotomie dans un tableau trié, renvoyer l'indice minimal en cas d'occurrences multiples.	63
10	Recherche séquentielle d'un élément dans un tableau	65
11	Tri par insertion, calcul du nombre d'opérations	65
12	Factoriel récursif	70
13	Recherche récursive d'un élément par dichotomie dans un tableau trié	72
14	Est-il mon aïeul?	72
15	Factoriel récursif terminal	72
16	Algorithme récursif terminal	73
17	Version itérative d'un algorithme récursif terminal	73
18	Factoriel itératif	73
19	Test de la terminaison de l'algorithme SOUS_TEST	86
20	Calcul des éléments de la suite de Syracuse	86
21	Produit de deux nombres entiers naturels, quel pourrait-être un variant?	88
22	Produit de deux nombres entiers naturels, quel pourrait-être un variant?	88
23	Boucle infinie (à ne pas reproduire!)	89
24	Boucle infinie, condition d'arrêt sur un flottant (ne pas reproduire!!!)	89
25	Factoriel récursif	90
26	Division euclidienne $a = bq + r$	92
27	Exponentiation rapide, version itérative	93
28	Calcul de a^n	100
29	Produit de deux vecteurs $(n, 1) \times (1, n) \longrightarrow (n, n)$	100
30	Somme de puissances $1 + 2^n + \dots + n^n$	101
31	Accumuler	101
32	Appliquer une fonction et accumuler	102
33	Recherche récursive d'un élément par dichotomie dans un tableau trié	103
34	Exponentiation naïve a^n	104

35	Exponentiation rapide a^n	104
36	Tri fusion	106
37	Découper en deux	106
38	Fusion de deux sous-tableaux triés	106
39	Diviser, résoudre et combiner (Divide And Conquer)	110
40	Recherche récursive d'un élément par dichotomie dans un tableau trié	112
41	Exponentiation naïve a^n	113
42	Exponentiation rapide a^n	114
43	Exponentiation rapide a^n (version itérative)	114
44	Principe d'un algorithme glouton	117
45	Réservation d'une place au port de Brest	118
46	Problème du sac à dos	120
47	Parcours en largeur d'un graphe	144
48	Parcours en profondeur d'un graphe (version récursive)	147
49	Parcours en profondeur d'un graphe (version itérative)	147
50	Longueur d'une chaîne via un parcours en largeur d'un graphe pondéré	148
51	Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné	151
52	Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné	155
53	Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet	158
54	A*	160
55	Fibonacci récursif	188
56	Fibonacci itératif, sans calculs redondants.	188
57	KP(n, π) par programmation dynamique, de bas en haut	193
58	KP(n, π) par programmation récursive brute	193
59	KP(n, π) par programmation dynamique et mémoïsation	194
60	k plus proches voisins (KNN)	234
61	k moyennes (k-means)	236
62	Calcul de l'attracteur du joueur J_1	250
63	Minimax	254
64	Minimax avec élagage $\alpha\beta$	256
65	A*	258

Première partie

Introduction

1

CRAYONS ET QUESTIONS

Proverbe chinois : «Celui qui pose une question risque cinq minutes d'avoir l'air bête, celui qui ne pose pas de question restera bête toute sa vie».

Bernard Werber, dans les Thanatonautes

Les crayons c'est pas du bois et de la mine, c'est de la pensée par les phalanges.

Henri de Toulouse Lautrec

À la fin de ce chapitre, je sais :

 pourquoi et comment travailler l'informatique.

A Pourquoi travailler l'informatique?

Il est pertinent de travailler l'informatique parce que ¹ :

1. l'informatique rapporte des points aux concours, points qui font la différence,
2. ces points sont plus faciles à gagner que les points en mathématiques,
3. on manque d'ingénieries et d'ingénieurs capables dans le domaine,
4. l'évolution de votre carrière ne sera pas la même si vous savez utiliser pertinemment l'informatique dans votre contexte professionnel,
5. l'informatique façonne et façonnera toute votre vie. Tout simplement.
6. l'informatique, c'est passionnant.

1. du plus pragmatique au plus important!

B L'ingénieur, la construction et le calcul

Le technicien est celui à qui on confie une mission qu'il sait faire. Un ingénieur, c'est celui à qui on confie une mission et qu'il va savoir faire : la réponse n'existe pas nécessairement sur étagère, il lui faut souvent la construire, la calculer. Or, dans le monde de l'industrie, vous ne pouvez rien construire aujourd'hui sans l'informatique.

Si vous voulez être un ingénieur capable, il vous faut donc maîtriser l'informatique, tout comme vos ancêtres avant vous ont appris à maîtriser le crayon, les abaques, la règle, le compas, la trigonométrie, la corde à noeuds, le calcul dans le système de numération de position, les tables de logarithmes, la règle à calcul et les machines à calculer. Ce n'était pas un choix mais une évidence qui s'imposait, cela facilitait la construction et les calculs. **Aujourd'hui, l'évidence informatique s'impose à vous de la même manière et tout en vous confiant une puissance de calcul inégalée.**

C Programme d'informatique et philosophie de la construction

[Le programme officiel d'informatique est à consulter en ligne](#). Il est très précis et vaste et indique clairement ce qu'il faut savoir et savoir faire. Il indique également à la fin les éléments du langage Python que l'on attend que vous maîtrisiez.

Un extrait me semble particulièrement important :

La pratique régulière de la résolution de problèmes par une approche algorithmique et des activités de programmation qui en résultent constitue un aspect essentiel de l'apprentissage de l'informatique.

Cette pratique régulière se fait en TP. Les TP qui vous sont proposés présentent des problèmes et montrent une voie pour construire une solution à ce problème. Car *résoudre par une approche algorithmique des problèmes*, cela signifie construire une solution à un problème. Or une solution est une information. **L'informatique est donc la discipline de la construction de l'information par le calcul.**

D'une manière générale, les TP sont importants, notamment car il n'est pas prévu dans le programme officiel de cours magistraux au premier semestre. Les TP abordent les sujets du programme dans un ordre logique, en commençant par les concepts les plus faciles. Par ailleurs, mes TP sont progressifs, les questions les plus faciles se trouvent au début et les plus difficiles à la fin. Un TP raconte une histoire : il s'agit souvent de construire² une information, de la faire apparaître : un message secret, la solution à un problème concret ou la stratégie gagnante d'un jeu. Cette construction se fait par étapes, progressivement, tout au long du TP. Cette philosophie s'inspire des épreuves de concours qui vous présentent également la construction d'une information autour d'un thème, si bien que chaque TP vous prépare davantage au concours.

Tout ingénieur sait que pour bien construire, il faut commencer par bien bricoler : bricoler en informatique, cela signifie imaginer des algorithmes sur des cas simples, les implémenter, tester les codes et recommencer sur des situations plus complexes. Se construire des outils

2. Programmer, c'est bricoler de l'information!

pour en construire d'autres encore plus puissants, tout comme le bricoleur élabore parfois un outil dédié pour réussir une opération mécanique délicate. L'avantage de l'informatique, c'est qu'on ne risque ni de casser quelque chose, ni de se faire mal : au pire, on a fabriqué un outil et on appréhende mieux une situation problème. Alors, bricolons !

D Comment travailler l'informatique ?

Mon cours en ligne suit exactement le programme officiel. Sur ce site, vous trouverez tous les cours et tous les TP de chaque semestre, avec les solutions. Ils sont à votre disposition dans un but précis : aider simultanément ceux qui ont le plus de difficultés et ceux qui veulent aller plus loin.

Sur le plan matériel, l'informatique étant aujourd'hui ubiquitaire, vous pouvez travailler sur n'importe quel ordinateur³ ainsi que sur vos tablettes ou smartphones. Après avoir installé l'interprète Python, un simple éditeur de texte suffit généralement pour composer les codes. Cependant, un environnement de développement logiciel (IDE en anglais) est souvent un plus en termes de lisibilité et d'intelligibilité du code ainsi qu'en termes de rapidité de développement. Généralement, les classes préparatoires utilisent Pyzo, un IDE orienté calcul scientifique qui a le mérite d'être simple, disponible sur tous les OS et libre.

Sur le plan pédagogique, les sections qui suivent donnent quelques conseils éclairés.

a Avant le TP

- Lire le cours associé au TP : cela veut dire se poser des questions sur le sens du chapitre en prenant des notes sur les concepts présentés (avec un crayon et du papier).
- Lire le TP en entier.
- Préparer les premières questions du TP à l'écrit (avec un crayon et du papier).

b Pendant le TP sur machine

- Ne pas hésiter à prendre le clavier et la souris. Les partager avec son binôme régulièrement. Il faut être acteur de son apprentissage et à l'écoute.
- Quand on n'a pas le clavier, on peut écrire, imaginer et proposer des solutions (avec un crayon et du papier).
- Commenter ou prendre des notes sur les points les plus difficiles ou les plus importants du TP (avec un crayon et du papier).
- Interpeler régulièrement l'enseignant et lui poser les questions en lien avec le TP et le cours.

c Après le TP

- Finir le TP si vous n'avez pas eu le temps de finir en classe.

3. car Python est un langage interprété

- Poser des questions s'il y a encore des zones d'ombres.
- Écrire une petite synthèse sur le TP que vous pourrez relire avant le devoir (avec un crayon et du papier).
- Suggérer des améliorations au professeur.

d Sur le long terme (18 mois)

- Lire le cours en avance de phase : on comprend toujours mieux la deuxième fois et encore mieux la troisième !
- Poser des questions : à l'oral, par écrit.
- Repérer ses lacunes et les combler : syntaxe du langage, compréhension des structures et des algorithmes, intérêts des concepts. Il est souvent plus chronophage d'avoir des lacunes que de les combler.
- S'entraîner avec un autre élève : se poser des questions, tester l'autre sur la compréhension, expliquer à l'autre les concepts que l'on a soi-même compris. C'est ainsi que vous progresserez vite.
- Créer un lien entre les différents thèmes étudiés. Par exemple, se demander pourquoi je vous présente la récursivité avant les approches divisor pour régner et gloutonne ?
- Associer un exemple concret à chaque concept (généralement abstrait) et être capable de l'expliquer à un autre. Par exemple, associer l'exemple du Morpion à l'algorithme Minimax ou la programmation dynamique au problème du sac à dos.

Pour plus d'éléments sur les techniques d'apprentissage, je vous conseille le site [The Learning Scientist](#).

2

COMMENT LIRE CE COURS ?

Raisonner sur la matière et l'énergie, et raisonner sur l'information, c'est très différent.

Gérard Berry

À la fin de ce chapitre, je sais :

 lire ce cours

A L'informatique et le langage : une histoire de mots

■ **Définition 1 — Informatique.** L'informatique est la science du traitement automatique et rationnel de l'information par un système concret ou abstrait.

L'informatique traitant de l'information, il y est naturellement question de langages. Comment parler de l'information sans utiliser et maîtriser le langage en général? Comment maîtriser un langage sans connaître le sens des mots? La plupart des difficultés de l'informatique peuvent être dépassées si l'on maîtrise la définition des concepts. C'est pourquoi celles-ci sont bien mises en évidence dans ce cours.

B L'anglais et le français

Le français est la langue privilégiée de ce cours, celle qui permet d'expliquer et de comprendre. Mais, les codes informatiques présentés privilégient l'anglais : la raison principale de ce choix réside dans le fait qu'il est beaucoup plus difficile pour un être humain d'appréhender un mélange de langues plutôt que d'en lire un seule à la fois. C'est pourquoi j'essaie de ne pas mélanger les langues, tout en les pratiquant toutes.

L'influence de la langue anglaise étant malgré tout très forte dans le domaine de l'informatique, je précise les correspondances avec le français et les nuances terminologiques dans des paragraphes dédiés nommés *vocabulary*.



Vocabulary 1 — Word ⇔ mot.

Si au cours de votre lecture, vous trouvez des erreurs ou des coquilles, n'hésitez pas à me les signaler!

a Des remarques particulières et des méthodes

P Certains paragraphes commencent par ce symbole P dans un rond turquoise : cela signifie qu'ils traitent plus particulièrement du langage Python. Si vous voulez aller vite, vous pouvez vous concentrer dans un premier temps sur leur lecture.

Certains de ces paragraphes sont essentiels pour bien réussir les épreuves d'informatique commune des concours. Ceux-ci finissent la plupart du temps ainsi :



Le paragraphe précédent est important pour l'épreuve d'informatique!

R D'autres remarques d'ordre plus général figurent ainsi, dans un paragraphe commençant par un R dans un rond.

M **Méthode 1 — Méthode** Certains paragraphes sont identifiés comme étant des méthodes. Ils sont à connaître car utiles pour progresser efficacement.

C À la frontière du programme officiel → HORS PROGRAMME

R Certaines sections marquées → HORS PROGRAMME présentent des notions hors **programme officiel**. Néanmoins, elles ont souvent un grand intérêt pour la compréhension de l'informatique en général et certaines épreuves de concours se jouent à la frontière du programme.

★ D Option informatique

R Certaines sections sont marquées d'une étoile signifiant que le contenu de la section est davantage destiné aux élèves qui suivent l'option informatique. La lecture de ce paragraphe est certainement intéressante pour comprendre en profondeur mais pas nécessaire pour l'épreuve d'informatique commune.

3

ALGORITHMES À MAÎTRISER

One learns from books and example only that certain things can be done. Actual learning requires that you do those things.

Frank Herbert

A Rechercher

- rechercher l'élément maximum ou minimum d'un tableau (cf. listing 7.1)
- rechercher l'indice du maximum ou du minimum d'un tableau (cf. listing 7.1)
- rechercher un élément de manière séquentielle dans un tableau non trié (cf algorithme 7)
- rechercher par dichotomie un élément dans un tableau trié (en itératif ou récursivement) (cf. algorithmes 8 et 13)

B Trier

- trier par insertion, complexité dans le pire et meilleur des cas (cf. algorithme 5)
- trier par sélection (cf. algorithme 4)
- trier par comptage ¹ (cf. algorithme 6)
- Tri fusion (cf. algorithme 36) ²

C Calculer

- calculer la moyenne d'une série de valeur dans un tableau (cf. code 7.1)

1. connaître le concept suffit
2. connaître les principes et les complexités dans le pire et meilleur des cas.

- calculer la médiane d'une série de valeur dans un tableau trié
- calculer factoriel (en itératif ou récursivement)
- calculer les termes de la suite de Fibonacci (en itératif ou récursivement)
- créer et initialiser un tableau Numpy de dimension quelconque.
- calculer une formule simple (de type $2\nu \sin(2\pi\omega t)$) vectoriellement avec Numpy
- calculer une exponentiation rapide (en récursif) (cf. algorithme 42)
- calculer le PGCD de deux entiers, algorithme d'Euclide (en itératif ou récursivement)
- évaluer un polynôme avec la méthode d'Horner (cf. TP Complexité)
- calculer le produit scalaire de deux vecteurs
- calculer la distance euclidienne de deux points dans un espace de dimension n (cf. TP Machine Learning)

D Calculs sur les types structurés

- Chaînes de caractères : concaténer, parcourir
- Listes : créer, initialiser, parcourir, indexer (négatif, plage), tronçonner, éliminer des doublons (unique), mettre à plat une liste imbriquée,
- Dictionnaire : rechercher une clef, insérer une (clef,valeur), modifier une (clef,valeur), compter les occurrences des éléments d'une liste, utiliser un dictionnaire en programmation dynamique (mémoïsation) comme alternative à un tableau,
- Tableaux numpy : créer, initialiser, obtenir les dimensions (shape), parcourir, indexer (négatif, plage), tronçonner, opérer élément par élément (calcul vectoriel),
- Graphe : manipuler sous la forme d'une liste d'adjacence ou de matrice d'adjacence, parcourir en largeur, transposer un graphe orienté.

E Concepts

- Programmation structurée impérative
- Programmation procédurale : les fonctions, les paramètres d'entrées, valeur renournée
- Récursivité
- Complexité
- Représentation des nombres en machine (entiers signés, non signés et flottants)
- Algorithmes de décomposition : diviser pour régner, gloutons et programmation dynamique de bas en haut et par mémoïsation
- Apprentissage, prédiction, classification et régression
- Arène/Arbre de jeu, attracteur, stratégie, minimax
- Requêtes SQL

4

AU COMMENCEMENT, DES CONCEPTS

Swamp King : One day all this will be yours!
Herbert : What, the curtains ?

The Holy Grail, Monty Python

À la fin de ce chapitre, je sais :

- ☛ Distinguer les concepts d'algorithme, de programme et de processus
- ☛ Lister les grandes caractéristiques du langage Python
- ☛ Importer des fonctions d'une bibliothèque (module Python)

Ce premier chapitre n'explique pas dans le détail tous les concepts présentés. Cependant les TP du premier semestre permettent d'approfondir ces concepts et complémentent cette lecture.

Parler d'informatique nécessite de parler de l'information et des langages informatiques qui véhiculent ces informations. Parler des langages, c'est tenter de décrire les mots de ce langage, de la structure qui relie ces mots entre eux et de leur pouvoir. Pour parler de mots, il faut tout d'abord se mettre d'accord sur le sens de ces mots.

A Algorithmes, programmes et processus

■ **Définition 2 — Informatique.** L'informatique est la science du traitement automatique et rationnel de l'information par un système concret ou abstrait.

Formulé plus simplement, on peut dire que **l'informatique est la discipline de la construction de l'information par le calcul.**

 **Vocabulary 2 — Computer Sciences and Computer Engineering** ↔ Les anglo-saxons font une distinction entre l'informatique comme science abstraite (Computer Sciences) et l'informatique comme science concrète appliquée (Computer Engineering). En français, l'informatique désigne les deux, ce qui est important, l'une n'allant pas sans l'autre. Que ferait-on de la physique théorique sans l'expérience? Que ferait-on de l'informatique sans l'électronique?

L'informatique, tout comme la physique ou la chimie, fait appel aux mathématiques pour **modéliser** un problème et démontrer des propriétés. Mais à la différence des mathématiques qui se permettent souvent de manipuler des objets qu'on ne sait pas construire, l'informatique est une science **constructiviste** : lorsqu'on résout un problème en informatique, on construit la solution à l'aide un algorithme.

■ **Définition 3 — Algorithme.** Un algorithme est une méthode pour résoudre un problème donné. Cette méthode est constituée d'une suite d'instructions qui permet de trouver une solution au problème.

■ **Exemple 1 — Produit de deux nombres.** L'algorithme 1 calcule le produit de deux nombres.

Algorithme 1 Produit de deux nombres

1: Fonction PRODUIT(a, b)	$\triangleright a \in \mathbb{N}$ et $b \in \mathbb{N}$.
2: $p \leftarrow 0$	
3: $c \leftarrow 0$	$\triangleright c$ est un entier.
4: tant que $c < a$ répéter	
5: $p \leftarrow p + b$	
6: $c \leftarrow c + 1$	
7: renvoyer p	

On s'efforce généralement :

- de rendre un algorithme non ambigu,
- d'identifier clairement les entrées et les sorties ,
- de décrire un algorithme avec suffisamment d'abstraction afin de rendre de son implémentation indépendante du langage de programmation choisi.

 L'élaboration d'un algorithme n'est pas un exercice théorique. Au contraire c'est dessinant au brouillon, en émettant des hypothèses simplificatrices, en bricolant et en tâtonnant que l'on parvient généralement à écrire un algorithme qui fonctionne.

Le développement de l'informatique s'appuie fortement sur la logique, la théorie des graphes, la théorie des langages et des automates. Ces théories contribuent à la fois à son développement théorique et à son développement concret, c'est-à-dire électronique.

■ **Définition 4 — Programme.** Un programme est un ensemble d'instructions dans un langage de programmation donné.

R Il est important de savoir passer de l'expression d'un algorithme à un programme et, dans le cadre des épreuves des concours, à un programme en Python! L'expression de l'algorithme dans les épreuves est le plus souvent sous la forme d'un texte, c'est à dire qu'on décrit l'algorithme avec des **phrases**^a. Le code 4.1 traduit l'algorithme 1 en Python.

a. ce qui ne simplifie pas la tâche du candidat qui doit savoir lire et interpréter...

Code 4.1 – Un exemple de programme en Python - traduction de l'algorithme produit

```

1 def produit(a, b):    # une fonction
2     p = 0            # Un commentaire
3     c = 0
4     while c < a:
5         p = p + b
6         c = c + 1
7     return p
8
9
10 # Debut du programme principal
11 print("Le produit de 2 par 21 vaut :", produit(21, 2))

```

R Si un algorithme est abstrait, un programme est concret, c'est à dire qu'il est interprétable soit par un processeur directement, soit par un interpréteur (machine virtuelle). On peut donc demander à une machine d'exécuter un programme, mais on ne peut pas lui demander d'exécuter directement un algorithme.

■ **Définition 5 — Processus.** Un processus est un programme en cours d'exécution sur une machine, c'est à dire un processeur muni d'une mémoire et d'un système d'exploitation.

B Langages, compilateurs et machines --> HORS PROGRAMME

■ **Définition 6 — Langage de programmation.** Un langage de programmation est une notation normée dont l'objectif est d'implémenter des algorithmes.

■ **Exemple 2 — Langages de programmation.** Parmi les langages de programmation les plus utilisés actuellement on peut citer : C, C++, Java, Python ou Javascript. On choisit un langage plutôt qu'un autre en fonction de l'application visée. Javascript est par exemple un langage plutôt orienté web. C est plutôt dédié aux systèmes embarqués et bas niveau.

■ **Définition 7 — Compilateur.** Un compilateur est un logiciel qui transforme un langage en un autre. Le langage des données d'entrée est dit *langage source* et celui des données de sortie *langage cible*.

■ **Définition 8 — Langage machine.** Le langage machine est le langage d'un processeur. Il est composé d'instructions très élémentaires de type :

- mettre une donnée dans une case mémoire,
- modifier le contenu d'une case mémoire (opérations arithmétiques et logiques),
- tester la valeur d'une case mémoire,
- effectuer des branchements et des sauts dans le code,
- appeler une routine de calcul.

Ces instructions élémentaires sont exécutées par le processeur à une fréquence très élevée ^a.

- a.* de l'ordre du GHz dans les processeurs actuels.

■ **Définition 9 — Langage compilé.** Un langage compilé ^a est un langage muni d'un compilateur capable de générer du langage machine à partir du code source.

- a.* On devrait dire compilable, mais les anglicismes et les traductions littérales sont légions dans le domaine informatique.

■ **Exemple 3 — Quelques compilateurs.** Les compilateurs sont ubiquitaires dans tous les domaines de l'ingénierie et la plupart du temps invisibles. On trouve :

- **gcc** est un compilateur capable de compiler de nombreux langages dont le langage C. Il permet de traduire des instructions en langage C en langage machine, c'est à dire le langage d'un processeur.
- **javac** est un compilateur capable de traduire du code java en bytecode java.
- **pdflatex** est un compilateur capable de traduire du code latex en PDF et de créer les pages que vous lisez.

■ **Définition 10 — Langage interprété.** Un langage interprété est un langage muni d'un interpréteur capable de l'exécuter sur une machine concrète.

■ **Définition 11 — Interpréteur ou machine virtuelle.** Un interpréteur ^a ou machine virtuelle est un **logiciel** qui permet d'interpréter des instructions dans un langage généralement désigné par le terme Bytecode. Ce logiciel est exécuté par un processeur, ce qui explique le qualificatif virtuel : on ne programme pas un processeur mais un interpréteur.

- a.* On devrait dire interprète mais il s'agit encore une fois d'un anglicisme.

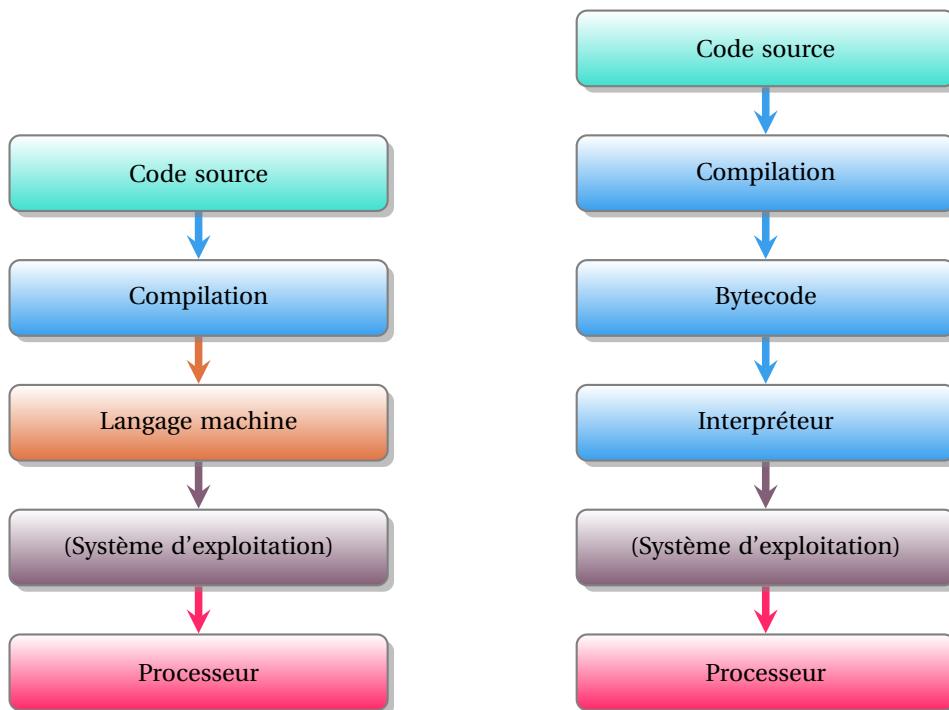


FIGURE 4.1 – Comparaison des chaînes d'exécution des langages compilés (à gauche) et des langages interprétés (à droite)

■ **Exemple 4 — Machines virtuelles.** Quelques exemples d'interpréteurs :

- Java Virtual Machine (JVM) est un interpréteur de code Java.
- Python Virtual Machine (PVM) est un interpréteur de code Python.
- Ocaml dispose d'un interpréteur et d'un compilateur en langage machine.

■ **Définition 12 — Bytecode.** Un bytecode est un langage intermédiaire entre un langage de programmation (de haut niveau) et les langages de plus bas niveau (proche du processeur). Il est exécuté par un interpréteur.

R Si un bytecode est indépendant de la machine sur laquelle il sera exécuté, un interpréteur dépend au contraire de la machine concrète sur laquelle il s'exécute c'est à dire du processeur et du système d'exploitation.

Le code Python, avant d'être exécuté par une machine virtuelle, c'est à dire un logiciel, est compilé dans un langage nommé Bytecode par la PVM.

P Python est donc un langage interprété.

```

0 LOAD_CONST           1 (0)
2 STORE_FAST          2 (p)
4 LOAD_CONST           1 (0)
6 STORE_FAST          3 (c)
8 LOAD_FAST            3 (c)
10 LOAD_FAST           0 (a)
12 COMPARE_OP          0 (<)
14 POP_JUMP_IF_FALSE  20 (to 40)
16 LOAD_FAST           2 (p)
18 LOAD_FAST           1 (b)
20 BINARY_ADD          2 (p)
22 STORE_FAST          3 (c)
24 LOAD_FAST           2 (1)
26 LOAD_CONST           3 (c)
28 BINARY_ADD          3 (c)
30 STORE_FAST          3 (c)
32 LOAD_FAST           0 (a)
34 LOAD_FAST           0 (<)
36 COMPARE_OP          8 (to 16)
38 POP_JUMP_IF_TRUE   2 (p)
40 LOAD_FAST           2 (p)
42 RETURN_VALUE

```

FIGURE 4.2 – Bytecode Python du code 4.1 de la fonction produit.

Le figure 4.2 donne, à titre d’illustration, le bytecode du programme 4.1. On perçoit bien la nature élémentaires des instructions impératives qui ressemblent beaucoup à celle d’un langage machine. Sauriez-vous l’interpréter à votre tour ?

C Des paradigmes différents --> HORS PROGRAMME

Tout comme il existe une multitude de langages humains, il existe une grande diversité de langages informatiques. Pour mieux cerner leurs différences, on utilise la notion de paradigme de programmation.

■ **Définition 13 — Paradigme de programmation.** Un paradigme de programmation est un ensemble de formes et de figures qui constitue un modèle propre à un langage.

■ **Définition 14 — Paradigme impératif.** Le paradigme impératif s’attache à décrire des séquences d’instructions (ordres) qui agissent sur un état interne de la machine (contexte). L’impératif explicite le *comment procéder* pour exécuter un programme. Cette programmation se rapproche de la logique électronique des processeurs.

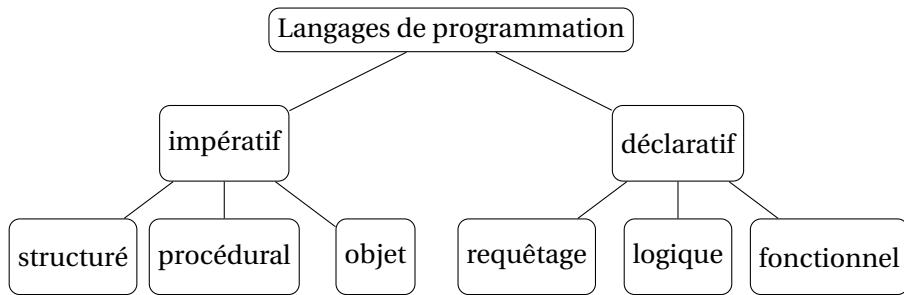


FIGURE 4.3 – Paradigmes des langages de programmation

■ **Définition 15 — Paradigme procédural.** Ce paradigme est une déclinaison de l’impératif et propose de regrouper des éléments réutilisables de code dans des routines. Ces routines sont appelées procédures (si elles ne renvoient rien) ou fonctions (si elles renvoient un résultat).

■ **Définition 16 — Paradigme objet.** Ce paradigme est une déclinaison de l’impératif et propose de décrire un programme comme l’interaction entre des objets à définir. Une classe est un type d’objet qui possède des attributs et des comportements. Ces caractéristiques sont encapsulées et peuvent être masquées à l’utilisateur d’un objet : cela permet de protéger l’intégrité de l’objet et de garantir une cohérence dans la manipulation des données.

■ **Définition 17 — Paradigme déclaratif.** Le paradigme déclaratif est une syntaxe qui s’attache à décrire le *quoi*, c’est à dire *ce que le programme doit faire*, non pas comment il doit le faire. Un langage déclaratif ne dépend pas de l’état interne d’une machine (contexte). Cette programmation se rapproche de la logique mathématique et délègue au compilateur la délicate question du *comment procéder*.

■ **Définition 18 — Paradigme fonctionnel.** Le paradigme fonctionnel est une déclinaison du déclaratif qui considère qu’un programme n’est qu’un calcul et qu’un calcul est le résultat d’une fonction. Le mot fonction est ici à prendre au sens mathématique du terme (lambda calcul) : une fonction appelée avec les mêmes paramètres produit le même résultat en toute circonstance.

Comme on peut le constater sur la figure 4.2, les langages de types Bytecode sont impératifs. L’objectif est de se rapprocher des langages machines qui sont également impératifs à cause de l’architecture des processeurs de type Von Neumann.

P

Python est un langage multiparadigme : impératif, objet et fonctionnel.

■ **Exemple 5 — Langages et paradigmes.** La plupart des langages contemporains sont mul-

tiparadigmes, c'est à dire qu'ils permettent de programmer de différentes manières.

- C : impératif, structuré et procédural,
- C++ : impératif, structuré, procédural, objet et fonctionnel,
- Java : impératif, objet, fonctionnel,
- Javascript : fonctionnel, objet (prototype), script,
- Smalltalk : objet,
- Prolog : logique,
- SQL : déclaratif, requêtage,
- Ocaml : fonctionnel, impératif et objet,
- Haskell : fonctionnel pur.

D Systèmes d'exploitations → HORS PROGRAMME

■ **Définition 19 — Système d'exploitation.** Un système d'exploitation est un ensemble de logiciels qui forme une interface abstraite entre les applications et la machine. Un système d'exploitation accède directement au matériel de l'ordinateur, c'est à dire les systèmes électroniques constitutifs comme le processeur, les mémoires et les périphériques (écran, clavier, trackpad...).

Les autres applications de l'ordinateur s'appuient sur le système d'exploitation pour accéder au matériel.

 **Vocabulary 3 — Operating System (OS)** ↪ Système d'exploitation.

Un système d'exploitation :

- garantit ainsi l'intégrité et la cohérence de la machine,
- masque la complexité du matériel aux utilisateurs, aux développeurs et aux applications en présentant une abstraction du matériel,
- gère le matériel en appliquant certaines stratégies prédéfinies (allocation mémoire, ordonnancement des processus),
- isole les applications les unes des autres.

 **Vocabulary 4 — Software, hardware** ↪ En anglais, on distingue les logiciels (software) du matériel électronique qui lui permet de fonctionner (hardware).

Un système d'exploitation est composé (cf. figure 4.4) :

- d'un système de fichiers (file system),
- de pilotes (drivers) spécifiques aux matériels électroniques,
- d'un ordonnanceur (scheduler) de processus,
- d'un gestionnaire de mémoire.



FIGURE 4.4 – Positionnement du système d'exploitation et des bibliothèques logicielles entre les éléments logiciels et le matériel électronique

L'ordonnanceur et le gestionnaire de mémoire appliquent des stratégies sophistiquées afin de garantir une certaine équité entre les processus qui s'exécutent, l'accès aux ressources de la machine et l'allocation de l'espace mémoire.

Un système d'exploitation offre la plupart du temps deux modes de fonctionnement différents :

- Le mode utilisateur, non privilégié : dans cet espace, l'utilisateur peut utiliser les ressources du système et l'interface du système d'exploitation. Cependant, il n'accède à aucun matériel de manière privilégiée. C'est l'espace de développement et de fonctionnement des applications et des services. Cet espace garantit que tout accès au système logiciel/matériel est légal, car contrôlé par le système d'exploitation.
- Le mode administrateur, super-utilisateur, privilégié (KERNEL SPACE) : dans cet espace, l'administrateur peut programmer le système d'exploitation, le modifier et accéder au matériel, sans restrictions.

■ **Exemple 6 — Quelques systèmes d'exploitation.** Parmi les plus courants, on peut citer :

- Windows,
- Mac OS X,
- Linux,

- la famille de Unix BSD.

Dans des domaines plus spécifiques, comme l'informatique temps réel ou les systèmes embarqués, on peut trouver d'autres systèmes d'exploitation comme Windows CE, FreeRTOS ou Xenomai.

■ **Définition 20 — Système de fichier.** Un système de fichier est une organisation hiérarchique des fichiers mis en œuvre par un système d'exploitation. Cette organisation se présente sous la forme d'un arbre (systèmes Unix) ou d'une forêt (systèmes Windows).

Lorsqu'on travaille sur un ordinateur, il est toujours important de savoir où l'on se trouve et où l'on peut aller dans un système de fichiers.

■ **Définition 21 — Chemin.** Un chemin est une chaîne de caractères qui permet de décrire la position d'un fichier ou d'un répertoire dans un système de fichier.

On peut lire directement un chemin sur l'arborescence de la figure 4.5. **La racine de l'arbre est dénotée /.** Ainsi, le chemin vers le répertoire personnel d'Olivier est `/home/olivier/`. Le sous-répertoire `python/tp1/` est désigné par le chemin absolu `/home/olivier/Desktop/python/tp1/`.

■ **Définition 22 — Chemin absolu.** Un chemin absolu est un chemin qui décrit la ressource désigné sans équivoque possible depuis la racine du système de fichier. Un chemin absolu commence par `/`. Par exemple, `/home/olivier/`.

■ **Définition 23 — Chemin relatif.** Un chemin relatif est un chemin qui est désigne une ressource par rapport à l'endroit où l'on se trouve dans le système de fichiers. Il ne commence pas par un `/`. Par exemple, `python/tp1`.

Lorsque l'on ouvre un environnement de développement comme Pyzo, celui-ci se place dans un répertoire de travail. C'est dans ce répertoire que sont lancés les commandes et les scripts Python. Il dépend de la configuration du système sur lequel on travaille. On peut éventuellement en changer si besoin, mais tout d'abord quel est-il?

Le module `os` est une bibliothèque qui permet d'interagir avec le système d'exploitation d'un ordinateur. La commande `import os` permet d'utiliser les commandes de ce module.

Par exemple pour connaître le répertoire dans lequel on se trouve on utilise l'instruction `os.getcwd()`. Il faut remarquer la notation pointée qui permet d'accéder à une fonction en particulier du module `os`, le nom de la fonction qui signifie *get current working directory* ainsi que les parenthèses qui permettent d'exécuter la fonction. Dans un script Python, pour afficher le résultat, il est nécessaire d'appeler la commande `print` de la manière suivante `print(os.getcwd())`. L'affichage est automatique dans le shell interactif.

E Bibliothèques logicielles

Tous les langages modernes possèdent des bibliothèques logicielles qui permettent d'accélérer le développement. Ce sont des collections de fonctions ou de classes réutilisables direc-



FIGURE 4.5 – Arborescence typique d'un système de fichier Unix/Linux/Mac OS X

tement par le développeur. Il s'agit de gagner du temps et de ne pas réinventer la poudre. Elles permettent souvent d'adresser un domaine particulier de l'informatique (calcul, traitement des images, création de graphiques) ou de simplifier les interfaces (interagir avec un système d'exploitation ou un périphérique (image, son, réseaux)).

Comme de nombreux langages, Python possède un grand nombre de bibliothèques¹. Les

1. On parle de l'Application Program Interface (API) Python

épreuves des concours se basent sur le cœur du langage et quelques bibliothèques. Parmi elles on peut citer :

- math,
- random,
- numpy.

P Il faut bien faire à attention à **respecter les consignes des épreuves** de concours car ces épreuves peuvent être de nature légèrement différente selon les concours. Si le sujet précise qu'on ne doit pas utiliser numpy, alors il ne faut pas l'utiliser. Si le sujet précise qu'on doit utiliser une fonction d'une bibliothèque, alors il faut l'utiliser!

 **Le paragraphe précédent est important pour l'épreuve d'informatique!**

P En python, on peut choisir d'utiliser tout ou partie d'une bibliothèque et plusieurs syntaxes sont possibles. Il faut les connaître (cf. codes 4.2, 4.3 et 4.4) et **ne pas les mélanger** : au concours, ces points sont faciles à obtenir, alors faites attention!

 **Le paragraphe précédent est important pour l'épreuve d'informatique!**

Code 4.2 – Importer un module et l'utiliser

```
1 import math
2
3 a = math.sqrt(math.log(7))
```

Code 4.3 – Importer toutes les fonctions d'un module et en utiliser certaines

```
1 from math import *
2
3 a = sqrt(log(7))
```

Code 4.4 – Importer quelques fonctions d'un module et les utiliser

```
1 from math import sqrt, log
2
3 a = sqrt(log(7))
```

R D'un point de vue mémoire, la dernière syntaxe est plus parcimonieuse puisqu'elle n'importe en mémoire que les fonctions dont le code a besoin.

Deuxième partie

Semestre 1

5

TYPES ET OPÉRATEURS

À la fin de ce chapitre, je sais :

- ☒ Utiliser et identifier les types simples (int, float, boolean, complex)
- ☒ Utiliser les opérateurs en lien avec les types simples numériques
- ☒ Utiliser une variable de type simple
- ☒ Reconnaître les principaux mots-clefs du langage Python

A Types

L'informatique traite l'information. Dans ce but, il faut être capable de stocker l'information sous une forme accessible au traitement informatique. Selon la nature de l'information, un type de données différent est choisi pour la représenter.

■ **Définition 24 — Typage.** Le typage désigne l'action de choisir une représentation à une donnée selon ses caractéristiques. Cette représentation est nommée type. Le typage est effectué soit par le programmeur, soit par le compilateur soit par l'interpréteur.

■ **Définition 25 — Type simple.** Les types simples correspondent à des informations comme les nombres, des constantes ou les valeurs booléennes.

Les types simples en Python sont :

- `int` permet de représenter un sous-ensemble des entiers relatifs,
- `float` permet de représenter un sous-ensemble des décimaux,
- `complex` permet de représenter un sous-ensemble des nombres complexes,
- `bool` permet de représenter une valeur booléenne, vrai ou faux.

Les codes 5.1 donnent des exemples d'usage de ces types en Python. Il faut noter que, comme de nombreux langages modernes, Python est un langage multiparadigme. Il implé-

mente notamment le paradigme objet, c'est pourquoi les types simples sont des classes d'objet.

Code 5.1 – Types simples

```
1 print(type(42))    # <class 'int'> an integer
2 print(type(2.37))  # <class 'float'> a floating point number
3 print(type(3-2j))  # <class 'complex'> a complex number
4 print(True, type(True))  # True <class 'bool'>
5 print(False, type(False)) # False <class 'bool'>
```

■ **Définition 26 — Inférence de type** . L'inférence de type est une action d'un compilateur ou d'un interpréteur qui permet de typer une données automatiquement d'après sa nature.

■ **Définition 27 — Typage explicite** . Un langage est dit à typage explicite s'il exige de toute donnée qu'elle soit déclarée selon un type. Le contraire est un typage **implicite**.

■ **Définition 28 — Typage dynamique** . Un langage est dit à typage dynamique si une variable peut changer de type au cours de l'exécution du programme. Inversement, on parle de typage **statique**.

Dans l'exemple ci-dessus, Python fait donc de l'inférence de type, puisque la donnée 42 est interprétée comme un `int`, 2.37 comme un `float`...

 Python est un langage à typage implicite et dynamique.

■ **Définition 29 — Transtypage.** Transtyper une variable c'est modifier son type en opérant une conversion de la donnée.

 **Vocabulary 5 — Type Casting** ⇔ En anglais, le transtypage est désigné par les termes *type casting* ou *type conversion*.

 En Python, on peut transtyper une variable numérique en utilisant un constructeur numérique `int` ou `float`. Par exemple :

```
1     b = float(3)
2     print(b, type(b))  # 3.0 <class 'float'>
3     c = int(3.56)
4     print(c, type(c))  # 3 <class 'int'>
```

■ **Définition 30 — Types composés.** Un type composé est un type de données qui agrège des types simples dans un objet homogène (tableau, énumération) ou inhomogène (structure, union). Dans les types composés (in)homogènes, les données (ne) sont (pas) toutes du

■ même type.

P En Python les types composés sont les chaînes des caractères, les tuples, les listes, les ensembles et les dictionnaires.

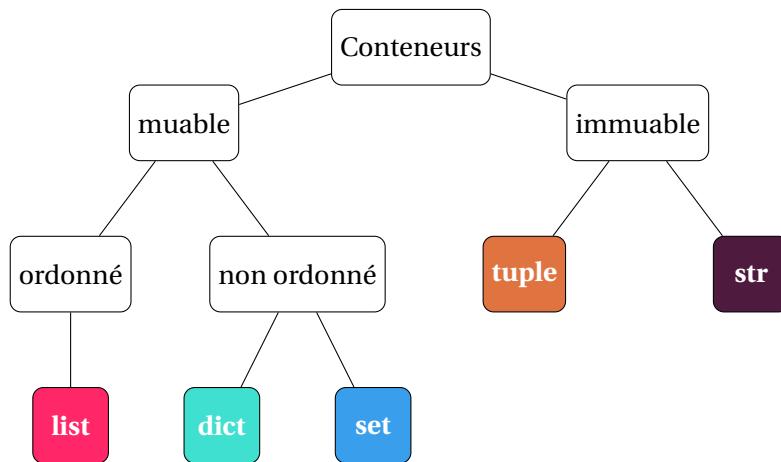


FIGURE 5.1 – Types de données composés du langage Python : les conteneurs

B Opérateurs

Les opérateurs peuvent être classés en catégories selon la nature de leur action, le type d'opérandes ou le nombre d'opérandes dont ils ont besoin.

a Opérateurs arithmétiques

Ce sont les transpositions informatiques des opérateurs mathématiques : ils agissent sur des représentations de nombre en machine, les types numériques.

- `a + b` l'addition,
- `a - b` la soustraction,
- `a * b` la multiplication,
- `a // b` la division entière,
- `a % b` l'opération modulo,
- `a / b` la division,
- `a ** n` l'élévation à la puissance `n` ou exponentiation.


Vocabulary 6 — Arithmetic operators ↗ En anglais, on les désigne par :

- Addition,
- Subtraction,
- Multiplication,
- Floor division,
- Modulus,
- Division,
- Exponentiation.

Merci le latin et Guillaume le Conquérant.

P Les opérateurs `+`, `-` et `*` s'appliquent indifféremment aux types `int` et `float`. Néanmoins, le résultat d'une opération entre `int` sera un `int`. Le résultat est un `float` si l'une des deux opérandes au moins en est un. On parle alors de transtypage implicite, les `int` sont transtypés par l'interpréteur en `float` pour réaliser l'opération, de manière transparente.

L'opérateur de division `/` renvoie en un `float` même si les opérandes sont entières. L'opérateur `//` peut renvoyer un `int` si les deux opérandes sont entières. Cette opération est en fait le quotient de la division euclidienne ou la partie entière du résultat de `/`, d'où le nom en anglais.

L'opérateur `**` s'applique aux `int` et aux `float` et renvoie un `int` si toutes les opérandes sont un `int`, un `float` sinon.

L'opérateur `%` renvoie généralement le reste de la division euclidienne de `a` par `b`. Cependant, lorsque les deux opérandes sont négatives, le modulo résulte en nombre négatif afin d'avoir également $a = (a//b)* b + a\%b$.

Tous les langages n'adoptent pas forcément les mêmes conventions que Python, il faut donc rester vigilant.

La syntaxe infixe de ces opérateurs fait qu'il peut être ambigu d'écrire certaines expressions. Par exemple, comment interpréter `2 + 3 * 4`? Doit-on calculer d'abord l'addition `2+3` puis multiplier le résultat? Ou bien doit-on effectuer la multiplication d'abord? Le résultat ne sera pas le même. Quelle est la bonne opération? Quel opérateur doit-on appliquer en premier? Sans parenthèses, cette ambiguïté demeure à moins qu'on ne décide d'attribuer une priorité différente aux opérateurs. C'est ce qui est fait par la plupart des langages.

P La priorité des opérateurs en Python est définie comme indiqué sur le tableau 5.1. C'est pourquoi l'expression `2 + 3 * 4` est évaluée en `14a`. De plus, lorsque le degré de priorité est identique, on associe en priorité de la gauche vers la droite. Ainsi, l'expression `15 / 5 * 2` est évaluée en `6b`.

a. et non pas 24

b. et non pas 1.5

Priorité	Opérateurs	Description
1	()	parenthèses
2	**	exponentiation
3	+x -x ~x	plus et moins unaire, négation bits à bits
4	* / // %	multiplication, division division entière, modulo
5	+ -	addition, soustraction
6	<< >> >>= <<=	décalage binaire
7	&	et bits à bits
8	^	ou exclusif bits à bits
9		ou bits à bits
10	==, != >, >=, <, <= is, is not, in, not in	identité comparaison appartenance
11	not	négation logique
12	and	et logique
13	or	ou logique

TABLE 5.1 – Priorités des opérateurs en Python : dans l'ordre d'apparition, du plus prioritaire (1) au moins prioritaire (13)

R Attention : certains opérateurs apparaissent identiques syntaxiquement mais se comportent différemment selon le type de donnée. Par exemple * ou + sont des opérateurs qui peuvent également agir sur les chaînes de caractères ou sur les listes. Dans ce cas, ils possèdent une autre signification. On dit qu'on a surchargé ces opérateurs.

b Opérateurs logiques

■ **Définition 31 — Opérateur logique.** Les opérateurs logiques produisent une valeur booléenne à partir d'autres valeurs booléennes en les combinant. Ils prennent le plus souvent deux opérandes.

Les opérateurs logiques Python sont :

- a and b la conjonction, renvoie `True` si a et b sont tous les deux à vrais,
- a or b la disjonction, renvoie `True` si a ou b sont vrais,
- not a la négation, renvoie `True` si a est faux, `False` sinon.

P En Python les valeurs booléennes sont `True` et `False`. Ce sont deux objets qui peuvent être interprétées numériquement par 0 ou 1.

Il faut noter également que d'autres objets peuvent être interprétés comme `False` :

- `None` qui est un l'objet qui représente *rien*,
- les représentations de zéro pour les types numériques dont `int`, `float`,
- les séquences et collections vides : `""`, `()`, `[]`, `set()`, `range(0)`

M **Méthode 2 — Tester un booléen** Pour tester la valeur d'un booléen `a`, on n'écrit jamais `if a == True`, mais simplement `if a`. De même, pour tester le cas faux, on écrira `if not a`.

c Opérateurs d'affectation

■ **Définition 32 — Affectation.** L'affectation est l'opération qui consiste à assigner une valeur à une variable et donc de modifier son état en mémoire.

Le fait que l'affectation soit une instruction est caractéristique des langages impératifs. En Python, il existe plusieurs opérateurs d'affectation :

- `a = 3` affectation simple,
- `a += 3` affectation simple combinée avec une addition équivalent à `a = a + 3`,
- `a -= 3` affectation simple combinée avec une soustraction équivalent à `a = a - 3`.

L'affectation combinée évite à l'interpréteur de créer une variable intermédiaire pour le calcul. L'opération se fait en place, c'est à dire sur l'espace mémoire même associé à la variable.

C Opérateurs sur les chaînes de caractères

Les chaînes de caractères sont des objets immuables en Python. Leurs valeurs sont initialisées en utilisant les guillemets `""`. Certains symboles sont utilisés pour désigner plusieurs opérateurs différents. C'est le cas par exemple de `+` et `*` qui peuvent désigner des opérateurs sur les chaînes de caractères. C'est ce qu'illustre l'exemple suivant

```
1  s = "Z comme "
2  print(s + " Zorglub")      # Z comme Zorglub
3  print(s + "42")           # Z comme 42
4  print(s*3)                # Z comme Z comme Z comme
```

 **Vocabulary 7 — Double and simple quotes ↴** Les guillemets `""` correspondent au mot anglais *double quote*. Il existe aussi le symbole apostrophe `'`, *simple quote* en anglais, qui permet également d'initialiser des chaînes de caractères.

D Une variable Python est une référence

Tout est dans le titre de la section, mais je vais l'écrire comme une règle d'or :

P En Python, une variable est une référence contenant l'adresse d'un objet en mémoire.

■ **Définition 33 — Type immuable.** Un type immuable est un type de donnée que l'on ne peut pas modifier. Inversement, un type mutable est un type dont on peut modifier la valeur.

P En Python, les types immuables sont :

- les types simples numériques : `int`, `float`, `bool`,
- les `str`,
- les `tuple`.

Les types muables sont :

- les listes `list`,
- les dictionnaire `dict`,
- les ensembles `set`.

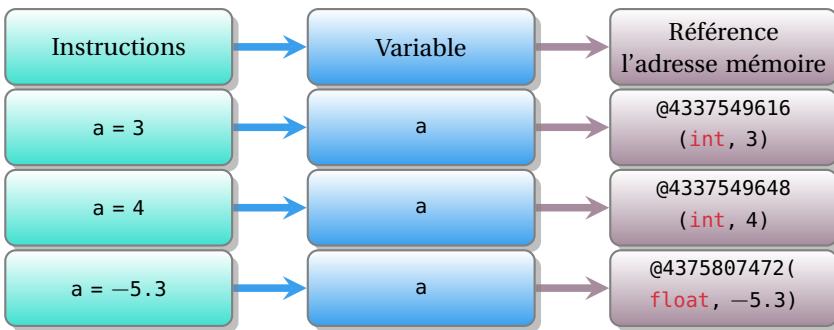
Ces types muables et immuables combinés à la règle d'or précédente permettent d'expliquer énormément de codes qui peuvent parfois sembler très peu clairs au débutant. C'est par exemple ce qui explique pourquoi l'affectation ne se comporte pas de la même manière avec toutes les variables, ce que l'on observera au cours des TP de l'année.

■ **Exemple 7 — Affectation d'immuables.** Un entier est une donnée immuable : on ne peut pas en modifier la valeur, 42 vaut 42. De même pour un nombre flottant ou un booléen, vrai reste vrai et faux, faux... En Python, une variable à qui on a affecté une donnée référence un objet d'un certain type représentant cette donnée en mémoire. Si on affecte une nouvelle donnée à une variable immuable, le type étant immuable, c'est donc à la référence de changer. L'interpréteur alloue donc un nouvel espace mémoire à cette donnée et la variable référence ce nouvel objet. C'est ce qu'illustrent les figures 5.2 et 5.3 avec un type `int` immuable et un type `list` mutable.

L'ancienne valeur peut demeurer un moment en mémoire, jusqu'à ce que le ramasse-miette de l'interpréteur (Garbage Collector) désalloue cet espace mémoire pour libérer la mémoire, si cette valeur n'est plus référencée par aucune variable.

E Mots-clefs du langage

Les mots-clefs d'un langage sont des mots réservés, c'est à dire qu'on ne peut et qu'on ne doit pas les utiliser pour nommer des variables pour des fonctions. Ils sont rassemblés sur le tableau 5.2.



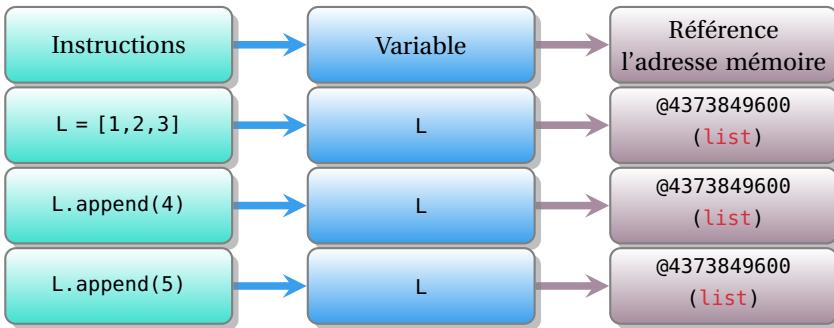
Code 5.2 – Affectations de type immuable

```

1 a = 3
2 print(a, id(a)) #3 4337549616
3 a = 4
4 print(a, id(a)) #4 4337549648
5 a = -5.3
6 print(a, id(a)) #5.3 4375807472

```

FIGURE 5.2 – Variable de type immuable, affectation et référencement en mémoire (à gauche) et programme Python pour la visualisation des adresses en mémoire (à droite)



Code 5.3 – Affectations de type mutable

```

1 L = [1, 2, 3]
2 print(L, id(L))
3 # [1, 2, 3] 4373849600
4 L.append(4)
5 print(L, id(L))
6 # [1, 2, 3, 4] 4373849600
7 L.append(5)
8 print(L, id(L))
9 # [1, 2, 3, 4, 5] 4373849600

```

FIGURE 5.3 – Variable de type mutable, affectation et référencement en mémoire (à gauche) et programme Python pour la visualisation des adresses en mémoire (à droite)

Mot-clef	Usage
<code>if</code> <code>elif</code> <code>else</code> <code>for</code> <code>while</code> <code>break</code> <code>continue</code> <code>pass</code>	structure conditionnelle structure conditionnelle structure conditionnelle pour créer une boucle pour pour créer une boucle tant que sortir d'une boucle continuer la boucle pour ne rien faire
<code>and</code> <code>not</code> <code>or</code> <code>False</code> <code>True</code> <code>assert</code> <code>None</code> <code>in</code> <code>is</code>	et logique négation logique ou logique valeur booléenne faux valeur booléenne vrai créer une assertion en programmation défensive constante pour représenter le rien pour vérifier si une valeur est présente dans une séquence pour tester l'égalité de deux variables
<code>from</code> <code>import</code> <code>as</code>	pour importer tout ou partie d'un module pour importer un module pour créer un alias
<code>def</code> <code>return</code> <code>lambda</code>	pour définir une fonction pour sortir d'une fonction et renvoyer une valeur pour créer des fonctions anonymes
<code>raise</code> <code>try</code> <code>except</code> <code>finally</code> <code>with</code>	pour lever une exception pour gérer les exception pour gérer les exceptions pour gérer les exceptions pour gérer les exceptions
<code>class</code> <code>del</code> <code>global</code> <code>nonlocal</code> <code>yield</code>	définir une classe pour supprimer un objet pour déclarer une variable globale et l'utiliser pour déclarer une variable non locale pour créer une coroutine (hors programme)

TABLE 5.2 – Mots-clefs du langage Python

6

STRUCTURONS

À la fin de ce chapitre, je sais :

- lire et interpréter un code Python structuré
- indenter et enchaîner correctement les blocs Python
- coder une structure conditionnelle (et l'expression conditionnelle)
- coder une boucle avec un range sur l'intervalle d'entiers ouvert $[0, n[$
- coder une boucle tant que en précisant la condition de sortie
- coder une fonction (paramètres formels, effectifs, retour)

A Anatomie d'un programme Python

■ **Définition 34 — Instruction.** Une instruction dans un programme informatique est un mot qui explique à l'ordinateur l'action qu'il doit exécuter.

■ **Définition 35 — Programme informatique.** Un programme informatique est une suite d'instructions.

Un programme informatique se présente sous la forme d'un fichier texte organisé en parties et sous-parties, comme le montre le code 6.1.

Code 6.1 – Anatomie d'un programme Python (variables globales, fonctions, indentation, blocs, programme principal)

```
1 from random import randint
2
3 N = 42                      # Global variables
4
5 def produit(a, b):    # BEGIN BLOC :
6     p = 0                  # INDENTATION local variable
```

```

7      c = 0          # INDENTATION local variable
8      while c < a:  # INDENTATION instruction BEGIN BLOC :
9          p = p + b # INDENTATION INDENTATION instruction
10         c = c + 1 # INDENTATION INDENTATION instruction
11     return p       # INDENTATION instruction
12
13
14
15 if __name__ == "__main__":    # MAIN PROGRAM
16     length = 10
17     results = []
18     while len(results) < length:
19         p = produit(randint(1, N), randint(1, N))
20         if p > N:
21             results.append(p)
22     print(results)
23 # —> [168, 280, 261, 740, 150, 507, 242, 288, 99, 84]

```

P Une des caractéristiques principales du langage Python est de donner un sens à l'indentation dans le code, c'est à dire que **les espaces en début de ligne ont une signification** : ils désignent un bloc d'instructions.

■ **Définition 36 — Bloc d'instructions Python.** Une suite d'instructions indentées au **même niveau** constituent un bloc d'instructions.

■ **Définition 37 — Programme principal.** Le programme principal est le point d'entrée de l'exécution d'un programme informatique : c'est par là que tout commence.

P En Python le programme principal s'appelle `__main__`. Il peut être introduit par les instructions `if __name__ == "__main__":`. L'intérêt de cette instruction est de permettre au fichier de se comporter soit comme un programme à exécuter, soit comme un module que l'on peut alors importer dans un autre programme sans exécuter le programme principal.

B Programmation structurée

■ **Définition 38 — Programmation structurée.** Dans le cadre de la programmation structurée, un programme informatique peut-être développé à partir de trois structures :

1. les séquences d'instructions,
2. les structures alternatives,
3. les structures itératives.

Ce paradigme de programmation est une déclinaison du paradigme impératif. Il est proche du fonctionnement des processeurs actuels.

a Séquences d'instructions

Tous les langages de programmation dispose d'un moyen pour exprimer une séquence d'instruction. La plupart du temps on utilise des accolades (en Java, en C), parfois des délimiteurs de type begin ... end (Ocaml). En Python, le point virgule ; et l'indentation (cf. définition 36) permettent de constituer une séquence d'instruction.

b Structures conditionnelles

■ **Définition 39 — Instruction de test ou test.** Une instruction de test est une opération dont le résultat est un booléen.

■ **Définition 40 — Instruction conditionnelle.** L'exécution d'une instruction conditionnelle est soumise à la validité d'un test effectué en amont de l'instruction.

P En Python, il existe deux structures alternatives : le bloc `if` `then` `elif` `else` ou bien l'évaluation sous condition d'une expression `expr if then else`.

Bloc conditionnel en Python

Le code 6.2 recense les syntaxes possibles.

R Lorsqu'on utilise les structures conditionnelles, il est nécessaire d'être vigilant à ce que les conditions s'excluent mutuellement. Dans le cas contraire, c'est le premier test validé qui déclenche le branchement et l'exécution d'une instruction conditionnelle. On peut utiliser `if` sans `else` ou/et sans `elif`, mais il faut être logique dans l'élaboration des conditions pour que l'exécution aboutisse à un algorithme cohérent.

Dans l'exemple 6.2, il n'est pas souhaitable par exemple que les tests soient écrits ainsi `if age <= 50 : et elif 50 <= age <= 60 :.` Logiquement, cela signifierait que si `age` valait 50, on devrait afficher "Still young !" et "Has to work...". Or, seule la première chaîne de caractères sera affichée dans ce cas.

Code 6.2 – blocs conditionnels

```

1 age = 57
2 if age < 50:
3     print("Great !")
4
5 if age < 50:
6     print("Still young !")
7 else:
8     print("Not dead yet !")
9
10 if age < 50:
11     print("Still young !")
12 elif 50 <= age <= 67:

```

```

13     print("Has to work...")
14 else:
15     print("Get some rest !")

```

Expressions conditionnelles en Python

Cette syntaxe est à rapprocher du paradigme fonctionnel. Elle permet de conditionner l'évaluation d'une expression. Le code 6.3 décrit la syntaxe Python.

Code 6.3 – Expression conditionnelle

```

1 autruche = True
2 age = 45 if autruche else 46
3 print(age) # 45

```

c Structures itératives

■ **Définition 41 — Structures itératives ou boucles.** Les structures itératives ou boucles permettent de répéter une séquence d'instructions un certain nombre de fois.

On distingue les boucles conditionnelles des boucles inconditionnelles.

Boucles inconditionnelles POUR

Les boucles inconditionnelles permettent de répéter une séquence d'instructions un nombre explicite de fois. Cela signifie que, lorsqu'on programme une boucle inconditionnelle, on connaît ce nombre et on peut donc l'écrire dans le code.

Algorithme 2 Produit de deux nombres entiers avec boucle inconditionnelle

```

1: a ← 3
2: b ← 4
3: p ← 0
4: pour k de 0 à b – 1 répéter                                ▷ b est un entier.
5:     p ← p + a

```

Code 6.4 – Boucle inconditionnelle for

```

1 a = 3
2 b = 4
3 p = 0
4 for k in range(b):
5     p += a
6 print(p) # 12

```

P En Python, la fonction `range` permet d'exprimer la plage de variation de la variable de boucle. **Les paramètres de cette fonction sont des entiers** et il faut être vigilant car ceci est une source majeure de pertes de points à l'épreuve d'informatique commune. Le résultat est une séquence immuable.

La signature de cette fonction est `range(start, stop[, step])`. Si on omet de préciser `start`, alors on commence la séquence par zéro. Si on omet de préciser `step`, le pas par défaut vaut un. **La valeur `stop` est toujours exclue de la séquence.** Donc `range(10)` va renvoyer une séquence `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` contenant **dix** éléments, comme le montre le code 6.5.

 Le paragraphe précédent est important pour l'épreuve d'informatique!

Code 6.5 – Utilisation de `range`

```

1 print("Simple range")
2 for i in range(10):
3     print(i, end="\t")      # 0 1 2 3 4 5 6 7 8 9
4
5 print("\nStart Stop range")
6 for i in range(3, 10):
7     print(i, end="\t")      # 3 4 5 6 7 8 9
8
9 print("\nStart Stop Step range")
10 for i in range(1, 10, 2):
11     print(i, end="\t")      # 1 3 5 7 9

```

Boucles conditionnelles TANT QUE

Les boucles conditionnelles permettent de conditionner la répétition d'une séquence d'instructions à une condition exprimée par un test. Cela signifie que, lorsqu'on programme une boucle conditionnelle, on ne sait pas nécessairement le nombre de répétitions à exécuter. Mais, on sait exprimer la condition d'arrêt des répétitions.

Algorithme 3 Produit de deux nombres entiers avec boucle conditionnelle

```

1:  $a \leftarrow 3$                                      ▷ On veut calculer  $a \times b$ 
2:  $b \leftarrow 4$ 
3:  $k \leftarrow 0$                                      ▷ Initialiser la variable de boucle
4:  $p \leftarrow 0$                                      ▷  $p$  contiendra le résultat
5: tant que  $k < b$  répéter
6:    $p \leftarrow p + a$ 
7:    $k \leftarrow k + 1$                                 ▷ Incrémenter la variable de boucle

```

Code 6.6 – Boucle conditionnelle `while`

```

1 a = 3
2 b = 4
3 k = 0

```

```

4 p = 0
5 while k < b :
6     p += a
7     k += 1
8 print(p) # 12

```

R D'une manière générale, on privilégiera l'utilisation des boucles `for`. On choisira une boucle `while` dans les cas où :

- l'on ne peut pas exprimer simplement le nombre de répétitions nécessaires,
- les conditions d'arrêt mettent en jeux des nombres flottants,
- on doit à la fois balayer tout un ensemble et respecter une certaine condition d'arrêt.

d Pourquoi compte-t-on à partir de 0 en informatique? --> HORS PROGRAMME

Cette question mérite qu'on s'y attarde un peu car, de nouveau, ceci est à l'origine de pertes de points significatives lors de l'épreuve d'informatique commune. Cette question est liée à une autre : pourquoi spécifie-t-on le paramètre `stop` dans la fonction `range` comme la borne supérieure plutôt que le maximum de la séquence ? Deux questions qui peuvent empêcher de dormir...

Dans une lettre restée célèbre [7], Dijkstra¹ explique les raisons théoriques qui ont poussé les informaticiens à adopter cette convention, car il s'agit bien là d'une convention, on aurait pu choisir de procéder différemment.

La convention choisie pour délimiter les bornes d'une séquence de nombres entiers est la suivante : $\min \leq i < \sup$, le minimum étant inclus et la borne supérieure exclue. Par exemple, la séquence $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$ est exprimée $0 \leq i < 13$.

Tout d'abord, Dijkstra observe qu'une séquence de nombres entiers naturels possède toujours un plus petit élément par lequel on commence ou finit. Donc l'exclure n'est pas judicieux. Ceci justifie le choix du symbole \leq . Ensuite, il observe qu'on pourrait très bien commencer à indexer les séquences par l'élément numéro 1. Mais si on adopte cette convention, alors la plage des indices s'exprime $1 \leq i < N + 1$, ce qui n'est pas très élégant et ne donne pas d'emblée la longueur de la séquence. Par contre, si on commence à zéro, alors cette plage des indices s'écrit $0 \leq i < N$ ce qui présente à la fois l'avantage :

- d'indiquer directement la longueur de la séquence (N),
- de ne pas à avoir à ajouter en permanence $+1$ pour préciser les plages d'indices.

Une autre raison, plus électronicienne, milite pour commencer à zéro. Un tableau en mémoire commence à une case numérotée. Quand on va chercher dans la mémoire le premier élément du tableau, on va donc récupérer cette case. Si on souhaite récupérer le second élément, on incrémente le numéro de la case mémoire de un. Le premier élément se trouve donc à l'indice 0 et le second à l'indice 1.

1. prononcer DailleKeStra

C'est pourquoi, comme Dijkstra, il faut considérer zéro comme le nombre entier le plus naturel pour commencer à dénombrer! D'ailleurs, le mot *zéro* vient de l'italien *zefiro* et qui provient lui-même de l'arabe *sifr* qui signifie à l'origine le vide, le néant. Quoi de plus naturel que le vide pour commencer?

C Fonctions Python

Dans un programme, il est inutile voire dangereux de répéter du code déjà écrit : cela nuit à la fiabilité², à la lisibilité³ et à la générativité. C'est pour éviter cela que le paradigme procédural a été inventé.

■ **Définition 42 — Paradigme procédural.** Le paradigme procédural incite à regrouper dans un même code les fonctionnalités importantes d'un programme dans un but d'intelligibilité et de réutilisabilité du code écrit. Les fonctionnalités sont regroupées dans des routines ou des procédures.

 En Python, les routines sont appelées fonctions. On les appellera donc ainsi par la suite.

■ **Définition 43 — Fonctions Python.** Une fonction Python est caractérisée par :

- son nom,
- ses paramètres formels,
- une valeur de retour (optionnelle).

■ **Définition 44 — Appel d'une fonction.** Appeler une fonction, c'est demander son exécution. Cela s'effectue en respectant le prototype de la fonction et en utilisant l'opérateur () qui demande l'exécution de la fonction à l'interpréteur.

■ **Définition 45 — Prototype d'une fonction.** Le prototype d'une fonction sert de mode d'emploi au développeur. Dans la plupart des langages, le prototype d'une fonction est constitué du nom de la fonction, des paramètres formels qu'elle accepte et de la valeur renvoyée.

■ **Définition 46 — Paramètres formels.** Ce sont les paramètres qui sont utilisés pour écrire la fonction. Ils sont situés dans le prototype. Ils sont formellement exigés par la fonction pour opérer.

■ **Définition 47 — Paramètres effectifs.** Ce sont les paramètres qui apparaissent lors de

2. car on peut se tromper en recopiant

3. car cela allonge le code inutilement

l'appel d'un fonction, c'est à dire lorsqu'on utilise la fonction. Il sont effectivement transmis à la fonction.

■ **Exemple 8** — . Le code 6.7 donne un exemple très simple de fonction. Sur cet exemple, on peut noter :

- le prototype de la fonction : `square(a)`,
- le paramètre formel `a`,
- la valeur renvoyée `x`,
- le paramètre effectif `3`.

Code 6.7 – Fonction Python à un paramètre renvoyant un entier

```
1 def square(a):      # prototype, a is a formal parameter
2     x = a * a        # function body
3     return x          # returned value
4
5 nine = square(3)    # 3 is an effective parameter (argument)
6 print(f"Squareing three gives {nine}") # Squareing three gives 9
```

🇬🇧 **Vocabulary 8 — Parameters - arguments** ↗ En anglais, les paramètres formels sont désignés par le terme *parameters*, les paramètres effectifs par *arguments*. Certains emploient ces mots en français également.

Le code 6.8 montre un exemple de procédure, c'est à dire une fonction qui ne renvoie rien. Elle ne fait, en apparence seulement, aucun calcul qu'on puisse sauvegarder. Par contre, elle interagit avec la console, c'est-à-dire avec l'écran de la machine via le système d'exploitation, pour faire afficher des chaînes de caractères⁴.

Code 6.8 – Procédure Python

```
1 def say_hello(name, n):  # prototype
2     for i in range(n):  # function body
3         print(f"Hello {name} !", end="\t")
4     # no returned value
5
6 say_hello("Olivier", 2) # Hello Olivier ! Hello Olivier !
```

Le code 6.9 montre un exemple de fonction avec paramètre optionnel. Si le paramètre `capitalized` n'est pas fourni lors de l'appel de la fonction alors celui-ci se voit attribuer la valeur `True`. S'il existe un paramètre effectif lors de l'appel, c'est celui-ci qui est utilisé.

Ce code montre également qu'avec une structure conditionnelle, il peut exister plusieurs chemins d'exécution avec une valeur retour différente.

Code 6.9 – Fonction avec paramètre optionnel et plusieurs chemin d'exécution avec valeur retour

4. ce que l'on appellera un effet au second semestre.

```

1 def say_my_name(name: str, capitalized=True):
2     # signature with optional parameter
3     if capitalized:
4         return name.capitalize()      # returned value
5     else:
6         return name.upper()        # returned value
7
8 my_name = say_my_name("olivier")    # calling function
9 print(my_name)                   # Olivier
10 my_name = say_my_name("olivier", False)  # calling function
11 print(my_name)                  # OLIVIER

```

D Conventions de nommage

■ **Définition 48 — Conventions.** Règles de conduite adoptées à l'intérieur d'un groupe social. Exemple : en mathématiques, l'inconnue c'est x .

D'une manière générale, en informatique contemporaine, on préfère les conventions aux configurations⁵, car cela fait gagner un temps précieux. Respecter des conventions dans l'écriture des programmes permet leur traitement automatique par d'autres outils pour générer d'autres programmes (tests, vérification, documentation, mise en ligne, interfaçage). Cela permet également aux développeurs de comprendre plus rapidement un code.

L'intelligibilité est certainement la plus grande qualité qu'on puisse exiger d'un code. Aujourd'hui, dans la plupart des entreprises, des normes de codage sont appliquées afin de rendre le travail collaboratif plus efficace. Pour le langage Python, un ensemble de recommandations a été produit (cf. [Python Enhancement Proposals - PEP 8](#)). Donc, si vous ne savez pas trop comment faire, il est toujours possible de s'y référer. Les IDE actuels implémentent ces recommandations et peuvent formater ou proposer des changements conformes aux Python Enhancement Proposals.

M Méthode 3 — Choisir un nom de variable ou de fonction *Mal nommer les choses c'est ajouter au malheur de ce monde*^a. Alors tâchons de bien les nommer.

À faire :

- choisir des noms de variables et de fonctions en minuscules.
- si plusieurs mots sont nécessaires, mettre un `_` entre les mots,
- préférer un verbe pour les fonctions,
- appeler un chat un chat.

À ne pas faire :

5. c'est à dire des arrangements spécifiques.

- utiliser les lettres l,O ou I pour un nom de variables. Elles sont confondues avec 1 ou 0.
- utiliser autre chose que les caractères ASCII (par exemple des lettres accentuées, des lettres grecques ou des kanjis),
- appeler une variable d'après un mot clef du langage Python (lambda, list, dict, set, global, try, True...)

a. citation apparemment de Brice Parrain dans une réflexion sur l'étranger d'Albert Camus. À vérifier cependant.

■ **Exemple 9 — Noms courants de variables.** Voici des noms possibles et courants pour :

- les types `int` : `i, j, k, m, n, p, q, a, b`
- les types `float` : `x, y, z, u, v, w`
- les accumulateurs : `acc, s, somme, prod, produit, product`
- les pas ^a : `dt, dx, dy, step, pas,`
- les chaînes de caractères : `c, ch, s,`
- les listes : `L, results, values,`
- les dictionnaires : `d, t, ht,`
- les constantes en majuscules : `MAX_SIZE.`

a. c'est à dire la distance entre chaque élément d'un vecteur temporel par exemple

M **Méthode 4 — Faire un commentaire** Un commentaire peut être utile dans une copie pour détailler un point du code ou expliquer un choix d'implémentation que vous avez fait. Mais d'une manière générale, il n'est pas forcément nécessaire, pourvu que votre code soit intelligible. Le choix des noms des variables, le respect de l'indentation et des conventions sont les clefs d'un code intelligible.

Que peut-il arriver de pire à quelqu'un qui raconte une blague? Devoir l'expliquer. Il en est de même du commentaire. Le commentaire peut être utile à l'intelligibilité mais il peut lui nuire également. Par ailleurs, dans le temps, un commentaire peut faire référence à une ligne modifiée ou une variable dont le nom a changé. Dans ce cas là, le commentaire nuit à la compréhension.

On s'attachera donc à :

- n'inscrire que des commentaires utiles et brefs,
- s'assurer qu'ils sont cohérents avec le code,
- à ne pas paraphraser le code.

On préfèrera écrire un code directement lisible. [Sur le site de ce cours](#), vous trouverez des exemples de copies de concours que je vous aide à analyser. Bien écrire et bien nommer permet de gagner des points.

7

LISTES PYTHON

À la fin de ce chapitre, je sais :

- ☒ créer une liste simplement ou en compréhension
- ☒ manipuler une liste pour ajouter, ôter ou sélectionner des éléments
- ☒ tester l'appartenance d'un élément à une liste
- ☒ utiliser la concaténation et le tronçonnage sur une liste
- ☒ itérer sur les éléments d'une liste avec une boucle for
- ☒ coder les algorithmes incontournables (count, max, min, sum, avg)

La liste Python est une collection très ¹ pratique. Elle est à la base de quasiment tous les algorithmes que l'on demande d'implémenter lors des épreuves de concours. C'est une liste **mutable** implémentée par un tableau dynamique ²

A Constructeurs de listes

Cette section s'intéresse à la manière dont on peut créer des listes en Python.

a Crochets

À tout seigneur tout honneur, les crochets sont la voie royale pour créer une liste.

```
1     L = [] # Empty list
2     print(L, type(L)) # [] <class 'list'>
3     L = [1, 2, 3]
4     print(L) # [1, 2, 3]
5     L = [True, False, False]
6     print(L) # [True, False, False]
```

-
1. trop?
 2. au programme du deuxième semestre.

```

7     L = ["cours", "td", "tp"]
8     print(L) # ['cours', 'td', 'tp']
9     L = [[2.3, 4.7], [5.9, 7.1], [1.8, 3.9]] # Nested List !
10    print(L) # [[2.3, 4.7], [5.9, 7.1], [1.8, 3.9]]

```

L'exemple précédent montre que :

- On peut créer des listes de n'importe quel type de donnée. Une liste est un conteneur avant toute chose.
- On peut créer des listes imbriquées (nested list), c'est à dire des listes de listes. Ces dernières sont très appréciées par les créateurs d'épreuves de concours.

b Constructeur list

La fonction `list()`³ permet également de créer une liste à partir de n'importe quel objet itérable. Elle s'utilise le plus souvent pour convertir un autre objet itérable⁴ en liste.

```

1     L = list() # Empty list
2     print(L, type(L)) # [] <class 'list'>
3     L = list("Coucou !")
4     print(L) # ['C', 'o', 'u', 'c', 'o', 'u', ' ', '!']
5     L = list(range(10))
6     print(L) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

c Construire une liste en compréhension

Tout comme les ensembles en mathématiques, les listes peuvent être construites à partir d'une description compréhensible des éléments de la liste. Cette méthode de création de liste est à rapprocher du paradigme fonctionnel.

```

1     L = [ i for i in range(10) ]
2     print(L) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3     L = [ i for i in range(10) if i %2 == 0]
4     print(L) # [0, 2, 4, 6, 8]

```

 Cette méthode de création de liste est puissante mais est très délicate à manipuler. C'est pourquoi il est préférable de ne l'utiliser que si on est vraiment sûr de soi, sinon c'est une perte de points assurée au concours. On peut l'éviter avec une boucle `for` et un `append` et ainsi assurer des points.



Le paragraphe précédent est important pour l'épreuve d'informatique!

-
3. On appelle cette fonction le **constructeur** des objets de type `list`.
 4. un dictionnaire ou une chaîne de caractères par exemple

B Opérations sur une liste

a Longueur d'une liste

La fonction `len` renvoie la longueur d'une séquence. Elle s'utilise donc aussi pour les listes.

```
1     L = [ 1, 2, 3 ]
2     print(len(L)) # 3
```

P

En Python, on peut tester si une liste est vide avec la fonction `len`.

```
1     L = [ 1, 2, 3 ]
2     while len(L) > 0:
3         print(L.pop()) # 3 2 1
```

b Appartenance à une liste

Les mots-clefs `in` et `not in` permettent de tester l'appartenance à une liste et renvoient les booléens correspondants.

```
1     L = [ 1, 2, 3 ]
2     print(2 in L) # True
3     print(42 in L) # False
4     print(42 not in L) # True
```

c Ajouter un élément à une liste

La méthode `append` de la classe `list` permet d'ajouter un élément à la fin d'une liste. Cette méthode modifie la liste.

```
1     L = [ 1, 2, 3 ]
2     L.append(4)
3     print(L) # [1, 2, 3, 4]
4     L.append(5)
5     print(L) # [1, 2, 3, 4, 5]
```

d Retirer le dernier élément d'une liste

La méthode `pop` de la classe `list` permet de retirer le dernier élément ajouté.

```
1     L = [ 1, 2, 3 ]
2     L.pop()
3     print(L) # [1, 2 ]
```

C Concaténation et démultiplication de listes

L'opérateur `+` permet de concaténer une liste à une autre liste pour en créer une nouvelle, c'est à dire de créer une nouvelle liste en fusionnant leurs éléments.

```

1      L = [1, 2, 3]
2      M = [4, 5, 6]
3      N = L + M
4      print(N) # [1, 2, 3, 4, 5, 6]

```

L'opérateur `*` permet de démultiplier un élément dans une liste.

```

1      L = [1] * 10
2      print(L) # [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

```

Il existe également l'opérateur de concaténation et d'affectation `+=` ainsi que l'opérateur de démultiplication et d'affectation `*=`.

```

1      L = [1, 2]
2      L += [3]
3      L *= 4
4      print(L) # [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

```

D Des listes indicables

L'intérêt des listes Python est qu'elles permettent d'accéder à un élément particulier en temps constant⁵. Pour cela, il suffit de connaître l'indice de l'élément dans la liste. Naturellement, comme expliqué dans le cours précédent :

- le premier élément d'une liste est l'élément d'indice `0`,
- le dernier élément d'une liste est l'élément d'indice `len(L) - 1`.

Tout comme les chaînes de caractères, les listes autorisent également les indices négatifs pour identifier un élément à partir de la fin de la chaîne.

```

1      L = [ 1, 2, 3 ]
2      print(L[0]) # 1
3      print(L[len(L) - 1]) # 3
4      print(L[-2]) # 2

```

E Des listes itérables

Une liste Python est itérable, c'est à dire qu'elle peut être l'objet d'une itération via une boucle `for`.

5. C'est parce qu'elles sont implémentées par des tableaux dynamiques.

```

1      L = [ 1, 2, 3 ]
2      for elem in L:
3          print(elem)      # 1 2 3   elem

```

Naturellement, on peut parcourir une liste d'après ses indices :

```

1      L = [ 1, 2, 3 ]
2      for i in range(len(L)):
3          print(L[i])      # 1 2 3

```

L'intérêt de la première syntaxe est qu'on ne peut pas se tromper sur les indices. Le premier inconvénient est que l'on ne peut pas modifier l'élément `elem`. Le second inconvénient est qu'on ne dispose pas l'indice alors qu'on pourrait en avoir besoin...

On choisira donc l'une ou l'autre syntaxe selon qu'il est nécessaire ou pas de disposer de l'indice ou de modifier les éléments.

F Des listes tronçonnables

 **Vocabulary 9 — Slicing ↵ ↵ Tronçonnage**

Tout comme les chaînes de caractères, les listes sont tronçonnables.

```

1      L = [1, 2, 3, 4, 5, 6]
2      print(L[1:3])  # slicing start stop —> [2, 3]
3      print(L[-3:-1]) # negative slicing —> [4, 5]
4      print(L[::-1]) # reverse list —> [6, 5, 4, 3, 2, 1]
5      print(L[0:-1:2])# slicing start stop step —> [1, 3, 5]

```

G Les algorithmes simples mais incontournables

Les listes Python sont très souvent utilisées pour agréger des éléments différents d'un même type. Lors d'un traitement automatisé des données, il est naturel de vouloir :

- compter les éléments d'un certain type,
- trouver le maximum ou le minimum des éléments s'il y en a un, ou trouver l'indice de cet extrémum,
- calculer une somme, une moyenne ou un écart type, si les éléments de la liste sont numériques.

Le code 7.1 implémente ces algorithmes incontournables en Python.

Code 7.1 – Algorithmes simples et incontournables

```

1 def count_elem_if(L, value):
2     c = 0
3     for elem in L:
4         if elem == value:

```

```

5             c += 1
6     return c
7
8
9 def max_elem(L):
10    if len(L) > 0:
11        m = L[0]
12        for elem in L:
13            if elem > m:
14                m = elem
15        return m
16    else:
17        return None
18
19
20 def min_elem(L):
21    if len(L) > 0:
22        m = L[0]
23        for elem in L:
24            if elem < m:
25                m = elem
26        return m
27    else:
28        return None
29
30
31 def sum_elem(L):
32     acc = 0
33     for elem in L:
34         acc += elem
35     return acc
36
37
38 def avg_elem(L):
39     if len(L) > 0:
40         acc = 0
41         for elem in L:
42             acc += elem
43     return acc / len(L)
44    else:
45        return None
46
47 if __name__=="__main__":
48     L = ["words", "letters", "words", "sentences", "words"]
49     print(count_elem_if(L, "words")) # 3
50     L = [ 13, 19, -7, 23, -29, 5, -3, 41]
51     print(max_elem(L), min_elem(L), sum_elem(L), avg_elem(L)) # 41 -29 62 7.75
52     L = []
53     print(max_elem(L), min_elem(L), sum_elem(L), avg_elem(L)) # None None 0 None
54     L = list("Anticonstitutionnellement")
55     print(max_elem(L), min_elem(L)) # u A

```

P En Python, il existe une fonction :

- `sum` pour calculer la somme des éléments d'une liste,
- `max` et `min` pour trouver le maximum et le minimum d'une liste.

```
1     L = [1, 2, 3]
2     print(sum(L), max(L), min(L)) # 6, 3, 1
```

Avant de les utiliser, lisez attentivement le sujet de l'épreuve. Il se peut qu'elles ne soient pas autorisées.

 Le paragraphe précédent est important pour l'épreuve d'informatique!

H Listes, copies et références

En manipulant les listes, il faut rester vigilant lorsqu'on souhaite partager des éléments ou copier des éléments. Il faut avoir en mémoire la règle d'or Python⁶.

P L'affection simple d'une liste à une autre ne recopie pas les éléments de la liste mais copie la référence de la liste. Sur l'exemple suivant, M et L désigne le même objet en mémoire. Si on modifie l'un, on modifie l'autre.

```
1     L = [1, 2, 3]
2     M = L
3     print(id(M) == id(L)) # True
4     M[0] = 42
5     print(L) # [42, 2, 3]
```

Il n'y a donc pas recopie des éléments dans ce cas.

Par contre, l'instruction suivante permet de recopier une liste, c'est à dire dupliquer ses éléments en mémoire. On dispose alors de deux références vers deux objets différents. L'un ne modifie pas l'autre.

```
1     L = [1, 2, 3]
2     M = L[:]
3     print(id(M) == id(L)) # False
4     M[0] = 42
5     print(L) # [1, 2, 3]
```

I Listes, paramètres et fonctions

Cette section s'intéresse au cas où une liste est un paramètre d'une fonction. Que se passe-t-il dans ce cas ?

6. Toute variable Python est une référence vers un objet en mémoire.

La liste étant une séquence mutable, lorsqu'une fonction utilise une liste qu'elle a reçue en paramètre, les opérations sont directement effectuées sur la liste en mémoire comme le montre le code 7.2. Ceci explique pourquoi on n'a pas besoin de renvoyer une liste modifiée par une fonction si celle-ci a été transmise en paramètre. **C'est ce qu'on appelle un passage par référence d'un type mutable.**

Comme le montre l'exemple 7.2, pour une variable immuable comme un `int`, le passage en paramètre n'accorde pas plus de droits sur l'objet : celui-ci est toujours immuable. Donc, d'autres références sont créées pour enregistrer les calculs. Si la dernière référence créée contenant le résultat n'est pas renvoyée par la fonction en utilisant `return`, le calcul est perdu.

On fera donc attention à utiliser `return` lorsqu'il le faut !

Code 7.2 – Passage en paramètre d'un type mutable et d'un type immuable à une fonction

```

1 def f(L):
2     print("inside f start : ", id(L), L)
3     for i in range(len(L)):
4         L[i] = L[i] + 100
5     print("inside f end : ", id(L), L)
6
7
8 def g(a):
9     print("inside g start : ", id(a), a)
10    for i in range(3):
11        a = 10 * a
12    print("inside g end : ", id(a), a)
13    # return a # —> should have return a !
14
15
16 if __name__ == "__main__":
17     M = [1, 2, 3]
18     f(M)
19     print("main M", id(M), M)
20
21     b = 2
22     g(b)
23     print("main b", id(b), b)

```

Voici le résultat de l'exécution de ce code :

```

1  inside f start : 4374619648 [1, 2, 3]
2  inside f end : 4374619648 [101, 102, 103]
3  main M 4374619648 [101, 102, 103]
4  inside g start : 4373872912 2
5  inside g end : 4373887856 2000
6  main b 4373872912 2

```

J Tuples

Les tuples Python sont des séquences immuables que l'on construit avec les parenthèses (). On peut les manipuler comme des listes : ils sont indexables, itérables et tronçonnables. Il ne faut juste pas tenter de modifier leurs éléments!

```
1     T = (1, 2, 3, 4, 5)
2
3     for elem in T:
4         print(elem)
5
6     for i in range(len(T)):
7         print(T[i])
8
9     print(T[-1])
10    print(T[::-1])
11    print(T[1:4:2])
```

8

TRIER ET RECHERCHER

À la fin de ce chapitre, je sais :

- ☒ expliquer le fonctionnement d'algorithmes de tri simple,
- ☒ compter le nombre d'opérations élémentaires nécessaires à l'exécution d'un algorithme,
- ☒ caractériser un tri (comparatif, en place, stable, en ligne)
- ☒ rechercher un élément dans un tableau trié par recherche dichotomique

Trier est une activité que nous pratiquons dès notre plus jeune âge : les couleurs, les formes, les tailles sont autant d'entrées possibles pour cette activité. Elle est également omniprésente dans l'activité humaine, dès l'instant où il faut rationaliser une activité. Dans le cadre du traitement de l'information, on y a si souvent recours qu'on n'y prête quasiment plus attention. C'est pourquoi les algorithmes de tri ont été intensément étudiés et raffinés à l'extrême : il s'agit d'être capable de trier n'importe quel jeu de données pourvu qu'un ordre puisse être défini et de sélectionner le tri le plus efficace selon le tri à effectuer.

Lors du traitement des informations, le tri est souvent associé à une autre activité : la recherche d'un élément dans un ensemble trié. En effet, la recherche dichotomique permet d'accélérer grandement la recherche d'un élément, pourvu que celui-ci soit trié.

Ce chapitre esquisse donc quelques algorithmes de tri simples à coder puis aborde la recherche recherche dichotomique dans un tableau trié.

Dans tout le chapitre, on considère :

- un tableau noté t , indiqué avec des crochets, c'est à dire que l'on peut accéder à un élément en écrivant $t[i]$,
- des éléments à trier dont le type importe peu, pourvu qu'on dispose d'un ordre sur ces éléments,

On donnera au cours de l'année une définition précise d'une relation d'ordre, mais, pour l'instant, on considère juste qu'il est possible de comparer deux éléments entre eux. C'est à

dire qu'on peut réaliser un test de type $t[i] < t[j]$ et récupérer une réponse booléenne à ce test. C'est d'ailleurs ainsi que la plupart des contexte informatique définissent un ordre dans un type de donnée.

P En Python, on implémentera ces tableaux par des listes.

A Comment caractériser un algorithme de tri?

Pour distinguer les différents algorithmes de tri, on dispose de nombreux qualificatifs. Les définitions suivantes en expliquent quelques uns parmi les plus courants.

■ **Définition 49 — Tri par comparaisons.** Un tri par comparaison procède en comparant les éléments deux à deux ^a.

a. ce qui sous-entend que certains algorithmes, comme l'algorithme de tri par comptage, procèdent différemment.

■ **Définition 50 — Tri en place.** Un tri est dit en place s'il peut être directement effectué dans le tableau initial et ne nécessite pas l'allocation d'une nouvelle structure en mémoire de la taille du tableau initial ^a.

a. Cette propriété est importante car la mémoire est un paramètre très couteux des systèmes électroniques, tant sur le plan énergétique que sur le plan financier.

■ **Définition 51 — Tri incrémental ou en ligne.** Il s'agit d'un tri capable de commencer le tri avant même d'avoir reçu l'intégralité des données ^a.

a. Cette propriété est très intéressante dans le cadre pour le traitement en temps réel des systèmes dynamiques (qui évoluent dans le temps)

■ **Définition 52 — Tri stable.** Un tri est dit stable s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux mais que l'on peut distinguer ^a.

a. par exemple, dans un jeu de cartes, deux valets (même type de carte) de couleurs différentes.

B Trier un tableau

a Tri par sélection

Le tri par sélection est un tri par comparaisons et échanges, en place, non stable et hors ligne.

Son principe est le suivant. On cherche le plus petit élément du tableau (de droite) et on l'insère à la fin du tableau trié de gauche. Au démarrage de l'algorithme, on cherche le plus petit élément du tableau et on l'insère à la première place. Puis on continue avec le deuxième plus petit élément du tableau que l'on ramène à la deuxième place et ainsi de suite.

Algorithme 4 Tri par sélection

```

1: Fonction TRIER_SELECTION(t)
2:    $n \leftarrow \text{taille}(t)$ 
3:   pour  $i$  de 0 à  $n - 1$  répéter
4:      $\text{min\_index} \leftarrow i$                                  $\triangleright$  indice du prochain plus petit
5:     pour  $j$  de  $i + 1$  à  $n - 1$  répéter                 $\triangleright$  pour tous les éléments non triés
6:       si  $t[j] < t[\text{min\_index}]$  alors
7:          $\text{min\_index} \leftarrow j$ 
8:       échanger( $t[i]$ ,  $t[\text{min\_index}]$ )                 $\triangleright$  c'est l'indice du plus petit non trié!
                                                  $\triangleright$  c'est le plus grand des triés!

```

b Tri par insertion

Le tri par insertion est un tri par comparaison, stable et en place. De plus, si les données ne sont pas toutes présentes au démarrage de l'algorithme, le tri peut quand même commencer, ce qui n'est pas le cas du tri par sélection. C'est donc un tri en ligne également.

Son principe est le suivant : on considère deux parties gauche (triée) et droite (non triée) du tableau. Puis, on cherche à insérer le premier élément non trié (du tableau de droite) dans le tableau trié (de gauche) à la bonne place.

Algorithme 5 Tri par insertion

c Tri par comptage

Le tri par comptage est un tri par dénombrement de valeurs entières. Il ne porte donc que sur des tableaux contenant des entiers. C'est un tri non comparatif, stable et hors ligne.

Le principe du tri par comptage est de compter le nombre d'occurrences de chaque valeur entière puis de construire un nouveau tableau à partir de ce comptage.

R Ce tri est certes non comparatif car il n'utilise pas explicitement de comparaisons. Mais l'ordre des entiers est utilisé lors du comptage.

Algorithme 6 Tri par comptage

```

1: Fonction TRIER_COMPTAGE( $t, v_{max}$ ) ▷  $v_{max}$  est le plus grand entier à trier
2:    $n \leftarrow \text{taille}(t)$ 
3:    $c \leftarrow \text{un tableau de taille } v_{max} + 1 \text{ initialisé avec des zéros}$ 
4:   pour  $i$  de 0 à  $n - 1$  répéter
5:      $c[t[i]] \leftarrow c[t[i]] + 1$  ▷ compter les occurrences de chaque élément du tableau.
6:    $\text{résultat} \leftarrow \text{un tableau de taille } n$ 
7:    $i \leftarrow 0$ 
8:   pour  $v$  de 0 à  $v_{max}$  répéter ▷ On prend chaque valeur possible dans l'ordre
9:     si  $c[v] > 0$  alors ▷ Si l'élément  $v$  est présent dans le tableau
10:      pour  $j$  de 0 à  $c[v] - 1$  répéter
11:         $\text{résultat}[i] \leftarrow v$  ▷ alors écrire autant de  $v$  que d'occurrences de  $v$ 
12:         $i \leftarrow i + 1$  ▷ à la bonne place, la  $i$ ème!
13:   renvoyer résultat

```

C Comparatif des tris

Les tableaux 8.1 et 8.2 résument les caractéristiques et les complexités des principaux algorithmes de tri au programme.

D Recherche séquentielle

Rechercher d'un élément dans un tableau est une opération nécessaire pour de nombreux algorithmes. Il est donc important de disposer d'algorithmes rapides pour exécuter cette tâche. La recherche séquentielle est l'algorithme le plus naïf pour réaliser cette tâche. L'algorithme 7 rappelle la recherche séquentielle d'un élément dans un tableau.

Dans le pire des cas, c'est à dire si l'élément recherché n'appartient pas au tableau, la recherche séquentielle nécessite un nombre d'instructions proportionnel à n , la taille du tableau. On dit que sa complexité est en $O(n)$.

Tris	Comparaison	Stable	En place	En ligne
par sélection	oui	non	oui	non
par insertion	oui	oui	oui	oui
par comptage	non	oui	non	non
fusion	oui	oui	non	non
rapide	oui	non	oui	non

TABLE 8.1 – Comparatif des caractéristiques des algorithmes de tri. Il existe de nombreuses variantes de ces algorithmes. Les informations de ce tableau concernent les versions implémentées en cours ou en TP.

Tris	Complexité temporelle pire des cas	Complexité temporelle moyenne	Complexité temporelle meilleur des cas	Complexité spatiale
par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
par insertion	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$
par comptage	$O(n + v_{max})$	$O(n + v_{max})$	$O(n + v_{max})$	$O(n + v_{max})$
fusion	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
rapide	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$

TABLE 8.2 – Complexité des différents algorithmes de tri.

E Recherche dichotomique

Si le tableau dans lequel la recherche est à effectuer est trié, alors la recherche dichotomique à privilégié. Celle-ci est illustrée sur la figure 8.1.

■ **Définition 53 — Recherche dichotomique.** La recherche dichotomique recherche un élément dans un tableau trié en éliminant à chaque itération la moitié du tableau qui ne contient pas l'élément. Le principe est le suivant :

Algorithme 7 Recherche séquentielle d'un élément dans un tableau

```

1: Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
2:   n ← taille(t)
3:   pour i de 0 à n – 1 répéter
4:     si t[i] = elem alors
5:       renvoyer i           ▷ élément trouvé, on renvoie sa position dans t
6:     renvoyer l'élément n'a pas été trouvé

```

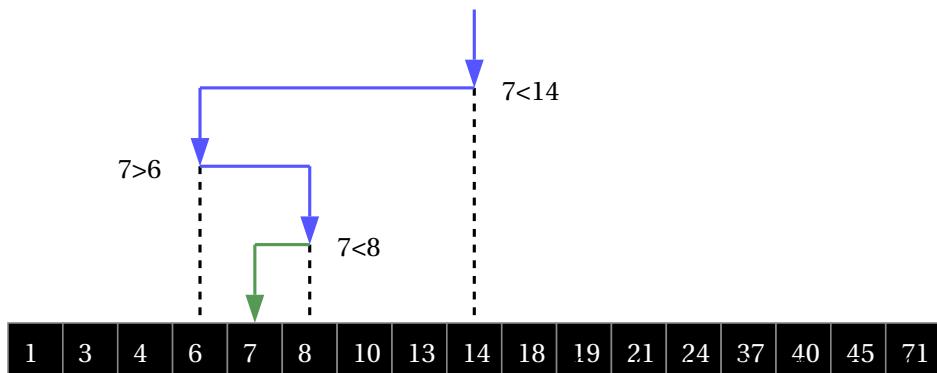


FIGURE 8.1 – Illustration de la recherche dichotomique de la valeur 7 dans un tableau trié (Source : Wikimedia Commons)

- comparer la valeur recherchée avec l’élément **médián** du tableau,
- si les valeurs sont égales, la position de l’élément dans le tableau a été trouvée,
- sinon, il faut poursuivre la recherche dans la moitié du tableau qui contient l’élément à coup sûr.

Si l’élément existe dans le tableau, alors l’algorithme renvoie son indice dans le tableau. Sinon, l’algorithme signifie qu’il ne l’a pas trouvé.

Le mot dichotomie vient du grec et signifie couper en deux. En prenant l’élément médian du tableau, on opère en effet une division de la taille du tableau en deux parties. On ne conserve que la partie qui pourrait contenir l’élément recherché.

Algorithme 8 Recherche d’un élément par dichotomie dans un tableau trié

```

1: Fonction RECHERCHE_DICHOTOMIQUE(t, elem)
2:   n ← taille(t)
3:   g ← 0
4:   d ← n-1
5:   tant que g ≤ d répéter                                ▷ ≤ cas où valeur au début, au milieu ou à la fin
6:     m ← (g+d)//2                                         ▷ Division entière : un indice est un entier!
7:     si t[m] < elem alors                                ▷ l’élément devrait se trouver dans t[m+1, d]
8:       g ← m + 1
9:     sinon si t[m] > elem alors                         ▷ l’élément devrait se trouver dans t[g, m-1]
10:    d ← m - 1
11:    sinon
12:      renvoyer m                                         ▷ l’élément a été trouvé
13:    renvoyer l’élément n’a pas été trouvé

```

À chaque tour de boucle, l’algorithme 8 divise par deux la taille du tableau considéré. Dans le pire des cas, le dernier tableau considéré comporte un seul élément. Supposons que la taille du tableau est une puissance de deux : $n = 2^p$. On a alors $1 = \frac{n}{2^k} = \frac{2^p}{2^k}$, si k est le nombre

d'itérations effectuées pour en arriver là. C'est pourquoi le nombre d'itérations peut s'écrire : $k = \log_2 2^p = p \log_2 2 = p = \log_2(n)$. La complexité de cet algorithme est donc logarithmique en $O(\log_2(n))$, ce qui est bien plus efficace qu'un algorithme de complexité linéaire.

R Si le tableau contient plusieurs occurrences de l'élément à chercher, alors l'algorithme 8 renvoie un indice qui n'est pas celui de la première occurrence de l'élément. C'est pourquoi, l'algorithme 9 propose une variation qui renvoie l'indice de la première occurrence de l'élément.

Algorithme 9 Recherche d'un élément par dichotomie dans un tableau trié, renvoyer l'indice minimal en cas d'occurrences multiples.

```

1: Fonction RECHERCHE_DICHOTOMIQUE_INDICE_MIN(t, elem)
2:   n ← taille(t)
3:   g ← 0
4:   d ← n-1
5:   tant que g < d répéter                                ▷ attention au strictement inférieur!
6:     m ← (g+d)//2                                         ▷ Un indice de tableau est un entier!
7:     si t[m] < elem alors
8:       g ← m + 1                                         ▷ l'élément devrait se trouver dans t[m+1, d]
9:     sinon
10:    d ← m                                              ▷ l'élément devrait se trouver dans t[g, m]
11:    si t[g] = elem alors
12:      renvoyer g
13:    sinon
14:      renvoyer l'élément n'a pas été trouvé

```

L'algorithme de la recherche dichotomique peut s'écrire de manière récursive, comme on le verra dans le chapitre suivant.

R Attention à ne pas confondre cet algorithme avec la recherche de zéro d'une fonction par dichotomie. Le principe est le même. Cependant, dans le cas de la recherche dichotomique, on s'intéresse à des indices de tableaux, donc à des nombres entiers. C'est pourquoi on choisit de prendre le milieu via la division entière $//$. Lorsqu'on cherche les zéros d'une fonction mathématique, on s'intéresse à des nombres réels représentés par des flottants en machine. C'est pourquoi lorsqu'on prend le milieu, on choisit alors la division de flottants $/$.

R Le milieu d'un segment $[a, b]$ vaut $m=(a+b)/2$, **non pas** $m=(b-a)/2$.

F Compter les opérations

Pour comparer l'efficacité des algorithmes, on compte le nombre d'opérations élémentaires nécessaires à leur exécution en fonction de la taille des données d'entrée. Dans notre cas, la taille des données d'entrée est souvent la taille du tableau à trier. Que se passe-t-il lorsque cette taille augmente ? Le temps d'exécution de l'algorithme augmente-t-il ? Et de quelle manière ? Linéairement, exponentiellement ou logarithmiquement par rapport à la taille ?

Dans ce qui suit, on suppose qu'on dispose d'une machine pour tester l'algorithme. On fait l'hypothèse réaliste que les opérations élémentaires suivantes sont réalisées en des temps constants par cette machine :

- opération arithmétique $+, -, *, /, //, \%$, coûts associé c_{op} ,
- tests $==, !=, <, >$, coûts associé c_t ,
- affectation \leftarrow , coût associé c_{\leftarrow}
- accès à un élément indicé $t[i]$, coût associé c_a
- structures de contrôles (structures conditionnelle et boucles), coût associé négligé,
- échange de deux éléments dans le tableau, coût c_e ,
- accès à la longueur d'un tableau, coût c_l .

■ **Exemple 10 — Compter le nombres d'opération élémentaires de la recherche séquentielle.**

Sur l'algorithme 10, on a reporté les coûts pour chaque instruction. On peut donc maintenant calculer le coût total. Ce coût C va dépendre de la taille du tableau que l'on va noter n . Il va également dépendre du chemin d'exécution : est-ce que le test ligne 4 est valide ou non ? On choisit de se placer dans le pire des cas, c'est à dire qu'on suppose que le test est invalidé à toutes les itérations ^a. Dans ce cas, on peut dénombrer les coûts élémentaires :

$$C(n) = c_{\leftarrow} + \sum_{i=0}^{n-1} c_a + c_t \quad (8.1)$$

$$= c_{\leftarrow} + (c_a + c_t)n \quad (8.2)$$

Dans le pire des cas, on observe donc que le coût de la recherche séquentielle est proportionnel à n . On notera cette au deuxième semestre $C(n) = O(n)$, signifiant par là que le coût est dominé asymptotiquement par n . Lorsque n augmente, le coût de la recherche séquentielle dans le pire des cas n'augmente pas plus vite que n .

Si on se place dans le meilleur des cas, c'est à dire lorsque l'élément recherché se situe dans la première case du tableau, on obtient un coût total de $C(n) = c_{\leftarrow} + c_a + c_t$, c'est à dire un coût qui ne dépend pas de n . On le notera $C(n) = O(1)$ et on dira que ce coût est constant.

^a. i.e. l'élément cherché n'est pas dans le tableau.

Algorithme 10 Recherche séquentielle d'un élément dans un tableau

```

1: Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
2:   n ← taille(t)                                     ▷ affectation :  $c_a$ 
3:   pour i de 0 à n – 1 répéter                   ▷ répéter n fois
4:     si t[i]= elem alors                         ▷ accès, test :  $c_a + c_t$ 
5:       renvoyer i
6:   renvoyer l'élément n'a pas été trouvé

```

(R) L'analyse des calculs précédents montre qu'on peut simplifier notre approche du décompte des opérations. Étant donné que ce qui nous intéresse, c'est l'évolution du coût en fonction de la taille du tableau, les valeurs des constantes de coût des opérations élémentaires importent peu, pourvu que celles-ci soient constantes. **C'est pourquoi, on supposera désormais qu'on dispose d'une machine de test pour nos algorithmes telle que chaque instruction élémentaire possède un coût constant que l'on notera c .**

Algorithme 11 Tri par insertion, calcul du nombre d'opérations

```

1: Fonction TRIER_INSERTION(t)
2:   n ← taille(t)                                     ▷ affectation :  $c_a$ 
3:   pour i de 1 à n-1 répéter
4:     à_insérer ← t[i]                                ▷ accès, affectation :  $2c$ 
5:     j ← i                                         ▷ affectation :  $c$ 
6:     tant que t[j-1] > à_insérer et j>0 répéter    ▷ accès, tests :  $3c$ 
7:       t[j] ← t[j-1]                                ▷ accès, affectation :  $3c$ 
8:       j ← j-1                                       ▷ affectation :  $c$ 
9:     t[j] ← à_insérer                                ▷ accès, affectation :  $2c$ 

```

■ **Exemple 11 — Compter le nombre d'opérations élémentaires pour le tri par insertion.** Sur l'algorithme 11, on a reporté les coûts pour chaque instruction. On peut donc maintenant calculer le coût total. On choisit de se placer dans le pire des cas, c'est à dire lorsqu'on insère l'élément systématiquement au début du tableau ^a. La boucle *tant que* est donc exécutée $i - 1$ fois.

$$C(n) = c + \sum_{i=1}^{n-1} (2c + c + (i-1)(3c + c) + 2c) \quad (8.3)$$

$$= c + c \sum_{i=1}^{n-1} (1 + 4i) \quad (8.4)$$

$$= c + c(n-1) + 4 \frac{n(n-1)}{2} \quad (8.5)$$

$$= (c-2)n + 2n^2 \quad (8.6)$$

Dans le pire des cas, le coût total du tri par insertion est donc en $O(n^2)$.

Le meilleur des cas, pour l'algorithme de tri par insertion, c'est lorsque l'élément à insérer n'a jamais à être déplacé ^b. Alors la condition de sortie de la boucle *tant que* est valide dès le premier test. Le coût total devient alors :

$$C(n) = c + \sum_{i=1}^{n-1} (2c + c + 3c) \quad (8.7)$$

$$= c + 6c(n-1) \quad (8.8)$$

$$= -5c + 6cn \quad (8.9)$$

Dans le meilleur des cas, on a donc un coût du tri par insertion en $O(n)$.

- a. Le tableau est donné en entrée dans l'ordre inverse à l'ordre souhaité.
- b. Le tableau donné en entrée est déjà trié!

■ **Exemple 12 — Compter le nombre d'opérations élémentaires pour le tri par comptage.** La première partie de l'algorithme 6 compte les occurrences de chaque élément. La complexité de cette opération est $O(n)$ puisqu'il s'agit de balayer tout le tableau d'entrée à chaque fois.

La seconde partie de l'algorithme 6 est composée deux boucles imbriquées qui dépendent de v_{max} et de n . La boucle imbriquée effectue $c[v]$ tours à chaque fois que le nombre est présent dans le tableau. On considère que le test et les deux opérations dans la boucle imbriquée ont un coût constant de 1. On peut compter le nombre d'opérations pour les deux boucles imbriquées :

$$C(n, v_{max}) = \sum_{v=0}^{v_{max}} \left(1 + \sum_{j=0}^{c[v]-1} 1 \right) \quad (8.10)$$

$$= \sum_{v=0}^{v_{max}} 1 + c[v] \quad (8.11)$$

$$= 1 + v_{max} + n \quad (8.12)$$

car $\sum_{j=0}^{c[v]-1} 1 = c[v]$ et $\sum_{v=0}^{v_{max}} c[v] = n$.

C'est pourquoi la complexité du tri par comptage est en $O(n+1+v_{max}+n) = O(v_{max}+n)$

Le coût total de l'algorithme de tri par comptage est donc linéaire par rapport aux deux paramètres d'entrée. Il est donc raisonnable de l'utiliser lorsque n est du même ordre que v_{max} . Par contre, il nécessite la création d'un tableau de même taille n ^a, ce qui peut avoir des conséquences sur la mémoire du système.

a. ce tri n'est pas en place

G Trier avec les fonctions natives de Python

P Il existe deux solutions natives pour trier une liste en Python :

1. appeler la fonction `sorted` sur une liste ou un dictionnaire. Cette fonction renvoie une nouvel objet trié sans modifier l'original. Par exemple, `T = sorted(L)`. Dans le cas d'un dictionnaire, ce sont les clefs qui sont triées.
2. appeler la méthode `sort` de la classe List. Cette méthode effectue un tri en place sur une liste et la modifie. Par exemple, `L.sort()`. Cette méthode n'existe pas pour les dictionnaires.

Les tris opérés par ces fonctions sont garantis stables. Cela signifie que lorsque plusieurs enregistrements on la même clef, leur ordre original est préservé.

[La documentation Python est à consulter ici pour les usages plus spécifiques de ces fonctions.](#) On peut notamment spécifier :

- un tri ascendant ou descendant à l'aide du paramètre optionnel booléen `reverse`,
- à la fonction l'élément à utiliser pour le tri (en cas d'élément multiple) et même le comparateur à l'aide du paramètre optionnel `key`.

P Par défaut, Python trie les n-uplets dans l'ordre lexicographique.

9

RÉCURSIVITÉ

À la fin de ce chapitre, je sais :

- expliquer le principe d'un algorithme récursif
- imaginer une version récursive d'un algorithme
- trouver et coder une condition d'arrêt à la récursivité
- coder des algorithmes à récursivité simple et multiple en Python
- identifier le type de récursivité d'un algorithme

A Principles

La récurrence est une méthode à la fois simple et puissante pour définir un objet ou résoudre un problème mathématique. La récursivité, c'est la projection de cette méthode en informatique. La plupart des langages contemporains sont récursifs, c'est à dire qu'ils permettent de programmer récursivement.

■ **Définition 54 — Algorithme récursif.** Un algorithme \mathcal{A} qui résout un problème \mathcal{P} est dit récursif si, où au cours de son exécution, il s'utilise lui-même pour résoudre le problème \mathcal{P} avec des données d'entrées différentes.

■ **Définition 55 — Appel récursifs d'une fonction.** On désigne par le terme appel récursif l'utilisation d'une fonction dans la définition de la fonction elle-même.

■ **Définition 56 — Pile d'exécution.** La pile d'exécution d'un processus est un emplacement mémoire destiné à mémoriser les appels de fonction, les paramètres, les variables locales ainsi que l'adresse de retour ^a de chaque fonction en cours d'exécution.

^a. c'est à dire la case mémoire qui va contenir la valeur renvoyée par la fonction.


Vocabulary 10 — Call stack \leadsto Pile d'exécution

L'algorithme 12 est un algorithme récursif qui permet de calculer la fonction factorielle d'un entier naturel. La ligne 5 contient l'appel récursif. À la lecture de cet algorithme, on peut être pris d'un vertige : cela va-t-il se terminer ? Est-il possible de s'appeler soi-même pour résoudre un problème ? Comment gérer en mémoire les appels successifs ?

Algorithme 12 Factoriel récursif

1: Fonction FACT(n)	
2: si n ≤ 1 alors	▷ Condition d'arrêt
3: renvoyer 1	
4: sinon	
5: renvoyer n \times FACT(n-1)	▷ Appel récursif

Pour calculer, cet algorithme procède comme suit : il suspend momentanément le calcul du résultat de la fonction à la réception du résultat de l'appel récursif en empilant l'appel récursif sur la pile d'exécution. La pile d'exécution est donc une zone mémoire essentielle qui garantit le bon déroulement des appels successifs des fonctions récursives ou non.

■ **Exemple 13 — Calculer 3! récursivement.** Dans le cas de 3!, l'exécution se présente ainsi :

1. appel de FACT(3),
2. attente du résultat de FACT(2) qui est empilé sur la pile d'exécution,
3. attente du résultat de FACT(1) qui est empilé sur la pile d'exécution,
4. FACT(1) renvoie 1, on dépile,
5. 1 est multiplié par 2, 2 est renvoyé par FACT(2), on dépile,
6. 2 est multiplié par 3 et 6 est renvoyé par FACT(3).

La figure 9.1 illustre ce fonctionnement.

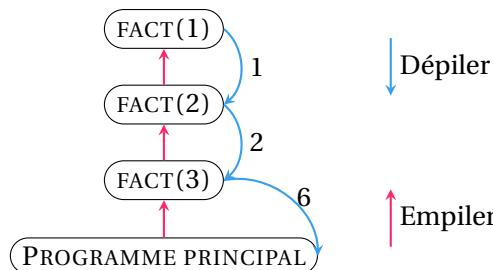


FIGURE 9.1 – Vision de la pile d'exécution de l'algorithme factoriel récursif 12 pour le calcul de 3!

R Comme toute zone mémoire, la pile d'exécution est limitée, ce qui a des conséquences concrètes en programmation : on ne peut pas effectuer une infinité d'appels récursif. Une limite existe à la profondeur de la récursivité : la nature est têtue, notre monde physique demeure fini.

R La définition 54 ne garantit pas le bon fonctionnement d'un algorithme récursif. Pour qu'un algorithme récursif aboutisse correctement, il est nécessaire que le nombre d'appels récursifs soit fini. Dans le cas contraire, la pile d'exécution explose en mémoire : il n'y a plus assez de place pour la stocker et le système s'effondre.

M **Méthode 5 — Formuler correctement un algorithme récursif** Afin d'éviter les appels récursifs infinis, il est nécessaire d'être vigilant lors de l'écriture des algorithmes récursifs. Il faut :

1. distinguer au moins un cas sans appel récursif qui permet de terminer l'algorithme, c'est à dire **une condition d'arrêt**,
2. appeler récursivement avec des données **plus proches** des données qui satisfont la condition d'arrêt.

Dans le cas de l'algorithme 12, la condition d'arrêt est la ligne 2. Par ailleurs, on voit bien que l'appel récursif stipule le paramètre $n - 1$ qui se rapproche de cette condition d'arrêt. Une formulation plus correcte et plus théorique repose sur l'induction et la notion d'ordre bien fondé (cf. option informatique).

B Types de récursivité

■ **Définition 57 — Récursivité simple.** La récursivité est dite simple si une exécution de l'algorithme aboutit à un seul appel récursif.

■ **Exemple 14 — Algorithmes simplement récursifs.** L'algorithme factoriel récursif 12 est une récursivité simple. Il est possible de formuler l'algorithme de recherche dichotomique récursivement (cf. algorithme ??). Cet algorithme est également de récursivité simple : chaque chemin d'exécution n'opère qu'un seul appel récursif.

■ **Définition 58 — Récursivité multiple.** La récursivité est dite multiple si une exécution de l'algorithme aboutit à plusieurs appels récursifs.

■ **Exemple 15 — Algorithmes à récursivité multiple.** Les algorithmes calculant la suite de Fibonacci, la solution au jeu des tours d'Hanoi ou l'exemple 14 sont à récursivité multiples. L'algorithme 14 est à récursivité multiple car la fonction est appelée deux fois à la ligne sept.

Algorithme 13 Recherche récursive d'un élément par dichotomie dans un tableau trié

```

1: Fonction REC_DICH(t, g, d, elem)
2:   si g > d alors                                ▷ Condition d'arrêt
3:     renvoyer l'élément n'a pas été trouvé
4:   sinon
5:     m ← (g+d)//2                                ▷ Diviser
6:     si t[m] = elem alors
7:       renvoyer m
8:     sinon si elem < t[m] alors
9:       REC_DICH(t, g, m-1, elem)                  ▷ résoudre
10:    sinon
11:      REC_DICH(t, m+1, d, elem)                  ▷ résoudre

```

Algorithme 14 Est-il mon aïeul?

```

1: Fonction EST_AÏEUL(p1, p2)
2:   si p1 est un des parents de p2 alors
3:     renvoyer Vrai
4:   sinon si p1 est plus jeune que p2 alors
5:     renvoyer Faux
6:   sinon
7:     renvoyer EST_AÏEUL(p1, mère(p2)) ou EST_AÏEUL(p1, père(p2))

```

■ **Définition 59 — Récursivité terminale.** Un algorithme est dit à récursivité terminale s'il renvoie directement le résultat de l'appel récursif sans opération supplémentaire. Cette propriété rend l'opération de dépilement de la pile d'exécution très efficace puisqu'on n'a pas à faire de calculs supplémentaires.

■ **Exemple 16 — Factoriel récursif terminal.** On peut transformer l'algorithme 12 pour qu'il présente une récursivité terminale (cf. 15)

Algorithme 15 Factoriel récursif terminal

```

1: Fonction FACT(n, acc)                                ▷ on ajoute un accumulateur
2:   si n <= 1 alors                                ▷ Condition d'arrêt
3:     renvoyer acc
4:   sinon
5:     renvoyer FACT(n-1, n × acc)                  ▷ Appel récursif avec accumulateur

```

C Du récursif à l'itératif

Dans la pratique, un algorithme est souvent énoncé récursivement car l'expression est plus puissante, plus intelligible et plus facile à trouver¹. Cependant, on convertit la plupart du temps l'algorithme récursif en itératif afin d'être plus performant : il s'agit d'éviter d'avoir à effectuer les appels récursifs successifs qui ralentissent souvent l'exécution sur les machines concrètes lorsque la récursivité n'est pas terminale.

M Méthode 6 — Du récursif à l'itératif La formulation d'un algorithme sous la forme d'un algorithme récursif **terminal** permet de facilement en déduire une version itérative. Le principe est expliqué par les algorithmes 16 et 17.

En appliquant la méthode décrite par ces algorithmes, on peut retrouver facilement la version itérative de factorielle (cf. algorithme 18).

Algorithme 16 Algorithme récursif terminal

```

1: Fonction REC_TERM(n)
2:   si ARRÊTER(n) alors
3:     renvoyer SOLUTION(n)
4:   sinon
5:     CALCULER_AVEC(n)
6:     renvoyer REC_TERM(MODIFIER(n))

```

Algorithme 17 Version itérative d'un algorithme récursif terminal

```

1: Fonction ITER(n)
2:   tant que non ARRÊTER(n) répéter
3:     CALCULER_AVEC(n)
4:     n  $\leftarrow$  MODIFIER(n)
5:   renvoyer SOLUTION(n)

```

Algorithme 18 Factoriel itératif

```

1: Fonction FACT(n)
2:   acc  $\leftarrow$  1
3:   tant que n > 1 répéter
4:     acc  $\leftarrow$  n  $\times$  acc
5:     n  $\leftarrow$  n - 1
6:   renvoyer acc

```

1. parfois...

R Programmer récursivement est une manière de penser très structurante. Elle repose sur le principe d'induction ^a. Les algorithmes proposés en TP permettent de développer cette compétence importante.

a. exposé en option informatique

10

INTRODUCTION À NUMPY

À la fin de ce chapitre, je sais :

- ☛ importer la bibliothèque Numpy
- ☛ utiliser un tableau Numpy mono et multidimensionnel
- ☛ écrire des opérations élément par élément

P Numpy est un bibliothèque logicielle Python dédiée au calcul numérique. Elle n'est pas explicitement au programme. Certaines épreuves de concours en interdisent son usage quand d'autres l'exigent explicitement. Il faut rester vigilant à la lecture de l'épreuve.

 **Le paragraphe précédent est important pour l'épreuve d'informatique!**

Numpy est très utilisé dans le monde entier. Les raisons principales sont les suivantes :

1. Numpy est open source,
2. Numpy est optimisée avec un cœur compilé en langage C (compilé, pas interprété donc plus rapide),
3. Numpy s'interface facilement avec d'autres langage de calcul scientifique (C et Fortran notamment),
4. Numpy procure les `array`, une structure de données de type tableau multidimensionnel de dimensions fixes. Cela complète habilement le langage Python qui ne propose que des listes qui sont implémentées par des tableaux dynamiques.
5. Numpy propose des opérations sur les tableaux élément par élément. Cela permet à la fois d'éviter d'écrire des boucles et de paralléliser les opérations.
6. Numpy permet l'écriture de calculs matriciels.
7. Ces deux derniers points font que les formules mathématiques apparaissent écrites quasi-naturellement dans le code.

A Importation

On peut importer comme on le désire la bibliothèque Numpy. Cependant une convention très utilisée fait qu'on l'importe souvent comme ceci :

```
1 import numpy as np
2
3 t = np.zeros(42)
```

B Créer des vecteurs, des matrices ou des tableaux

■ **Définition 60 — Vecteur numpy.** Le terme *vecteur* désigne des `array` numpy de dimension 1.

■ **Définition 61 — Matrice numpy .** Le terme *matrice* désigne les `array` numpy de dimension deux.

Lorsque le tableau possède plus de deux dimensions, on parle de tableau.

On peut créer, comme le code 10.1 le montre, un vecteur, une matrice ou un tableau numpy à partir :

- du constructeur `np.array()`,
- d'une liste ou d'une liste imbriquée,
- de fonctions spéciales en précisant les dimensions du tableau et la valeur d'initialisation des cases du tableau,
- d'un tableau existant : la tableau créé aura les mêmes dimensions. On précise la valeur à laquelle on veut initialiser les cases.

Code 10.1 – Créer des tableaux Numpy

```
1 import numpy as np
2
3 v1 = np.array([1, 2, 3])
4 print(v1.shape, v1)  # (3,) [1 2 3]
5 v2 = np.array([[10], [20], [30]])
6 print(v2.shape, v2)  # (3, 1) [[10] [20] [30]]
7
8 v3 = np.zeros((1, 3))
9 print(v3.shape, v3)  # (1, 3) [[0. 0. 0.]]
10 v4 = np.ones((1, 7))
11 print(v4.shape, v4)  # (1, 7) [[1. 1. 1. 1. 1. 1. 1.]]
12
13 m1 = np.zeros((5, 5))
14 print(m1.shape, m1)
15 # (5, 5)
16 # [[0. 0. 0. 0. 0.]
17 # [0. 0. 0. 0. 0.]]
```

```

18 # [0. 0. 0. 0. 0.]
19 # [0. 0. 0. 0. 0.]
20 # [0. 0. 0. 0. 0.]
21
22 m1 = np.ones((2, 3))
23 print(m1.shape, m1)
24 # (2, 3)
25 # [[1. 1. 1.]
26 # [1. 1. 1.]]
27
28 t0 = np.zeros((42, 21, 84, 3))
29 print(t0.shape) # (42, 21, 84, 3)
30
31 t1 = np.zeros_like(v1)
32 print(t1.shape, t1) # (3,) [0 0 0]
33 t2 = np.ones_like(v3)
34 print(t1.shape, t1) # (1, 3) [[1. 1. 1.]]
35 t3 = np.full_like(m1, 42)
36 print(t1.shape, t1)
37 # (2, 3)
38 # [[42. 42. 42.]
39 # [42. 42. 42.]]
40
41 s = np.zeros((4, 4, 500)).shape
42 print(s[0], s[1], s[2]) # 4 4 500
43
44 r = np.arange(0, 1, 0.1) # like range but for float !
45 print(r) # [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
46 time = np.linspace(0, 1, 11)
47 print(time) # [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]

```

P Comme le montre la fin du code 10.1, la fonction `shape` renvoie un tuple qui spécifie les dimensions du tableau. Comme c'est un tuple, on peut accéder à la taille de chaque dimension par l'opérateur `[]`. Pour un matrice, par exemple, on obtient le nombre de lignes et le nombre de colonnes.

C Accéder aux éléments d'un tableau

Les cases d'un `array` numpy sont numérotées à partir de zéro, comme pour les listes Python. Le tronçonnage ainsi que l'indexage négatif sont également disponibles comme le montre le code 10.2.

P Cependant, la syntaxe de la manipulation des tableaux multidimensionnels est différente de celle des listes imbriquées : une virgule sépare les indices des dimensions différentes. Il faut rester vigilant et ne pas confondre ces syntaxes.

 Le paragraphe précédent est important pour l'épreuve d'informatique!

Code 10.2 – Accéder aux éléments d'un tableau numpy

```

1 import numpy as np
2
3 m = np.array([[1, 2], [3, 4]])
4 print(m)
5 # [[1 2]
6 #  [3 4]]
7 print(m[0, 0], m[0, 1], m[1, 0], m[1, 1]) # 1 2 3 4
8 # slicing
9 print(m[:, 0]) # [1 3]
10 print(m[:, 1]) # [2 4]
11 print(m[0, :]) # [1 2]
12 print(m[1, :]) # [3 4]
13
14 # negative indexing
15 print(m[-1, -1])
16 # reversing
17 print(m[::-1])
18 # assignment
19 m[0, :] = 42
20 print(m)
21 # [[42 42]
22 #  [ 3  4]]
23
24 # resizing
25 m = m.reshape((1, 4))
26 print(m) # [[1 2 3 4]]

```

P Comme on peut le voir à la fin du code 10.2, on peut facilement redimensionner un tableau, avec ses éléments dedans. Cela vient du fait que numpy stocke toujours les éléments d'un tableau de manière contiguë en mémoire et ce, quelle que soit les dimensions du tableau.

La bibliothèque stocke aussi avec le tableau ses dimensions apparentes pour l'utilisateur. L'accès aux éléments au travers ses dimensions `t[3,2]` n'est donc qu'une commodité (du sucre syntaxique) fournie par le génie logiciel de la bibliothèque. Un indice réel est calculé par numpy à partir de `[3,2]` pour accéder à la case l'élément dans la zone contiguë en mémoire.

 **Vocabulary 11 — Syntactic sugar ↪** Sucre syntaxique. Élément de la syntaxe d'un langage visant à faciliter l'utilisation, l'écriture et la lecture associées à un concept. Cela adoucit le travail humain.

D Opérations élément par élément

On a très souvent besoin d'appliquer les mêmes formules à tous les éléments d'un tableau, à toute une série de données. Par exemple, pour filtrer un signal, le moyenner, pour changer de couleur tous les pixels d'une image ou tout simplement pour calculer tous le prix TTC de tous les éléments d'une liste de prix HT.

Le succès de numpy repose principalement sur la capacité à opérer des calculs sur un tableau comme si on les effectuait sur une variable scalaire (cf. code 10.3). **Il devient alors inutile de faire une boucle pour traiter tous les éléments du tableau** et l'écriture et la lecture des formules physiques et mathématiques s'en trouvent facilitées :

Code 10.3 – Opérer élément par élément

```

1 import numpy as np
2
3 x = np.ones(4)
4 print(x + x)  # [2. 2. 2. 2.]
5 print(x - x)  # [0. 0. 0. 0.]
6 print(4 * x * x / 5)  # [0.8 0.8 0.8 0.8]
7 print(x / x)  # [1. 1. 1. 1.]
8
9 r = np.arange(0, 1, 0.3)
10 print(r)  # [0. 0.3 0.6 0.9]
11 s = np.pi * r ** 2
12 print(s)  # [0. 0.28274334 1.13097336 2.54469005]
13
14 time = np.linspace(0, 1, 11)
15 print(time)  # [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]
16 signal = 5 * np.cos(2 * np.pi * 7 * time)
17 print(signal)
18 # [ 5. -1.54508497 -4.04508497 4.04508497 1.54508497 -5.
19 # 1.54508497 4.04508497 -4.04508497 -1.54508497 5. ]

```

 Les opérateurs arithmétiques `+, -, *, /, */` utilisés dans le code 10.3 sont des opérateurs qui agissent sur des tableaux numpy. En informatique, on dit alors qu'on a surchargé les opérateurs `+, -, *, /, */` pour qu'ils puissent agir sur d'autres données que les `int` ou les `float`.

E Opérations matricielles

Numpy possède également la multiplication matricielle comme le montre le code 10.4. C'est l'opérateur `@` qui permet de réaliser cette opération. Il est également très facile de faire de l'algèbre linéaire¹ avec le module `numpy.linalg`.

Code 10.4 – Calcul matriciel

```

1 import numpy as np
2
3 A = np.array([[1, 2], [3, 4]])
4 X = np.array([[.2], [.1]])
5 B = np.array([[.1, .2, .3], [.4, .5, .6]])
6 U = np.array([[0.1], [0.2], [0.3]])
7
8 Xp = A @ X + B @ U

```

1. Calculer une matrice inverse par exemple.

```

9 print(Xp.shape)  # (2, 1)
10 print(Xp)
11 # [[0.54]
12 # [1.32]]

```

F Types de données

La bibliothèque numpy fournit également des types de données supplémentaires à Python. Ce sont en fait des types utilisés par le langage C qui sous-tend les calculs. On y trouve notamment les types `int` non signés désignés par `uint` ou les `float` sur 32 bits.

De nombreuses données concrètes sont représentées par des entiers non signés. C'est le cas par exemple des pixels d'une image dont on code généralement l'intensité sur huit bits, c'est à dire par une valeur comprise entre 0 et $2^8 - 1 = 255$. Numpy propose le type `np.uint8` pour représenter ce type de données.

Code 10.5 – Types de données numpy

```

1 import numpy as np
2
3 unsigned_integer = np.uint8(42)
4 print(unsigned_integer) # 42
5 image = np.zeros((1024, 512), dtype=np.uint8)
6 image[:] = 237
7 print(image)
8 # [[237 237 237 ... 237 237 237]
9 # [237 237 237 ... 237 237 237]
10 # [237 237 237 ... 237 237 237]
11 # ...
12 # [237 237 237 ... 237 237 237]
13 # [237 237 237 ... 237 237 237]
14 # [237 237 237 ... 237 237 237]]

```

R Utiliser le type de données le plus adapté à la nature de la donnée est très important : cela permet d'économiser radicalement l'espace mémoire et d'accélérer les calculs.

G Autres fonctions

Numpy regorge de fonctionnalités. N'hésitez pas à consulter [la documentation en ligne](#). Cette bibliothèque est complétée par `scipy`, bibliothèque scientifique.

- fonctions numpy standards à connaître : `min`, `max`, `std`, `mean`, `sum`,
- `unique` : permet de ne conserver qu'un seul exemplaire de chaque valeur dans un tableau,
- `argmax`, `argmin` : permet de récupérer l'indice du maximum ou du minimum d'un tableau,
- `concatenate` agrège deux tableaux,

- le module `numpy.random` et notamment `shuffle` pour mélanger un tableau.

■ **Exemple 17 — Normalisation d'un ensemble de données par colonne.** On suppose qu'on dispose d'un ensemble de données sous la forme d'un tableau numpy de dimension (n, m) . Chaque colonne représente un paramètre que l'on souhaite normaliser : centrer en zéro et d'écart type égal à un.

Pour chaque échantillon x_{ij} de la ligne i et colonne j , on veut :

$$x_{ij}^{norm} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

où μ_j et σ_j sont les moyennes et écart-type de la colonne j .

On peut réaliser ceci en une seule instruction :

```

1 import numpy as np
2
3 def normalized(data):
4     return (data - np.mean(data, axis=0)) / np.std(data, axis=0)
5
6
7 data = np.random.random((100, 5))
8 data = normalized(data)
9
10 print(np.max(data, axis=0), np.min(data, axis=0), np.mean(data, axis=0), np.std(data,
    axis=0))

```

Troisième partie

Semestre 2

11

TERMINAISON ET CORRECTION

À la fin de ce chapitre, je sais :

- expliquer le problème de la terminaison d'un algorithme
- utiliser un variant de boucle pour prouver la terminaison d'un algorithme
- expliquer la notion de correction d'un algorithme
- utiliser un invariant de boucle pour prouver la correction d'un algorithme

A Un programme se termine-t-il?

Si un programme ne s'arrête pas de lui-même, c'est soit qu'il :

- converge très lentement vers une solution,
- il ne s'arrêtera jamais, tournant en rond, répétant **indéfiniment** les mêmes séquences.

Cette dernière possibilité est à la fois utile, par exemple pour une système d'exploitation et des serveurs qui doivent servir en permanence, et en même temps guère admissible lorsqu'on souhaite avoir la réponse à un problème donné. C'est pourquoi les informaticiens s'intéressent tout particulièrement à la question de l'arrêt de leurs programmes.

Parmi les éléments de notre pseudo-langage algorithmique, les seuls qui pourraient engendrer une exécution infinie d'un programme sont les instructions de boucle, **pour** et **tant que**. C'est pourquoi les sections qui suivent y accordent un intérêt tout particulier.

■ **Exemple 18 — S'arrêtera ? S'arrêtera pas ?.** Imaginons un instant que nous disposions d'un algorithme capable de se prononcer sur la terminaison d'un autre algorithme et nommons le **termine?**. Cet algorithme renvoie vrai si l'algorithme passé en paramètre se termine et faux sinon.

Imaginons que l'on souhaite tester la terminaison d'un autre algorithme nommé **SOUS_TEST** comme dans l'algorithme 19. Que penser alors de cet algorithme? À quelle condition se

termine-t-il? Peut-on le prouver? Naturellement, ce paradoxe souligne l'impossibilité de répondre à la question de la terminaison d'un algorithme.

On dit que le problème de l'arrêt est **indécidable**, tout comme toute propriété non triviale sur un programme (Théorème de Rice [20]).

Algorithme 19 Test de la terminaison de l'algorithme SOUS_TEST

1: **Fonction** TEST_TERMAISON
2: **tant que** TERMINE?(SOUS_TEST) est faux **répéter**
3: ne rien faire.

Théorème 1 — Problème de l'arrêt. Il n'existe pas de programme permettant de dire si un algorithme termine toujours ou non.

■ **Exemple 19 — Suite de Syracuse** . Prenons par exemple l'algorithme de calcul des éléments de la suite de Syracuse (cf. algorithme 20).

Une implémentation dans le langage de votre choix vous montrera que cette fonction renvoie la valeur 1 au bout d'un certain nombre d'itérations ^a. Cet algorithme se termine donc, au moins en apparence.

Il existe d'ailleurs une conjecture nommée «conjecture de Syracuse» qui formule l'hypothèse que la suite de Syracuse de n'importe quel entier strictement positif atteint 1.

Cependant, aucune théorie mathématique ou informatique n'a pu le démontrer à ce jour. Il se peut même qu'on ne puisse pas le démontrer.

a. À partir de ce rang, la suite de Syracuse est cyclique.

Algorithme 20 Calcul des éléments de la suite de Syracuse

```

1: Fonction SYRACUSE( $n$ )
2:    $s \leftarrow n$ 
3:   tant que  $s > 1$  répéter
4:     si  $s$  est pair alors
5:        $s \leftarrow s/2$ 
6:     sinon
7:        $s \leftarrow 3 * s + 1$ 
8:   renvoyer  $s$ 

```

▷ n est un paramètre formel d'entrée

R D'une manière générale, on en peut donc pas conclure sur la terminaison d'un algorithme. Néanmoins, dans de nombreux cas particuliers et en utilisant des propriétés mathématiques, on peut démontrer que des algorithmes se terminent. Ceci est l'objet des deux sections suivantes.

B Variant de boucle

■ **Définition 62 — Variant de boucle (fonction de terminaison).** Un variant de boucle v : $\mathcal{D} \rightarrow \mathbb{N}$ est une fonction :

1. à valeurs entières et positives (\mathbb{N}),
2. strictement décroissante à chaque itération de la boucle,
3. qui dépend des variables de la boucle.

■ **R** Le variant de boucle est une suite de valeurs entières strictement décroissante et positive et donc minorée par 0. Le **théorème de la limite monotone** affirme qu'il peut **atteindre n'importe quelle valeur positive inférieure à sa valeur de départ en un nombre d'itérations fini**. Dès lors, si la condition d'arrêt de la boucle est le franchissement de cette valeur, l'arrêt se déclenche et le programme termine.

■ **R** D'une manière générale, la condition d'arrêt d'une boucle doit impliquer le franchissement d'une valeur particulière par le variant de boucle.

C Démontrer la terminaison d'un algorithme

■ **M** **Méthode 7 — Démontrer la terminaison d'un algorithme** Pour prouver la terminaison d'un algorithme, si cela est possible, il suffit souvent de prouver que les boucles se terminent et donc de :

1. trouver un variant de boucle (entier, positif, strictement décroissant),
2. montrer que le variant est minoré, qu'il franchit nécessairement une valeur limite liée à la condition d'arrêt.

■ **R** La boucle **pour** ne pose généralement ^a pas de problèmes au niveau de la terminaison car le nombre d'itérations est connu et explicité clairement dans la syntaxe de la boucle. Il suffit alors de choisir un variant lié à la variable d'itération de la boucle, typiquement $n - i$ ou i . Ce n'est pas le cas des boucles tant que.

^a généralement, car on peut aussi mal formuler une boucle **pour** dans n'importe quel langage et faire que cela ne soit pas explicite...

■ **Exemple 20 — Fonction produit.** Prenons l'exemple simple du calcul du produit de deux nombres entiers naturels a et b par l'algorithme 21 : la condition d'arrêt de la boucle tant que $c < b$ peut s'écrire $b - c > 0$. Par ailleurs, c est incrémenté à chaque tour de boucle. La

fonction

$$\begin{aligned} v : \mathbb{N} &\longrightarrow \mathbb{N} \\ c &\longmapsto b - c \end{aligned}$$

est une fonction à valeurs entières, strictement décroissante, initialisée à une valeur positive, c'est à dire un variant de boucle. v atteint nécessairement la valeur 0, au bout d'un certain nombre d'itérations. Il s'en suit que la condition d'arrêt $b - c > 0$ devient fausse et déclenche l'arrêt de la boucle. Donc, l'algorithme 21 se termine.

On procède de la même manière pour l'algorithme 22. Construisons un variant de boucle à partir de la variable i . On peut poser :

$$\begin{aligned} v : \mathbb{N} &\longrightarrow \mathbb{N} \\ i &\longmapsto b - i \end{aligned}$$

Il s'agit bien d'une fonction à valeurs entières. Au démarrage, $v(0)$ vaut b qui est une valeur entière positive. À chaque tour de boucle, i est incrémenté implicitement de 1. Donc v est une fonction entière strictement décroissante, un variant de boucle. La condition de sortie de la boucle est $i > b - 1$. On a $v(b) = b - b = 0$. D'après le théorème de la limite monotone, v atteindra la valeur 0 au bout d'un certain nombre d'itération. Donc l'algorithme se termine.

Algorithme 21 Produit de deux nombres entiers naturels, quel pourrait-être un variant?

1: Fonction PRODUIT(a, b)	$\triangleright a \in \mathbb{N}$ et $b \in \mathbb{N}$.
2: $p \leftarrow 0$	
3: $c \leftarrow 0$	$\triangleright c$ est un entier.
4: tant que $c < b$ répéter	
5: $p \leftarrow p + a$	
6: $c \leftarrow c + 1$	
7: renvoyer p	

Algorithme 22 Produit de deux nombres entiers naturels, quel pourrait-être un variant?

1: Fonction PRODUIT(a, b)	$\triangleright a \in \mathbb{N}$ et $b \in \mathbb{N}$.
2: $p \leftarrow 0$	
3: pour $i = 0$ à $b - 1$ répéter	$\triangleright i$ est la variable d'itération de la boucle pour
4: $p \leftarrow p + a$	
5: renvoyer p	

R Il est important de noter que le test de la boucle **tant que** s'exprime avec le signe inférieur. On veillera à procéder de même, c'est à dire à exprimer ces tests à l'aide de l'opérateur $<$ ou $>$, non pas avec l'opérateur \neq . Ceci afin d'éviter l'erreur suivante illustrée par l'exemple

21.

■ **Exemple 21 — Boucle infinie (à ne pas reproduire!).** Reprenons l'algorithme 21 en introduisant une **petite erreur** à la ligne 6 : on incrémente de 2. La condition d'arrêt de la boucle $c \neq b$ ne garantit pas que la fonction $\nu(c) = b - c$ soit minorée par zéro, notamment si b est impair : ν peut prendre des valeurs négatives, ce n'est donc pas un variant de boucle. Cet algorithme 23 ne termine pas toujours.

Algorithme 23 Boucle infinie (à ne pas reproduire!)

```

1: Fonction BINF( $a, b$ ) ▷  $a \in \mathbb{N}$  et  $b \in \mathbb{N}$ .
2:    $p \leftarrow 0$ 
3:    $c \leftarrow 0$  ▷  $c$  est un entier.
4:   tant que  $c \neq b$  répéter
5:      $p \leftarrow p + a$ 
6:      $c \leftarrow c + 2$ 
7:   renvoyer  $p$ 

```

(R) De manière similaire, on évitera les tests d'égalité sur les flottants dans les conditions d'arrêt : on a toutes les chances de ne pas la vérifier, comme le montre l'exemple 22 et l'algorithme 24.

■ **Exemple 22 — Boucle infinie, de la théorie à la pratique (à ne pas reproduire!).** Dans l'algorithme 24, la condition d'arrêt doit pouvoir être vérifiée, en théorie. Dans les faits, elle ne l'est pas pour deux raisons :

1. on ne sait pas si r est un multiple de 0.1,
2. même si c'était le cas, l'utilisation des flottants (norme IEEE 754) ne permet pas d'atteindre la valeur 0.1 exactement, quelque soit le langage de programmation comme cela est expliqué au chapitre ??.

Il faut donc éviter les conditions d'arrêt de boucle comprenant des flottants et, si jamais on y est contraint, utiliser les opérateurs de comparaison $<$ ou $<$, jamais les opérateurs \neq ou $==$.

Algorithme 24 Boucle infinie, condition d'arrêt sur un flottant (**ne pas reproduire!!!**)

```

1: Fonction CAFLOT( $r$ ) ▷  $r \in \mathbb{R}$ .
2:    $x \leftarrow 0.0$  ▷  $x \in \mathbb{R}$ .
3:   tant que  $x \neq r$  répéter
4:      $x \leftarrow x + 0.1$ 

```

R Dans le cas où l'algorithme est récursif, on peut démontrer la terminaison par récurrence sur la variable de la récursivité si celle-ci décroît par sous-traction. On initialise pour la condition d'arrêt de l'algorithme, puis on fait l'hypothèse de récurrence est que l'algorithme p se termine pour $p(n)$ et on montre qu'il se termine pour $p(n+1)$. On conclut sur la terminaison pour tout $n \in \mathbb{N}$.

Si la récurrence décroît par division, on peut essayer de montrer que les paramètres des appels récursifs forment une suite des $(u_n)_{n \in \mathbb{N}}$ strictement décroissante, à valeurs positives et minorée par zéro.

■ **Exemple 23 — Terminaison de l'algorithme récursif 25 de calcul de $n!$.** On procède par récurrence sur n .

Initialisation : pour $n = 0$ ou 1 , l'algorithme renvoie la valeur 1 et termine.

Hérité : Supposons que l'algorithme termine pour le paramètre n . Le calcul de $\text{fact}(n+1)$ termine, car par hypothèse de récurrence $\text{fact}(n)$ termine et l'algorithme renvoie alors la multiplication de n et de ce résultat.

Conclusion : L'algorithme de factoriel termine pour tout $n \in \mathbb{N}$.

On aurait également pu dire que la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_{n+1} = u_n - 1$ et $u_0 = N$ est à valeurs entières, strictement décroissante et initialisée à une valeur positive. Le nombre d'appels récursifs avant d'atteindre la condition d'arrêt est donc fini. L'algorithme se termine.

Algorithme 25 Factoriel récursif

1 : Fonction FACT(n)	
2 : si n $\leqslant 1$ alors	▷ Condition d'arrêt
3 : renvoyer 1	
4 : sinon	
5 : renvoyer n \times FACT(n-1)	▷ Appel récursif

D Le résultat est-il correct?

Le résultat d'un algorithme est-il le bon ? Cette question hante de nombreux informaticiens et plus récemment tout ceux qui sont passés par Parcoursup !

■ **Définition 63 — Correction d'un algorithme.** La correction d'un algorithme est sa capacité à :

1. se terminer,
2. produire un résultat correct, conforme à ce qu'on attend de lui, quelles que soient les entrées.

On dit que la correction est :

partielle si le résultat est correct lorsque l'algorithme se termine,

totale si elle est partielle et que l'algorithme se termine.

Dans le cadre de la programmation structurée, on peut prouver formellement la correction des opérations d'affectation, des enchaînements d'instructions, des structures conditionnelles et des boucles. Néanmoins, nous nous intéressons dans ce qui suit plus particulièrement aux boucles.

E Invariant de boucle

Si la terminaison d'une boucle permet de conclure sur le fait qu'elle s'arrête, elle ne prouve en aucun cas la validité de son résultat. La correction d'une boucle dans le cadre de tests unitaires non exhaustifs n'est pas une preuve non plus. Or, il est difficile voire impossible de tester de manière exhaustive un algorithme et il faut être capable de garantir la correction, qu'elles que soient les entrées.

La démarche générale de démonstration de la correction des boucles est une forme de démonstration par récurrence. La propriété à démontrer est nommée *invariant de boucle*. On procède donc en trois phases : l'initialisation de l'invariant, l'hérédité et la conclusion.

■ **Définition 64 — Invariant de boucle.** Un invariant de boucle est une **propriété** liée aux variables d'un algorithme qui :

1. est vraie avant la boucle,
2. est invariante par les instructions de la boucle à chaque itération,
3. donne le résultat escompté si la condition de boucle est invalidée.

F Démontrer que le résultat d'une boucle est correct

■ **Exemple 24 — Correction de la division euclidienne.** On considère l'algorithme de la division euclidienne (cf. algorithme 26). Pour démontrer sa correction, on peut choisir l'invariant de boucle \mathcal{I} : *on a l'égalité $a = bq + r$* . On procède alors en trois temps :

1. \mathcal{I} est vérifié avant la boucle car $a = qb + r = r$ car $q = 0$.
2. Au cours de l'exécution de la boucle, on choisit une itération quelconque. On suppose qu'au début de cette itération, l'invariant \mathcal{I} est vérifié et on a $a = bq + r$. À la fin de l'itération, $b(q + 1) + (r - b) = bq + b + r - b = bq + r = a$. Donc, \mathcal{I} est vérifié à la fin de l'itération, si elle est vraie à l'entrée de celle-ci.
3. Comme l'invariant de boucle \mathcal{I} est vrai au démarrage de la boucle et invariant par les instructions de la boucle, on a donc, à la fin de la boucle, $a = bq + r$ et $0 \leq r < b$. Cet algorithme délivre donc bien le résultat escompté.

R La plupart du temps, l'invariant de boucle est une propriété liée à l'expression que l'on souhaite calculer grâce à cette boucle. Il est nécessaire de l'exprimer soit mathématiquement

Algorithme 26 Division euclidienne $a = bq + r$

1: Fonction DIVISER(a, b)	▷ les entrées sont des entiers naturels.
2: $r \leftarrow a$	
3: $q \leftarrow 0$	
4: tant que $r \geq b$ répéter	
5: $r \leftarrow r - b$	
6: $q \leftarrow q + 1$	
7: renvoyer (q, r)	

comme dans l'exemple 24 soit en langage naturel. Par exemple, lors de l'étude des graphes, la preuve de la correction du parcours en largeur d'un graphe est donnée. L'invariant utilisé est le suivant : «Pour chaque sommet v ajouté à V et enfilé dans F , il existe un chemin de s à v ». Comme cet invariant n'est pas une expression arithmétique, il est d'autant plus important de soigner la rédaction.

R Dans tous les cas, une démonstration de correction est une rédaction et demande, comme toute preuve, un effort de structuration du propos et de l'intelligibilité.

R D'une manière pratique, on peut vérifier un invariant de boucle en ajoutant une assertion à chaque tour de boucle qui exige de le vérifier, comme dans le code ci-dessous.

```

1 def euclid_div( $a, b$ ):
2      $r=a$ 
3      $q=0$ 
4     assert  $a == b*q + r$ 
5     while  $r >= b$ :
6          $r = r - b$ 
7          $q = q + 1$ 
8         assert  $a == b*q + r$ 
9     return  $q, r$ 

```

■ **Exemple 25 — Correction de l'exponentiation rapide.** L'invariant de la boucle *tant que* de l'algorithme 27 est le suivant :

$\mathcal{I}: i \times e^n = a^n$.

1. Initialisation : \mathcal{I} est vérifié avant la boucle, car $i \times e^n = 1 \times a^n = a^n$.
2. Hérédité : supposons que \mathcal{I} est vérifié à l'entrée de la boucle. On a donc $i \times e^n = a^n$.
 - Si n est pair, i n'est pas modifié par les instructions. Par contre, n est divisé par deux et e devient e^2 . On a donc, à la suite des instructions de la boucle : $i \times e^n \longrightarrow i \times (e^2)^{n/2} = i \times e^n = a^n$, car n est pair et $2 \times (n/2) = n$.

- Si n est impair, les instructions font évoluer i , e et n . Ainsi

$$i \times e^n \longrightarrow (i \times e) \times (e^2)^{n/2} = i \times e^{1+n/2} = i \times e^n = a^n$$

car n est impair.

- Conclusion : \mathcal{I} est invariant par les instructions de la boucle. Comme il est vérifié à l'entrée, il est donc vérifié à la fin de la boucle. Comme, on sort de la boucle lorsque $n = 0$, à la fin on a $i \times e^1 = ie = a^n$. L'algorithme est donc correct.

Algorithme 27 Exponentiation rapide, version itérative

```

1: Fonction EXP( $a, n$ ) ▷  $n$  est un entier naturel.
2:   si  $n == 0$  alors
3:     renvoyer 1
4:   sinon
5:      $e \leftarrow a$ 
6:      $i \leftarrow 1$ 
7:     tant que  $n > 1$  répéter
8:       si  $n$  est pair alors
9:          $e \leftarrow e \times e$ 
10:         $n \leftarrow n/2$ 
11:       sinon
12:          $i \leftarrow e \times i$ 
13:          $e \leftarrow e \times e$ 
14:          $n \leftarrow n/2$ 
15:       renvoyer  $i * e$ 
  
```

```

1 def ite_quick_exp( $a, n$ ):
2   if  $n == 0$ :
3     return 1
4   else:
5      $e = a$ 
6      $i = 1$ 
7      $N = n$ 
8     while  $n > 1$ :
9       assert  $i * e ** n == a ** N$  # Loop invariant
10      if  $n \% 2 == 0$ :
11         $e = e * e$ 
12         $n = n // 2$ 
13      else:
14         $i = i * e$ 
15         $e = e * e$ 
16         $n = n // 2$ 
17      # last computed n was 1 →  $i * e ** 1 = a ** n$ 
18      return  $e * i$ 
  
```

12

COMPLEXITÉ

À la fin de ce chapitre, je sais :

- ☒ définir les concepts de complexité temporelle et complexité mémoire
- ☒ calculer la complexité d'algorithmes simples
- ☒ calculer la complexité d'algorithmes récursifs

A Complexités algorithmiques

Lorsque la taille des données d'entrée à traiter d'un algorithme augmente, le résultat est que l'algorithme met généralement plus de temps à s'exécuter. Pourquoi est-ce un problème ? C'est un problème car, dans les activités humaines, lorsqu'un système fonctionne, on a souvent tendance à lui en demander plus, très vite. Or, si le système développé est capable de gérer trois utilisateurs, peut-il en gérer un million en un temps raisonnable et sans s'effondrer ? C'est peu probable.

La question est donc de savoir :

- comment mesurer la sensibilité d'un algorithme au changement d'échelle des données d'entrée,
- comment mesurer cette sensibilité indépendamment des machines concrètes (processeurs), car on se doute bien que selon la puissance de la machine, le résultat ne sera pas le même.

■ **Définition 65 — Complexité temporelle.** La complexité temporelle est une mesure de l'évolution du temps nécessaire à un algorithme pour s'exécuter correctement en fonction de la taille des données d'entrée. La complexité temporelle est directement liée au nombre d'instructions à exécuter.

■ **Définition 66 — Complexité mémoire ou spatiale.** La complexité mémoire est une mesure de l'évolution de l'espace nécessaire à un algorithme pour s'exécuter correctement en fonction de la taille des données d'entrée. La complexité mémoire est associée à la taille de l'espace mémoire occupé par un algorithme au cours de son exécution.

D'un point de vue opérationnel, si la taille des données d'entrées augmente (n croît), un bon algorithme doit pouvoir délivrer des résultats en un temps fini, même si le nombre d'instructions lié aux boucles ou aux appels récursifs dépend de n . C'est pourquoi la complexité est un calcul asymptotique : on s'intéresse au comportement de l'algorithme lorsque n tend vers l'infini.

B Notation asymptotique

On utilise la notation de Landau O pour la qualifier le comportement asymptotique de la complexité¹. Le tableau 12.1 recense les principales complexités et donne un exemple associé.

■ **Définition 67 — Notation de Landau O .** Soit $f : \mathbb{N} \rightarrow \mathbb{R}_+$ et $g : \mathbb{N} \rightarrow \mathbb{R}_+$. On dit que f ne croît pas plus vite que g et on note $f = O(g)$ si et seulement si :

$$\exists C \in \mathbb{N}, \exists n_i \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_i \Rightarrow f(n) \leq Cg(n)$$

R Cette définition signifie simplement qu'au bout d'un certain rang, la fonction f ne croît jamais plus vite que la fonction g .

Théorème 2 — Propriétés de O . Soit f, f_1, f_2, g, g_1 et $g_2 : \mathbb{N} \rightarrow \mathbb{R}_+$.

1. $\forall k \in \mathbb{N}, O(k.f) = O(f)$
2. $f = O(g) \Rightarrow f + g \in O(g)$
3. $f_1 = O(g_1), f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
4. $f_1 = O(g_1), f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
5. $\forall k \in \mathbb{N}, f = O(g) \Rightarrow k.f = O(g)$

■ **Exemple 26 — Simplification de notations asymptotiques.** Supposons qu'on ait compté le nombre d'opérations d'un algorithme en fonction de n et qu'on ait trouvé : $2n^2 + 4n + 3$. Alors la complexité de l'algorithme est $O(n^2)$. En effet, on a bien $\forall n \in \mathbb{N}, 2n^2 + 4n + 3 < 10n^2$. Si vous avez un doute, étudiez le signe du trinôme $-8n^2 + 4n + 3$.

De même, $10\log(n) + 5(\log(n))^3 + 7n + 3n^2 + 6n^3 = O(n^3)$. Pour le montrer, il suffit d'utiliser le théorème sur les croissances comparées.

1. $O(n)$ se dit «grand o de n».

Complexité	Nom	Description
$O(1)$	Constante	Instructions exécutées un nombre constant de fois Indépendante de la taille de l'entrée
$O(\log(n))$	Logarithmique	Légèrement plus lent lorsque n augmente.
$O(n)$	Linéaire	L'algorithme effectue une tâche constante pour chaque élément de l'entrée.
$O(n \log(n))$	Linéarithmique	L'algorithme effectue une tâche logarithmique pour chaque élément de l'entrée.
$O(n^2)$	Quadratique	L'algorithme effectue une tâche linéaire pour chaque élément de l'entrée.
$O(n^k)$	Polynomiale	Typiquement k tâches linéaires imbriquées.
$O(k^n)$	Exponentielle	L'algorithme effectue une tâche constante sur tous les sous-ensembles de l'entrée.
$O(n!)$	Factorielle	L'algorithme effectue une tâche dont la complexité est multipliée par une quantité croissante proportionnelle à n .

TABLE 12.1 – Hiérarchie des complexités temporelles de la moins complexe à la plus complexe.

Taille de l'entrée	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10^2	2,3 ns	50 ns	230 ns	5 μ s	500 μ s	335 années
10^3	3,4 ns	500 ns	3,45 μ s	500 μ s	500 ms	10^{282} années
10^4	4,6 ns	5 μ s	46 μ s	50 ms	500 s	...
10^5	5,7 ns	50 μ s	575 μ s	5 s	2h20 min	...
10^6	6,9 ns	500 μ s	6,9 ms	500 s	96 jours	...
10^9	10 ns	500 ms	10,4 s	96 jours

TABLE 12.2 – Sur une machine cadencée à 2 Ghz, quelle est la durée prévisible d'exécution d'un algorithme en fonction de la taille des données d'entrée et de sa complexité? On suppose qu'une seule période d'horloge est nécessaire au traitement d'une donnée.

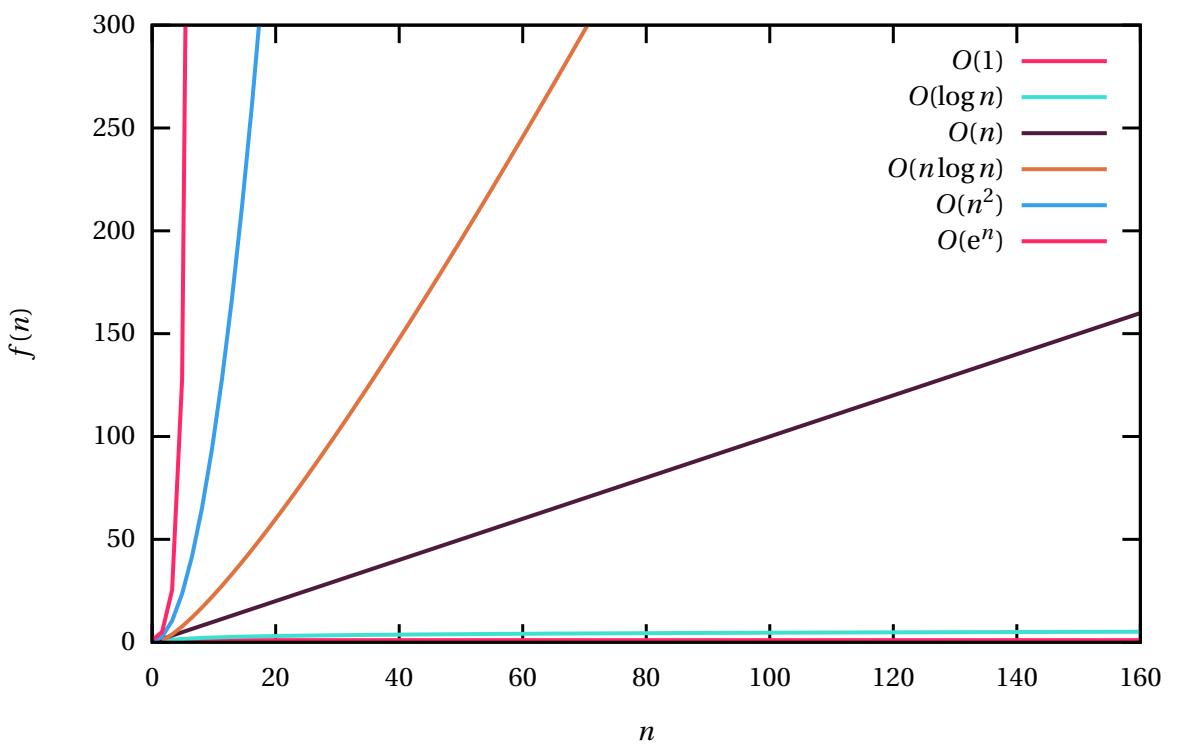


FIGURE 12.1 – Comparaison des croissances des complexités usuelles

C Typologie de la complexité

Selon l'algorithme étudié, on est amené à s'intéresser à différentes complexités :

- La complexité dans le pire des cas, c'est à dire l'estimation du nombre d'instructions nécessaires dans le cas le plus défavorable.
- La complexité dans le meilleur des cas, idem dans le cas le plus favorable.
- La complexité moyenne, c'est à dire une moyenne de la complexité de tous les cas possibles.

R Ces trois calculs de complexité sont parfois nécessaires pour faire un choix d'algorithme et la connaissance statistique de la nature des données d'entrée peut influer sur le ce choix.

D Calcul du coût d'une instruction

Il est difficile de savoir exactement en combien de temps une instruction d'un programme s'exécute pour plusieurs raisons :

1. les compilateurs disposent de fonctions d'optimisation en langage machine ou en code interprétable qui font que le code source n'est pas nécessairement représentatif du code exécuté. Il serait donc nécessaire d'examiner le code exécutable pour statuer.
2. selon les architectures électroniques et les machines virtuelles, le coût d'une même opération varie.

Cependant, dans le cadre d'un calcul de complexité d'un algorithme, on peut s'abstraire de ces considérations électroniciennes et considérer qu'une opération élémentaire i possède un coût constant qu'on notera c .

Dans ce qui suit, on suppose qu'on dispose d'une machine pour tester l'algorithme. On fait l'hypothèse réaliste que les opérations élémentaires suivantes sont réalisées en un temps constant c par cette machine :

- opération arithmétique $+, -, *, /, //, \%$,
- tests $==, !=, <, >$,
- affectation \leftarrow ,
- accès à un élément indicé $t[i]$,
- structures de contrôles (structures conditionnelle et boucles), coût associé négligé,
- échange de deux éléments dans le tableau,
- accès à la longueur d'un tableau.

Finalement, on fait l'approximation supplémentaire qu'une combinaison simple de ces opérations est également réalisée en un temps constant c .

E Calculs classiques de complexité

■ **Exemple 27 — Calcul d'une complexité linéaire.** On souhaite calculer la complexité de l'algorithme 28. La taille du problème dépend de n , c'est à dire la puissance à laquelle on veut calculer le nombre a . En effet, pour différents a , plus petits ou plus grands, l'exécution ne sera pas plus chronophage. Le coût total $C(n)$ associé à cet algorithme peut donc s'écrire :

$$C(n) = c + n \times c = O(nc) = cO(n) = O(n) \quad (12.1)$$

Comme un coût c constant est $O(1)$, car constant en fonction de n , la complexité de l'algorithme 28 est donc linéaire.

Algorithme 28 Calcul de a^n

1: Fonction PUISSANCE(a, n)	▷ a et n sont des entiers naturels
2: $p \leftarrow 1$	▷ coût : c
3: pour $i = 1, \dots, n$ répéter	▷ on répète n fois
4: $p \leftarrow p \times a$	▷ coût : c
5: renvoyer p	

■ **Exemple 28 — Calcul d'une complexité quadratique.** On souhaite calculer la complexité de l'algorithme 29. Le coût total associé à cet algorithme peut donc s'écrire :

$$C(n) = c + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c = c + c \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = c + n^2 c = O(n^2) \quad (12.2)$$

C'est pourquoi, la complexité de l'algorithme 29 est en $O(n^2)$.

Algorithme 29 Produit de deux vecteurs $(n, 1) \times (1, n) \longrightarrow (n, n)$

1: Fonction PVEC(u, v)	▷ u est $(n, 1)$ et v est $(1, n)$
2: $t \leftarrow$ nouveau tableau de taille (n, n)	▷ coût : c
3: pour i de 0 à $n - 1$ répéter	▷ on répète n fois
4: pour j de 0 à $n - 1$ répéter	▷ on répète n fois
5: $t[i, j] \leftarrow u[i] \times v[j]$	▷ coût : c
6: renvoyer t	▷ le résultat

■ **Exemple 29 — Calcul d'une complexité quadratique plus subtile.** On souhaite calculer la complexité de l'algorithme 30 qui fait lui-même appel à un autre algorithme qui calcule une puissance en une complexité linéaire $c_p n$.

Le coût total associé à cet algorithme 30 peut s'écrire :

$$C(n) = c + n \times (c + c_p n) = c + cn + c_p n^2 = O(n^2) \quad (12.3)$$

C'est pourquoi, la complexité de l'algorithme 30 est en $O(n^2)$. C'est pourquoi, il faut veiller à bien étudier tous les coûts, directs et indirects, afin de ne pas conclure hâtivement parce qu'il n'y a qu'une seule boucle que la complexité est linéaire...

a. Oui, on peut faire mieux!

Algorithme 30 Somme de puissances $1 + 2^n + \dots + n^n$

1 : Fonction SOMME_PUISSANCE(n)	▷ n est un entier naturel
2 : $acc \leftarrow 0$	▷ coût : c
3 : pour k de 1 à n répéter	▷ on répète n fois
4 : $acc \leftarrow acc + PUISSANCE(k, n)$	▷ coût : $c + c_p n$
5 : renvoyer acc	▷ le résultat

■ **Exemple 30 — Complexité quadratique.** On souhaite calculer la complexité de l'algorithme 31. On peut calculer le coût total de l'algorithme 31 comme suit :

$$C(n) = c + \sum_{k=1}^n \sum_{i=1}^k c \quad (12.4)$$

$$= c + c \sum_{k=1}^n \sum_{i=1}^k 1 \quad (12.5)$$

$$= c + c \sum_{k=1}^n k \quad (12.6)$$

$$= c + c \frac{n(n+1)}{2} \quad (12.7)$$

$$= O(n^2) \quad (12.8)$$

Algorithme 31 Accumuler

1 : Fonction QACC(n)	
2 : $a \leftarrow 0$	▷ coût : c
3 : pour k de 1 à n répéter	▷ on répète n fois
4 : pour i de 1 à k répéter	▷ on répète k fois
5 : $a \leftarrow a + i$	▷ coût : c
6 : renvoyer a	▷ le résultat

■ **Exemple 31 — Complexité polynomiale.** On souhaite calculer la complexité de l'algorithme 32. On suppose qu'on connaît la complexité de f et qu'elle est linéaire. On peut donc

calculer le coût total de l'algorithme 32 comme suit :

$$c = c + \sum_{k=1}^n \sum_{i=1}^k c + c_f i = c + c \sum_{k=1}^n \sum_{i=1}^k 1 + c_f \sum_{k=1}^n \sum_{i=1}^k i \quad (12.9)$$

$$= c + c \frac{n(n+1)}{2} + \sum_{k=1}^n \frac{k(k+1)}{2} = c + c \frac{n(n+1)}{2} + \sum_{k=1}^n \frac{k}{2} + \frac{k^2}{2} \quad (12.10)$$

$$= c + c \frac{n(n+1)}{2} + c_f \frac{n(n+1)}{4} + c_f \frac{n(n+1)(2n+1)}{12} \quad (12.11)$$

$$= O(n^3) \quad (12.12)$$

Algorithme 32 Appliquer une fonction et accumuler

```

1: Fonction FACC( $n$ )                                ▷ Applique  $f$  et accumule  $n$  fois
2:    $a \leftarrow 0$                                      ▷ coût :  $c$ 
3:   pour  $k$  de 1 à  $n$  répéter                  ▷ on répète  $n$  fois
4:     pour  $i$  de 1 à  $k$  répéter          ▷ on répète  $k$  fois
5:        $a \leftarrow a + f(i)$                       ▷ coût :  $c + c_f i$ 
6:   return  $c$                                      ▷ le résultat

```

D'autres exemples de calcul de complexité sont abordés dans ce cours, notamment au chapitre 8 et au chapitre 13.

F Exemple de la recherche dichotomique

■ **Exemple 32 — Recherche dichotomique.** L'algorithme de recherche dichotomique 40 est un exemple d'algorithme de type diviser pour régner : la division du problème en sous-problèmes est opérée via la ligne 5. La résolution des sous-problèmes est effectuée par des appels récursifs. La combinaison des résultats n'est pas explicite mais s'effectue sur le tableau lui-même grâce aux indices g et d .

R La recherche dichotomique est donc bien un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est à dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux.

Supposons que le tableau d'entrée de l'algorithme possède n éléments et que le nombre d'opérations nécessaires à l'algorithme est $T(n)$. Pour simplifier le calcul, on fait l'hypothèse que n est une puissance de deux. On peut expliciter formellement la relation de récurrence qui existe entre $T(n)$ et $T(n/2)$: on a $T(n) = T(n/2) + c$, car en dehors de l'appel récursif, le coût de l'exécution vaut c . Les différents appels récursifs sont illustrés sur la figure 13.3.

Algorithme 33 Recherche récursive d'un élément par dichotomie dans un tableau trié

```

1: Fonction REC_DICH(t, g, d, elem)
2:   si g > d alors                               ▷ Condition d'arrêt
3:     renvoyer l'élément n'a pas été trouvé
4:   sinon
5:     m ← (g+d)//2                                ▷ Diviser
6:     si t[m] = elem alors
7:       renvoyer m
8:     sinon si elem < t[m] alors
9:       renvoyer REC_DICH(t, g, m-1, elem)          ▷ résoudre
10:    sinon
11:      renvoyer REC_DICH(t, m+1, d, elem)          ▷ résoudre

```

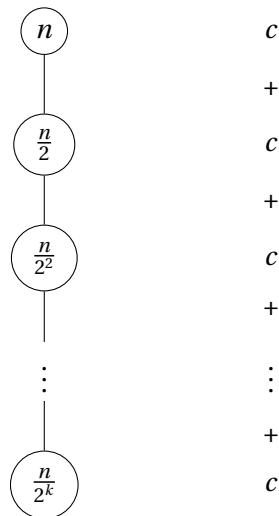


FIGURE 12.2 – Structure d'arbre et appels récursifs pour la récurrence de la recherche dichotomique : $T(n) = T(n/2) + c$ et $\frac{n}{2^k} = 1$. Hors appel récursif, la fonction opère un nombre constant d'opérations c .

On peut donc écrire :

$$T(n) = T(n/2) + c \quad (12.13)$$

$$= T(n/4) + c + c = T(n/4) + 2c \quad (12.14)$$

$$= T(n/8) + 3c \quad (12.15)$$

$$= \dots \quad (12.16)$$

$$= T(n/2^k) + kc \quad (12.17)$$

$$= T(1) + kc \quad (12.18)$$

D'après l'algorithme 40, la condition d'arrêt s'effectue en un nombre constant d'opérations : $T(1) = O(1)$. Donc on a $T(n) = O(k)$. Or, on a $\frac{n}{2^k} = 1$. Donc $k = \log_2 n$ et $T(n) = O(\log n)$.

On peut également le montrer plus mathématiquement en considérant $k = \log_2 n$ et la suite $(u_k)_{k \in \mathbb{N}^*}$ telle que $u_k = u_{k-1} + c$ et $u_1 = c$. C'est une suite arithmétique, $u_k = kc$. D'où le résultat.

G Exemple de l'exponentiation rapide

L'algorithme naïf de l'exponentiation (cf. algorithme 41) qui permet d'obtenir a^n en multipliant a par lui-même n fois n'est pas très efficace : sa complexité étant en $O(n)$.

Algorithme 34 Exponentiation naïve a^n

```

1: Fonction EXP_NAIVE(a,n)
2:   api  $\leftarrow 1$ 
3:   pour i de 0 à  $n - 1$  répéter
4:     api  $\leftarrow$  api  $\times$  a
5:   renvoyer api

```

Or, l'exponentiation est une opération très récurrente qu'il est nécessaire de pouvoir exécuter le plus rapidement possible. L'exponentiation rapide (cf. algorithme 42) propose une version récursive de type diviser pour régner dont la complexité est en $O(\log n)$.

Algorithme 35 Exponentiation rapide a^n

```

1: Fonction EXP_RAPIDE(a,n)
2:   si n = 0 alors                                $\triangleright$  Condition d'arrêt
3:     renvoyer 1
4:   sinon si n est pair alors
5:     p  $\leftarrow$  EXP_RAPIDE(a, n//2)                 $\triangleright$  Appel récursif
6:     renvoyer p  $\times$  p
7:   sinon
8:     p  $\leftarrow$  EXP_RAPIDE(a, (n-1)//2)            $\triangleright$  Appel récursif
9:     renvoyer p  $\times$  a

```

L'analyse de l'algorithme 42 montre que :

- c'est un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est à dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux²,
- l'évolution du coût ne dépend pas de a mais de n , c'est à dire l'exposant.

On peut procéder de la même manière qu'avec l'algorithme 40 pour calculer la complexité et s'appuyer sur l'arbre de la figure 13.3. Pour simplifier le calcul, on peut considérer que la taille du problème est divisée par deux. Le coût hors appel récursif est constant car il s'agit de multiplications. On a donc $T(n) = O(\log n)$.

2. à un près si n est pair

H Exemple du tri fusion

Les tris génériques abordés jusqu'à présent, par sélection ou insertion, présentent des complexités polynomiales en $O(n^2)$ dans le pire des cas. L'algorithme de tri fusion a été inventé par John von Neumann en 1945. C'est un bel exemple d'algorithme de type diviser pour régner avec $r = 2$ et $d = 2$, c'est à dire deux appels récursifs par chemin d'exécution et une division de la taille du problème par deux (cf. figure 12.3). Il permet de dépasser cette limite et d'obtenir un tri générique de complexité logarithmique. Ce tri est comparatif, il peut s'effectuer en place et les implémentations peuvent être stables.

Son principe (cf. algorithmes 36, 38 et 37) est simple : transformer le tri d'un tableau à n éléments en sous-tableaux ne comportant qu'un seul élément³ puis les recombiner en un seul tableau en conservant l'ordre. L'algorithme est divisé en deux fonctions :

- TRI_FUSION qui opère concrètement la division et la résolution des sous-problèmes,
- FUSION qui combine les solutions des sous-problèmes en fusionnant deux sous-tableaux triés.

Il n'y a pas de pire ou meilleur cas : l'algorithme effectue systématiquement la découpe et la fusion des sous-tableaux.

Pour le calcul de la complexité, on a la relation de récurrence $T(n) = 2T(n/2) + f(n)$ où $f(n)$ représente le nombre d'opérations élémentaires nécessaires pour fusionner deux sous-tableaux de taille $n/2$. La complexité de la fonction FUSION est linéaire, car on effectue n fois les instructions élémentaires de la boucle. Donc on peut simplifier la récurrence en $T(n) = 2T(n/2) + n$.

On fait l'hypothèse que n est une puissance de deux pour simplifier le calcul. Soient les suites auxiliaires $u_k = T(2^k)$ et $v_k = u_k/2^k$. La récurrence s'écrit alors :

$$T(2^k) = T(2^{k-1}) + 2^k = u^k = u^{k-1} + 2^k$$

On en déduit que la suite v_k vérifie : $v_k = v_{k-1} + 1$. v^k est une suite arithmétique de raison 1. Si on suppose que $u_0 = 0$, c'est-à-dire le coût de traitement d'un tableau vide est nul, alors $v_0 = 0$. On en déduit que : $v_k = v_0 + k \times 1 = k$ et donc :

$$u_k = k2^k = T(2^k)$$

La taille du tableau étant $n = 2^k$, la complexité de l'algorithme est :

$$T(n) = n \log_2 n$$

I Synthèse

3. et donc déjà triés!

Algorithme 36 Tri fusion

```

1: Fonction TRI_FUSION(t)
2:    $n \leftarrow$  taille de t
3:   si  $n < 2$  alors
4:     renvoyer t
5:   sinon
6:      $t_1, t_2 \leftarrow$  DÉCOUPER_EN_DEUX(t)
7:     renvoyer FUSION(TRI_FUSION( $t_1$ ), TRI_FUSION( $t_2$ ))

```

Algorithme 37 Découper en deux

```

1: Fonction DÉCOUPER_EN_DEUX(t)
2:    $n \leftarrow$  taille de t
3:    $t_1, t_2 \leftarrow$  deux listes vides
4:   pour  $i = 0$  à  $n//2 - 1$  répéter
5:     AJOUTER( $t_1$ , t[i])
6:   pour  $j = n//2$  à  $n - 1$  répéter
7:     AJOUTER( $t_2$ , t[j])
8:   renvoyer  $t_1, t_2$ 

```

Algorithme 38 Fusion de deux sous-tableaux triés

```

1: Fonction FUSION( $t_1, t_2$ )
2:    $n_1 \leftarrow$  taille de  $t_1$ 
3:    $n_2 \leftarrow$  taille de  $t_2$ 
4:    $n \leftarrow n_1 + n_2$ 
5:   t  $\leftarrow$  une liste vide
6:    $i_1 \leftarrow 0$ 
7:    $i_2 \leftarrow 0$ 
8:   pour k de 0 à  $n - 1$  répéter
9:     si  $i_1 \geq n_1$  alors
10:       AJOUTER(t,  $t_2[i_2]$ )
11:        $i_2 \leftarrow i_2 + 1$ 
12:     sinon si  $i_2 \geq n_2$  alors
13:       AJOUTER(t,  $t_1[i_1]$ )
14:        $i_1 \leftarrow i_1 + 1$ 
15:     sinon si  $t_1[i_1] \leq t_2[i_2]$  alors
16:       AJOUTER(t,  $t_1[i_1]$ )
17:        $i_1 \leftarrow i_1 + 1$ 
18:     sinon
19:       AJOUTER(t,  $t_2[i_2]$ )
20:        $i_2 \leftarrow i_2 + 1$ 
21:   renvoyer t

```

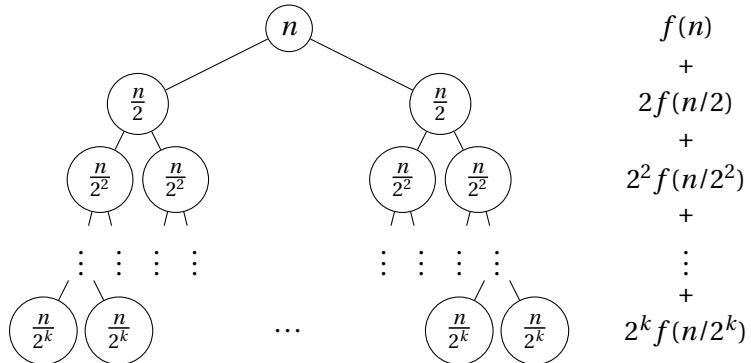


FIGURE 12.3 – Structure d’arbre et appels récursifs pour le tri fusion : $T(n) = 2T(n/2) + f(n)$ et $\frac{n}{2^k} = 1$. La fonction FUSION opère un nombre d’opérations $f(n)$.



Méthode 8 — Complexité d'une fonction

Pour trouver la complexité d'une fonction :

1. Trouver le(s) paramètre(s) de la fonction étudiée qui influe(nt) sur la complexité.
2. Déterminer si, une fois ce(s) paramètre(s) fixé(s), il existe un pire ou un meilleur des cas.
3. Calculer la complexité en :
 - calculant éventuellement une somme d’entiers (fonction itérative),
 - posant une formule récurrente sur la complexité (fonction récursive).

Le tableau 12.3 récapitule les complexités des algorithmes récursifs à connaître.

Réurrence	Complexité	Algorithmes
$T(n) = 1 + T(n - 1)$	$\rightarrow O(n)$	factorielle
$T(n) = 1 + T(n/2)$	$\rightarrow O(\log n)$	dichotomie, exponentiation rapide
$T(n) = n + 2T(n/2)$	$\rightarrow O(n \log n)$	tri fusion, transformée de Fourier rapide

TABLE 12.3 – Récurrences et complexités associées utiles et à connaître

13

DIVISER POUR RÉGNER

À la fin de ce chapitre, je sais :

- ☒ énoncer le principe d'un algorithme de type diviser pour régner
- ☒ distinguer les cas d'utilisation de ce principe (sous-problèmes indépendants)
- ☒ évaluer la complexité d'un algorithme diviser pour régner

A Diviser pour régner

■ **Définition 68 — Algorithme de type diviser pour régner.** L'idée centrale d'un algorithme de type diviser pour régner est de décomposer le problème étudié en plusieurs sous-problèmes de taille réduite. Ces algorithmes peuvent éventuellement être exprimés récursivement et sont souvent très efficaces.

On distingue trois étapes lors de l'exécution d'un tel algorithme :

1. la division du problème en sous-problèmes qu'on espère plus simples à résoudre,
2. la résolution des sous-problèmes, c'est à cette étape que l'on peut faire appel à la récursivité,
3. la combinaison des solutions des sous-problèmes pour construire la solution au problème.

R On devrait donc logiquement nommer ces algorithmes diviser, résoudre et combiner!

 **Vocabulary 12 — Divide and conquer** ⇔ diviser pour régner.

Très souvent¹, les algorithmes de type diviser pour régner sont exprimés récursivement.

1. mais pas toujours

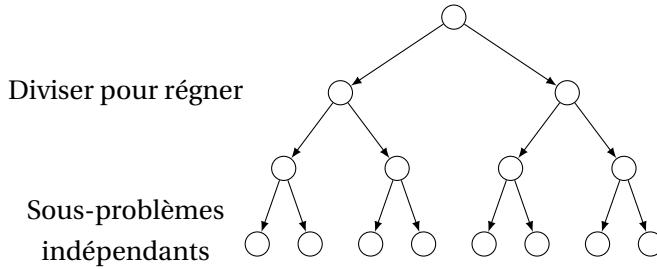


FIGURE 13.1 – Principe de la décomposition d'un problème en sous-problèmes indépendants pour un algorithme de type diviser pour régner.

À partir d'une taille de problème \mathcal{P} de taille n , la division en sous-problèmes aboutit soit à $n = 1$ soit à $n = s$, s étant alors une taille pour laquelle on sait résoudre le problème facilement et efficacement. Les étapes 7 et 9 de l'algorithme 39 sont facilement descriptibles à l'aide d'un arbre, comme le montre la figure 13.2. La hauteur de cet arbre peut être appréciée et quantifiée. Elle sert notamment à calculer la complexité liée aux algorithmes récursifs.

Algorithme 39 Diviser, résoudre et combiner (Divide And Conquer)

```

1: Fonction DRC( $\mathcal{P}$ )                                 $\triangleright \mathcal{P}$  est un problème de taille  $n$ 
2:    $r \leftarrow$  un entier  $\geq 1$                        $\triangleright$  pour générer  $r$  sous-problèmes à chaque étape
3:    $d \leftarrow$  un entier  $> 1$                        $\triangleright$  on divise par  $d$  la taille du problème
4:   si  $n < s$  alors                             $\triangleright$  Condition d'arrêt,  $s$  est un seuil à déterminer
5:     renvoyer RÉSOUUDRE( $n$ )
6:   sinon
7:      $(\mathcal{P}_1, \dots, \mathcal{P}_r) \leftarrow$  Diviser  $\mathcal{P}$  en  $r$  sous problèmes de taille  $n/d$ .
8:     pour  $i$  de 1 à  $r$  répéter
9:        $S_i \leftarrow$  DRC( $\mathcal{P}_i$ )                       $\triangleright$  Appels récursifs
10:      renvoyer COMBINER( $S_1, \dots, S_r$ )

```

Sur la figure 13.2, on a choisi de représenter un algorithme de type diviser pour régner dont le problème associé \mathcal{P} est de taille n . L'étape de division en sous-problèmes divise par d le problème initial et nécessite r appels récursifs. Si $D(n)$ est la complexité de la partie division et $C(n)$ la complexité de l'étape de combinaison des résultats, alors on peut décrire la complexité $T(n)$ d'un tel algorithme par la relation de récurrence $T(n) = rT(n/d) + D(n) + C(n)$ et $T(s)$ une constante. Sur la figure 13.2 on a choisi $r = 3$ pour la représentation graphique.

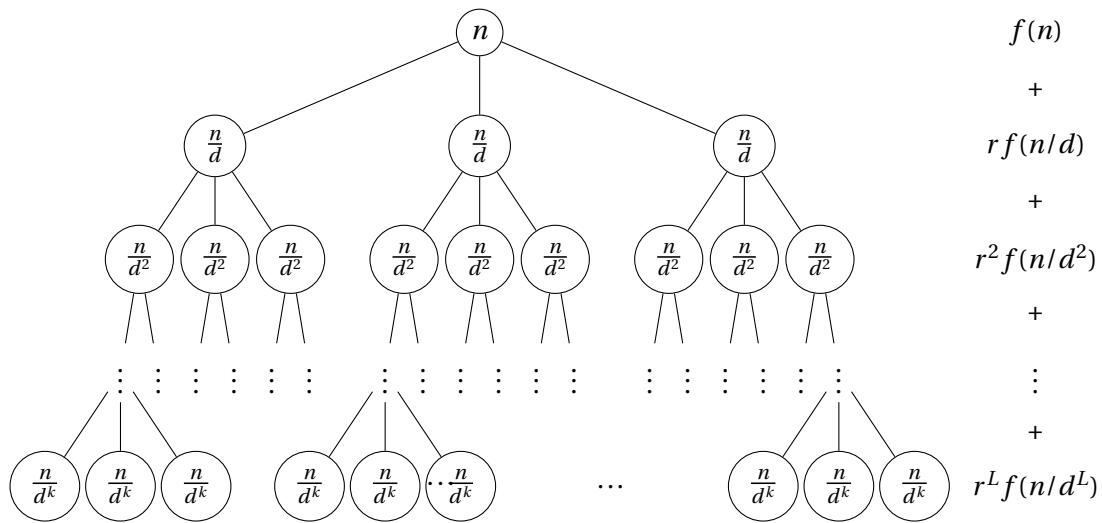


FIGURE 13.2 – Structure d’arbre et appels récursifs pour la récurrence : $T(n) = rT(n/d) + f(n)$ ET $n/d^k = s$. On a choisi $r = 3$ pour l’illustration, c’est-à-dire chaque nœud possède trois enfants au maximum : on opère trois appels récursifs à chaque étape de l’algorithme.

B Exemple de la recherche dichotomique

■ **Exemple 33 — Recherche dichotomique.** L'algorithme de recherche dichotomique 40 est un exemple d'algorithme de type diviser pour régner : la division du problème en sous-problèmes est opérée via la ligne 5. La résolution des sous-problèmes est effectuée par des appels récursifs. La combinaison des résultats n'est pas explicite mais s'effectue sur le tableau lui-même grâce aux indices g et d.

R La recherche dichotomique est donc bien un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est à dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux.

Algorithme 40 Recherche récursive d'un élément par dichotomie dans un tableau trié

```

1: Fonction REC_DICH(t, g, d, elem)
2:   si g > d alors                                ▷ Condition d'arrêt
3:     renvoyer l'élément n'a pas été trouvé
4:   sinon
5:     m ← (g+d)/2                                ▷ Diviser
6:     si t[m] = elem alors
7:       renvoyer m
8:     sinon si elem < t[m] alors                ▷ résoudre
9:       REC_DICH(t, g, m-1, elem)
10:    sinon
11:      REC_DICH(t, m+1, d, elem)                ▷ résoudre

```

Grâce à la figure 13.3, on peut calculer le nombre d'opérations élémentaires $T(n)$ nécessaires à l'exécution de l'algorithme 40. On fait l'hypothèse que, hors appel récursif, la fonction REC_DICH nécessite un nombre constant d'opérations c . On peut expliciter formellement la relation de récurrence qui existe entre $T(n)$ et $T(n/2)$: on a $T(n) = T(n/2) + c$. on peut donc écrire :

$$T(n) = T(n/2) + c \quad (13.1)$$

$$= T(n/4) + c + c = T(n/4) + 2c \quad (13.2)$$

$$= T(n/8) + 3c \quad (13.3)$$

$$= \dots \quad (13.4)$$

$$= T(n/2^k) + kc \quad (13.5)$$

$$= T(1) + kc \quad (13.6)$$

D'après l'algorithme 40, la condition d'arrêt s'effectue en un nombre constant d'opérations : $T(1) = O(1)$. Donc on a $T(n) = O(k)$. Or, on a $\frac{n}{2^k} = 1$. Donc $k = \log_2 n$ et $T(n) = O(\log n)$.

On peut également le montrer plus mathématiquement en considérant $k = \log_2 n$ et la suite $(u_k)_{k \in \mathbb{N}^*}$ telle que $u_k = u_{k-1} + c$ et $u_1 = c$. C'est une suite arithmétique, $u_k = kc$. D'où le résultat.

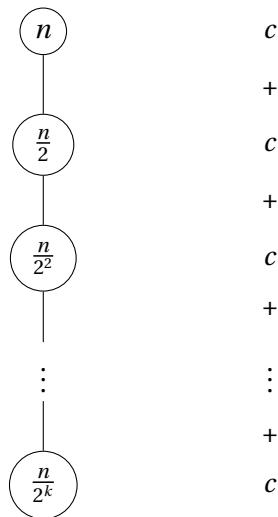


FIGURE 13.3 – Structure d’arbre et appels récursifs pour la récurrence de la recherche dichotomique : $T(n) = T(n/2) + c$ et $\frac{n}{2^k} = 1$. Hors appel récursif, la fonction opère un nombre constant d’opérations c .

C Exemple de l'exponentiation rapide

L'algorithme naïf de l'exponentiation (cf. algorithme 41) qui permet d'obtenir a^n en multipliant a par lui-même n fois n'est pas très efficace : sa complexité étant en $O(n)$.

Algorithme 41 Exponentiation naïve a^n

```

1: Fonction EXP_NAIVE(a,n)
2:   api ← 1
3:   pour i de 0 à  $n - 1$  répéter
4:     api ← api × a
5:   renvoyer api

```

Or, l'exponentiation est une opération très récurrente qu'il est nécessaire de pouvoir exécuter le plus rapidement possible. L'exponentiation rapide (cf. algorithme 42) propose une version récursive de type diviser pour régner dont la complexité est en $O(\log n)$.

L'analyse de l'algorithme 42 montre que :

- c'est un cas particulier d'algorithme diviser pour régner avec $r = 1$ et $d = 2$, c'est à dire un seul appel récursif par chemin d'exécution et une division de la taille du problème par deux²,
- l'évolution du coût ne dépend pas de a mais de n , c'est à dire l'exposant.

On peut procéder de la même manière qu'avec l'algorithme 40 pour calculer la complexité et s'appuyer sur l'arbre de la figure 13.3. Pour simplifier le calcul, on peut considérer que la

2. à un près si n est pair

Algorithme 42 Exponentiation rapide a^n

```

1: Fonction EXP_RAPIDE(a,n)
2:   si n = 0 alors                                $\triangleright$  Condition d'arrêt
3:     renvoyer 1
4:   sinon si n est pair alors
5:     p  $\leftarrow$  EXP_RAPIDE(a, n//2)                 $\triangleright$  Appel récursif
6:     renvoyer p  $\times$  p
7:   sinon
8:     p  $\leftarrow$  EXP_RAPIDE(a, (n-1)//2)            $\triangleright$  Appel récursif
9:     renvoyer p  $\times$  p  $\times$  a

```

taille du problème est divisée par deux. Le coût hors appel récursif est constant car il s'agit de multiplications. On a donc $T(n) = O(\log n)$.

R L'algorithme 42 n'est pas à récursivité terminale. Sa version itérative est moins évidente. Celle-ci est détaillée par l'algorithme 43

Algorithme 43 Exponentiation rapide a^n (version itérative)

```

1: Fonction EXP_RAPIDE_ITE(a, n)                   $\triangleright$  a et n sont des entiers naturels
2:   si n = 0 alors
3:     renvoyer 1
4:   p  $\leftarrow$  a
5:   m  $\leftarrow$  1
6:    $n_{it} \leftarrow \lfloor \log_2(n) \rfloor$ 
7:   pour i de 1 à  $n_{it}$  répéter
8:     m  $\leftarrow$  n//2 $^{n_{it}-i}$ 
9:     si m est pair alors
10:      p  $\leftarrow$  p  $\times$  p
11:    sinon
12:      p  $\leftarrow$  p  $\times$  p  $\times$  a
13:   renvoyer p

```

14

ALGORITHMES GLOUTONS

À la fin de ce chapitre, je sais :

- expliquer le principe d'un algorithme glouton
- citer des cas d'utilisation classiques de ce principe
- coder un algorithme glouton en Python

A Princip

■ **Définition 69 — Optimisation.** Un problème d'optimisation \mathcal{P} nécessite de déterminer les conditions dans lesquelles ce problème présente une caractéristique optimale au regard d'un critère.

■ **Exemple 34 — Problèmes d'optimisation.** La plupart des problèmes de tous les jours sont des problèmes d'optimisation :

- comment répartir équitablement des tâches selon certains critères ?
- comment choisir le plus court chemin pour aller d'un point à un autre ?
- comment choisir ses actions pour optimiser un portefeuille et son rendement ?
- comment choisir le régime moteur pour économiser un maximum de carburant ?
- comment choisir des articles dans un supermarché en respectant un budget et d'autres contraintes simultanément ?
- comment faire ses valises en emportant à la fois le plus d'affaires possibles et le plus de valeur possible au global ?

■ **Définition 70 — Algorithme glouton.** Un algorithme glouton décompose un problème en

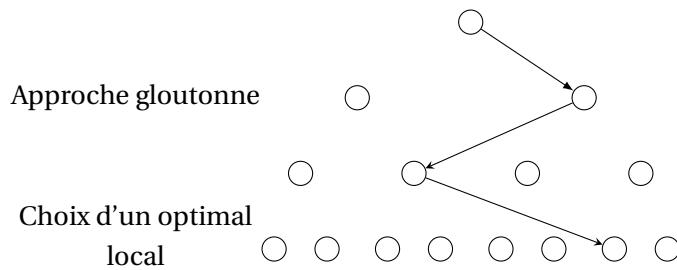


FIGURE 14.1 – Étape de résolution d'un problème par décomposition en sous-problèmes et approche gloutonne.

sous-problèmes et le résout en :

1. construisant une solution partielle en effectuant à chaque étape le meilleur choix local,
2. espérant que ces choix locaux conduiront à un résultat global optimal.



Vocabulary 13 — Greedy algorithms ↪ les algorithmes gloutons

La figure 14.1 illustre le fonctionnement d'un algorithme glouton. Un problème de décomposition devient une suite de choix optimaux localement.

R La plupart du temps, un algorithme glouton est appliqué à un problème d'optimisation. Le résultat n'est pas toujours optimal. Mais l'espérance fait vivre : certains algorithmes gloutons obtiennent une solution optimale ! Comme ils sont souvent assez simples à implémenter par rapport aux autres algorithmes d'optimisation, ils représentent une solution précieuse.

■ **Définition 71 — Glouton optimal.** On dit qu'un algorithme glouton est optimal s'il produit une solution optimale au problème d'optimisation associé.

■ **Exemple 35 — Algorithmes gloutons optimaux.** Parmi les algorithmes au programme, il existe des algorithmes gloutons optimaux :

- Dijkstra (plus court chemin),
- Prim et Kruskal (arbres recouvrants),
- codage d'Huffman.

B Modélisation

On considère un ensemble \mathcal{E} d'éléments parmi lesquels on doit faire des choix pour optimiser le problème \mathcal{P} . On construit une solution \mathcal{S} séquentiellement via un algorithme glouton en suivant la procédure décrite sur l'algorithme 44. Il ne reste plus qu'à préciser, selon le problème

considéré :

- le choix de l'élément **optimal localement** dans \mathcal{E} ,
- le test d'une solution pour savoir si celle-ci est complète ou partielle,
- l'ajout d'un élément à une solution.

Algorithme 44 Principe d'un algorithme glouton

```

1: Fonction GLOUTON( $\mathcal{E}$ )                                 $\triangleright \mathcal{E}$  un ensemble d'éléments
2:    $\mathcal{S} \leftarrow \emptyset$ 
3:   tant que  $\mathcal{S}$  pas complète et  $\mathcal{E}$  pas vide répéter
4:      $e \leftarrow \text{CHOISIR\_ÉLÉMENT\_LOCAL\_OPTIMAL}(\mathcal{E})$            $\triangleright \text{le meilleur localement!}$ 
5:     si l'ajout de  $e$  à  $\mathcal{S}$  est une solution possible alors
6:        $\mathcal{S} \leftarrow \mathcal{S} + e$ 
7:     Retirer  $e$  de  $\mathcal{E}$                                           $\triangleright$  Si pas déjà fait en 4
8:   renvoyer  $\mathcal{S}$ 
  
```

a Exemple de l'occupation de la place au port

Un port de plaisance gère l'occupation d'une place vacante et ouverte à la réservations pour une durée limitée. Certains plaisanciers sont de passage et font des réservations. L'objectif fixé est de sélectionner **un maximum de réservations compatibles** afin de satisfaire un maximum de clients.

On désigne par \mathcal{E} l'ensemble des demandes des clients. Pour toute demande $e \in \mathcal{E}$, on a la possibilité d'accéder à la date de début $d(e)$ de la demande ainsi qu'à la date de fin $f(e)$. On cherche donc à trouver un sous-ensemble de \mathcal{E} constitué de demandes compatibles et de cardinal maximum.

On dénote l'intervalle de temps d'occupation associé à une demande e par $[d(e), f(e)]$. La compatibilité de deux demandes e_i et e_j peut alors être formalisée ainsi :

$$[d(e_i), f(e_i)] \cap [d(e_j), f(e_j)] = \emptyset \quad (14.1)$$

L'algorithme glouton 45 permet de résoudre ce problème de planning.

Cet algorithme est bien glouton car :

1. la construction de l'ensemble $\mathcal{S} = \{s_1, \dots, s_k\}$ s'effectue de manière séquentielle,
2. le choix effectué à chaque tour de boucle est le meilleur en terme de compatibilité : les demandes peuvent éventuellement s'enchaîner grâce à la ligne 8.

L'ensemble \mathcal{S} étant constitué de demandes compatibles, l'algorithme aboutit à une solution. Cependant, on peut se demander si celle-ci est optimale, c'est à dire de cardinal maximum : a-t-on satisfait un maximum de clients ? Ne peut-on faire mieux ?

Algorithme 45 Réservation d'une place au port de Brest

```

1: Fonction GÉRER_LA_PLACE( $\mathcal{E}$ )                                $\triangleright \mathcal{E}$  un ensemble de demandes  $e_i$ 
2:    $\mathcal{S} \leftarrow \emptyset$ 
3:   Trier  $\mathcal{E}$  par ordre de valeurs  $f(e_i)$  croissantes            $\triangleright$  Dates de fin croissantes
4:   Ajouter à  $\mathcal{S}$  la demande qui se termine le plus tôt.
5:   tant que  $\mathcal{E}$  pas vide répéter
6:      $e \leftarrow$  retirer l'élément de  $\mathcal{E}$  qui se termine le plus tôt
7:      $s \leftarrow$  dernier élément ajouté dans  $\mathcal{S}$ 
8:     si  $d(e) \geq f(s)$  alors                                 $\triangleright$  Est-ce une solution?
9:       Ajouter  $e$  à  $\mathcal{S}$ 
10:  renvoyer  $\mathcal{S}$ 

```

★ **b Preuve de l'optimalité** \dashrightarrow HORS PROGRAMME

Théorème 3 — L'algorithme 45 aboutit à une solution optimale.

Démonstration par induction sur les ensembles solutions $\mathcal{S} = \{s_1, s_2, \dots, s_j\}$, $j \in \llbracket 1, k \rrbracket$. On suppose qu'on peut ordonner les ensembles \mathcal{E} et \mathcal{S} par date de fin croissante et qu'il y a une date à laquelle on a démarré le service de location¹. Pour un ensemble de demandes $\Delta = \{\delta_1, \dots, \delta_r\}$ ainsi ordonné, les demandes $\delta_1, \dots, \delta_r$ sont compatibles si et seulement si :

$$d(\delta_1) < f(\delta_1) \leq d(\delta_2) < f(\delta_2) \leq \dots \leq d(\delta_r) < f(\delta_r) \quad (14.2)$$

On procède en deux temps en montrant :

1. d'abord qu'il existe une solution optimale $\mathcal{O} = \{o_1, \dots, o_m\}$ telle que les premiers éléments de $\mathcal{S} = \{s_1, \dots, s_k\}$ coïncident avec ceux de \mathcal{O} ,
2. puis que $\mathcal{S} = \mathcal{O}$.

Initialisation : pour $j = 1$, on a $\mathcal{S} = s_1$, s_1 étant la demande qui se termine le plus tôt. Par rapport à la solution optimale $\mathcal{O} = \{o_1, \dots, o_m\}$, on a nécessairement $f(s_1) \leq f(o_1)$ et donc :

$$d(s_1) < f(s_1) \leq d(o_2) < f(o_2) \leq \dots \leq d(o_m) < f(o_m) \quad (14.3)$$

On en conclut que $\{s_1, o_2, \dots, o_m\}$ est compatible avec \mathcal{O} . Pour $j = 1$ et \mathcal{S} est donc optimale.

Pour l'héritage, on suppose que la solution $\mathcal{S} = \{s_1, \dots, s_j\}$ est compatible avec $\mathcal{O} = \{o_1, \dots, o_m\}$, c'est à dire que : $\{s_1, s_2, \dots, s_j, o_{j+1}, \dots, o_m\}$ est optimale. L'algorithme 45 assure que s_{j+1} est choisi dans $\mathcal{E} \setminus \{s_1, \dots, s_j\}$, de telle manière que la date de fin est minimale. D'après notre hypothèse, o_{j+1} est choisi dans $\mathcal{E} \setminus \{s_1, \dots, s_j\}$, de telle manière que la date de fin est minimale. Donc $f(s_{j+1}) \leq f(o_{j+1})$. On en conclut que $\{s_1, s_2, \dots, s_j, s_{j+1}, o_{j+2}, \dots, o_m\}$ est compatible avec \mathcal{O} et donc optimale.

Par induction, on peut donc affirmer que \mathcal{S} est optimale, quelque soit $j \in \llbracket 1, k \rrbracket$. Est-ce que les deux solutions optimales \mathcal{S} et \mathcal{O} coïncident ? C'est à dire, est-ce que $k = m$? Si ce n'était pas le

1. sous-entendu, on ne pourra pas choisir une date plus petite que celle-ci pour le début et la fin de la location.

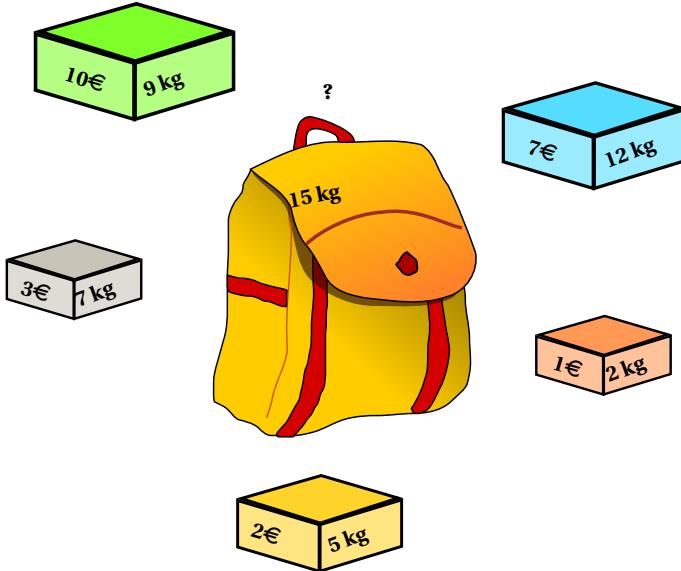


FIGURE 14.2 – Illustration du problème du sac à dos (d'après [Wikipedia](#)). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 . Le poids total admissible dans le sac est 15kg.

cas, c'est à dire $k > m$, alors on pourrait choisir dans $\mathcal{E} \setminus \{s_1, \dots, s_k\}$ une demande dont la date de fin serait compatible avec la solution optimale. Cette solution ne serait donc plus optimale, ce qui est une contradiction. ■

Cet algorithme glouton est donc optimal, mais cela est essentiellement dû aux contraintes qu'on a mise sur l'optimisation : **tous les clients ne seront pas satisfaits. On cherche juste à en satisfaire un maximum.**

C Exemple du sac à dos

On cherche à remplir un sac à dos comme indiqué sur la figure 14.2. Chaque objet que l'on peut insérer dans le sac est **insécable**² et possède une valeur entière (€) et un poids entier connus (kg). On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant³ le poids à π kg.

 **Vocabulary 14 — Knapsack problem** ⇔ Le problème du sac dos.

On peut chercher à résoudre le problème du sac à dos de manière gloutonne en utilisant un algorithme glouton (cf. algorithme 46).

En terme de complexité, l'algorithme 46 est plutôt intéressant puisqu'en $O(n)$. Cependant, la solution trouvée n'est pas nécessairement optimale : cet algorithme est donc un point de

2. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas en mettre qu'une partie.

3. On accepte un poids total inférieur ou égal à π .

Algorithme 46 Problème du sac à dos

```

1: Fonction SAC_À_DOS( $\mathcal{E}, \pi$ ) ▷  $\mathcal{E} = \{(v_1, p_1), \dots, (v_n, p_n)\}$ 
2:    $\mathcal{S} \leftarrow \emptyset$ 
3:    $p_{total} \leftarrow 0$ 
4:    $v_{total} \leftarrow 0$ 
5:   Trier  $\mathcal{E}$  pas ordre de valeurs  $v_i$  décroissantes ▷ le choix sera facile
6:   tant que  $p_{total} \leq \pi$  et  $\mathcal{E}$  pas vide répéter
7:      $v, p \leftarrow$  retirer l'élément de  $\mathcal{E}$  le plus valué ▷ choix de  $v$  maximale
8:     si  $p_{total} + p \leq \pi$  alors ▷ Est-ce une solution?
9:       Ajouter  $(v, p)$  à  $\mathcal{S}$ 
10:  renvoyer  $\mathcal{S}$ 

```

départ mais pas la solution définitive à ce problème.

R Intuitivement, on comprend bien qu'on ne pourra trouver de solution optimale que si on peut compléter le sac à dos au maximum. Il faut donc que les objets qui restent à mettre dans le sac après le plus valué soient :

- de la bonne taille, ce qui revient à invoquer la chance,
- ou sécable afin de se conformer à l'espace restant et l'optimiser. Or on a choisi des objets insécables.

D'une manière plus générale, le problème du sac à dos reflète un problème d'allocation de ressources pour lequel le temps (ou le budget) est fixé et où l'on doit choisir des éléments indivisibles parmi un ensemble tâches (ou de projets).

D Gloutonnerie et dynamisme

Tenter sa chance est le plus souvent payant! Si un algorithme glouton donne une solution rapidement alors que l'algorithme donnant la solution optimale est de complexité exponentielle $O(2^n)$ alors tenter sa chance en *gloutonnant* permet souvent de progresser vers une solution acceptable.

R La programmation dynamique qui sera étudiée au semestre trois permet de résoudre les problèmes d'optimisation sur lesquels butent certains algorithmes gloutons. Elle est pertinente lorsque les sous-problèmes générés se recoupent.

15

LES MOTS DES GRAPHES

À la fin de ce chapitre, je sais :

- utiliser des mots pour décrire les graphes
- énumérer quelques graphes remarquables
- distinguer un parcours d'une chaîne et d'un cycle
- distinguer un graphe orienté, non orienté, pondéré et un arbre

La théorie des graphes en mathématiques discrètes étudie les graphes comme objet mathématique. En informatique, en plus de les étudier, on a la chance de pouvoir les programmer, de jouer avec pour résoudre une infinité de problèmes. Les domaines d'application des graphes sont innombrables : les jeux, la planification, l'organisation, la production, l'optimisation, les programmes et modèles informatiques, les trajets dans le domaine des transports, le tourisme, la logistique ou tout simplement la géométrie... **Les graphes sont des objets simples que tout le monde peut dessiner.** Même s'il ne vous apparaît pas immédiatement que résoudre un sudoku est équivalent à la coloration d'un graphe, la pratique de ces derniers vous amènera à regarder le monde différemment.

A Typologie des graphes

■ **Définition 72 — Graphe.** Un graphe G est un couple $G = (S, A)$ où S est un ensemble fini et non vide d'éléments appelés **sommets** et A un ensemble de paires d'éléments de S appelées **arêtes**.

R A est donc un ensemble de paires de sommets : l'arête qui, sur le graphe 15.1, relie le sommet a au sommet b est noté (a, b) ou tout simplement ab .

🇬🇧 **Vocabulary 15 — Graph** ⇔ Graphe

🇬🇧 **Vocabulary 16 — Vertex (plural : vertices)** ⇔ Sommet

🇬🇧 **Vocabulary 17 — Edge** ⇔ Arête

La définition 72 est en fait celle des **graphes simples et non orientés** : ce sont eux que l'on considérera la plupart du temps.

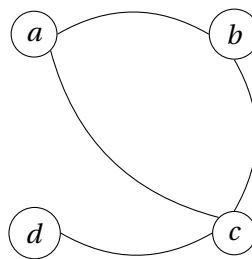


FIGURE 15.1 – Graphe simple

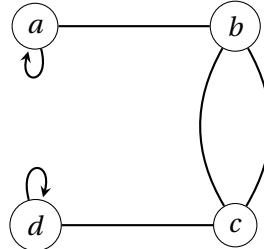


FIGURE 15.2 – Multigraph à deux boucle et deux arêtes parallèles → HORS PROGRAMME

- **Définition 73 — Boucle.** Une boucle est une arête reliant un sommet à lui-même.
- **Définition 74 — Arêtes parallèles.** Deux arêtes sont parallèles si elles relient les mêmes sommets.
- **Définition 75 — Graphe simple.** Un graphe simple est un graphe sans arêtes parallèles et sans boucles.
- **Définition 76 — Multigraph.** Un multigraph est un graphe avec des boucles et des arêtes parallèles. → HORS PROGRAMME

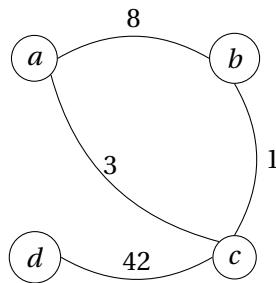


FIGURE 15.3 – Graphe pondéré

■ **Définition 77 — Graphe pondéré.** Un graphe $G = (S, A)$ est pondéré s'il existe une application $w : A \rightarrow \mathbb{R}$. Le poids de l'arête ab vaut $w(ab)$.

■ **Définition 78 — Graphe orienté.** Un graphe $G = (S, A)$ est orienté si ses arêtes sont orientées selon une direction. Les arêtes sont alors désignées par le mot arc.

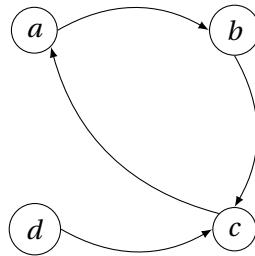
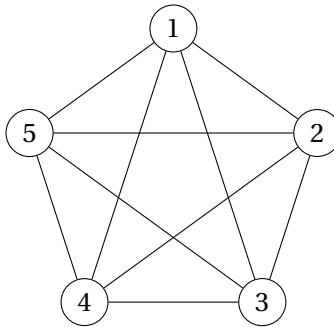


FIGURE 15.4 – Graphe orienté

R Sur le graphe orienté de la figure 15.4, il existe une arc de a vers b , matérialisé sur le graphe par une flèche. On note l'arc (a, b) ou ab .

■ **Définition 79 — Graphe complet.** Un graphe $G = (S, A)$ est complet si et seulement si une arête existe entre chaque sommet, c'est-à-dire si tous les sommets sont voisins.

En hommage à Kuratowski, on désigne les graphes complets par la lettre K_o indicée par l'ordre du graphe (cf. définition 83). La figure 15.5 représente le graphe complet d'ordre cinq K_5 . Kuratowski a notamment démontré [16] que K_5 n'est pas planaire : quelle que soit la manière de représenter ce graphe sur un plan, des arêtes se croiseront. Le graphe de la figure 15.1 est planaire.

FIGURE 15.5 – Graphe complet K_5

■ **Définition 80 — Graphe planaire.** Une graphe planaire est un graphe que l'on peut représenter sur un plan sans qu'aucune arête ne se croise.

■ **Définition 81 — Graphe biparti.** un graphe $G = (S, A)$ est biparti si l'ensemble S de ses sommets peut être divisé en deux sous-ensembles disjoints U et W tels que chaque arête de A ait une extrémité dans U et l'autre dans W .

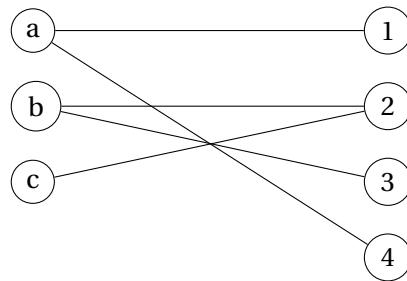
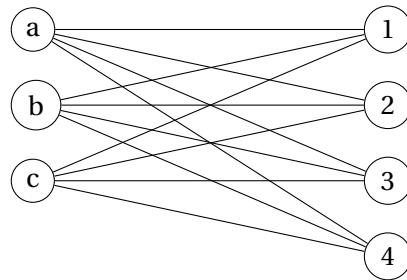


FIGURE 15.6 – Graphe biparti

FIGURE 15.7 – Graphe biparti complet K_{34}

B Implémentation des graphes

On peut représenter graphiquement un graphe comme sur les figures précédentes 15.1, 15.3 ou 15.4. On peut également chercher à les implémenter sous la forme d'ensembles, de matrices ou de listes.

■ **Exemple 36 — Graphe et ensembles.** Le graphe de la figure 15.1 est un graphe simple que l'on peut noter $G = \{S = \{a, b, c, d\}, A = \{(a, b), (a, c), (b, c), (c, d)\}\}$ ou plus simplement $G = \{S = \{a, b, c, d\}, A = \{ab, ac, bc, cd\}\}$.

■ **Définition 82 — Adjacent ou voisin.** Deux sommets a et b sont adjacents ou voisins si le graphe contient une arête ab . Deux arêtes sont adjacentes ou voisines s'il existe un sommet commun à ces deux arêtes.

■ **Exemple 37 — Graphe et matrice d'adjacence.** Grâce au concept d'adjacence, on peut représenter un graphe par une matrice d'adjacence. Pour construire une telle matrice, il faut d'abord numérotter arbitrairement les sommets du graphes. Par exemple, pour le graphe de la figure 15.1, on choisit l'ordre $(a, b, c, d) \rightarrow (0, 1, 2, 3)$. Les coefficients m_{ij} de la matrice d'adjacence sont calculés selon la règle suivante :

$$m_{ij} = \begin{cases} 1 & \text{s'il existe une arête entre le sommet } i \text{ et le sommet } j \\ 0 & \text{sinon} \end{cases} \quad (15.1)$$

Pour le graphe de la figure 15.1, on obtient :

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (15.2)$$

La matrice d'adjacence d'un graphe simple non orienté est de diagonale nulle (pas de boucles) et symétrique.

La matrice d'un graphe orienté n'est pas forcément symétrique.

Dans le cas d'un graphe pondéré, on peut remplacer le coefficient de la matrice par le poids de l'arête considérée.

En Python, on pourra utiliser un tableau Numpy pour implémenter une matrice d'adjacence.

■ **Exemple 38 — Graphe et liste d'adjacence.** On peut représenter un graphe par la liste des voisins de chaque sommet. Par exemple, si on dénote les sommets a , b , c et d par les indices 0, 1, 2, et 3, alors le graphe de la figure 15.1 peut être décrit par la liste `[[1,2], [0,2], [0,1,3], [2]]`.

Pour résumé, en Python :

```
1           G = [[1,2], [0,2], [0,1,3], [2]]
```

```
2           n = len(G)    # graph order
```

Par ailleurs, si le graphe est pondéré, la liste d'adjacence est une liste de listes de tuples, chaque tuple étant un couple (sommets, poids). Par exemple, le graphe orienté de la figure 15.3 s'écrit :

```
1           G = [(1,8),(2,3)], [(0,8),(2,1)], [(0,3),(1,1),(3,42)], [(2,42)]]
```

R Ni la représentation graphique de la figure 15.1, ni la matrice d'adjacence M de l'équation 15.2, ni les listes d'adjacence ne sont des représentations uniques. On peut tracer différemment le graphe ou choisir un autre ordre pour les sommets et obtenir une autre matrice ou une autre liste d'adjacence. Cela traduit l'isomorphisme des graphes.

R Le choix d'une implémentation ou d'une autre est avant tout lié aux choix des algorithmes que l'on va utiliser. La structure de donnée utilisée est souvent le facteur clef qui permet d'améliorer ou de détériorer les performances d'un algorithme.

C Caractérisation structurelle des graphes

■ **Définition 83 — Ordre d'un graphe.** L'ordre d'un graphe est le nombre de ses sommets. Pour $G = (S, A)$, l'ordre du graphe vaut donc le cardinal de l'ensemble S que l'on note généralement $|S|$. On note parfois l'ordre d'un graphe $|G|$.

R Si $|S| = n$, alors une matrice d'adjacence de $G = (S, A)$ est de dimension $n \times n$. Une liste d'adjacence de G a pour taille n .

■ **Définition 84 — Taille d'un graphe.** La taille d'un graphe désigne le nombre de ses arêtes. On le note $|A|$ et parfois $||G||$

■ **Définition 85 — Voisinage d'un sommet.** L'ensemble de voisins d'un sommet a d'un graphe $G = (S, A)$ est le voisinage de ce sommet. On le note généralement $V_G(a)$.

■ **Définition 86 — Incidence.** Une arête est dite incidente à un sommet si ce sommet est une des extrémités de cette arête

■ **Définition 87 — Degré d'un sommet.** Le degré $d(a)$ d'un sommet a d'un graphe G est le nombre d'arêtes incidentes au sommet a . C'est aussi $|V_G(a)|$.

■ **Définition 88 — Degrés d'un graphe orienté.** Dans un graphe orienté G et pour un som-

met s de ce graphe, on distingue :

- le degré entrant $d_+(s)$: le nombre d'arêtes incidentes à s et dirigées sur s ,
- le degré sortant $d_-(s)$: le nombre d'arêtes incidentes qui sortent de s et qui sont dirigées vers un autre sommet.

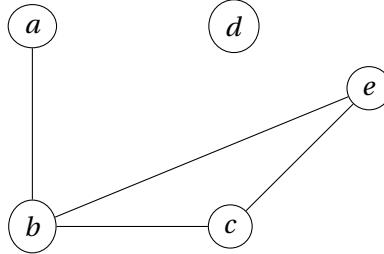


FIGURE 15.8 – Graphe d'ordre cinq, de taille quatre et de séquence $[0, 1, 2, 2, 3]$. Le sommet d est isolé. Ce graphe n'est ni complet ni connexe.

■ **Définition 89 — Sommet isolé.** Un sommet isolé est un sommet dont le degré vaut zéro.

■ **Définition 90 — Séquence des degrés.** La séquence des degrés d'un graphe G est la liste ordonnée par ordre croissant des degrés des sommets de G .

Sur la figure 15.8, on a représenté un graphe d'ordre cinq avec un sommet isolé. Ce graphe n'est pas connexe ni complet et sa séquence des degrés est $[0, 1, 2, 2, 3]$.

■ **Définition 91 — Graphe complémentaire.** Soit $G = (S, A)$ un graphe. On dit que $\bar{G} = (S, \bar{A})$ est le complémentaire de G si les arêtes de \bar{G} sont les arêtes possibles qui ne figurent pas dans G . On note ces arêtes \bar{A} .

Par exemple, le complémentaire du graphe 15.8 est représenté sur la figure 15.9.

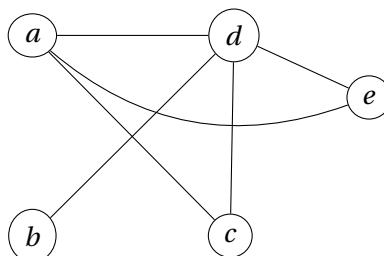


FIGURE 15.9 – Graphe complémentaire du graphe de la figure 15.8

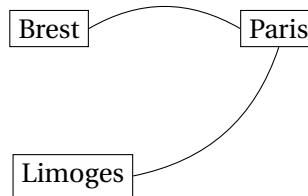


FIGURE 15.10 – Graphe d'ordre trois, de taille deux et de séquence [1,1,2]

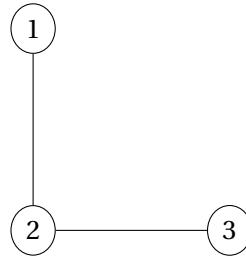


FIGURE 15.11 – Graphe d'ordre trois, de taille deux et de séquence [1,1,2]

D Isomorphisme des graphes

Considérons les graphes des figures 15.10 et 15.11. Ils ne diffèrent que par les noms des sommets et la signification des arêtes. Si on ne prête pas attention aux noms des sommets ni à la signification des arêtes, ces deux graphes sont identiques, leurs caractéristiques sont les mêmes : ordre, degré, taille. On dit qu'ils sont isomorphes ou qu'ils sont identiques à un isomorphisme près. Finalement, c'est la structure du graphe telle qu'on peut la caractériser qui importe, pas son apparence.

■ **Définition 92 — Graphes isomorphes.** Deux graphes $G = (S, A)$ et $G' = (S', A')$ sont isomorphes si et seulement s'il existe une bijection σ de S vers S' pour laquelle, si ab est un arête de A , alors $\sigma(a)\sigma(b)$ est à une arête de A' .

■ **Exemple 39 — Isomorphes et bijection.** Considérons les deux graphes G et G' représentés sur les figures 15.12 et 15.13.

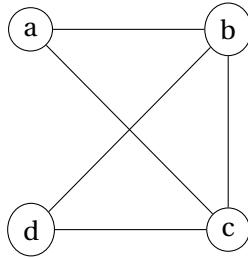
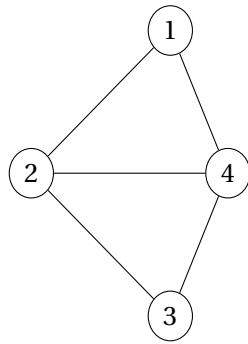
On peut les définir par des ensembles de la manière suivante :

$$G = (S = \{a, b, c, d\}, A = \{ab, ac, bc, bd, dc\}) \quad (15.3)$$

$$G' = (S' = \{1, 2, 3, 4\}, A' = \{12, 14, 24, 23, 43\}) \quad (15.4)$$

Formulé de la sorte, on pourrait croire que ces deux graphes ne sont pas isomorphes. Pourtant, c'est le cas. Comment le montrer? En exhibant une bijection ad-hoc!

On cherche donc une bijection entre les deux graphes en comparant les degrés des som-

FIGURE 15.12 – Graphe d'exemple $G = (S = \{a, b, c, d\}, A = \{ab, ac, bc, bd, dc\})$ FIGURE 15.13 – Graphe d'exemple $G' = (S' = \{1, 2, 3, 4\}, A' = \{12, 14, 24, 23, 43\})$

mets et en observant leurs arêtes.

On peut proposer la bijection $\sigma : S \longrightarrow S'$ telle que :

$$\sigma(a) = 1 \quad (15.5)$$

$$\sigma(b) = 2 \quad (15.6)$$

$$\sigma(c) = 4 \quad (15.7)$$

$$\sigma(d) = 3 \quad (15.8)$$

On a également la correspondance des arêtes :

$$\sigma(a)\sigma(b) = 12 \quad (15.9)$$

$$\sigma(a)\sigma(c) = 14 \quad (15.10)$$

$$\sigma(b)\sigma(c) = 24 \quad (15.11)$$

$$\sigma(b)\sigma(d) = 23 \quad (15.12)$$

$$\sigma(c)\sigma(d) = 43 \quad (15.13)$$

R On peut compter le nombre de graphes isomorphes pour un ordre donné. Par exemple, il y a deux graphes isomorphes d'ordre 2 et 8 d'ordre 3.

E Chaînes, cycles et parcours

■ **Définition 93 — Chaîne.** Une chaîne reliant deux sommets a et b d'un graphe non orienté est une suite finie d'arêtes consécutives reliant a à b . Dans le cas d'un graphe orienté on parle de chemin.

■ **Définition 94 — Chaîne élémentaire.** Une chaîne élémentaire ne passe pas deux fois par un même sommet : **tous ses sommets sont distincts**.

■ **Définition 95 — Chaîne simple.** Une chaîne simple ne passe pas deux fois par une même arête : **toutes ses arêtes sont distinctes**.

■ **Définition 96 — Longueur d'une chaîne.** La longueur d'une chaîne \mathcal{C} est :

- le nombre d'arêtes que comporte la chaîne dans un graphe simple non pondéré,
- la somme des poids des arêtes de la chaîne, c'est à dire $\sum_{e \in \mathcal{C}} w(e)$, dans le cas d'un graphe simple pondéré dont la fonction de valuation est w .

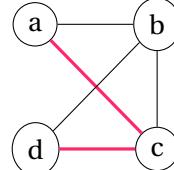


FIGURE 15.14 – Exemple de chaîne simple reliant a à d en rouge

■ **Définition 97 — Cycle.** Un cycle est une chaîne simple dont les deux sommets extrémités sont identiques.

La longueur d'un cycle est le nombre d'arêtes qu'il contient. Dans le cas des graphes orientés on parle de circuit.

■ **Définition 98 — Chaîne eulérienne.** Une chaîne eulérienne est une chaîne simple qui passe par toutes les arêtes d'un graphe.

■ **Définition 99 — Cycle eulérien.** Un cycle eulérien est un cycle passant exactement une fois par chaque arête d'un graphe.

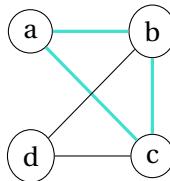


FIGURE 15.15 – Exemple de cycle en turquoise

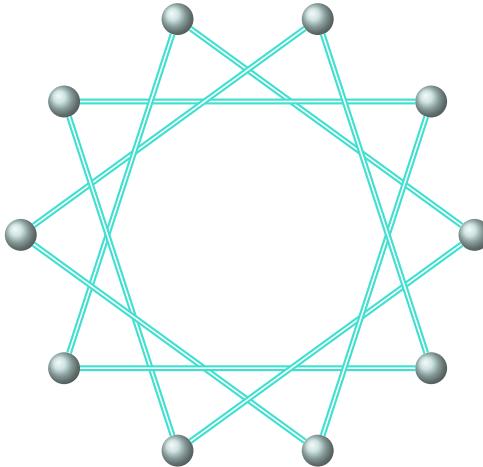


FIGURE 15.16 – Saurez-vous trouver le cycle eulérien de ce graphe ?

■ **Définition 100 — Cycle hamiltonien.** Un cycle hamiltonien est un sous-graphe couvrant qui est un cycle. Autrement dit, c'est un cycle qui passe par tous les sommets d'un graphe.

R Rowan Hamilton était un astronome irlandais qui a inventé le jeu icosien ^a en 1857.

a. The icosian game, jeu équivalent à l'icosagonal d'Édouard Lucas [17]

R Les graphes complets K_n sont eulériens et hamiltoniens : ils possèdent à la fois un cycle eulérien et un cycle hamiltonien.

R On peut également définir une chaîne comme le graphe d'ordre n isomorphe au graphe $P_n = \{S_n = \{1, 2, \dots, n\}, A_n = \{\{1, 2\}, \{2, 3\}, \dots, \{n-1, n\}\}\}$. Par convention, on pose $P_1 = \{S_1 = \{1\}\}$ et $A_1 = \emptyset$. Les extrémités de la chaînes sont les deux sommets de degré 1.

Avec la même approche et les mêmes notations, un cycle devient alors un graphe isomorphe au graphe $C_n = \{S_n, A'_n = A_n \cup \{n, 1\}\}$, c'est à dire que la chaîne finit là où elle a commencé. L'ordre de C_n est supérieur ou égal à trois.

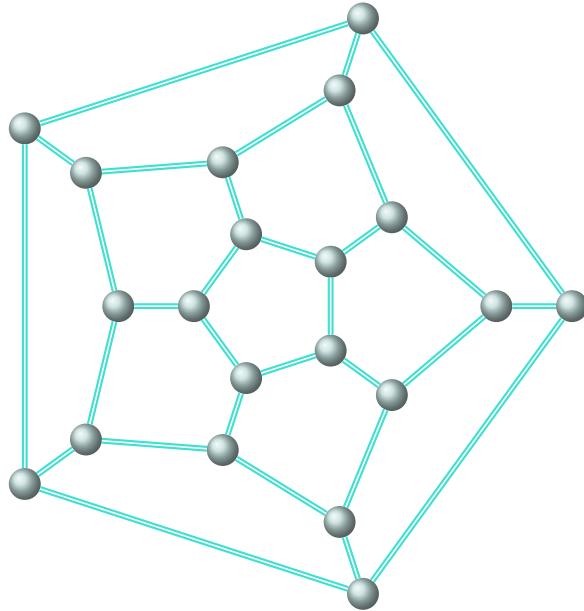


FIGURE 15.17 – Graphe du jeu icosien et du dodécahèdre (solide régulier à 12 faces pentagonales). C'est un graphe cubique car chaque sommet possède trois voisins. Ce graphe possède un cycle hamiltonien. Saurez-vous le trouver?

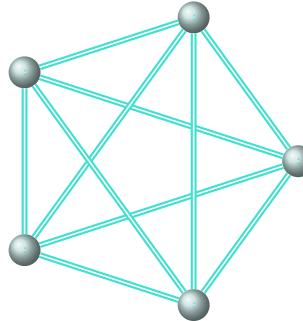


FIGURE 15.18 – Graphe K_5 : saurez-vous trouver des cycles hamiltonien et eulérien de ce graphe?

■ **Définition 101 — Parcours.** Un parcours d'un graphe G est une liste non vide et ordonnées de sommets de G telles que deux sommets consécutifs sont adjacents dans G . Il peut y avoir des répétitions de sommets dans un parcours, mais il n'y a pas de répétitions d'arêtes dans un cycle ou une chaîne simple.

Par exemple, $\pi = \{a, b, c, d, b\}$ est un parcours sur le graphe de la figure 15.15.

F Sous-graphes et connexité

■ **Définition 102 — Graphe connexe.** Un graphe $G = (S, A)$ est connexe si et seulement si pour tout couple de sommets (a, b) de G , il existe une chaîne d'extrémités a et b .

Par exemple, le graphe de la figure 15.1 est connexe, mais pas celui figure 15.8.

■ **Définition 103 — Sous-graphe.** Soit $G = (S, A)$ un graphe, alors $G' = (S', A')$ est un sous-graphe de G si et seulement si $S' \subseteq S$ et $A' \subseteq A$.

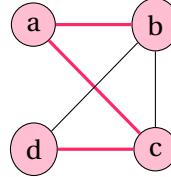


FIGURE 15.19 – Exemple de sous-graphe couvrant G en rouge : $G = (S = \{a, b, c, d\}, A = \{ab, ac, cd\})$

■ **Définition 104 — Sous-graphe couvrant.** G' est un sous-graphe couvrant de G si et seulement si G' est un sous-graphe de G et $S' = S$.

■ **Définition 105 — Sous-graphe induit.** Soit $S' \subset S$ non vide. G' est un sous-graphe de G induit par S' et on note $G[S']$ si et seulement si G' admet pour arêtes celles de G dont les deux extrémités sont dans S' .

■ **Définition 106 — Clique.** Une clique est d'un graphe $G = (S, A)$ un sous-ensemble $C \subseteq S$ des sommets dont le sous-graphe induit $G[C]$ est complet.

Sur la figure 15.20, le graphe $G = (S = \{b, c, d\}, A = \{bc, bd, cd\})$ est une clique.

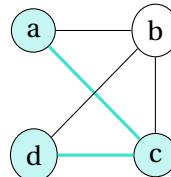


FIGURE 15.20 – Exemple de sous-graphe induit par les sommets $S = \{a, c, d\}$ en turquoise. $G[S] = (S = \{a, c, d\}, A = \{ac, cd\})$

G Coloration de graphes

■ **Définition 107 — Coloration.** Une coloration d'un graphe simple est l'attribution d'une couleur aux sommets de ce graphe.

■ **Définition 108 — Coloration valide.** Une coloration est valide lorsque deux sommets adjacents n'ont jamais la même couleur.

■ **Définition 109 — Nombre chromatique.** Le nombre chromatique d'un graphe G est le plus petit nombre de couleurs nécessaires pour obtenir une coloration valide de ce graphe. On le note généralement $\chi(G)$.

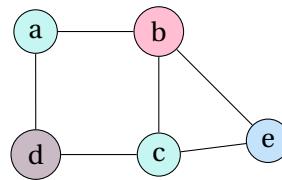


FIGURE 15.21 – Exemple de 4-coloration valide d'un graphe. Cette coloration n'est pas optimale.

■ **Définition 110 — k -coloration.** Lorsqu'une coloration de graphe utilise k couleurs, on dit d'elle que c'est une k -coloration.

■ **Définition 111 — Coloration optimale.** Une $\chi(G)$ -coloration valide est une coloration optimale d'un graphe G .

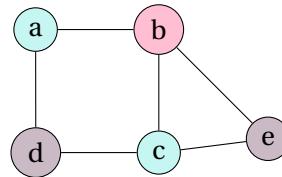


FIGURE 15.22 – Exemple de 3-coloration valide d'un graphe. Cette coloration est optimale.

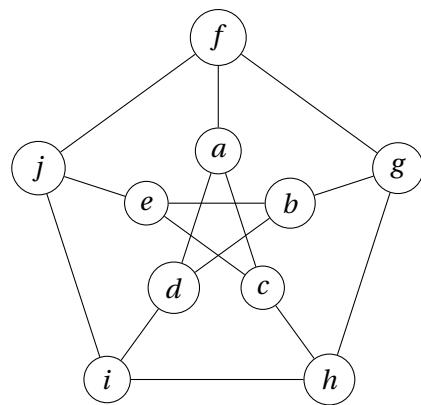


FIGURE 15.23 – Graphe de Petersen : saurez-vous proposer une coloration optimale de ce graphe sachant que son nombre chromatique vaut trois ?

H Distances

■ **Définition 112 — Distance dans un graphe simple non pondéré.** La distance d'un sommet a à un sommet b dans un graphe simple non pondéré G est la longueur de la plus courte chaîne d'extrémités a et b . On la note $d_G(a, b)$.

R Cette définition coïncide avec la notion de distance en mathématiques. Pour les sommets a , b et c de G :

- $d_G(a, b) = 0 \Leftrightarrow a = b$
- $d_G(a, b) = d_G(b, a)$
- $d_G(a, b) \leq d_G(a, c) + d_G(c, b)$

■ **Définition 113 — Valuation d'une chaîne dans un graphe pondéré.** La valuation d'une chaîne dans un graphe pondéré est la somme des poids de chacune de ses arêtes. Pour une chaîne P , on la note $v(P)$.

■ **Définition 114 — Distance dans un graphe pondéré.** La distance d'un sommet a à un sommet b dans un graphe pondéré G est la valuation minimum des chaînes d'extrémités a et b . On la note $d_{G,v}(a, b)$.

I Arbres

■ **Définition 115 — Arbre.** Un arbre est un graphe connexe et acyclique.

■ **Définition 116 — Feuilles.** Dans un arbre, les sommets de degré un sont appelés les feuilles.

■ **Définition 117 — Arbre recouvrant.** Un arbre recouvrant d'un graphe G est un sous-graphe couvrant de G qui est un arbre.

■ **Définition 118 — Arbre enraciné.** Parfois, on distingue un sommet particulier dans un arbre A : la racine r . Le couple (A, r) est un nommé arbre enraciné. On le représente un tel arbre verticalement avec la racine placée tout en haut comme sur la figure 15.24.

■ **Définition 119 — Arbre binaire.** Un arbre binaire est un graphe connexe acyclique pour lequel le degré de chaque sommet vaut au maximum trois. Le degré de la racine vaut au maximum deux.

■ **Définition 120 — Arbre binaire parfait.** Un arbre binaire parfait est un arbre dans lequel

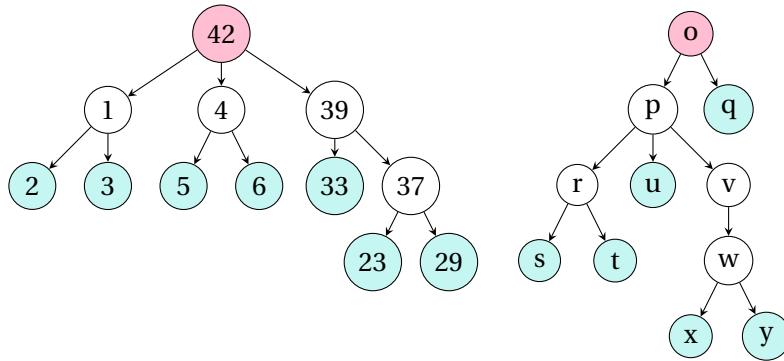


FIGURE 15.24 – Exemples d’arbres enracinés. Les racines des arbres sont en rouge, les feuilles en turquoise. Le tout forme une forêt.

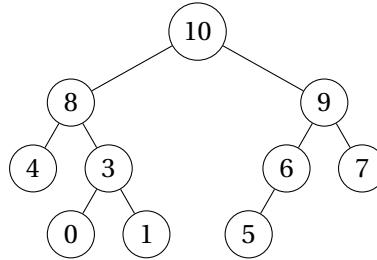


FIGURE 15.25 – Arbre binaire

tous les niveaux sauf le dernier doivent être totalement remplis. Si le dernier n'est pas rempli totalement alors il doit être rempli de gauche à droite.

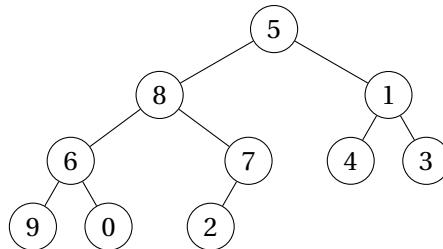


FIGURE 15.26 – Arbre binaire parfait

Ces arbres sont illustrés sur les figure 15.25 et 15.26.

16

PROPRIÉTÉS DES GRAPHES

À la fin de ce chapitre, je sais :

- expliquer le lemme des poignées de mains
- caractériser un cycle eulérien
- caractériser un graphe connexe, acyclique ou un arbre

A Des degrés et des plans

Théorème 4 — Somme des degrés d'un graphe. Le nombre d'arêtes d'un graphe simple est égale à la moitié de la somme des degrés des sommets de ce graphe.

Plus formellement, soit $G = (S, A)$ un graphe simple alors on a :

$$2|A| = \sum_{s \in S} d(s) \quad (16.1)$$

R On appelle souvent ce théorème le lemme des poignées de mains car il peut se traduire par le fait que dans un graphe il y a toujours un nombre pair de sommets de degré impair.

Théorème 5 — Formule d'Euler pour les graphes planaires. Soit $G = (S, A)$ un graphe simple. G est planaire si le nombre de régions du plan qu'il délimite R vaut :

$$R = 2 + |A| - |S| \quad (16.2)$$

R Pour vérifier cette formule, il ne faut pas oublier la région extérieur au graphe qui compte également.

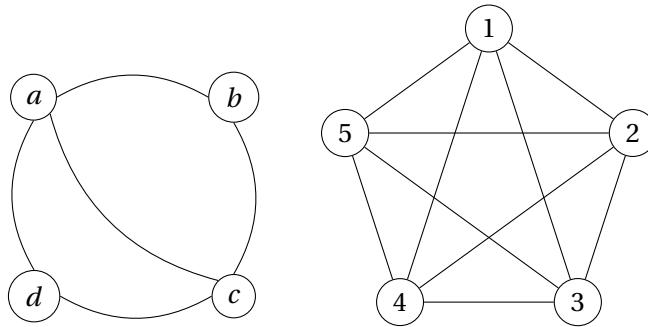


FIGURE 16.1 – Sur ces graphes, on peut vérifier les théorèmes de caractérisation des chaînes eulériennes et des cycles eulériens

B Caractérisation des chaînes, des cycles et des graphes

Théorème 6 — Caractérisation d'une chaîne eulérienne. Il existe une chaîne eulérienne dans un graphe lorsque seuls les sommets de départ et d'arrivée sont de degré impair.

Théorème 7 — Caractérisation d'un cycle eulérien. Il existe un cycle eulérien dans un graphe si tous les sommets sont de degré pair.

Pour bien visualiser ces caractérisations, on peut s'entraîner sur les graphes de la figure 16.1. Le graphe d'ordre quatre possède une chaîne eulérienne mais pas de cycle eulérien. Le graphe complet K_5 possède les deux.

Théorème 8 — Chaînes extraites et existence de chaînes. S'il existe un parcours d'un sommet a vers un sommet b dans un graphe G alors il existe une chaîne de a vers b dont les arêtes sont des arêtes du parcours.

Par transitivité, s'il existe une chaîne de a à b et une de a à c alors il existe une chaîne de b à c .

On appelle graphe hamiltonien un graphe qui possède un cycle hamiltonien (cf. définition 100). Un graphe hamiltonien :

- est connexe,
- d'ordre supérieur ou égal à trois,
- n'a pas de sommets de degré un.

Théorème 9 — Condition nécessaire pour un graphe non hamiltonien. Soit $G = (S, A)$ un graphe. Soit $U \subseteq S$ un ensemble de sommets de G . Si le nombre de composantes connexes de $G = (S \setminus U, A)$ est strictement supérieur au nombre de sommets de U , alors G n'est pas hamiltonien.

Théorème 10 — Condition nécessaire pour un graphe hamiltonien. Soit $G = (S, A)$ un graphe d'ordre supérieur ou égal à deux. Si pour toute paire de sommets a et b de G on a :

$$d(a) + d(b) \geq |S| \quad (16.3)$$

alors G est hamiltonien.

C Graphes acycliques et connexes

Théorème 11 — Condition nécessaire d'acyclicité d'un graphe. Soit un graphe $G = (S, A)$ possédant au moins une arête et acyclique alors G possède au moins deux sommets de degré un et on a :

$$|A| \leq |S| - 1 \quad (16.4)$$

Théorème 12 — Condition nécessaire de connexité d'un graphe. Si un graphe $G = (S, A)$ est connexe alors on a :

$$|A| \geq |S| - 1 \quad (16.5)$$

R On déduit des deux théorèmes précédents qu'un arbre (cf. définition 115) possède exactement $|S| - 1$ arêtes.

D Coloration, graphes planaires et nombre chromatique

Théorème 13 — Trois couleurs. Si tous les degrés des sommets d'un graphe planaire sont pairs, alors trois couleurs suffisent pour obtenir une coloration valide.

Théorème 14 — Quatre couleurs. Le nombre chromatique d'un graphe planaire ne dépasse jamais quatre.

On peut chercher à encadrer le nombre chromatique d'un graphe. Dans une premier temps, on peut remarquer que :

- $\chi(G) \leq |S|$, autrement dit, l'ordre d'un graphe est supérieur ou égal au nombre chromatique. L'égalité est atteinte pour les graphes complets : tous les sommets étant reliés les uns aux autres, on ne peut qu'utiliser des couleurs différentes pour chaque sommet.
- Pour un sous-graphe G' de G , on a $\chi(G') \leq \chi(G)$.

■ **Définition 121 — Degré maximum des sommets d'un graphe.** On note $\Delta(G)$ le degré maximum des sommets d'un graphe G .

■ **Définition 122 — Ordre du plus grand sous-graphe complet d'un graphe.** On note $\omega(G)$ l'ordre du plus grand sous-graphe **complet** d'un graphe G .

Théorème 15 — Encadrement du nombre chromatique. Pour un graphe G , on a :

$$\omega(G) \leq \chi(G) \leq \Delta(G) + 1 \quad (16.6)$$

E Principe d'optimalité et plus court chemin dans un graphe

Théorème 16 — Optimalité et plus court chemin dans graphe. Si $a \rightsquigarrow b$ est le plus court chemin passant par un sommet c , alors les sous-chemins $a \rightsquigarrow c$ et $c \rightsquigarrow b$ sont des plus courts chemins.

Démonstration. Soit $a \rightsquigarrow b$ le plus court chemin passant par un sommet c dans un graphe G . Si $a \rightsquigarrow c$ n'est pas le plus court chemin, alors il suffit de prendre le plus court chemin entre a et c et de le joindre à $c \rightsquigarrow b$ pour obtenir un chemin plus court de a vers b . Ce qui est en contradiction avec notre hypothèse que $a \rightsquigarrow b$ est le plus court chemin. ■

17

ALGORITHMES ET GRAPHES

À la fin de ce chapitre, je sais :

- ☒ parcourir un graphe en largeur et en profondeur
- ☒ utiliser une file ou un pile pour parcourir un graphe
- ☒ énoncer le principe de l'algorithme de Dijkstra (plus court chemin)

A Parcours d'un graphe

Le parcours d'un graphe est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A*. On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. L'implémentation la plus simple et la plus commune est récursive. Mais on peut aussi l'implémenter de manière itérative en utilisant une *pile* de sommets de type Last In First Out.
3. L'algorithme de **Dijkstra** s'applique à un graphe pondéré : il passe par le voisin le plus proche d'un sommet avant de parcourir les autres voisins de ce sommet. C'est un parcours en largeur qui utilise une **file de priorités** : lorsqu'on insère un nouvel élément dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance la plus faible : la plus petite distance se situe en tête de la file.

a Parcours en largeur

Parcourir en largeur un graphe signifie qu'on cherche à visiter tous les voisins situé à une même distance d'un sommet (sur un même niveau) avant de parcourir le reste du graphe.

 **Vocabulary 18 — Breadth First Search** \leftrightarrow Parcours en largeur

Le parcours en largeur d'un graphe (cf. algorithme 47) est un algorithme à la base de nombreux développements comme l'algorithme de Dijkstra et de Prim (cf. algorithmes 51 et ??). Il utilise une file FIFO¹ afin de gérer la découverte des voisins dans l'ordre de la largeur du graphe.

Pour matérialiser le parcours en largeur, on opère en repérant les sommets à visiter. Lorsqu'un sommet est découvert, il intègre l'ensemble des éléments à visiter, c'est-à-dire la file F . Lorsque le sommet a été traité, il quitte la file. Il est donc également nécessaire de garder la trace du passage sur un sommet afin de ne pas traiter plusieurs fois un même sommet : si un sommet a été visité alors il intègre l'ensemble des éléments visités.

Au fur et à mesure de sa progression, cet algorithme construit un arbre de parcours en largeur dans le graphe. La racine de cet arbre est l'origine du parcours. Comme un sommet de cet arbre n'est découvert qu'une fois, il a au plus un parent. L'algorithme 47 peut ne rien renvoyer et servir pour un traitement particulier sur chaque noeud. Il peut aussi renvoyer une trace du parcours dans l'arbre dans une structure de type liste (P) qui enregistre le parcours.

Algorithme 47 Parcours en largeur d'un graphe

```

1: Fonction PARCOURS_EN_LARGEUR( $G, s$ )                                 $\triangleright s$  est un sommet de  $G$ 
2:    $F \leftarrow$  une file FIFO vide                                          $\triangleright F$  comme file FIFO
3:    $V \leftarrow \emptyset$                                                   $\triangleright V$  ensemble des sommets visités
4:    $P \leftarrow$  une liste vide                                          $\triangleright P$  comme parcours
5:   ENFILER( $F, s$ )
6:   AJOUTER( $V, s$ )
7:   tant que  $F$  n'est pas vide répéter
8:      $v \leftarrow$  DÉFILER( $F$ )
9:     AJOUTER( $P, v$ )                                          $\triangleright$  ou traiter le sommet en place
10:    pour chaque voisin  $x$  de  $v$  dans  $G$  répéter
11:      si  $x \notin V$  alors                                          $\triangleright x$  n'a pas encore été découvert
12:        AJOUTER( $V, x$ )
13:        ENFILER( $F, x$ )
14:    renvoyer  $P$                                           $\triangleright$  Facultatif, on pourrait traiter chaque sommet  $v$  en place

```

b Terminaison et correction du parcours en largeur

La terminaison du parcours en largeur peut être prouvée en considérant le variant de boucle $|F| + |\bar{V}|$, c'est à dire la somme des éléments présents dans la file et du nombre de noeuds non visités. En effet, au début de la boucle, si n est l'ordre du graphe, on a $|F| + |\bar{V}| = 1 + n - 1 = n$.

1. First In First Out

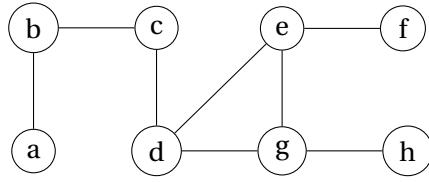


FIGURE 17.1 – Exemple de parcours en largeur au départ de a : $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow g \rightarrow f \rightarrow h$.

Puis, à chaque tour de boucle on retire un élément de la file et on ajoute ses p voisins en même temps qu'on marque les p voisins comme visités. L'évolution du variant s'écrit :

$$|F| - 1 + p + |\bar{V}| - p = |F| + |\bar{V}| - 1 \quad (17.1)$$

À chaque tour de boucle, le variant décroît donc strictement de un et atteint nécessairement zéro au bout d'un certain nombre de tours. Lorsque le variant vaut zéro $|F| + |\bar{V}| = 0$, on a donc $|F| = 0$ et $|\bar{V}| = 0$. La file est nécessairement vide et tous les nœuds ont été visités. L'algorithme se termine alors. La structure de donnée file permet de garantir la correction du parcours en largeur d'abord.

Parcourir un graphe, à partir d'un sommet de départ s , cela veut dire trouver un chemin partant de s vers tous les sommets du graphe². On remarque que tous les sommets du graphe sont à un moment ou un autre de l'algorithme **visités** et admis dans l'ensemble V : ceci vient du fait qu'on procède de proche en proche en ajoutant tous les voisins d'un sommet, sans distinction.

La correction peut se prouver en utilisant l'invariant de boucle \mathcal{I} : «Pour chaque sommet v ajouté à V et enfilé dans F , il existe un chemin de s à v .»

- Initialisation : à l'entrée de la boucle, s est ajouté à V et est présent dans la file F . Le chemin de s à s existe trivialement.
- Conservation : on suppose que l'invariant est vérifié jusqu'à une certaine itération. On cherche à montrer qu'il l'est toujours à la fin de l'itération suivante. Lors de l'exécution de cette itération, un sommet v est défilé. Ce sommet faisait déjà parti de V et par hypothèse, comme l'invariant était vérifié jusqu'à présent, il existe un chemin de s à v . Puis, les voisins de v sont ajoutés à V et enfilés. Comme ils sont voisins, il existe donc un chemin de v à ces sommets et donc il existe un chemin de s à ces sommets. À la fin de l'itération, l'invariant est donc vérifié.
- Terminaison : à la fin de l'algorithme, il existe un chemin de s vers tous les sommets du graphe visités (ajoutés à V). Tous les sommets ont été parcourus.

2. On fait l'hypothèse que le graphe est connexe. S'il ne l'est pas, il suffit de recommencer la procédure avec un des sommets n'ayant pas été parcouru.

c Complexité du parcours en largeur

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit G un graphe d'ordre n et de taille m implémenté par une liste d'adjacence. On a choisi une file FIFO pour laquelle les opérations ENFILER et DÉFILER sont en $O(1)$. On parcours tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n + m)$.

R Utiliser un liste d'adjacence pour implémenter le graphe est très important dans ce cas car cela permet d'accéder rapidement aux voisins : le coût de cette opération est l'accès à un élément de la liste. Si on avait utilisé une matrice, on aurait été obligé de rechercher les voisins à chaque étape.

R Si le graphe est complet, on note que la complexité $O(n + m)$ est en fait une complexité en $O(n^2)$ car $2|E| = n(n - 1)$ d'après le lemme des poignées de main (cf t4).

R Si l'on avait choisi un type tableau dynamique (typiquement le type `list` en Python) au lieu d'une file FIFO pour implémenter F , alors l'opération DÉFILER ferait perdre du temps : en effet, le tableau serait réécrit dans sa totalité à chaque fois qu'une opération DÉFILER aurait lieu car on retirerait alors le premier élément du tableau et il faudrait donc allouer un autre espace mémoire à ce nouveau tableau. Une fois encore, le choix de la structure de données est important pour que l'algorithme soit efficace.

R L'ensemble V n'est pas indispensable dans l'algorithme 47. On pourrait se servir de la liste qui enregistre le parcours. Néanmoins, son utilisation permet de bien découpler la sortie de l'algorithme (le chemin C) de son fonctionnement interne et ainsi de prouver la terminaison.

d Parcours en profondeur

Parcourir en profondeur un graphe signifie qu'on cherche à emprunter d'abord les arêtes du premier sommet trouvé avant de parcourir les voisins de ce sommet et le reste du graphe.

 **Vocabulary 19 — Depth First Search** ↗ Parcours en profondeur

Le parcours en profondeur d'un graphe s'exprime naturellement récursivement (cf. algorithme 48). Il peut également s'exprimer de manière itérative (cf. algorithme 49) en utilisant une pile P afin gérer la découverte des voisins dans l'ordre de la profondeur du graphe.

La complexité de cet algorithme est lié, comme toujours, aux structures de données utilisées. Soit G un graphe d'ordre n et de taille m implémenté par une liste d'adjacence. On a choisi une pile LIFO pour laquelle les opérations EMPILER et DÉPILER sont en $O(1)$. On parcours tous les sommets et chaque liste d'adjacence est parcourue une fois. Ces opérations sont donc en $O(n + m)$.

Algorithme 48 Parcours en profondeur d'un graphe (version récursive)

```

1: Fonction REC_PARCOURS_EN_PROFONDEUR( $G, s, V$ )            $\triangleright s$  est un sommet de  $G$ 
2:   AJOUTER( $V, s$ )                                          $\triangleright s$  est marqué visité
3:   pour chaque voisin  $x$  de  $s$  dans  $G$  répéter
4:     si  $x \notin V$  alors                                      $\triangleright x$  n'a pas encore été découvert
5:       REC_PARCOURS_EN_PROFONDEUR( $G, x, V$ )

```

Algorithme 49 Parcours en profondeur d'un graphe (version itérative)

```

1: Fonction PARCOURS_EN_PROFONDEUR( $G, s$ )            $\triangleright s$  est un sommet de  $G$ 
2:    $P \leftarrow$  une file vide                                 $\triangleright P$  comme pile
3:    $V \leftarrow$  un ensemble vide                             $\triangleright V$  comme visités
4:    $C \leftarrow$  un liste vide                                 $\triangleright C$  pour le parcours
5:   EMPILER( $P, s$ )
6:   tant que  $P$  n'est pas vide répéter
7:      $v \leftarrow$  DÉPILER( $P$ )
8:     AJOUTER( $C, v$ )
9:     si  $v \notin V$  alors                                      $\triangleright v$  n'a pas encore été découvert
10:    AJOUTER( $V, v$ )
11:    pour chaque voisin  $x$  de  $v$  dans  $G$  répéter
12:      EMPILER( $P, x$ )
13:    renvoyer  $C$                                       $\triangleright$  Facultatif, on pourrait traiter chaque sommet  $v$  en place

```

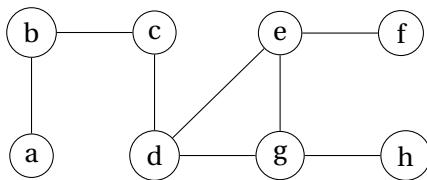


FIGURE 17.2 – Exemple de parcours en profondeur au départ de a : $a \rightarrow b \rightarrow c \rightarrow d \rightarrow g \rightarrow h \rightarrow e \rightarrow f$

B Trouver un chemin dans un graphe

On peut modifier l'algorithme 47 de parcours en largeur d'un graphe pour trouver un chemin reliant un sommet à un autre et connaître la longueur de la chaîne qui relie ces deux sommets. Il suffit pour cela de :

- garder la trace du prédécesseur (parent) du sommet visité sur le chemin,
- sortir du parcours dès qu'on a trouvé le sommet cherché (early exit),
- calculer le coût du chemin associé.

Le résultat est l'algorithme 50. Opérer cette recherche dans un graphe ainsi revient à chercher dans toutes les directions, c'est à dire sans tenir compte des distances déjà parcourues.

Algorithme 50 Longueur d'une chaîne via un parcours en largeur d'un graphe pondéré

```

1: Fonction CD_PEL( $G, a, b$ )            $\triangleright$  Trouver un chemin de  $a$  à  $b$  et la distance associée
2:    $F \leftarrow$  une file FIFO vide            $\triangleright F$  comme file FIFO
3:    $V \leftarrow$  un ensemble vide            $\triangleright V$  comme visités
4:    $P \leftarrow$  un dictionnaire vide            $\triangleright P$  comme parent
5:   ENFILER( $F, s$ )
6:   AJOUTER( $V, s$ )
7:   tant que  $F$  n'est pas vide répéter
8:      $v \leftarrow$  DÉFILER( $F$ )
9:     si  $v$  est le sommet  $b$  alors            $\triangleright$  Objectif atteint, early exit
10:    sortir de la boucle
11:    pour chaque voisin  $x$  de  $v$  dans  $G$  répéter
12:      si  $x \notin V$  alors            $\triangleright x$  n'a pas encore été découvert
13:        AJOUTER( $V, x$ )
14:        ENFILER( $F, x$ )
15:         $P[x] \leftarrow v$             $\triangleright$  Garder la trace du parent
16:         $c \leftarrow 0$             $\triangleright$  Coût du chemin
17:         $s \leftarrow b$ 
18:      tant que  $s$  est différent de  $a$  répéter
19:         $c \leftarrow c + w(s, P[s])$             $\triangleright w$  est la fonction de valuation du graphe  $G$ 
20:        sommet  $\leftarrow P[s]$             $\triangleright$  On remonte à l'origine du chemin
21:      renvoyer  $c$ 

```

R Il faut noter néanmoins que le chemin trouvé et la distance associée issue de l'algorithme 50 n'est pas nécessairement la meilleure, notamment car on ne tient pas compte de la distance parcourue jusqu'au sommet recherché.

R Si le graphe n'est pas pondéré, cet algorithme fonctionne néanmoins, il suffit de compter le nombre de sauts pour évaluer la distance (la fonction de valuation vaut toujours 1).

C Plus courts chemins dans les graphes pondérés

R Un graphe non pondéré peut-être vu comme un graphe pondéré dont la fonction de valuation vaut toujours 1. La distance entre deux sommets peut alors être interprétée comme le nombre de sauts nécessaires pour atteindre un sommet.

Théorème 17 — Existence d'un plus court chemin. Dans un graphe pondéré **sans pondérations négatives**, il existe toujours un plus court chemin.

Démonstration. Un graphe pondéré possède un nombre fini de sommets et d'arêtes (cf. définition 72). Il existe donc un nombre fini de chaînes entre les sommets du graphe. Comme les valuations du graphe ne sont pas négatives, c'est à dire que $\forall e \in E, w(e) \geq 0$, l'ensemble des longueurs de ces chaînes est une partie non vide de \mathbb{N} : elle possède donc un minimum. Parmi ces chaînes, il en existe donc nécessairement une dont la longueur est la plus petite, le plus court chemin. ■

■ **Définition 123 — Plus court chemin entre deux sommets d'un graphe.** Le plus court chemin entre deux sommets a et b d'un graphe G est une chaîne \mathcal{C}_{ab} qui relie les deux sommets a et b et :

- qui comporte un minimum d'arêtes si G est un graphe non pondéré,
- dont le poids cumulé est le plus faible, c'est à dire $\min_{\mathcal{C}_{ab} \in G} \left(\sum_{e \in \mathcal{C}_{ab}} w(e) \right)$, dans le cas d'un graphe pondéré de fonction de valuation w .

■ **Définition 124 — Distance entre deux sommets.** La distance entre deux sommets d'un graphe est la longueur d'un plus court chemin entre ces deux sommets. Pour deux sommets a et b , on la note δ_{ab} . On a enfin :

$$\delta_{ab} = \min_{\mathcal{C}_{ab} \in G} \left(\sum_{e \in \mathcal{C}_{ab}} w(e) \right) \quad (17.2)$$

On se propose maintenant d'étudier les algorithmes les plus célèbres qui illustrent, dans différentes configurations, le concept de plus court chemin dans un graphe. La majorité de ces algorithmes reposent sur le principe d'optimalité de Bellman et la programmation dynamique et qui a déjà été énoncé dans le théorème 16. On peut formuler ce principe ainsi : **toute sous-chaîne entre p et q d'un plus court chemin entre a et b est un plus court chemin entre p et q .**

a Algorithme de Dijkstra

L'algorithme de Dijkstra³[6] s'applique à des **graphes pondérés** $G = (V, E, w)$ **dont la valuation est positive**, c'est à dire que $\forall e \in E, w(e) \geq 0$. C'est un algorithme glouton optimal (cf.

3. à prononcer "Daillekstra"

informatique commune) qui trouve les plus courts chemins entre un sommet particulier $a \in V$ et tous les autres sommets d'un graphe. Pour cela, l'algorithme classe les différents sommets par ordre croissant de leur distance minimale au sommet de départ. Dans ce but, il **parcourt en largeur le graphe en choisissant les voisins les plus proches en premier**.

Algorithme 51 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```

1: Fonction DIJKSTRA( $G = (V, E, w)$ ,  $a$ )       $\triangleright$  Trouver les plus courts chemins à partir de  $a \in V$ 
2:    $\Delta \leftarrow a$                        $\triangleright \Delta$  est l'ensemble des sommets dont on connaît la distance à  $a$ 
3:    $\Pi \leftarrow \emptyset$                    $\triangleright \Pi[s]$  est le parent de  $s$  dans le plus court chemin de  $a$  à  $s$ 
4:    $d \leftarrow \emptyset$                    $\triangleright$  l'ensemble des distances au sommet  $a$ 
5:    $\forall s \in V, d[s] \leftarrow w(a, s)$        $\triangleright w(a, s) = +\infty$  si  $s$  n'est pas voisin de  $a$ , 0 si  $s = a$ 
6:   tant que  $\bar{\Delta}$  n'est pas vide répéter       $\triangleright \bar{\Delta}$  : sommets dont la distance n'est pas connue
7:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $d[u] = \min(d[v], v \in \bar{\Delta})$ 
8:      $\Delta = \Delta \cup \{u\}$                        $\triangleright$  On prend la plus courte distance à  $a$  dans  $\bar{\Delta}$ 
9:     pour  $x \in \bar{\Delta}$  répéter       $\triangleright$  Ou bien  $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$ , pour tous les voisins de  $u$  dans  $\bar{\Delta}$ 
10:    si  $d[x] > d[u] + w(u, x)$  alors
11:       $d[x] \leftarrow d[u] + w(u, x)$            $\triangleright$  Mises à jour des distances des voisins
12:       $\Pi[x] \leftarrow u$                    $\triangleright$  Pour garder la tracer du chemin le plus court
13:   renvoyer  $d, \Pi$ 

```

R Il faut remarquer que les boucles imbriquées de cet algorithme peuvent être comprises comme deux étapes successives de la manière suivante :

1. On choisit un nouveau sommet u de G à chaque tour de boucle *tant que* qui est tel que $d[u]$ est la plus petite des valeurs accessibles dans $\bar{\Delta}$. **C'est le voisin d'un sommet de $\bar{\Delta}$ le plus proche de a .** Ce sommet u est alors inséré dans l'ensemble Δ : c'est la **phase de transfert** de u de $\bar{\Delta}$ à Δ .
2. Lors de la boucle *pour*, on met à jour les distances des voisins de u qui n'ont pas encore été découverts. En effet, si x n'est pas un voisin de u , alors il n'existe pas d'arête entre u et x et $w(u, x) = +\infty$. La mise à jour de la distance n'a donc pas lieu, on n'a pas trouvé une meilleure distance à x . C'est la **phase de mise à jour des distances** des voisins de u .

L'algorithme de Dijkstra procède donc de proche en proche.

■ **Exemple 40 — Application de l'algorithme de Dijkstra.** On se propose d'appliquer l'algorithme 51 au graphe représenté sur la figure 17.3. Le tableau 17.1 représente les distances successivement trouvées à chaque tour de boucle *tant que* de l'algorithme. En rouge figurent les distances les plus courtes à a à chaque tour. On observe également que certaines distances sont mises à jour sans pour autant que le sommet soit sélectionné au tour suivant.

À la fin de l'algorithme, on note donc que les distances les plus courtes de a à b, c, d, e, f sont $[5, 1, 8, 3, 6]$. Le chemin le plus court de a à b est donc $a \rightarrow c \rightarrow e \rightarrow b$. Le plus court de a à f est $a \rightarrow c \rightarrow e \rightarrow f$. C'est la structure de données Π qui garde en mémoire le prédécesseur (parent) d'un sommet sur le chemin le plus court qui permettra de reconstituer les chemins.

Δ	a	b	c	d	e	f	$\bar{\Delta}$
\emptyset	0	7	1	$+\infty$	$+\infty$	$+\infty$	$\{a, b, c, d, e, f\}$
$\{a\}$.	7	1	$+\infty$	$+\infty$	$+\infty$	$\{b, c, d, e, f\}$
$\{a, c\}$.	6	.	$+\infty$	3	8	$\{b, d, e, f\}$
$\{a, c, e\}$.	5	.	8	.	6	$\{b, d, f\}$
$\{a, c, e, b\}$.	.	.	8	.	6	$\{d, f\}$
$\{a, c, e, b, f\}$.	.	.	8	.	.	$\{d\}$
$\{a, c, e, b, f, d\}$	$\{\}$

TABLE 17.1 – Tableau d des distances au sommet a successivement trouvées au cours de l’algorithme de Dijkstra appliqué au graphe de la figure 17.3

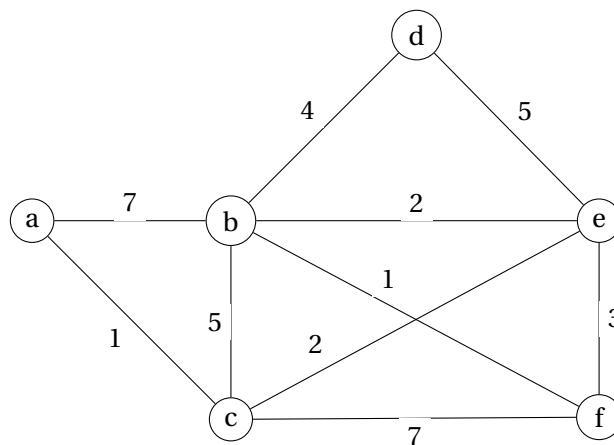


FIGURE 17.3 – Graphe pondéré à valeurs positives pour l’application de l’algorithme de Dijkstra.

Théorème 18 — L’algorithme de Dijkstra se termine et est correct.

Démonstration. Correction de l’algorithme : à chaque étape de cet algorithme, on peut distinguer deux ensembles de sommets : l’ensemble Δ est constitué des éléments dont on connaît la distance la plus courte à a et l’ensemble complémentaire $\bar{\Delta}$ qui contient les autres sommets.

D’après le principe d’optimalité, tout chemin plus court vers un sommet de $\bar{\Delta}$ passera nécessairement par un sommet de Δ . Ceci s’écrit :

$$\forall u \in \bar{\Delta}, d[u] = \min(d[v] + w[v, u], v \in \Delta) \quad (17.3)$$

On souhaite montrer qu’à la fin de chaque tour de boucle tant que (lignes 6-12), d contient les distances les plus courtes vers tous les sommets de Δ . On peut formuler cet invariant de boucle.

\mathcal{I} : à chaque fin de tour de boucle on a

$$\forall u \in \Delta, d[u] = \delta_{au} \quad (17.4)$$

$$\forall u \in \bar{\Delta}, d[u] = \min(d[v] + w[v, u], v \in \Delta) \quad (17.5)$$

À l'entrée de la boucle, l'ensemble Δ ne contient que le sommet de départ a . On a $d[a] = 0$, ce qui est la distance minimale. Pour les autres sommets de $\bar{\Delta}$, d contient :

- une valeur infinie si ce sommet n'est pas un voisin de a , ce qui, à cette étape de l'algorithme est le mieux qu'on puisse trouver,
- le poids de l'arête venant de a s'il s'agit d'un voisin, ce qui, à cette étape de l'algorithme est le mieux que l'on puisse trouver également.

On peut donc affirmer que d contient les distances entre a et tous les sommets de Δ . L'invariant est vérifié à l'entrée de la boucle.

On se place maintenant à une étape quelconque de la boucle. Notre hypothèse \mathcal{H} est que toutes les itérations précédentes sont correctes. À l'entrée de la boucle on sélectionne un sommet u , le premier de la file de priorités. Il nous alors montrer que $d[u] = \delta_{au}$.

u entre dans Δ , c'est à dire que $u \in \bar{\Delta}$ et $\forall v \in \bar{\Delta}, d[u] \leq d[v]$. Considérons un autre chemin de a à u passant par un sommet v de $\bar{\Delta}$. Comme on a $d[u] \leq d[v]$, cet autre chemin sera au moins aussi long que $d[u]$, sauf s'il existe des arêtes de poids négatif (ce qui n'est pas le cas).

Formellement, on peut écrire cela ainsi

$$\delta_{au} = \delta_{av} + \delta_{vu} \quad (17.6)$$

$$\delta_{au} \geq \delta_{av} \quad (17.7)$$

Par ailleurs, comme v appartient à $\bar{\Delta}$, il vérifie l'hypothèse d'induction. On a donc :

$$d[v] = \min(d[x] + w[x, v], x \in \Delta) \quad (17.8)$$

$$= \min(\delta_{ax} + w[x, v], x \in \Delta) \quad (17.9)$$

$$= \delta_{av} \quad (17.10)$$

la deuxième ligne étant obtenue grâce à l'hypothèse d'induction également.

$$d[u] \leq d[v] = \delta_{av} \quad (17.11)$$

$$\leq \delta_{av} \quad (17.12)$$

$$\leq \delta_{au} \quad (17.13)$$

Or, $d[u]$ ne peut pas être plus petit que la distance de a à u . On a donc finalement $d[u] = \delta_{au}$.

d contient donc les distances vers tous les sommets à la fin de l'exécution de l'algorithme.

Terminaison de l'algorithme : avant la boucle *tant que*, $\bar{\Delta}$ possède $n - 1$ éléments, si $n \in \mathbb{N}^*$ est l'ordre du graphe. À chaque tour de boucle *tant que*, l'ensemble $\bar{\Delta}$ décroît strictement d'un élément et atteint donc nécessairement zéro. Le cardinal de $\bar{\Delta}$ est donc un variant de boucle. L'algorithme se termine lorsque le cardinal de $\bar{\Delta}$ atteint zéro. ■

La complexité de l'algorithme de Dijkstra dépend de l'ordre n du graphe considéré et de sa taille m . La boucle *tant que* effectue exactement $n - 1$ tours. La boucle *pour* effectue à chaque fois un nombre de tour égal au nombre d'arêtes non découvertes qui partent du sommet u considéré et vont vers un sommet voisin de $\bar{\Delta}$. On ne découvre une arête qu'une seule fois, puisque le sommet u est transféré dans Δ au début de la boucle. Au final, on exécute donc la mise à jour des distances un nombre de fois égal à la taille m du graphe, c'est à dire son nombre d'arêtes. En notant la complexité du transfert c_t et la complexité de la mise à jour des distances c_d et en déroulant la boucle *tant que*, on peut écrire :

$$C(n, m) = (n - 1)c_t + mc_d \quad (17.14)$$

Les complexités c_d et c_t dépendent naturellement des structures de données utilisées pour implémenter l'algorithme.

Si on choisit une implémentation de d par un tableau, alors on a besoin de rechercher le minimum des distances pour effectuer le transfert : cela s'effectue au prix d'un tri du tableau au minimum en $c_t = O(n \log n)$. Un accès aux éléments du tableau pour la mise à jour est en $c_d = O(1)$. On a donc $C(n) = (n - 1)O(n \log n) + mO(1) = O(n^2 \log n)$.

Si d est implémentée par une file à priorités (un tas) comme le propose Johnson [14], alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. La complexité est alors en $C(n) = (n + m) \log n$. Cependant, pour que le tas soit une implémentation pertinente, il est nécessaire que $m = O(\frac{n^2}{\log n})$, c'est à dire que le graphe ne soit pas complet, voire un peu creux !

■ **Exemple 41 — Usage de l'algorithme de Dijkstra** . Le protocole de routage OSPF implémente l'algorithme de Dijkstra. C'est un protocole qui permet d'automatiser le routage sur les réseaux internes des opérateurs de télécommunication. Les routeurs sont les sommets du graphe et les liaisons réseaux les arêtes. La pondération associée à une liaison entre deux routeurs est calculée à partir des performances en termes de débit de la liaison. Plus liaison possède un débit élevé, plus la distance diminue.

OSPF est capable de relier des centaines de routeurs entre eux, chaque routeur relayant les paquets IP de proche en proche en utilisant le plus court chemin de son point de vue ^a. Le protocole garantit le routage des paquets par les plus courts chemins en temps réel. Chaque routeur calcule ses propres routes vers toutes les destinations, périodiquement. Si une liaison réseau s'effondre, le routeurs en sont informés et recalculent d'autres routes immédiatement. La puissance de calcul nécessaire pour exécuter l'algorithme sur un routeur, même dans le cas d'un réseau d'une centaine de routeur, est relativement faible car la plupart des réseaux de télécommunications sont des graphes relativement creux. Ce n'est pas rentable de créer des graphes de télécommunications complets, même si ce serait intéressant pour le consommateur et très robuste !

^a. Cela fonctionne grâce au principe d'optimalité de Bellman!

b Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford calcule les plus courts chemins depuis un sommet de départ, comme l'algorithme de Dijkstra. Cependant, il s'applique à des **graphes pondérés et orientés**

dont les pondérations peuvent être négatives mais sans cycles de longueur négative[4, 11, 19].

R Soit un graphe pondéré non orienté qui possède des poids négatifs. Comme les arêtes d'un graphe non orienté sont des cycles, on ne peut pas appliquer l'algorithme de Bellman-Ford. On lui préférera alors Dijkstra.

R Les poids négatifs peuvent représenter des transferts de flux (chaleur en chimie, argent en économie) et sont donc très courants.

Algorithme 52 Algorithme de Bellman-Ford, plus courts chemins à partir d'un sommet donné

```

1 : Fonction BELLMAN_FORD( $G = (V, E, w)$ ,  $a$ )
2 :    $\Pi \leftarrow$  un dictionnaire vide       $\triangleright \Pi[s]$  est le parent de  $s$  dans le plus court chemin de  $a$  à  $s$ 
3 :    $d \leftarrow$  ensemble des distances au sommet  $a$ 
4 :    $d[s] \leftarrow w(a, s)$                    $\triangleright w(a, s) = +\infty$  si  $s$  n'est pas voisin de  $a$ , 0 si  $s = a$ 
5 :   pour  $_$  de 1 à  $|V| - 1$  répéter           $\triangleright$  Répéter  $n - 1$  fois
6 :     pour  $(u, v) = e \in E$  répéter           $\triangleright$  Pour toutes les arêtes du graphe
7 :       si  $d[v] > d[u] + w(u, v)$  alors           $\triangleright$  Si le chemin est plus court par là...
8 :          $d[v] \leftarrow d[u] + w(u, v)$            $\triangleright$  Mises à jour des distances des voisins
9 :          $\Pi[v] \leftarrow u$                        $\triangleright$  Pour garder la tracer du chemin
10:    renvoyer  $d, \Pi$ 

```

■ **Exemple 42 — Application de l'algorithme de Bellman-Ford.** On se propose d'appliquer l'algorithme 52 au graphe pondéré et orienté représenté sur la figure 17.4. On note qu'il contient une pondération négative de b à f mais pas de cycle à pondération négative. Le tableau 17.2 représente les distances successivement trouvées à chaque itération.

On observe que le chemin de a à f emprunte bien l'arc de pondération négative.

Il faut noter que l'algorithme a convergé avant la fin de l'itération dans cas. C'est un des axes d'amélioration de cet algorithme.

Nº d'itération	a	b	c	d	e	f
1	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	0	5	1	8	3	6
3	0	5	1	8	3	4
4	0	5	1	8	3	4
5	0	5	1	8	3	4

TABLE 17.2 – Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Bellman-Ford appliqué au graphe de la figure 17.4

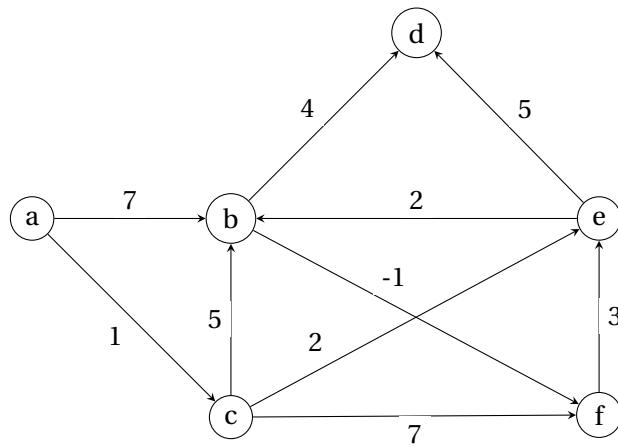


FIGURE 17.4 – Graphe pondéré et orienté à valeurs positives et négatives pour l’application de l’algorithme de Bellman-Ford.

La complexité de l’algorithme 52 est en $O(nm)$ si n est l’ordre du graphe et m sa taille.

■ **Exemple 43 — Protocole de routage RIP.** Le protocole de routage RIP utilise l’algorithme de Bellman-Ford pour trouver les plus courts chemins dans un réseau de routeur. Il est moins adapté que OSPF pour les grands réseaux.

c Algorithme de Floyd-Warshall

L’algorithme de Floyd-Warshall [10, 21, 25] est l’application de la programmation dynamique à la recherche de l’existence d’un chemin entre toutes les paires de sommets d’un graphe orienté et pondéré. Les distances trouvées sont les plus courtes. Les pondérations du graphe peuvent être négatives mais on exclut tout circuit de poids strictement négatif.

Soit un graphe orienté et pondéré $G = (V, E, w)$. G peut être modélisé par une matrice d’adjacence M

$$\forall i, j \in \llbracket 0, |V| - 1 \rrbracket, M = \begin{cases} w(v_i, v_j) & \text{si } (v_i, v_j) \in E \\ +\infty & \text{si } (v_i, v_j) \notin E \\ 0 & \text{si } i = j \end{cases} \quad (17.15)$$

Un exemple de graphe associé à la matrice d’adjacence :

$$M_{\text{init}} = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \quad (17.16)$$

est donné sur la figure 17.5. Sur cet exemple, le chemin le plus court de v_4 à v_3 vaut 3 et passe par v_2 .

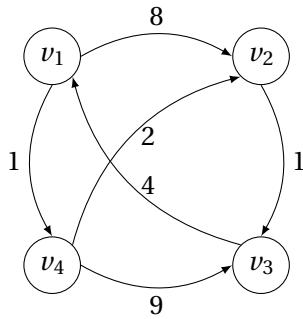


FIGURE 17.5 – Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.

Pour trouver le plus court chemin entre deux sommets, on essaye tous les chemins de toutes les longueurs possibles et on ne garde que les plus courts. Chaque étape p de l'algorithme de Floyd-Warshall est donc constitué d'un allongement **éventuel** du chemin par le sommet v_p . À l'étape p , on associe une matrice M_p qui contient la longueur des chemins les plus courts d'un sommet à un autre passant par des sommets de l'ensemble $\{v_0, v_1, \dots, v_p\}$. On construit ainsi une suite de matrice finie $(M_p)_{p \in \llbracket 0, n-1 \rrbracket}$ et on initialise la matrice M avec M_{init} .

Supposons qu'on dispose de M_{p-1} . Considérons un chemin \mathcal{C} entre v_i et v_j dont la longueur est minimale et dont les sommets intermédiaires sont dans $\{v_0, v_1, \dots, v_{p-1}\}$, $p \leq n$. Pour un tel chemin :

- soit \mathcal{C} passe par v_p . Dans ce cas, \mathcal{C} est la réunion de deux chemins dont les sommets sont dans $\{v_0, v_1, \dots, v_p\}$: celui de v_i à v_p et celui de v_p à v_j .
- soit \mathcal{C} ne passe pas par v_p .

Entre ces deux chemins, on choisira le chemin le plus court.

Disposer d'une formule de récurrence entre M_p et M_{p-1} permettrait de montrer que le problème du plus court chemin entre deux sommets d'un graphe orienté et pondéré est à sous-structure optimale. On pourrait alors utiliser la programmation dynamique pour résoudre le problème. Or, on peut traduire notre explication ci-dessus par la relation de récurrence suivante :

$$\forall p \in \llbracket 0, n-1 \rrbracket, \forall i, j \in \llbracket 0, n-1 \rrbracket, M_p(i, j) = \min(M_{p-1}(i, j), M_{p-1}(i, p) + M_{p-1}(p, j)) \quad (17.17)$$

Pour $p = 0$, on pose $M_{-1} = M_{\text{init}}$.

L'algorithme de Floyd-Warshall 53 n'est que le calcul de la suite de ces matrices. C'est un bel exemple de programmation dynamique.

R Cet algorithme effectue le même raisonnement que Bellman-Ford mais avec un vision globale, à l'échelle du graphe tout entier, pas uniquement par rapport à un sommet de départ.

Algorithme 53 Algorithme de Floyd-Warshall, plus courts chemins entre toutes les paires de sommet

```

1: Fonction FLOYD_WARSHALL( $G = (V, E, w)$ )
2:    $M \leftarrow$  la matrice d'adjacence de  $G$ 
3:   pour  $p$  de  $0$  à  $|V| - 1$  répéter
4:     pour  $i$  de  $0$  à  $|V| - 1$  répéter
5:       pour  $j$  de  $0$  à  $|V| - 1$  répéter
6:          $M(i, j) = \min(M(i, j), M(i, p) + M(p, j))$ 
7:   renvoyer  $M$ 

```

■ **Exemple 44 — Application de l'algorithme de Floyd-Warshall.** Si on applique l'algorithme au graphe de la figure 17.5, alors on obtient la série de matrices suivantes :

$$M_0 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & +\infty & 0 & +\infty \\ +\infty & 2 & 9 & 0 \end{pmatrix} \quad (17.18)$$

$$M_1 = \begin{pmatrix} 0 & 8 & +\infty & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 9 & 0 \end{pmatrix} \quad (17.19)$$

$$M_2 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ +\infty & 0 & 1 & +\infty \\ 4 & 12 & 0 & 5 \\ +\infty & 2 & 3 & 0 \end{pmatrix} \quad (17.20)$$

$$M_3 = \begin{pmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \quad (17.21)$$

$$M_4 = \begin{pmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \quad (17.22)$$

d A*

L'algorithme A*⁴ est un algorithme couteau suisse qui peut être considéré comme un algorithme de Dijkstra muni d'une heuristique : là où Dijkstra ne tient compte que du coût du chemin déjà parcouru, A* considère ce coût et une heuristique qui l'informe sur le reste du chemin à parcourir. Il faut bien remarquer que le chemin qu'il reste à parcourir n'est pas nécessairement le plus court.

4. prononcer *A étoile* ou *A star*

rement déjà exploré : parfois il est même impossible d'explorer tout le graphe. Si l'heuristique pour évaluer le reste du chemin à parcourir est bien choisie, alors A* converge aussi vite voire plus vite que Dijkstra[23].

■ **Définition 125 — Heuristique admissible.** Une heuristique \mathcal{H} est admissible si pour tout sommet du graphe, $\mathcal{H}(s)$ est une borne inférieure de la plus courte distance séparant le sommet de départ du sommet d'arrivée.

■ **Définition 126 — Heuristique cohérente.** Une heuristique \mathcal{H} est cohérente si pour tout arête (s, p) du graphe $G = (V, E, w)$, $\mathcal{H}(s) \leq \mathcal{H}(p) + w(s, p)$.

■ **Définition 127 — Heuristique monotone.** Une heuristique \mathcal{H} est monotone si l'estimation du coût **total** du chemin ne décroît pas lors du passage d'un sommet à ses successeurs. Pour un chemin (s_0, s_1, \dots, s_n) , on $\forall 0 \leq i < j \leq n, c(s_j) \geq c(s_i)$.

Soit $G = (V, E, w)$ un graphe orienté. Soit d la fonction de distance utilisée par l'algorithme de Dijkstra (cf. algorithme 51). A*, muni d'une fonction h permettant d'évaluer l'heuristique, calcule alors le coût total pour aller jusqu'à un sommet p comme suit :

$$c(p) = d(p) + h(p) \quad (17.23)$$

Le coût obtenu n'est pas nécessairement optimal, il dépend de l'heuristique.

Supposons que l'on cherche le chemin le plus court entre les sommets s_0 et p . Supposons que l'on connaisse un chemin optimal entre s_0 et un sommet s . Alors on peut écrire que le coût total vers le sommet p vaut :

$$c(p) = d(p) + h(p) \quad (17.24)$$

$$= d(s) + w(s, p) + h(p) \quad (17.25)$$

$$= d(s) + h(s) + w(s, p) - h(s) + h(p) \quad (17.26)$$

$$= c(s) + w(s, p) - h(s) + h(p) \quad (17.27)$$

Ainsi, on peut voir l'algorithme A* comme un algorithme de Dijkstra muni :

- de la distance $\tilde{d} = c$,
- et de la pondération $\tilde{w}(s, p) = w(s, p) - h(s) + h(p)$.

L'algorithme 65 donne le détail de la procédure à suivre.

Algorithme 54 A*

```

1: Fonction ASTAR( $G = (V, E, w)$ ,  $a$ ) ▷ Sommet de départ  $a$ 
2:    $\Delta \leftarrow a$ 
3:    $\Pi \leftarrow$ 
4:    $\tilde{d} \leftarrow$  l'ensemble des distances au sommet  $a$ 
5:    $\forall s \in V, \tilde{d}[s] \leftarrow \tilde{w}(a, s)$  ▷ Le graphe est partiel, l'heuristique fait le reste
6:   tant que  $\bar{\Delta}$  n'est pas vide répéter ▷  $\bar{\Delta}$  : sommets dont la distance n'est pas connue
7:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $\tilde{d}[u] = \min(\tilde{d}[v], v \in \bar{\Delta})$ 
8:      $\Delta = \Delta \cup \{u\}$ 
9:     pour  $x \in \bar{\Delta}$  répéter ▷ Ou bien  $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$ , pour tous les voisins de  $u$  dans  $\bar{\Delta}$ 
10:    si  $\tilde{d}[x] > \tilde{d}[u] + \tilde{w}(u, x)$  alors
11:       $\tilde{d}[x] \leftarrow \tilde{d}[u] + \tilde{w}(u, x)$ 
12:       $\Pi[x] \leftarrow u$  ▷ Pour garder la tracer du chemin le plus court
13:    renvoyer  $\tilde{d}, \Pi$ 

```

Quatrième partie

Semestre 3

STRUCTURES DE DONNÉES

---> HORS PROGRAMME

À la fin de ce chapitre, je sais :

- ☒ distinguer les différentes structures de données
- ☒ choisir une structure de données adaptée à un algorithme

Écrire un programme optimal en terme de complexité nécessite l'identification des structures de données utilisées très tôt dans le développement. En effet, le choix d'une structure de données plutôt qu'une autre, par exemple choisir une liste à la place d'un tableau ou d'un dictionnaire, peut rendre inefficace un algorithme selon le choix effectué.

Ce chapitre a pour but d'approfondir la définition des structures de données afin de permettre un choix éclairé. C'est pourquoi on définit d'abord ce qu'est un type de données abstrait en illustrant ce concept sur les listes, les tableaux, les piles et les files. Puis on fait le lien avec les implémentations possibles de ces types en structures de données, notamment en langage Python.

A Type abstrait de données et structure de données

■ **Définition 128 — Type abstrait de données (TAD).** Un type de données abstrait est une abstraction d'une structure de données qui ne se préoccupe pas de son implémentation sur une machine : sa structure interne est indiscernable, le type abstrait est vu de l'extérieur comme une boîte noire.

Un TAD spécifie le quoi, c'est à dire le type de données contenues ainsi que les opérations qu'on peut faire dessus. Par contre, il ne spécifie pas comment les données sont stockées ni comment les opérations sont implémentées.

■ **Définition 129 — Structure de données.** Une structure de données est une mise en œuvre

concrète d'un type abstrait, une implémentation d'un type abstrait sur dans un langage de programmation.

■ **Exemple 45 — Un entier.** Un entier est un TAD qui :

- contient une suite de chiffres *a* éventuellement précédés par un signe – ou +,
- fournit les opérations +, –, ×, //, %.

Selon le langage, ce TAD entier est implémenté en machine par un type concret différent :

- `int` en Python,
- `Integer` ou `int` en Java,
- `char, short, int, uint, long int` en C,
- `int` en OCaml.

a. peu importe la base pour l'instant...

■ **Exemple 46 — Un booléen.** De la même manière, on peut définir un TAD qui désigne un booléen. Un booléen est un TAD qui :

- se note Vrai ou Faux,
- fournit les opérations logiques ET, OU, NON...

Selon le langage, ce TAD booléen est implémenté en machine par un type concret différent :

- `bool` valant `True` ou `False` en Python,
- `boolean` valant `true` ou `false` en Java,
- `bool` valant `1` ou `0` en C,
- `bool` valant `true` ou `false` en OCaml.

R Un type abstrait de données est à une structure de donnée ce qu'un algorithme est à un programme. On spécifie un algorithme ou un type abstrait de données, mais on implémente un programme ou une structure de données.

Les exemples précédents de types abstraits de données étaient limités à des types simples. Mais il est possible de définir des types abstraits de données composés.

■ **Exemple 47 — Types abstraits de données composés.** Voici quelques types abstraits composés parmi les plus courants :

- liste,
- file,
- pile,
- arbre binaire,
- dictionnaire ou tableau associatif,

- ensemble,
- graphe.

R Il faut faire attention, car si les objets de type `list` en Python implémentent le TAD liste, ce n'est pas la seule implémentation possible.

B TAD tableaux et listes

Pour les informaticiens, les listes et les tableaux sont avant tout des TAD qui permettent de stocker de façon ordonnée des éléments.

■ **Définition 130 — TAD tableau.** Un TAD tableau représente une structure finie indiquable par des entiers. Cela signifie qu'on peut accéder à la lecture ou à l'écriture de n'importe quel élément directement en utilisant un indice, par exemple `t[3]`.

On peut décliner le TAD tableau de manière :

- statique : la taille du tableau est fixée, on ne peut pas ajouter ou enlever d'éléments.
- évolutive : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

■ **Définition 131 — TAD liste.** Un TAD liste représente une séquence finie d'éléments d'un même type qui possède un rang dans la séquence. Les données sont traitées séquentiellement, dans l'ordre du rang.

Un TAD liste est évolutif, c'est à dire qu'on peut ajouter ou enlever n'importe quel élément.

Les opérations sur un TAD liste sont :

- l'ajout ou suppression en début et/ou en fin de liste,
- l'accès à la fin de la liste.

La longueur d'une liste est le nombre d'éléments qu'elle contient. On dit qu'une liste est vide si elle ne contient aucun élément, sa longueur vaut zéro. Le début de la liste est désigné par le terme tête de liste (head), le dernier élément de la liste par la fin de la liste (tail).

Ce chapitre détaille les implémentations de trois du TAD : les tableaux (vus cette fois-ci comme structure de données concrète), les listes chaînées et les tableaux dynamiques (ou vecteurs, ou encore listes-tableaux).

C Implémentation des tableaux

a Implémentation d'un tableau statique

Dans sa version statique, un TAD tableau de taille fixe n est implémenté par un bloc de mémoire contiguë contenant n cases. Ces cases sont capables d'accueillir le type d'élément

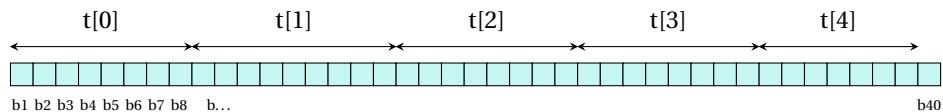


FIGURE 18.1 – Représentation d'un tableau statique en mémoire. Il peut représenter un tableau t de cinq entiers codés sur huit bits. On accède directement à l'élément i en écrivant $t[i]$.

que contient le tableau.

Par exemple, pour un TAD tableau statique de cinq entiers codés sur huit bits, on alloue un espace mémoire de 40 bits subdivisés en cinq octets comme indiqué sur la figure 18.1. Dans la majorité des langages, l'opérateur `[]` permet alors d'accéder aux éléments, par exemple $t[3]$. Les éléments sont numérotés à partir de zéro : $t[0]$ est le premier élément.

On peut estimer les coûts associés à l'utilisation d'un tableau statique comme le montre le tableau 18.1.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	créer un nouveau tableau
Ajout d'un élément à la fin	$O(n)$	créer un nouveau tableau
Suppression d'un élément au début	$O(n)$	créer un nouveau tableau
Suppression d'un élément à la fin	$O(n)$	créer un nouveau tableau

TABLE 18.1 – Complexité des opérations associées à l'utilisation d'un tableau statique.

b Implémentation d'un tableau dynamique

Un tableau dynamique est implémenté par un tableau statique de taille n_{max} supérieure à la taille nécessaire pour stocker les données. Les n données contenues dans un tel tableau le sont donc simplement entre les indices 0 et $n - 1$. Si la taille n_{max} n'est plus suffisante pour stocker toutes les données, on crée un nouveau tableau statique plus grand de taille kn_{max} et on recopie les données dedans.

Toute la subtilité des tableaux dynamiques réside dans la manière de gérer les nouvelles allocations mémoires lorsque le tableau doit être modifié.



Les tableaux dynamiques sont parfois appelés vecteurs.

R Comme le montre le tableau 18.2, l'intérêt majeur du tableau dynamique est de proposer un accès direct constant comme dans un tableau statique tout en évitant les surcouits liés à l'ajout d'éléments.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	décaler tous les éléments contigus
Ajout d'un élément à la fin	$O(1)$	amorti : il y a de la place ou pas
Suppression d'un élément au début	$O(n)$	décaler tous les éléments contigus
Suppression d'un élément à la fin	$O(1)$	amorti : il y a de la place, parfois trop

TABLE 18.2 – Complexité des opérations associées à l'utilisation d'un tableau dynamique.

R Le coût amorti signifie généralement le coût est constant. Mais, lorsqu'il n'y a plus de place, il faut bien créer la nouvelle structure adaptée au nombre d'éléments et cela a un coût linéaire $O(n)$. Donc ce coût amorti en $O(1)$ signifie c'est constant la plupart du temps mais que parfois cela peut être linéaire.

P En python le type `list` est implémenté par un tableau dynamique mais se comporte bien comme un TAD liste!

Cela a pour conséquence que :

- `L.pop()` et `L.append()` sont de complexité $O(1)$, donc supprimer ou ajouter en fin ne coûte pas cher,
- alors que `L.pop(0)` et `L.insert(0, elem)` sont de complexité $O(n)$ et donc supprimer ou ajouter en tête coûte cher.

Lorsqu'un algorithme doit supprimer ou ajouter en tête, il vaut mieux utiliser une autre structure de données qu'une `list` Python. Dans la bibliothèque `collections`, le type `deque` représente une liste sur laquelle les opérations d'ajout et de suppression en tête ou en fin sont en $O(1)$.

R Rechercher un élément dans un tableau statique ou dans un tableau dynamique présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

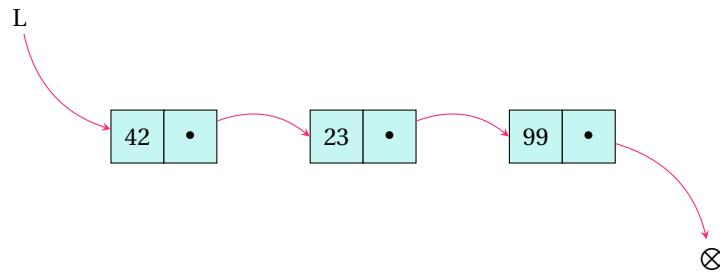


FIGURE 18.2 – Représentation d'une liste simplement chaînée d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

★ D Implémentations des listes

a Listes simplement chaînées

Un élément d'une liste simplement chaînée est une cellule constituée de deux parties :

- la première contient une donnée, par exemple un entier pour une liste d'entiers,
- la seconde contient un pointeur, c'est à dire une adresse mémoire, vers un autre élément (l'élément suivant) ou rien.

Une liste simplement chaînée se présente donc comme une succession d'éléments composites, chacun pointant sur le suivant et le dernier sur rien. En général, la variable associée à une liste simplement chaînée n'est qu'un pointeur vers le premier élément.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 18.3 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

b Listes doublement chaînées

Un élément d'une liste doublement chaînée est une cellule constituée de trois parties :

- la première contient un pointeur vers l'élément précédent,

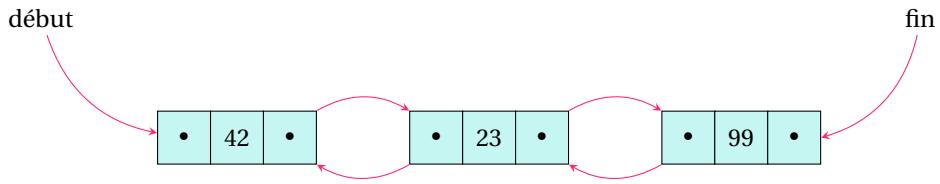


FIGURE 18.3 – Représentation d'une liste doublement chaînée d'entiers L. On conserve un pointeur sur le premier élément et un autre sur le dernier élément de la liste.

- la deuxième contient une donnée,
- la troisième contient un pointeur vers l'élément suivant.

Une liste doublement chaînée enregistre dans sa structure un pointeur vers le premier élément et un pointeur vers le dernier élément. Ainsi on peut toujours accéder directement à la tête et à la fin de liste. Par contre, c'est un peu plus lourd en mémoire et plus difficile à implémenter qu'une liste simplement chaînée. Le tableau 18.4 recense les coûts associés aux opérations sur les listes doublement chaînées.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	pointeur sur le premier élément
Accès à un élément à la fin	$O(1)$	pointeur sur le dernier élément
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	pointeur sur le premier élément
Ajout d'un élément à la fin	$O(1)$	pointeur sur le dernier élément
Suppression d'un élément au début	$O(1)$	pointeur sur le premier élément
Suppression d'un élément à la fin	$O(1)$	pointeur sur le dernier élément

TABLE 18.4 – Complexité des opérations associées à l'utilisation d'une liste doublement chaînée.

R La recherche d'un élément dans une liste simplement ou doublement chaînée présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

E Bilan de complexités des opérations sur les structures listes et tableaux

Opération	Tableau statique	Liste chaînée	Liste doublement chaînée	Tableau dynamique
Accès à un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Accès à un élément à la fin	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Accès à un élément au milieu	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ajout d'un élément au début	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Ajout d'un élément à la fin	$O(n)$	$O(n)$	$O(1)$	$O(1)$ amorti
Suppression d'un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(n)$
Suppression d'un élément à la fin	$O(n)$	$O(1)$	$O(1)$	$O(1)$ amorti

TABLE 18.5 – Complexité des opérations associées à l'utilisation des listes et des tableaux.

19

DICTIONNAIRES

À la fin de ce chapitre, je sais :

- ☒ décrire le TAD dictionnaire
- ☒ créer et utiliser un dictionnaire en Python
- ☒ expliquer l'implémentation d'un dictionnaire par une table de hachage

A Types abstraits de données (TAD)

La définition d'un TAD est donnée dans le chapitre précédent (cf. définition 128).

■ **Définition 132 — TAD Dictionnaire.** Un dictionnaire est une extension du TAD tableau dont les éléments \mathcal{V} , au lieu d'être indiqués par un entier sont indiqués par des clefs appartenant à un ensemble \mathcal{K} . Soit $k \in \mathcal{K}$, une clef d'un dictionnaire D . Alors $\mathsf{D}[k]$ est la valeur v de \mathcal{V} qui correspond à la clef k .

On dit qu'un dictionnaire est un **tableau associatif** qui associe une clef k à une valeur v .

Les opérations sur un dictionnaire sont :

1. rechercher la présence d'une clef dans le dictionnaire,
2. accéder à la valeur correspondant à une clef,
3. insérer une valeur associée à une clef dans le dictionnaire,
4. supprimer une valeur associée à une clef dans le dictionnaire.

P Un dictionnaire relie donc directement une clef, qui n'est pas nécessairement un entier, à une valeur : pas besoin d'index intermédiaire pour rechercher une valeur comme dans une liste ou un tableau. Par contre, cette clef est nécessairement d'un type immuable. Considérons l'exemple donné sur l'exemple de la figure 19.2. On suppose que les éléments chimiques

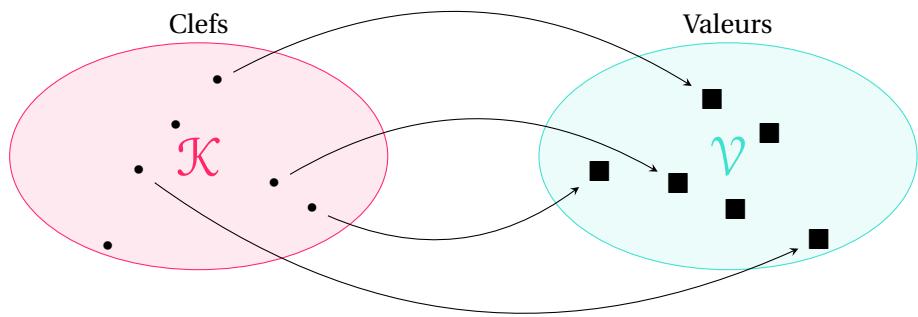


FIGURE 19.1 – Illustration du concept de dictionnaire

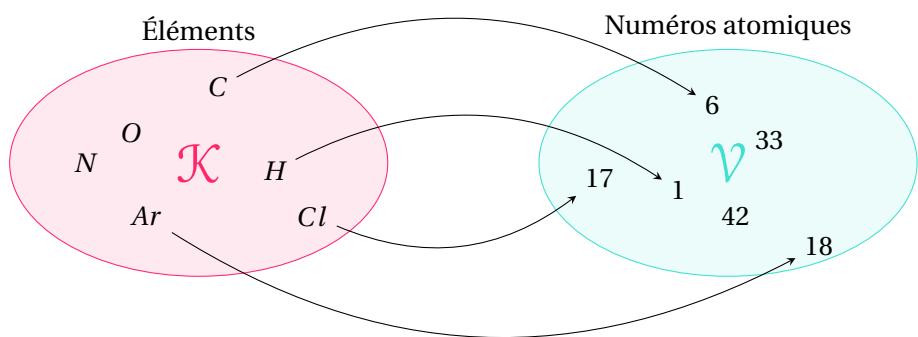


FIGURE 19.2 – Illustration du concept de dictionnaire, ensembles concrets

sont enregistrés via une chaîne de caractères : "C", "O", "H", "Cl", "Ar", "N". Soit `d`, un dictionnaire correspondant à la figure 19.2. Accéder au numéro atomique de l'élément `c` s'écrit : `d["C"]`.

(R) Un dictionnaire n'est pas une structure ordonnée, à la différence des listes ou des tableaux (cf. 5.1).

(R) L'intérêt majeur d'un dictionnaire est que la complexité pour accéder et recherche un élément est constante en $O(1)$.

■ **Exemple 48 — Usage des dictionnaires.** Les dictionnaires sont utiles notamment dans le cadre de la programmation dynamique pour la mémoisation, c'est à dire l'enregistrement des valeurs d'une fonction selon ses paramètres d'entrée. Par exemple :

- a-t-on déjà rencontré un sommet lorsqu'on parcourt un graphe?
- a-t-on déjà calculé la suite de Fibonacci pour $n = 4$?

Répondre à ces questions exige de savoir si pour une clef donnée il existe une valeur.

Si on utilise une liste pour stocker ces informations, par exemple `(n, fibo(n))`, les performances de l'exécution d'un algorithme peuvent être mauvaises : en effet, rechercher un élément dans une liste peut présenter dans le cas le pire une complexité linéaire (cf. tableau 18.5). Si on implémente bien un dictionnaire, tester l'appartenance à un dictionnaire est de complexité constante, ce qui peut accélérer grandement l'exécution d'un algorithme.

B Constructeurs de dictionnaires

Cette section s'intéresse à la manière dont on peut créer des dictionnaires en Python.

a Accolades

À tout seigneur tout honneur, les accolades sont la voie royale pour créer un dictionnaire.

```

1  d = {} # Empty dict
2  print(d, type(d)) # {} <class 'dict'>
3  d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
4  # keys are strings, values integers
5  print(d) # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1}
6
7  d = {"Alix": 14: True, "Guillaume": 7: False, ("Hannah": 24): 17}
8  print(d, type(d)) # keys are tuples, values booleans
9  # {('Alix', 14): True, ('Guillaume', 7): False, ('Hannah', 24): 17} <class 'dict'>
10
11
12 d = {13: [1, 3], 219: [2, 1, 9], 42: [4, 2]}
13 print(d, type(d)) # keys are integers, values lists
14 # {13: [1, 3], 219: [2, 1, 9], 42: [4, 2]} <class 'dict'>

```

L'exemple précédent montre que :

- Les clefs d'un dictionnaire sont nécessairement constituées de types **immuables**, c'est à dire ni une liste ni un dictionnaire par exemple. Si la clef était muable, alors le code calculé par la fonction de hachage serait variable pour une même clef : on ne saurait donc plus identifier la valeur associée ou bien celle-ci changerait.
- Les valeurs peuvent être de n'importe quel type de données.

b Constructeur `dict`

La fonction `dict()` permet également de créer un dictionnaire à partir de n'importe quel objet itérable. Elle s'utilise le plus souvent pour convertir un liste de tuples en dictionnaire.

```

1  d = dict() # Empty dict
2  print(d, type(d)) # {} <class 'dict'>
3  d = dict([('Ar', 18), ('Na', 11), ('Cl', 17), ('H', 1)])
4  # keys are strings, values integers
5  print(d) # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1}
6  d = dict(zip(['Alix', 'Guillaume', 'Hannah'], [14, 7, 24]))
7  # zip is really useful here !
8  print(d) # {'Alix': 14, 'Guillaume': 7, 'Hannah': 24}

```

c Définir un dictionnaire en compréhension

Tout comme les ensembles en mathématiques, les dictionnaires peuvent être construits à partir d'une description compréhensible de ses éléments. Cette méthode de création de dictionnaires est à rapprocher du paradigme fonctionnel.

```

1  pairs = [('Alix', 14), ('Guillaume', 7), ('Hannah', 24)]
2  d = {name: age for name, age in pairs} # comprehension dictionary
3  print(d) # {'Alix': 14, 'Guillaume': 7, 'Hannah': 24}

```

P Cette méthode de création de dictionnaire est puissante mais est très délicate à manipuler. C'est pourquoi il est préférable de ne l'utiliser que si on est vraiment sûr de soi, sinon c'est une perte de points assurée au concours. On peut l'éviter avec une boucle `for` et ainsi assurer des points.



Le paragraphe précédent est important pour l'épreuve d'informatique!

C Opérations sur un dictionnaire

a Nombres d'éléments d'un dictionnaire

La fonction `len` renvoie le nombre de clefs d'un dictionnaire.

```

1  d = dict([('Ar', 18), ('Na', 11), ('Cl', 17), ('H', 1)])
2  print(len(d)) # 4

```

P

En Python, on peut tester si un dictionnaire est vide avec la fonction `len`.

```

1  empty_dict = {}
2  if not empty_dict:    # direct
3      print("Dictionary is empty!")
4  else:
5      print("Dictionary is not empty!")
6
7  if len(empty_dict) == 0: # nb of elements
8      print("Dictionary is empty!")
9  else:
10     print("Dictionary is not empty!")
11
12 if empty_dict == {}: # compare to an empty dict
13     print('Dictionary is empty!')
14 else:
15     print('Dictionary is not empty!')

```

b Appartenance à un dictionnaire

Les mots-clefs `in` et `not in` permettent de tester l'appartenance à un dictionnaire et renvoient les booléens correspondants.

```

1  d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
2  if "Ar" in d:
3      print("Ar", d["Ar"]) # Ar 18
4  if "O" not in d:
5      print("O is not in d") # O is not in d

```

c Ajouter et supprimer un élément sur un dictionnaire

L'opérateur `[]` est nécessaire pour ajouter une valeur d'après sa clef. On peut ajouter plusieurs éléments avec `update`. Enfin, les fonctions qui retirent un élément modifient le dictionnaire en place.

```

1  d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
2  d["O"] = 8
3  print(d) # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'O': 8}
4  del d["Ar"]
5  print(d) # {'Na': 11, 'Cl': 17, 'H': 1, 'O': 8}
6  d.pop("Na")
7  print(d) # {'Cl': 17, 'H': 1, 'O': 8}
8  d.update({"F": 9, "Br": 35})
9  print(d) # {'Cl': 17, 'H': 1, 'O': 8, 'F': 9, 'Br': 35}

```

D Fusionner des dictionnaires

Pour fusionner deux dictionnaires, on peut soit utiliser la méthode `update` qui modifie le dictionnaire en place, soit utiliser un syntaxe dépendante des versions de Python.

```

1  d1 = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
2  d2 = {"F": 9, "Br": 35}
3  d1.update(d2) # simple, in place
4  print(d1) #{'Cl': 17, 'H': 1, 'O': 8, 'F': 9, 'Br': 35}
5
6  d1 = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
7  d2 = {"F": 9, "Br": 35}
8  d = {**d1, **d2} # Python 3.5
9  print(d) #{'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'F': 9, 'Br': 35}
10
11
12 d1 = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
13 d2 = {"F": 9, "Br": 35}
14 d = d1 | d2 # Python 3.9
15 print(d) #{'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'F': 9, 'Br': 35}
```

E Des dictionnaires itérables

Un dictionnaire Python est itérable, c'est à dire qu'il peut être l'objet d'une itération via une boucle `for`.

```

1  d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
2
3  for atom in d:
4      print(atom, d[atom])
5
6  for atom, number in d.items():
7      print(atom, number)
8
9  # Same result :
10 # Ar 18
11 # Na 11
12 # Cl 17
13 # H 1
14 # F 9
15 # Br 35
```

F Implémentation d'un TAD dictionnaire

On peut implémenter efficacement un TAD dictionnaire à l'aide

1. des tables de hachage,
2. des arbres rouges et noirs,

Opération	Table de hachage	Arbre de recherche équilibré
Ajout (pire des cas)	$O(n)$	$O(\log n)$
Accès (pire des cas)	$O(n)$	$O(\log n)$
Ajout (en moyenne)	$O(1)$	$O(\log n)$
Accès (en moyenne)	$O(1)$	$O(\log n)$

TABLE 19.1 – Complexité des opérations associées à l'utilisation des tables de hachage ou des arbres pour implémenter un TAD dictionnaire. Les coûts indiqués sont dans le pire des cas ou en moyenne.

3. d'arbres binaires de recherche.

Il existe de nombreuses implémentations possibles du TAD dictionnaire. On peut, par exemple, les implémenter avec des listes, mais l'efficacité n'est pas au rendez-vous. C'est pourquoi, dans la suite ce chapitre, on s'intéresse aux tables de hachage pour implémenter un TAD dictionnaire.

G Tables de hachage

a Principe

■ **Définition 133 — Table de hachage.** Une table de hachage est constituée d'un tableau t et d'une fonction de hachage h . Elle implémente un dictionnaire sur un ensemble de clefs \mathcal{K} et un ensemble de valeurs \mathcal{V} .

Pour tout élément k de \mathcal{K} , $h(k)$ est l'indice de la case du tableau t auquel on stocke la valeur v , ce qui peut s'exprimer ainsi :

$$\forall (k, v) \in (\mathcal{K} \times \mathcal{V}), t[h(k)] \leftarrow v \quad (19.1)$$

La figure 19.3 illustre l'implémentation d'un dictionnaire par une table de hachage.

■ **Définition 134 — Fonction de hachage.** Un fonction de hachage prend une clef en entrée et génère un index entier associé à cette clef.

Plus formellement, si \mathcal{K} est l'un ensemble des clefs (non nécessairement numériques) et \mathcal{V} l'ensemble des valeurs associées aux clefs de cardinal n , on peut définir une fonction de hachage :

$$h : \mathcal{K} \longrightarrow \llbracket 0, n-1 \rrbracket \quad (19.2)$$

$$k \longmapsto i \quad (19.3)$$

Soit k le cardinal de \mathcal{K} , c'est à dire le nombre de clefs possibles. On pourrait représenter un TAD dictionnaire à l'aide d'un tableau de dimension k et une fonction bijective $h : \mathcal{K} \longrightarrow \llbracket 0, n-1 \rrbracket$. L'accès aux éléments et l'ajout d'un élément seraient en $O(1)$, le temps de calculer la

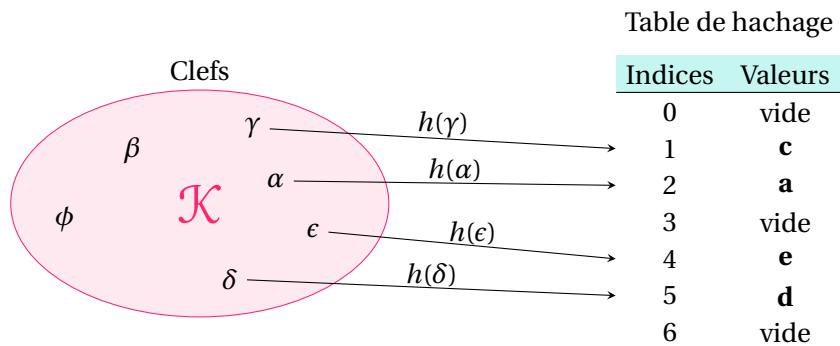


FIGURE 19.3 – Illustration de l’implémentation d’un dictionnaire par table de hachage. La fonction de hachage h permet de calculer les indices du tableau. **La valeur a associée à α se trouve à la case $h(\alpha)$ du tableau.** Toutes les clefs n’ont pas forcément de valeur associée à un moment donné de l’algorithme. Dans ce cas, à l’indice associé à cette clef, le tableau est vide.

valeur de la fonction bijective pour une clef donnée. Néanmoins, d’un point de vue complexité mémoire, cette solution n’est pas réalisable : le nombre de clefs possibles est souvent immense alors que les clefs effectivement utilisées sont moins nombreuses. Ce qui nous amèneraient à réservier un espace mémoire bien supérieur aux besoins réels.

On adopte donc l’hypothèse réaliste suivante : la taille m du tableau qu’on utilise est petite devant le nombre de clefs possibles, c’est à dire $m \ll c$. En procédant ainsi, on renonce à l’injectivité de la fonction de hachage et donc à sa bijectivité, car on engendre des collisions : il pourra exister des clefs différentes pour lesquelles le code calculé par la fonction de hachage sera le même. Tout dépend du nombre de clefs utilisées et de la fonction de hachage.

b Choix d’une fonction de hachage

Le choix d’une fonction de hachage n’est pas évident. Ces fonctions doivent permettre de générer un index dont la taille est inférieure à celle du tableau, tout en distinguant au mieux les clefs, tout en évitant le plus possible les collisions. On recherche donc des fonctions de hachage qui possèdent les caractéristiques suivantes :

1. son calcul doit être rapide,
2. pour une même clef, on obtient un même code (**cohérence**),
3. pour des clefs différentes, on obtient des codes différents (**injectivité**). Dans le cas contraire, on obtient un **collision** qu’on cherchera à minimiser.
4. pour des clefs qui se ressemblent, les codes obtenus doivent être très différents. D’une manière générale, les codes doivent présenter une distribution uniformément répartie sur l’espace des indices (**répartition uniforme**).

Pour minimiser les collisions, c’est à dire lorsque la fonction de hachage appliquée à deux clefs différentes c_1 et c_2 produit le même résultat $h(c_1) = h(c_2)$, on cherche à répartir uniformé-

ment les clefs dans les différentes cases du tableau. Ceci revient à faire en sorte que la probabilité qu'une valeur v associée à une clef c occupe la case i du tableau devrait être proche de $1/m$, si la taille de ce tableau est m . En faisant cette hypothèse, si on suppose qu'on utilise g clefs, alors la probabilité p d'obtenir des codes différents lors du calcul des g clefs par la fonction de hachage vaut :

$$p = \frac{m(m-1)\dots(m-g+1)}{m^g} = \frac{m!}{(m-g)!m^g} \quad (19.4)$$

On en déduit la probabilité de collision p_c :

$$p_c = 1 - \frac{m(m-1)\dots(m-g+1)}{m^g} = \frac{m!}{(m-g)!m^g} \quad (19.5)$$

g/m	5000	10000	100000	1000000
100	0,63	0,39	0,05	0,005
500	0,99	0,99	0,71	0,12
1000	1	1	0,99	0,39
1500	1	1	0,99	0,68
2000	1	1	0,99	0,86
2500	1	1	0,99	0,96

TABLE 19.2 – Probabilité de collision p_c dans l'hypothèse d'une répartition uniforme des valeurs dans le tableau d'une table de hachage. Même dans le cas d'un tableau à un million d'éléments, la probabilité de collision est quasi-certaine dès que la taille des clefs utilisées est supérieure à 2500. C'est le paradoxe des anniversaires.

Une rapide évaluation de cette probabilité est donnée sur le tableau 19.2. On en conclut que, quelle que soit la taille du tableau, les collisions existeront. Il faut donc trouver un moyen de les gérer.

c Fonctions de hachages possibles

Une fonction de hachage h peut procéder en deux étapes :

1. une fonction h_e qui encode la clef d'entrée,
2. et une fonction h_c qui compresse le code dans l'ensemble des indexs.

Plus formellement, on la fonction de hachage comme une fonction composée :

$$h_e : \mathcal{K} \longrightarrow \mathbb{N} \quad (19.6)$$

$$h_c : \mathbb{N} \longrightarrow \llbracket 0, n-1 \rrbracket \quad (19.7)$$

et

$$h = h_e \circ h_c \quad (19.8)$$

■ **Exemple 49 — Fonctions d'encodage des clefs.** Pour compresser une clef, il faut d'abord l'encoder, c'est à dire la convertir en un nombre entier. Dans ce but, on peut par exemple, si la clef est une chaîne de caractères $c_0c_1\dots c_{k-1}$, utiliser le code ASCII associé à un caractère $\text{ascii}(c_i)$ pour calculer $\sum_{i=0}^{k-1} \text{ascii}(c_i)2^{8+k}$. On verra en TP que faire juste la somme des valeurs ASCII des caractères n'est pas forcément une bonne idée. Prendre l'adresse en mémoire non plus.

L'important est que l'encodage génère un même code pour une même clef et un code vraiment différent si les clefs sont différentes.

■ **Exemple 50 — Hachage par modulo simple.** Si la taille du tableau de la table de hachage est m , on peut choisir la fonction $h : c \rightarrow c \bmod m$. Le choix de m mérite néanmoins quelques points d'attention : on évitera de prendre des nombres du type 2^q ou 2^{q-1} . On choisit généralement un nombre premier éloigné pas trop proche d'une puissance de 2 pour garantir une bonne répartition des clefs.

■ **Exemple 51 — Hachage par multiplication et modulo .** On peut également choisir la fonction $h : c \rightarrow \lfloor (\phi c \bmod 1) \times m \rfloor$. Si ϕ est le nombre d'or, cette méthode garantit une bonne répartition des clefs sans restreindre les valeurs de m .

d Gestion des collisions

Deux grandes méthodes permettent de gérer les collisions :

1. **par chainage** : stocker valeurs associées aux collisions dans une même case sous la forme d'une liste comme l'illustre la figure 19.4. Cette solution induit une augmentation de la complexité car en cas de collision, on n'accède plus à l'élément directement : il faut le chercher dans une liste. C'est ce qui explique le $O(n)$ dans le pire des cas sur le tableau 19.1.
2. **par adressage ouvert** : chercher une place vide dans le tableau en le sondant et placer la valeur dedans. Cela induit que le nombre de clefs utilisées est inférieur à la taille du tableau et ralentit l'accès à un élément. Lorsque le tableau est trop petit, on peut en changer pour un plus grand : cela a un coût également.

e Implémentation des `dict` en Python

L'implémentation du TAD dictionnaire en Python fait l'objet de travaux et d'évolutions en permanence et c'est un sujet complexe. En résumé, pour implémenter les `dict`, Python utilise des tables de hachage en adressage ouvert dont la taille évolue dynamiquement (comme le type `list`).

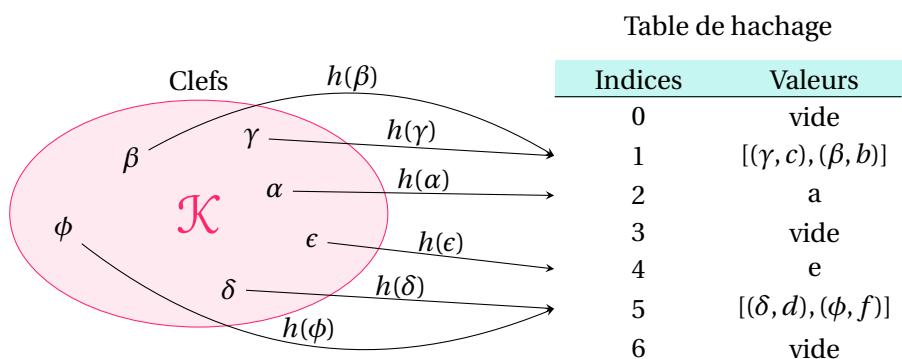


FIGURE 19.4 – Illustration de l'implémentation d'un dictionnaire par table de hachage avec chainage. Les clefs β et ϕ , engendrent des collisions. On stocke donc les valeurs possibles pour un même code (indice) dans une liste.

20

PROGRAMMATION DYNAMIQUE

À la fin de ce chapitre, je sais :

- ☒ énoncé les principes de la programmation dynamique
- ☒ distinguer cette méthode des approches gloutonnes et diviser pour régner
- ☒ formuler récursivement le problème du sac à dos

A Motivations

Dans la famille des algorithmes de décomposition, c'est à dire les algorithmes qui cherchent à décomposer un problème en sous-problèmes afin de le résoudre, on distingue trois grandes familles :

1. les algorithmes gloutons (cf. chapitre 14),
2. les algorithmes de type diviser pour régner (cf. chapitre 13)
3. la programmation dynamique.

La figure 20.1 schématise ces trois approches sous la forme d'arbres de décomposition de problèmes en sous-problèmes. Les algorithmes de type gloutons ou de type diviser pour régner ont des limites :

1. même s'il existe des algorithmes gloutons optimaux¹, c'est à dire qui produisent une solution optimale au problème, la plupart du temps ce n'est pas le cas.
2. même si l'approche diviser pour régner est très efficace pour de nombreux problèmes², elle nécessite que les sous-problèmes soient indépendants. Or, parfois, il n'en est rien,

1. On peut citer notamment : la planification de tâches dans le temps qui ne se chevauchent pas, l'algorithme de Prim ou de Huffmann.

2. On peut citer notamment : la transformée de Fourier rapide (FFT), l'exponentiation rapide, les approches dichotomiques.

certains sous-problèmes ont des sous-problèmes en commun, ils ne sont pas indépendant, ils se chevauchent. Dans ce cas, l'approche diviser pour régner devient inefficace puisqu'elle résout plusieurs fois les mêmes sous-problèmes.

Afin de dépasser ces limites, on étudie la programmation dynamique.

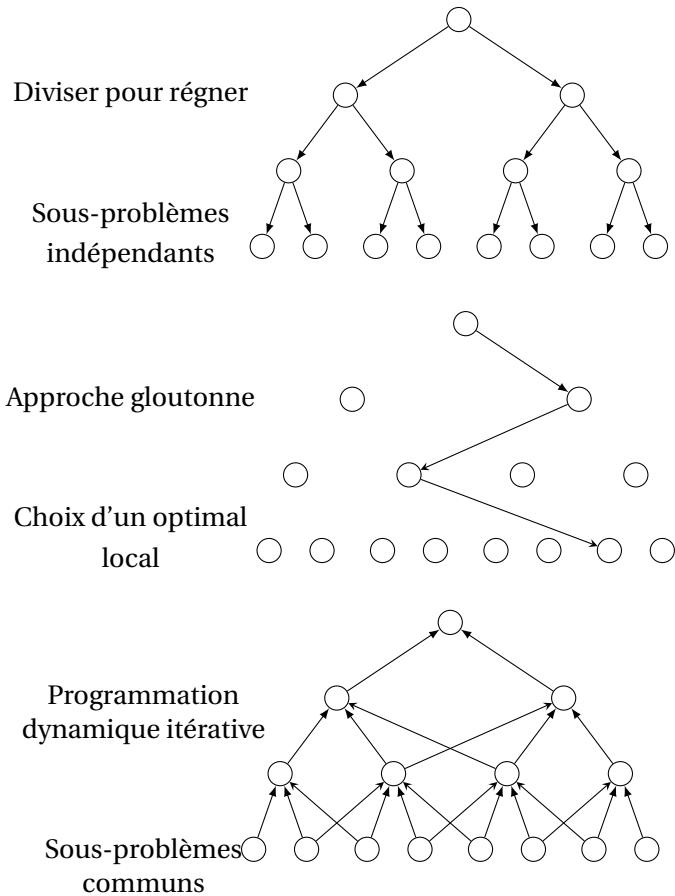


FIGURE 20.1 – Schématisation des différentes approches des algorithmes de décomposition d'un problème en sous-problèmes : diviser pour régner, approche gloutonne et programmation dynamique itérative.

B Principles de la programmation dynamique

■ **Définition 135 — Programmation dynamique.** La programmation dynamique est une méthode de construction des solutions optimales d'un problème par combinaison des solutions optimales de sous-problèmes. Certaines combinaisons sont implicitement rejetées si elles appartiennent à un sous-ensemble qui n'est pas intéressant : on ne construit que les solutions optimales des sous-problèmes utiles à la construction de la solution optimale. Cette approche :

- décompose le problème \mathcal{P} en sous-problèmes de taille moindre,
- ne résout un sous-problème qu'une seule fois.

En général, le cadre de l'application de la programmation dynamique sont les problèmes d'optimisation combinatoire dont la sous-structure est optimale. Pour ce genre de problèmes, il y a de nombreuses solutions possibles³. Chaque solution possède une valeur propre que l'on peut quantifier. On cherche alors soit à la minimiser soit à la maximiser, dans tous les cas, on cherche au moins une valeur optimale.

M Méthode 9 — Développement d'un algorithme de programmation dynamique Pour créer un algorithme de programmation dynamique répondant à un problème, il faut :

1. Formuler récursivement le problème en sous-problèmes ordonnés,
2. Créer et initialiser un tableau de résolution,
3. Compléter ce tableau du bas vers le haut en calculant les solutions des sous-problèmes dans l'ordre croissant de la récurrence trouvée.

■ **Définition 136 — Principe d'optimalité de Bellman.** La solution optimale à un problème d'optimisation combinatoire présente la propriété suivante : quel que soit l'état initial et la décision initiale prise, les décisions qui restent à prendre pour construire une solution optimale forment une solution optimale par rapport à l'état qui résulte de la première décision^a.

a. PRINCIPLE OF OPTIMALITY. An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions[3].

■ **Définition 137 — Sous-structure optimale.** En informatique, un problème présente une sous-structure optimale si une solution optimale peut être construire à partir des solutions optimales à ses sous-problèmes.

Le principe d'optimalité et la notion de sous-structure optimale sont illustrés par la figure 20.2 qui représente le problème du plus court chemin dans un graphe. Résoudre ce problème via la programmation dynamique suppose qu'on a un problème à sous-structure optimale, ce qui est le cas.

3. Penser au problème du sac à dos par exemple.

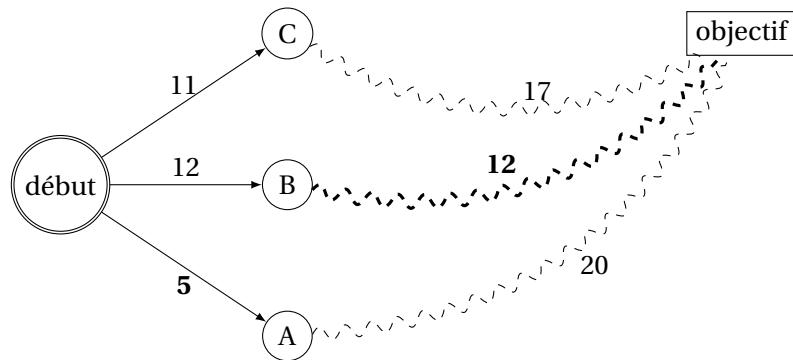


FIGURE 20.2 – Illustration du principe d'optimalité et de sous-structure optimale : trouver le plus court chemin dans ce graphe. Les nombres représentent la longueur d'un chemin. Les lignes droites sont des arrêtes. Les lignes ondulées indiquent les plus courts chemins dans le graphe : il faut imaginer qu'on n'a pas représenté tous les sommets.

On peut exprimer le plus court chemin récursivement en fonction du premier chemin choisi et du reste du graphe. Cela peut s'écrire formellement mathématiquement ou, plus simplement, comme suit :

Le plus court chemin est le chemin le plus court à choisir parmi :

- le chemin qui passe par A suivi par le chemin le plus court de A à l'objectif,
- le chemin qui passe par B suivi par le chemin le plus court de B à l'objectif,
- le chemin qui passe par C suivi par le chemin le plus court de C à l'objectif.

La résolution du problème amènera à choisir le chemin qui passe par B.

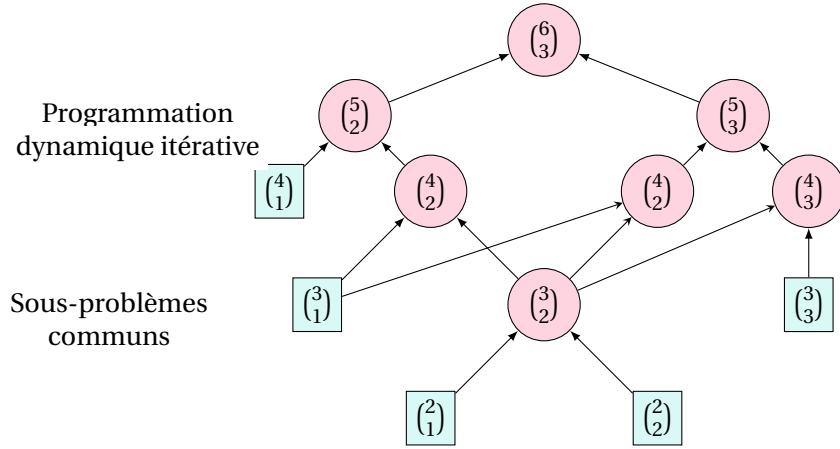
C Exemples simples de programmation dynamique

■ **Exemple 52 — Construction des solutions du calcul de $\binom{n}{k}$.** La formule de récurrence

$$\binom{n}{k} = \begin{cases} 0 & \text{si } k > n \\ 1 & \text{si } k = n \text{ ou } k = 0 \\ n & \text{si } k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sinon} \end{cases} \quad (20.1)$$

permet de construire le triangle de Pascal (cf. tableau 20.1). On peut considérer ce triangle comme le tableau de résolution du problème du calcul de $\binom{n}{k}$ par une méthode de programmation dynamique (cf. figure 20.3).

■ **Exemple 53 — Algorithme de calcul des termes de la suite Fibonacci.** On considère la suite de Fibonacci : $(u_n)_{n \in \mathbb{N}}$ telle que $u_0 = 0$, $u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$.

FIGURE 20.3 – Programmation dynamique du calcul de $\binom{6}{3}$.

$n \backslash k$	0	1	2	3	4	5	6
6	1	6	15	20	15	6	1
5	1	5	10	10	5	1	0
4	1	4	6	4	1	0	0
3	1	3	3	1	0	0	0
2	1	2	1	0	0	0	0
1	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0

TABLE 20.1 – Triangle de Pascal : valeurs de $\binom{n}{k}$. En couleur apparaissent les éléments nécessaires pour calculer $\binom{6}{3}$ qu'il faut mettre en parallèle de la figure 20.3. On a représenté le triangle du bas vers le haut afin de montrer le lien avec la programmation dynamique.

Pour calculer le u_n , on peut procéder de plusieurs manières différentes comme le montre les algorithmes 55 et 56. Cependant, ces deux approches n'ont pas la même efficacité. La première est une approche récursive multiple descendante. Les appels multiples montrent qu'on se trouve dans le cadre d'un problème pour lequel **les sous-problèmes ne sont pas indépendants**. Dans ce cas, l'algorithme 55 calcule inutilement plusieurs fois les mêmes termes et est inefficace. Par exemple, pour calculer u_4 selon cette approche on doit calculer $u_3 + u_2$. Mais le calcul de u_3 va lancer le calcul $u_2 + u_1$. On va donc calculer deux fois u_2 .

L'algorithme 56 propose une version itérative ascendante dans l'ordre des termes : on calcule d'abord le premier terme puis le second et ainsi il n'y a pas de calculs redondants. Cette approche est dans l'esprit de la programmation dynamique : on cherche à calculer du bas vers le haut comme indiqué sur la figure 20.1 pour éviter les calculs redondants. On pourrait construire un graphe similaire à celui de la figure 20.3.

Algorithme 55 Fibonacci récursif

```

1: Fonction REC_FIBO(n)
2:   si n = 0 alors                                     ▷ Condition d'arrêt
3:     renvoyer 0
4:   sinon si n = 1 alors                               ▷ Condition d'arrêt
5:     renvoyer 1
6:   sinon
7:     renvoyer REC_FIBO(n-1) + REC_FIBO(n-2)          ▷ Appels récursifs multiples
  
```

Algorithme 56 Fibonacci itératif, sans calculs redondants.

```

1: Fonction ITE_FIBO(n)
2:   si n = 0 alors
3:     renvoyer 0
4:   sinon si n = 1 alors
5:     renvoyer 1
6:   sinon
7:      $u_0 \leftarrow 0$ 
8:      $u_1 \leftarrow 1$ 
9:     pour i de 2 à n répéter
10:      tmp  $\leftarrow u_0$ 
11:       $u_0 \leftarrow u_1$ 
12:       $u_1 \leftarrow \text{tmp} + u_1$ 
13:    renvoyer  $u_1$ 
  
```

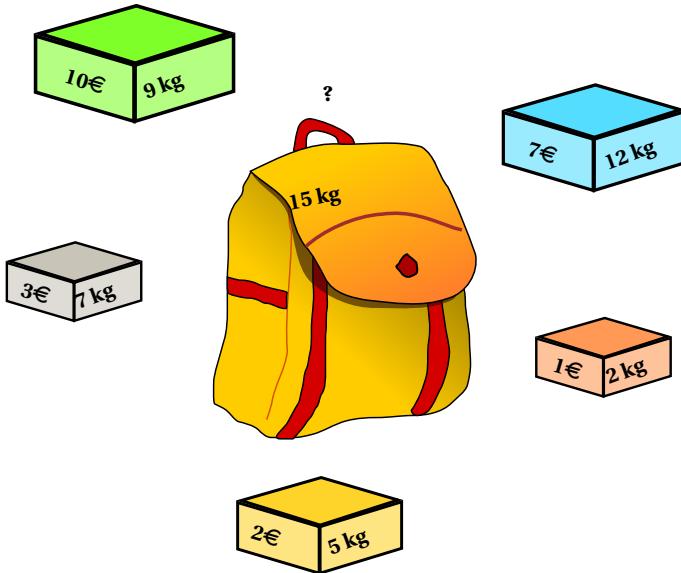


FIGURE 20.4 – Illustration du problème du sac à dos (d'après [Wikipedia](#)). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 €. Le poids total admissible dans le sac est 15kg.

D Le retour du sac à dos

a Position du problème

On cherche à remplir un sac à dos comme indiqué sur la figure 20.4. Chaque objet que l'on peut insérer dans le sac est **insécable**⁴ et possède une valeur et un poids connus. On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant⁵ le poids à π .

On a vu au chapitre 14 que l'approche gloutonne ne donnait pas toujours le résultat optimal. On se propose donc de résoudre le problème par la programmation dynamique en appliquant la méthode 9.

b Modélisation du problème

Soit un ensemble $\mathcal{O}_n = \{o_1, o_2, \dots, o_n\}$ de n objets de valeurs v_1, v_2, \dots, v_n et de poids respectifs p_1, p_2, \dots, p_n . Soit un sac à dos n'admettant pas un poids emporté supérieur à π . On note également qu'on peut mettre au plus n objets dans le sac.

Les objets sont rangés dans une liste et dans un ordre quelconque. Ils sont indicés par i variant de 1 à n . Un objet o_i possède une valeur v_i et un pèse p_i .

Avec ces notations, on peut formuler le problème du sac à dos comme suit.

4. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas en mettre qu'une partie.

5. On accepte un poids total inférieur ou égal à π .

■ **Définition 138 — Problème du sac à dos.** Comment remplir un sac à dos en maximisant la valeur totale emportée $V = \sum_{i=1}^n v_i$ tout en ne dépassant pas le poids maximal admissible par le sac à dos, c'est à dire en respectant la contrainte $\sum_{i=1}^n p_i \leq \pi$? On note ^a le problème du sac à dos $KP(n, \pi)$ et une solution optimale à ce problème $S(n, \pi)$.

a. en anglais, ce problème est nommé Knapsack Problem, d'où le KP.

c Formulation récursive du problème en sous-problèmes non indépendants

Pour chaque objet o_i , si $p_i \leq \pi$, on peut le mettre dans le sac. La formulation récursive s'énonce alors ainsi :

- Soit l'objet o_i fait partie d'une solution optimale. Alors la fonction à maximiser vaut la valeur de l'objet o_i plus la valeur maximale atteignable avec les $n-1$ objets restants, sachant qu'on ne peut plus mettre que $\pi - p_i$ kg dans le sac.
- Soit l'objet o_i ne fait pas partie d'une solution optimale. Alors la fonction à maximiser vaut la valeur maximale atteignable avec les $n-1$ objets restants une fois cet objet o_i écarté. On peut toujours mettre π kg dans le sac.

Formellement, on exprime cette récursivité ainsi :

$$S(n, \pi) = \begin{cases} 0 & \text{si } n = 0 \text{ ou si } \pi = 0 \\ \max(v_i + S(n-1, \pi - p_i), S(n-1, \pi)) & \text{si } p_i \leq \pi \\ S(n-1, \pi) & \text{sinon} \end{cases} \quad (20.2)$$

Cette formulation prouve que **le problème du sac à dos possède une sous-structure optimale et que les problèmes se chevauchent** : pour un poids π maximal donné, **calculer une solution optimale de $KP(\mathcal{O}_n, \pi)$ nécessite de savoir calculer une solution optimale pour les $n-1$ premiers objets de la liste et pour des poids π et $\pi - p_i$** . Schématiquement, on peut représenter cette démarche comme sur la figure 20.5.

R Attention au sens des notations : $S(i, P)$ est une solution au problème $KP(i, P)$. Pour ce problème, on ne peut prendre que les i premiers objets de la liste et le poids maximum admissible est P . Si on considère $S(i-1, P - p_i)$, alors on ne peut prendre que les $i-1$ premiers objets de la liste et le poids maximal admissible est $P - p_i$. Les objets sont rangés dans un ordre quelconque.

d Crédit et initialisation du tableau de résolution

On cherche donc maintenant à créer un tableau à double entrée qui recense toutes les solutions optimales nécessaires à la résolution du problème $KP(\mathcal{O}_n, \pi)$. Ce tableau a pour dimension $(n+1, \pi+1)$:

- le nombre i d'objets dans le sac d'un côté à valeur dans $\llbracket 0, n \rrbracket$. La valeur pour $i = 0$ est nulle, on ne prend pas d'objet.

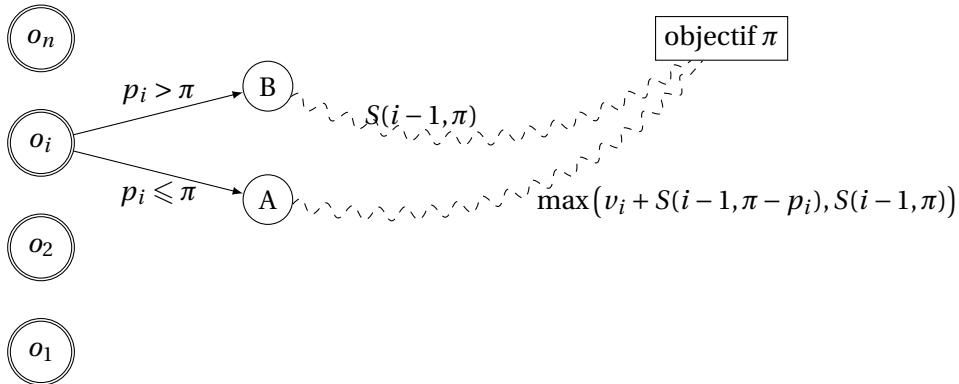


FIGURE 20.5 – Formulation récursive du problème du sac à dos.

Indice i de l'objet	1	2	3	4	5
valeur (€)	10	7	1	3	2
poids (kg)	9	12	2	7	5

TABLE 20.2 – Synthèse des informations relatives au problème de la figure 20.4.

- les poids P atteignables de l'autre à valeur dans $\llbracket 0, \pi \rrbracket$. La valeur pour $P = 0$ est nulle, on ne prend pas d'objet.

La valeur d'une case du tableau est $S(i, P)$.

e Complémentation du tableau par résolution dans l'ordre croissant des sous-problèmes

On considère la liste d'objets décrite sur le tableau 20.2 et qui correspond au problème décrit sur la figure 20.4. L'ordre des objets est **arbitraire**. Le résultat du calcul est donné sur le tableau 20.3. On a construit ce tableau du bas vers le haut en suivant l'algorithme 57.

Naturellement, sur l'exemple donné sur la figure 20.4, il est possible de calculer à la main les valeurs du tableau. Ce n'est guère le cas dans des situations réalistes, c'est pourquoi il faut maintenant écrire l'algorithme qui va permettre de compléter ce tableau dans l'ordre et ainsi de résoudre notre problème.

R Pour bien comprendre, il peut être utile de reproduire la figure 20.3 pour l'exemple du sac à dos KP(5, 7) par exemple!

E Programmation dynamique itérative

L'algorithme 57 donne la procédure de résolution de $KP(n, \pi)$ par programmation dynamique, de bas en haut et de manière itérative. Aucun calcul redondant n'est effectué puisqu'on

Indice i																
5	0 0 1 1 1 2 2 3 3 10 10 11 11 11 12 12															
4	0 0 1 1 1 1 1 3 3 10 10 11 11 11 11 11															
3	0 0 1 1 1 1 1 1 1 10 10 11 11 11 11 11															
2	0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10															
1	0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10															
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0															
Poids (kg) \rightarrow	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

TABLE 20.3 – Tableau de résolution du sac à dos dans le cas de la figure 20.4 donnant les valeurs de $S(i, P)$, avec $i \in \llbracket 0, 5 \rrbracket$ et $P \in \llbracket 0, 15 \rrbracket$.

0	$S(n, 1)$	$S(n, \pi)$
0
0	$S(i, 1)$	$S(i, P)$
0	$S(i-1, 1)$...	$S(i-1, P-p_i)$...	$S(i-1, P)$
0
0	0	0	0	0	0	0	0	0	0

FIGURE 20.6 – Schéma de remplissage du tableau pour le problème KP(n, π). Pour calculer $S(i, P)$ on a besoin de $S(i-1, P)$ et de $S(i-1, P-p_i)$.

complète le tableau de bas en haut.

Algorithme 57 KP(n, π) par programmation dynamique, de bas en haut

```

1: Fonction KP_DP( $p, v, \pi, n$ )                                $\triangleright p$  la liste de poids,  $v$  celle des valeurs
2:    $S \leftarrow$  un tableau d'entiers de taille  $(n + 1, \pi + 1)$ 
3:   pour  $i$  de 0 à  $n$  répéter                                 $\triangleright$  de bas en haut
4:     pour  $P$  de 0 à  $\pi$  répéter                             $\triangleright$  d bas en haut
5:       si  $i = 0$  ou  $P = 0$  alors
6:          $S[i, P] \leftarrow 0$ 
7:       sinon si  $p_i \leq P$  alors
8:          $S[i, P] \leftarrow \max(v_i + S[i - 1, P - p_i], S[i - 1, P])$ 
9:       sinon
10:       $S[i, P] \leftarrow S[i - 1, P]$ 
11:   renvoyer  $S[n, \pi]$ 
  
```

Les complexités temporelle et spatiale de l'algorithme 57 sont en $O(n\pi)$.

F Programmation dynamique récursive : mémoïsation

Il est possible de résoudre le problème du sac à dos de manière récursive comme le montre l'algorithme 58. Néanmoins, comme les sous-problèmes se chevauchent, de nombreux calculs redondants sont effectués. Pour des valeurs importantes de n et π , cet algorithme est totalement inefficace.

Algorithme 58 KP(n, π) par programmation récursive brute

```

1: Fonction KP_REC( $p, v, \pi, n$ )                                $\triangleright p$  la liste de poids,  $v$  celle des valeurs
2:   si  $i = 0$  ou  $P = 0$  alors
3:     renvoyer 0
4:   sinon si  $p[i] \leq P$  alors
5:     renvoyer  $\max(v_n + \text{KP\_REC}(p, v, \pi - p_n, n - 1), \text{KP\_REC}(p, v, \pi, n - 1))$ 
6:   sinon
7:     renvoyer KP_REC( $p, v, \pi, n - 1$ )
  
```

■ **Définition 139 — Mémoïsation.** La mémoïsation est une approche récursive de la programmation dynamique qui stocke les résultats intermédiaires dans un tableau et qui, avant chaque appel récursif, vérifie si le calcul à faire récursivement a déjà été effectué. Si c'est le cas, la solution stockée dans le tableau est utilisée. Sinon la récursivité s'exécute.

L'algorithme 59 donne la procédure de résolution de KP(n, π) par mémoïsation. Cette technique récursive est une approche de la programmation dynamique récursive.

Algorithme 59 KP(n, π) par programmation dynamique et mémoïsation

```

1: Fonction KP_MEM( $p, v, \pi, n, S$ )            $\triangleright S$  est un tableau d'entiers de taille  $(n + 1, \pi + 1)$ 
2:   si  $i = 0$  ou  $P = 0$  alors
3:     renvoyer 0
4:   sinon
5:     si la solution  $S[n, P]$  a déjà été calculée alors
6:       renvoyer  $S[n, P]$ 
7:     sinon si  $p_n \leq \pi$  alors
8:        $S[n, P] \leftarrow \max(v_n + \text{KP\_MEM}(p, v, \pi - p_n, n - 1, S), \text{KP\_MEM}(p, v, \pi, n - 1, S))$ 
9:       renvoyer  $S[n, P]$ 
10:    sinon
11:       $S[n, P] \leftarrow \text{KP\_MEM}(p, v, \pi, n - 1)$ 
12:      renvoyer renvoyer  $S[n, P]$ 

```

R La mémoïsation est une technique qui se prête bien à l'usage de dictionnaires : on peut stocker chaque solution déjà calculée dans un dictionnaire dont la clef est le couple (i, P) et la valeur la solution au problème KP(i, P). L'accès à la solution est rapide, en $O(1)$.

21

BASES DE DONNÉES RELATIONNELLES

Flight reservation systems decide whether or not you exist. If your information isn't in their database, then you simply don't get to go anywhere.

Arthur Miller

À la fin de ce chapitre, je sais :

- faire la différence entre une base de données et un SGBD
- modéliser des entités relatives à une réalité simple
- modéliser une association simple entre deux entités en précisant les cardinalités
- traduire une association dans un modèle relationnel
- interpréter un modèle relationnel de base de données

A Pourquoi ?

L'ubiquité de l'information n'est plus une vue de l'esprit aujourd'hui. La vie contemporaine n'est parfois qu'une suite ininterrompue de sollicitations de systèmes d'information qui ne cessent de générer des opérations sur des bases de données via des systèmes de gestion de bases de données. Par exemple :

- prendre un train, un bus, un avion, un bateau, une autoroute,
- louer un vélo, une voiture, une place de stationnement,
- réserver un hôtel, un article d'un magasin, un activité sportive ou culturelle, recevoir un colis,
- consulter un médecin, acheter des médicaments en pharmacie, faire des analyses médicales,

- payer ses impôts, demander une aide à la CAF, chercher un texte légal en vigueur dans le journal officiel,
- prendre une photo ou une vidéo sur son smartphone, envoyer un message à un proche, partager des informations via une application,
- jouer à un jeu en ligne, démarcher des clients ou chercher des collaborateurs via un réseau social...

Ces systèmes interagissent¹ entre eux à la suite de nos actions, souvent à l'insu même de l'instigateur. La plupart des requêtes sont tellement rapides qu'elles sont imperceptibles à l'être humain, à moins d'analyser finement ce que l'on est en train de faire et l'information que l'on manipule. Nous produisons collectivement et à travers eux des quantités inimaginables d'information.

■ **Exemple 54 — Quantité d'information sur les serveurs à l'échelle mondiale.** À l'échelle mondiale, on estime actuellement la quantité d'information créée et sauvegardée annuellement^a à une centaine de Zio (zebiocbts), soit 100×2^{70} octets. Pour se donner une idée, si on estime qu'un machine personnelle possède en standard un disque dur de 128 Gio, cela représente 858993459200 disques durs, soit environ 858 milliards. Nous sommes pour l'instant moins de 8 milliards d'habitants sur Terre... Gageons que ces données créées, sauvegardées, diffusées et recopiées sont toujours utiles!?

a. Ces données ne tiennent pas compte des machines personnelles (ordinateurs ou smartphones) ni des objets connectés.

🇬🇧 **Vocabulary 20 — bit** ↗ bit Unité binaire de l'information. Une information est pour l'instant représentée en machine par un ensemble de symboles binaires généralement notées 0 et 1.

🇬🇧 **Vocabulary 21 — Byte** ↗ octet. Un octet est constitué de huit bits.

Les bases de données étaient déjà au cœur des systèmes d'information, même avant l'ère informatique. Elles sont présentes aujourd'hui dans les systèmes d'exploitation, les applications des smartphones et même sur les objets connectés.

Ce chapitre traite de la modélisation de l'information pour les bases de données. Car, avant de pouvoir manipuler concrètement l'information de manière cohérente, il faut dégager un modèle qui décrit la réalité qu'on est en train de manipuler pour lui donner un sens. C'est l'enjeu de la modélisation conceptuelle et logique et du modèle relationnel associé.

B Données et gestion des données

■ **Définition 140 — Base de données.** Une base de données est un ensemble de données enregistré représentant une partie du monde ou une activité humaine.

1. Les canadiens parlent d'infonuagique pour l'informatique dans le cloud.

Il est de l'intérêt du concepteur et de l'utilisateur de limiter le périmètre d'une base de données. Par exemple, on s'intéressera aux documents d'une médiathèque, aux pneumatiques distribués en France, aux légumes cultivables en agriculture biologique ou aux dossiers scolaires des étudiants.

Une base de données est une idée que l'on peut concrétiser en écrivant sur des cahiers ou en programmant des ordinateurs selon ce que l'on veut en faire.

■ **Définition 141 — Système de Gestion de Base de Données (SGBD).** Un système de gestion de base de données est un ensemble logiciel dont le but est de gérer une base de données stockée sur support informatique. Il s'agit de garantir l'accès, la lecture, l'écriture, l'interrogation ou la modification de la base de données d'une manière cohérente et efficace.

■ **Exemple 55 — SGBD.** Parmi les SGBD les plus utilisés, on peut citer Oracle, MySQL, PostgreSQL, SQLite ou Microsoft SQL Server.

L'interaction avec les SGDB contemporains a été standardisée et on peut aujourd'hui utiliser le langage SQL pour interagir avec la plupart des SGBD. Au-delà de la manipulation des données, un SGBD doit pouvoir veiller à la cohérence et à l'intégrité des informations toute en garantissant des transactions rapides.

■ **Définition 142 — Structured Query Language.** SQL est un langage informatique normalisé servant à interagir avec des bases de données relationnelles. Il permet de manipuler (rechercher, ajouter ou supprimer, modifier) des informations, mais aussi de créer et de contrôler l'intégrité des données et des transactions sur une base de données relationnelle.

C De la conception à l'implémentation physique

La réalisation d'une base de données n'est pas une agrégation brute de données : c'est la réalisation d'une structure cohérente avec un contexte. Pour que les recherches s'effectuent simplement et efficacement, il est nécessaire de décrire, au-delà des données elles-mêmes, les liens sémantiques qui existent et qui relient les données entre elles, mais également la manière dont on stocke logiquement et physiquement les données ainsi que les transactions possibles.

La conception d'une base de données s'effectue la plupart du temps en trois étapes :

1. on établit d'abord un modèle **conceptuel** de la réalité à modéliser,
2. puis on choisit une représentation **logique** de ce modèle (généralement le modèle relationnel)
3. enfin, on implémente **physiquement** la base de données à l'aide d'un SGBD.

R Un modèle de base de données n'est pas unique. Selon le degré des exigences spécifiées par les utilisateurs de la base, différents modèles peuvent convenir. Les modèles proposés

par la suite sont donc toujours perfectibles. C'est normal. En général en ingénierie, on peut parvenir à des solutions différentes pour résoudre un même problème. On parle alors d'équifinalité.

a Modélisation conceptuelle

L'étape de modélisation conceptuelle s'affranchit de toute considération d'implémentation de la base de données pour se concentrer sur la **description sémantique de la réalité à modéliser**, c'est à dire le sens de cette réalité. Un réalité consiste le plus souvent en une activité humaine décrite simplement par un texte comme le montre l'exemple 59. L'analyse de ce texte et des mots le composant permet de dégager les éléments à modéliser. Cette modélisation est donc avant tout une activité du *langage* et une activité d'abstraction². Elle s'appuie généralement sur le modèle entité-association ou le langage Unified Modeling Language (UML).

■ **Définition 143 — Entité ou classe.** Une entité est un concept qui représente une abstraction d'un ensemble de données que l'on peut regrouper et caractériser par des attributs communs

■ **Exemple 56 — La voiture, une entité simple.** Une voiture est une entité : elle représente les véhicules à quatre roues motorisés. Elle est caractérisée d'être d'une marque particulière et par le fait de posséder quatre roues, des portes, un moteur, un poste de pilotage.

■ **Exemple 57 — Un conducteur de voiture, une entité simple.** Un conducteur est une entité : elle représente la personne qui conduit un véhicule. Elle est caractérisée un nom et un numéro de permis.

■ **Définition 144 — Association.** Une association entre entités définit un ensemble de liens entre ces entités. Généralement, une association est représentée par un verbe. Elle est pondérée par des cardinalités qui précisent le nombre d'entités mises en jeu dans cette association. Une association peut réunir une, deux ou trois entités, on dit alors qu'elle est unaire, binaire ou ternaire.

■ **Exemple 58 — Conduire, une association simple.** Conduire est une association binaire : elle relie une entité conducteur à une entité véhicule. Selon la réalité à modéliser, on pourra pondérer cette association différemment. Par exemple, une personne peut ne conduire qu'une seule voiture. Mais certains conducteurs conduisent plusieurs voitures. De la même manière, une voiture peut être conduite par un ou plusieurs conducteurs selon la situation.

■ **Définition 145 — Modèle entité-association.** Le modèle entité-association cherche à représenter sur un même schéma des entités et des associations identifiables dans une réalité à modéliser.

2. Il n'y a pas que les mathématiques dans la vie, il y a le langage aussi!

■ **Exemple 59 — Une réalité à modéliser.** Afin d'illustrer le modèle entité-association, on a capturé une réalité à modéliser. Cette capture se fait la plupart du temps grâce à des entretiens avec les personnes impliquées dans le contexte. Voici le texte décrivant la réalité à modéliser :

Une société de production recense les concerts donnés par des orchestres sur une saison. Chaque orchestre est décrit par son nom. Elle qualifie les orchestres en catégories selon leur style (baroque, symphonique, chambre, jazz, expérimental), sachant qu'un même orchestre ne peut faire partie que d'une seule catégorie. Un orchestre est dirigé par un chef d'orchestre dont on connaît le nom et le prénom. Au cours d'une même saison, un même orchestre peut accueillir plusieurs chefs différents. De la même manière, un chef peut diriger plusieurs orchestres. La résidence d'un orchestre définit son lieu de répétition. Une résidence est occupée par un seul orchestre mais un orchestre peut ne pas avoir de résidence. Elle est caractérisée par une ville et un pays.

On souhaite disposer d'une base de données pour accéder facilement à :

- *la liste des chefs d'un orchestre,*
- *la liste des pays de résidence des orchestres,*
- *la liste des orchestres de même style.*

La réalité décrite dans l'exemple 59 peut être modélisée et représentée graphiquement comme sur la figure 21.1. L'intérêt de cette modélisation n'est pas uniquement son intelligibilité pour l'être humain : des outils³ capables de transformer ce modèle graphique en modèle logique voire physique⁴ existent et sont couramment utilisés.

Sur la figure 21.1, les cardinalités viennent quantifier le nombre d'entités qui participent à une association. Le premier nombre quantifie la cardinalité minimale de l'association, le second la cardinalité maximale.

■ **Définition 146 — Types d'association.** On définit plusieurs types d'association en fonction des cardinalités **maximales** qu'elles mettent en jeu :

1. de un à un,
2. de un à plusieurs,
3. de plusieurs à plusieurs.

■ **Exemple 60 — Interprétations des cardinalités des associations de l'exemple 59.** La notation 0 – 1 à droite de l'association *occupe* sur la figure 21.1 signifie qu'un orchestre occupe zéro ou une seule résidence, conformément à l'énoncé de l'exemple 59.

Le 1 – 1 au-dessous de l'association *occupe* signifie qu'une résidence est occupée au plus par un orchestre et au moins par un orchestre, c'est à dire exactement par un orchestre, conformément à l'énoncé.

Cette association **occupe est de type un à un**, car de chaque côté la cardinalité maximale

3. des compilateurs

4. c'est à dire en base de données concrète et utilisable

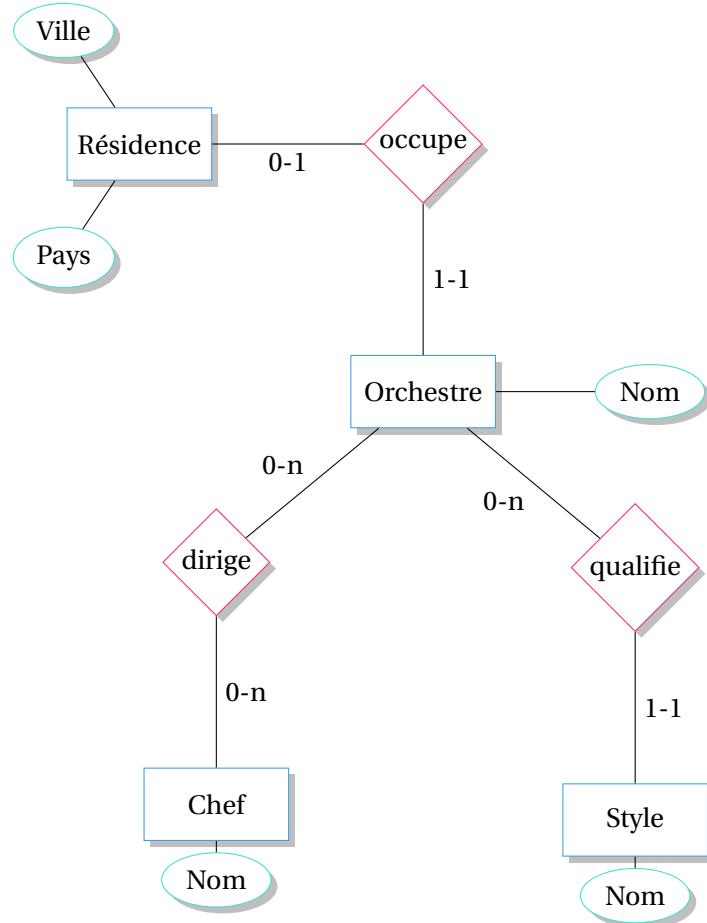


FIGURE 21.1 – Un modèle conceptuel de type entité-association associé à la réalité de l'exemple 59. Les entités sont inscrites dans des rectangles, les attributs des entités dans des ovales et les associations entre les entités dans des losanges. Des cardinalités sont précisées sur les arcs qui relient les entités aux associations

vaut un.

La notation $0-n$ au dessus de l'association ***qualifie*** sur la figure 21.1 signifie qu'un style qualifie zéro ou plusieurs orchestres.

Le $1-1$ au-dessous de l'association ***qualifie*** signifie qu'un orchestre est qualifié exactement par un style.

Cette association ***qualifie est de type un à plusieurs***, car la cardinalité maximale de part et d'autre de l'association vaut un et n.

La notation $0-n$ au dessus de l'association ***dirige*** sur la figure 21.1 signifie qu'un chef d'orchestre peut diriger zéro ou plusieurs orchestres. Le zéro laisse la possibilité à un chef d'être libre de tout engagement.

Le $0-n$ au dessous de l'association ***dirige*** signifie qu'un orchestre peut être dirigé par zéro ou plusieurs chefs. Le zéro laisse la possibilité à un orchestre de ne pas avoir de chef.

L'association ***dirige est de type plusieurs à plusieurs***, car la cardinalité maximale de chaque côté de l'association vaut n.

R Pour comprendre les cardinalités sur la figure 21.1, une bonne maîtrise de la voix active et de la voix passive en français est nécessaire. On lit une association dans un sens à la voix active et dans l'autre sens à la voix passive. L'interprétation des cardinalités découle de cette lecture à double voix.

b Modélisation logique

■ **Définition 147 — Modèle logique.** Un modèle logique s'élabore à partir d'un modèle conceptuel. Il explique comment sont structurées les données dans le logiciel de base de données. Il ne dit pas comment implémenter les structures mais spécifie :

- le type de structure qui enregistre les entités et les associations,
- le type de données des attributs et les valeurs possibles,
- les clefs qui identifient les entités,
- les contraintes référentielles liées à l'intégrité entre les éléments du modèle.

R Par essence, une base de données est partagée entre plusieurs utilisateurs ^a. Certains utilisateurs ont plus de droits que d'autres. Par exemple, ils peuvent avoir le droit d'insérer des nouveaux enregistrements. Il n'est pas rare de trouver dans les bases de données mal conçues :

- des doublons,
- des valeurs aberrantes,
- des valeurs incohérentes.

Pour remédier à ceci, on impose des contraintes sur la modèle logique de la base de données.

^a. voir des millions...

■ **Définition 148 — Contrainte d'unicité.** Une contrainte d'unicité précise qu'un attribut ou un ensemble d'attributs d'une entité est unique dans la base de données. On ne peut donc plus insérer deux informations différentes dont ces attributs seraient identiques. Cela permet d'éviter l'insertion d'informations redondantes. Elle est réalisée grâce à une clef primaire.

■ **Définition 149 — Clef primaire.** Une clef primaire est un attribut ou ensemble d'attributs d'une entité qui permet de distinguer un représentant de cette entité d'un autre dans une base de données. C'est un identifiant.

■ **Exemple 61 — Identifier une voiture et un conducteur.** L'identifiant choisi pour une entité voiture est généralement sa plaque d'immatriculation qui est unique au moins à l'échelle d'un pays. Ainsi, on ne pourra pas confondre deux voitures. De même, pour un conducteur, le plus sûr est de choisir son numéro de permis de conduire. On pourrait imaginer utiliser la combinaison de son nom et de sa date de naissance. Cela peut malheureusement ne pas suffire dans certaines situations ^a.

a. Un conducteur nommé Jean Martin, étant donné qu'il porte le prénom et le nom les plus courants en France, peut avoir un homonyme né le même jour. Ce n'est pas exclu.

Si la contrainte d'unicité protège des doublons, elle ne protège pas des valeurs aberrantes ou incohérentes.

■ **Définition 150 — Contrainte référentielle.** Une contrainte référentielle spécifie qu'un attribut d'une entité faisant référence à une autre doit nécessairement référencer l'autre entité en question pour ne pas dupliquer les informations ^a. Ce référencement se fait via une clef étrangère.

a. et les rendre incohérente par la même occasion...

■ **Définition 151 — Clef étrangère.** Une clef étrangère est un attribut ou ensemble d'attributs d'une entité qui permet de faire référence à un représentant d'une autre entité dans une base de données. C'est, le plus souvent, la clef primaire d'une autre entité dans la base.

Construire un modèle logique d'une base de données, c'est donc se poser de nombreuses questions sur les données et ce qu'on peut ou doit en faire. Toutes les réponses formulées ont un impact sur la structure logique de la base de données.

Le modèle logique généralement utilisé dans le domaine des bases de données est le modèle relationnel qui est décrit à la section D.

c Modélisation physique

■ **Définition 152 — Modèle physique.** Un modèle physique est l'implémentation d'un modèle logique dans un SGBD.

Pour implémenter ce modèle physique en mémoire, le SGBD crée alors :

- la base de données (sur un serveur ou dans un fichier),
- toutes les structures de données nécessaires pour enregistrer les données,
- les contraintes logicielles d'unicité et de référencement,
- des vues pour simplifier la visualisation des données,
- des index pour accélérer l'exécution des requêtes.

La réalisation d'une base de données physiquement est hors programme. Dans toutes les épreuves, on considère que la base de données a déjà été implémentée, elle est donc fournie.

D Le modèle relationnel

Cette section décrit plus précisément le modèle relationnel qui est au programme de l'épreuve d'informatique commune. C'est à partir de ce modèle que l'on peut effectuer des requêtes sur des bases de données.

Le modèle relationnel repose sur l'algèbre relationnelle, de solides fondements mathématiques qui ne sont pas exposés ici car hors programme, mais qui justifient pleinement son adoption massive comme modèle logique de référence.

■ **Définition 153 — Modèle relationnel.** Le modèle relationnel est un modèle logique de base de données pour lequel :

- l'information est structurée en tableaux à deux dimensions : les relations.
- les entités décrites dans ces tableaux doivent l'être indépendamment des associations qui les relient.

Un exemple de modèle relationnel est donné sur la figure 21.2 : on y retrouve les définitions données ci-dessous.

■ **Définition 154 — Relation.** Une relation est tableau à deux dimensions constitué :

1. d'un entête qui contient la liste des noms des attributs associés à leur domaine,
2. d'un corps qui est un ensemble de lignes dont les éléments sont des valeurs qui appartiennent aux domaines des attributs correspondant dans l'entête.

Les attributs d'une relation relèvent tous d'une entité commune.

■ **Définition 155 — Ligne ou enregistrement.** Un enregistrement est une ligne dans le corps d'une relation. Un enregistrement représente une instance d'une entité. Généralement on parle de ligne au niveau logique et d'enregistrement au niveau physique.

R Au niveau physique, on désigne une relation par le terme **table**, ce qui est légitime car son implémentation est un tableau. Les **lignes** d'une relation sont les enregistrements de la table. Les colonnes d'une table sont les attributs de la relation.

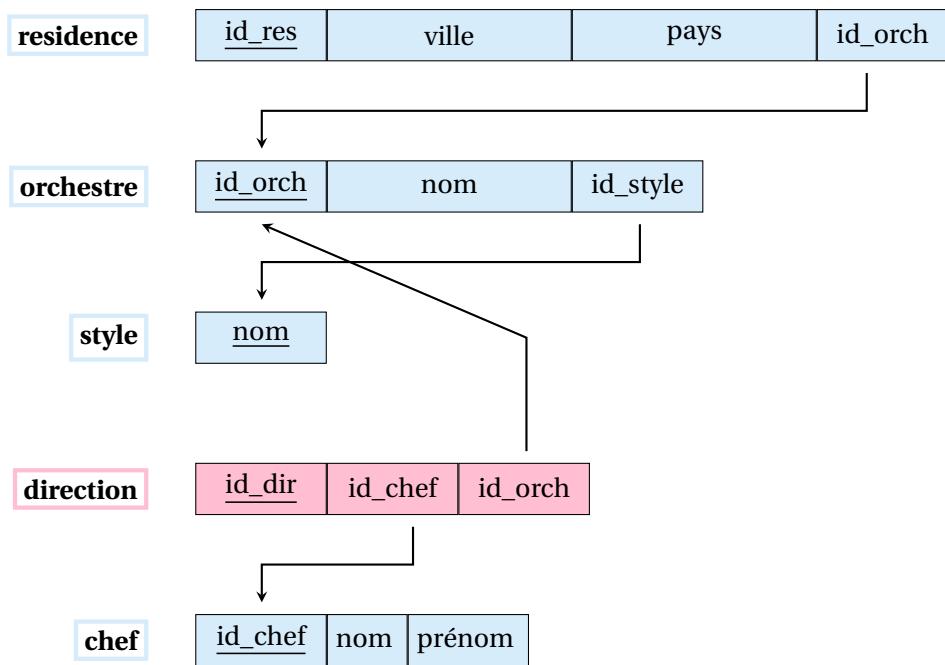


FIGURE 21.2 – Modèle relationnel construit à partir du modèle conceptuel 21.1. Seules les en-têtes de colonne des relations sont représentées. Les associations du modèle conceptuel ont été intégrées aux tables en ajoutant des colonnes qui contiennent les clefs primaires et les clefs étrangères nécessaires ou en créant une nouvelle table (direction). Les flèches mettent en évidence ces associations.

■ **Définition 156 — Domaine d'un attribut.** Le domaine d'un attribut est l'ensemble des valeurs qu'il peut prendre.

■ **Exemple 62 — Domaine d'un attribut *paiement*.** Si l'attribut d'une entité *paiement* est une monnaie, alors le domaine associé à cet attribut est l'ensemble de toutes monnaies possibles : euro, dollar, rouble, yen, livre sterling...

R Certains SGBD n'utilisent le modèle relationnel, on parle de systèmes NoSQL. Ils sont souvent utilisés lorsque les modèles de données à gérer varient rapidement.

La représentation des entités du modèle conceptuel est naturel dans le modèle relationnel : une entité est représentée par une table. Par contre, représenter une association demande un travail supplémentaire.

E Traduction des associations dans le modèle relationnel

Cette section s'attache à décrire comment on peut traduire les associations d'un modèle conceptuel dans un modèle relationnel.

M **Méthode 10 — Transformation d'une association de un à un** Une association de un à un se traduit par l'apparition d'une clef étrangère dans une des deux tables du modèle relationnel.

Par exemple, sur le modèle conceptuel de la figure 21.1, la relation *occupe* est une association de un à un. Concrètement, au niveau des tables du modèle relationnel, on peut ajouter une colonne à la table *résidence*, colonne de même type que la clef primaire de la table *orchestre*. Cette colonne de la table *résidence* devient une clef étrangère qui référence la table *orchestre*, signifiant ainsi qu'une résidence est occupée par un orchestre comme le montre la figure 21.3.

R Les colonnes ajoutées aux tables et représentant des clefs étrangères portent souvent le même nom que la clef primaire à laquelle elles se réfèrent. C'est une bonne pratique. Cependant, il est possible de leur donner des noms différents.

R Les cardinalités minimales sont généralement traduites dans le modèle relationnel par des contraintes qui n'apparaissent pas sur les schémas mais dans la manière de compléter les colonnes des clefs étrangères.

Pour une association dont la cardinalité est minimale est 1, on impose à la clef étrangère d'être définie : par exemple, pour le modèle de la figure 21.1, le fait qu'une résidence soit nécessairement occupée par un orchestre implique que le SGBD doit vérifier qu'à chaque nouvelle insertion d'une résidence, cette clef soit présente et référence un orchestre en particulier.

Pour une cardinalité minimale 0, lors de l'insertion d'un nouvel enregistrement, une clef étrangère peut ne pas être définie.

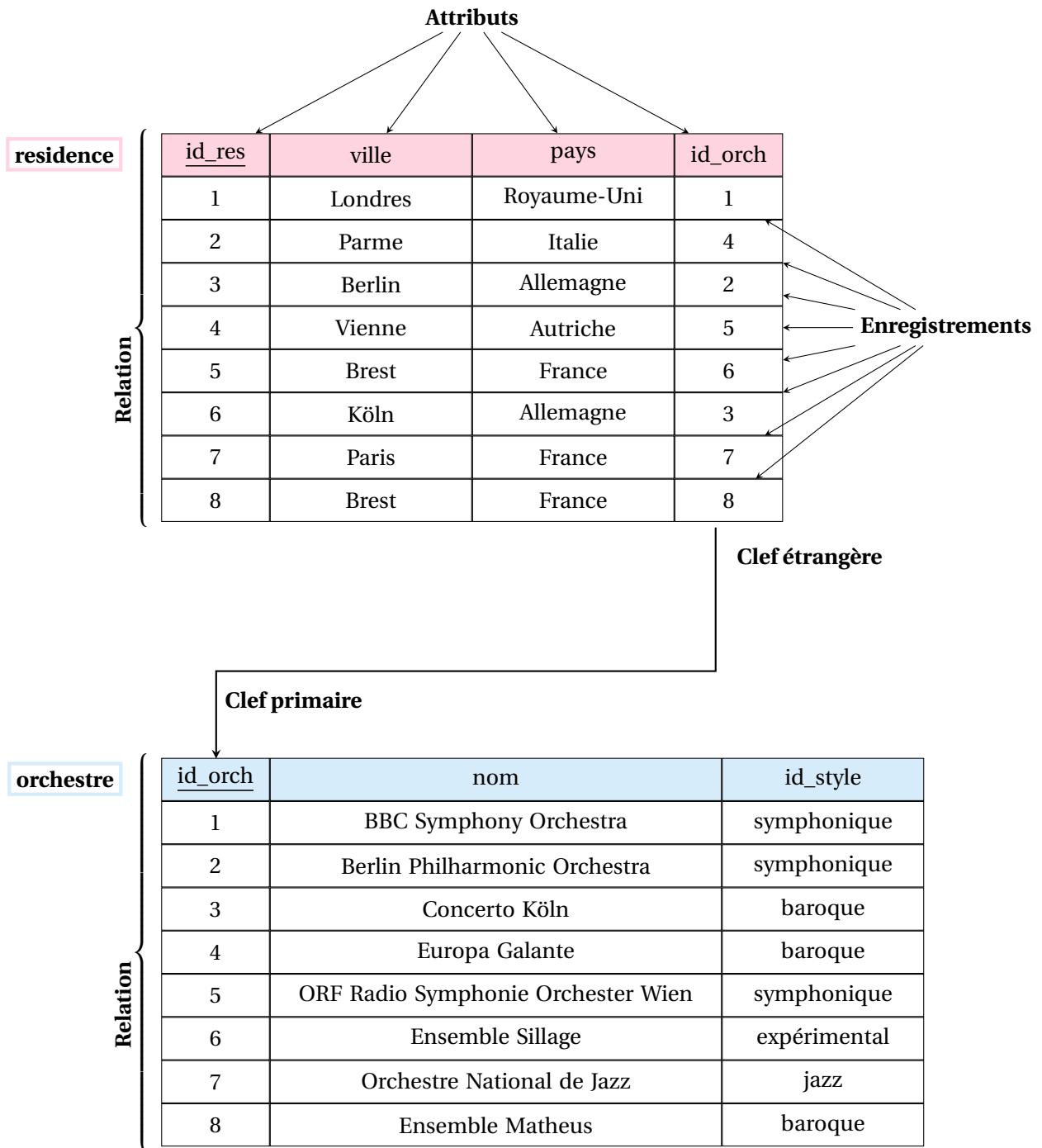


FIGURE 21.3 – Traduction d'une association de un à un dans un modèle relationnel. Modèle relationnel spécifiant les tables résidence et orchestre du modèle relationnel de la figure 21.2.

M Méthode 11 — Transformation d'une association de un à plusieurs Une association de un à plusieurs se traduit par l'apparition d'une clef étrangère dans la table dont la cardinalité maximale vaut un.

Par exemple, sur le modèle conceptuel de la figure 21.1, la relation *qualifie* est une association de un à plusieurs. Concrètement, au niveau des tables du modèle relationnel, on peut ajouter une colonne à la table *orchestre*, colonne de même type que la clef primaire de la table *style*. Cette colonne de la table *orchestre* devient une clef étrangère qui référence la table *style*, signifiant ainsi qu'un orchestre est qualifié par un style, comme le montre comme le montre la figure 21.4.

orchestre

<u>id_orch</u>	nom	<u>id_style</u>
1	BBC Symphony Orchestra	symphonique
2	Berlin Philharmonic Orchestra	symphonique
3	Concerto Köln	baroque
4	Europa Galante	baroque
5	ORF Radio Symphonie Orchester Wien	symphonique
6	Ensemble Sillage	expérimental
7	Orchestre National de Jazz	jazz
8	Ensemble Matheus	baroque

style

<u>nom</u>
symphonique
chambre
jazz
expérimental
baroque

Clef étrangère

Clef primaire

FIGURE 21.4 – Traduction d'une association de un à plusieurs dans un modèle relationnel. Modèle relationnel spécifiant les tables *orchestre* et *style* du modèle relationnel de la figure 21.2.

M **Méthode 12 — Transformation d'une association de plusieurs à plusieurs** Une association de plusieurs à plusieurs se traduit par la création d'une nouvelle table dans la base de données. On choisit généralement de nommer cette table d'après le nom de l'association. Cette nouvelle table contient deux clefs étrangères, chacune pointant vers une extrémité différente de l'association.

Par exemple, sur le modèle conceptuel de la figure 21.1, la relation *dirige* est une association de plusieurs à plusieurs. Concrètement, au niveau des tables du modèle relationnel, on ajoute une table nommée direction. Cette table comporte au moins deux colonnes, deux clefs étrangères : l'une pointe vers la clef primaire de la table orchestre et l'autre vers la clef primaire de la table chef. Une direction est le fait qu'au cours de la saison, un chef dirige un orchestre. La figure 21.5 montre comment transformer une relation de plusieurs à plusieurs dans le cas du modèle conceptuel de la figure 21.1.

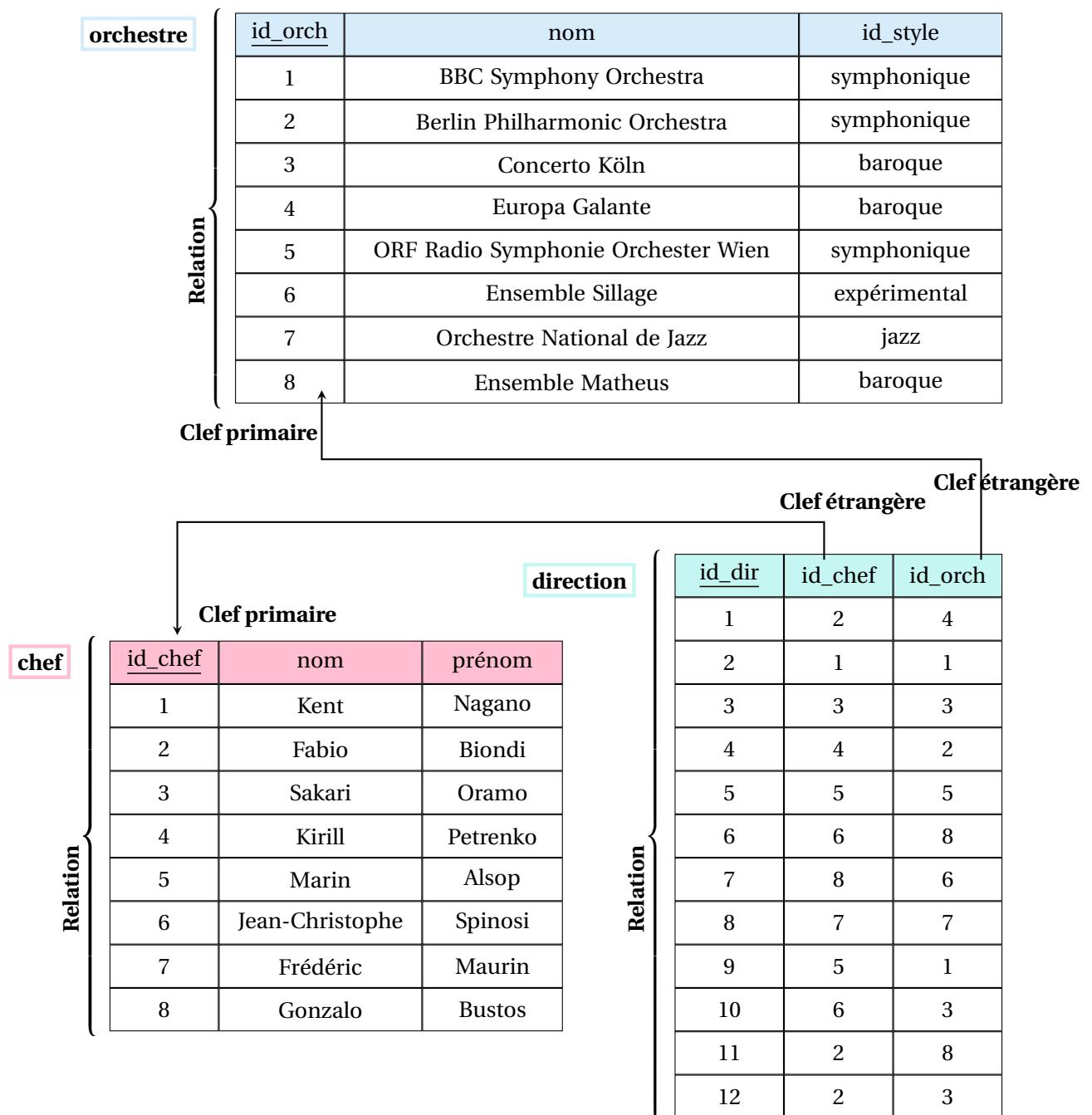


FIGURE 21.5 – Traduction d'une association de plusieurs à plusieurs dans un modèle relationnel. Modèle relationnel spécifiant les tables orchestre, direction et chef qui implémente l'association *dirige* du modèle conceptuel de la figure 21.1.

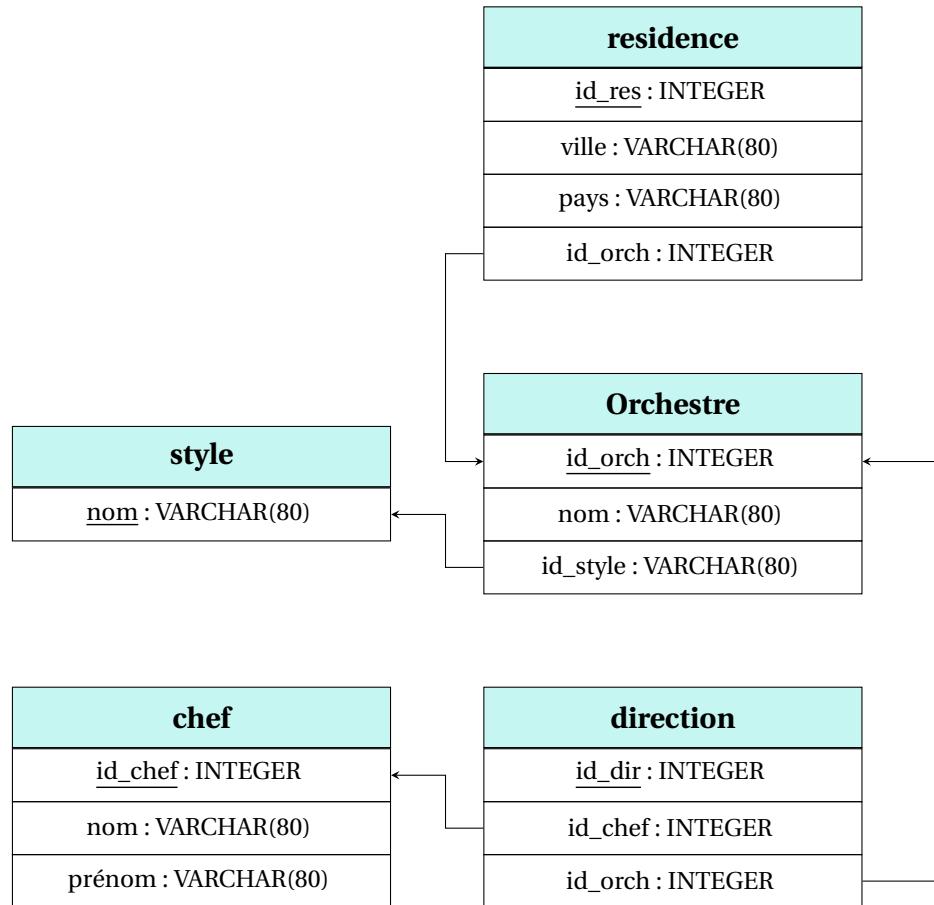


FIGURE 21.6 – Modèle physique de la base de données du modèle relationnel 21.2.

F Du modèle physique à la création des tables

→ HORS PROGRAMME

Il existe donc trois types de modèle pour décrire les base de données selon que l'on se place au niveau :

1. conceptuel,
2. relationnel (logique),
3. ou physique.

Sur la figure 21.6 est schématisé le modèle physique associé au modèle relationnel de la figure 21.2 et au modèle conceptuel de la figure 21.1.

Sur ce modèle physique sont notamment précisés les types des attributs en essayant de minimiser la taille en mémoire de chaque attribut.

Le langage SQL permet la création de ce modèle physique. Le code 21.1 montre comment créer les tables concrètes associées au modèle physique de la figure 21.6 dans la base de don-

nées. En plus des types des attributs y figurent également les contraintes de clefs primaires et étrangères.

Code 21.1 – Crédation du modèle physique d'une base de données

```

1  CREATE TABLE "chef"
2  (
3      "id_chef"  INTEGER,
4      "nom"      VARCHAR(80),
5      "prenom"    VARCHAR(80),
6      PRIMARY KEY ("id_chef")
7  );
8
9  CREATE TABLE "style"
10 (
11     "nom"      VARCHAR(80),
12     PRIMARY KEY ("nom")
13 );
14
15 CREATE TABLE "orchestre"
16 (
17     "id_orch"  INTEGER,
18     "nom"      VARCHAR(80),
19     "id_style" VARCHAR(80),
20     PRIMARY KEY ("id_orch"),
21     FOREIGN KEY ("id_style") REFERENCES "style"
22 );
23
24 CREATE TABLE "residence"
25 (
26     "id_res"   INTEGER,
27     "ville"    VARCHAR(80),
28     "pays"     VARCHAR(80),
29     "id_orch"  INTEGER,
30     PRIMARY KEY (id_res),
31     FOREIGN KEY ("id_orch") REFERENCES orchestre (id_orch)
32 );
33
34
35 CREATE TABLE "direction"
36 (
37     "id_dir"   INTEGER,
38     "id_chef"  INTEGER,
39     "id_orch"  INTEGER,
40     PRIMARY KEY ("id_dir"),
41     FOREIGN KEY ("id_chef") REFERENCES chef (id_chef),
42     FOREIGN KEY ("id_orch") REFERENCES orchestre (id_orch)
43 );

```

22

MODÈLE RELATIONNEL ET LANGAGE SQL

Did you really name your son ROBERT') ; DROP TABLE
students;--?

Exploits of a Mom, XKCD

À la fin de ce chapitre, je sais :

- ☒ interpréter et utiliser un modèle relationnel de base de données
- ☒ utiliser les opérateurs de projection et de sélection sur un modèle simple (select from, where)
- ☒ utiliser les clefs primaires et étrangères dans une requête simple
- ☒ opérer une jointure interne entre plusieurs tables (join on)
- ☒ utiliser les fonctions d'agrégation pour un calcul simple (min, max, sum, avg, count)
- ☒ filtrer des agrégations d'après un critère (having)
- ☒ utiliser des opérateurs ensemblistes (intersect, union, except)

A Requêtes SQL

SQL est le langage de manipulation et de création du modèle relationnel. Il implémente les opérations de l'algèbre relationnelle. De type déclaratif, ce langage décrit donc ce que l'on veut faire sur la base de données, pas comment on veut le faire car le SGBD s'en charge.

Les épreuves d'informatique commune proposent essentiellement de rédiger des requêtes SQL sur un base de données proposée. C'est un sous-ensemble de possibilité du langage qui permet déjà de se divertir un peu!

■ **Définition 157 — Requête SQL.** Une requête SQL consiste à extraire des données issues d'une base de données relationnelle en utilisant les opérateurs relationnels. Ces opérateurs sont :

- SELECT ...FROM
- WHERE
- GROUP BY
- HAVING
- ORDER BY

S'ajoutent à ces opérateurs, les opérations ensemblistes (union, intersection et différence) ainsi que l'opérateur de jointure (JOIN).

La structure générique d'une requête SQL est donnée sur le figure 22.1.

Code 22.1 – Structure de base d'une requête SQL : projection et sélection

```
1 — Commentaire
2 SELECT nom           — OPERATEUR attribut
3 FROM ville           — OPERATEUR table
4 WHERE nom_pays = "France"; — OPERATEUR condition
```

Dans tout le chapitre, on considère qu'on manipule le modèle relationnel donné sur la figure 22.1 via le langage SQL.

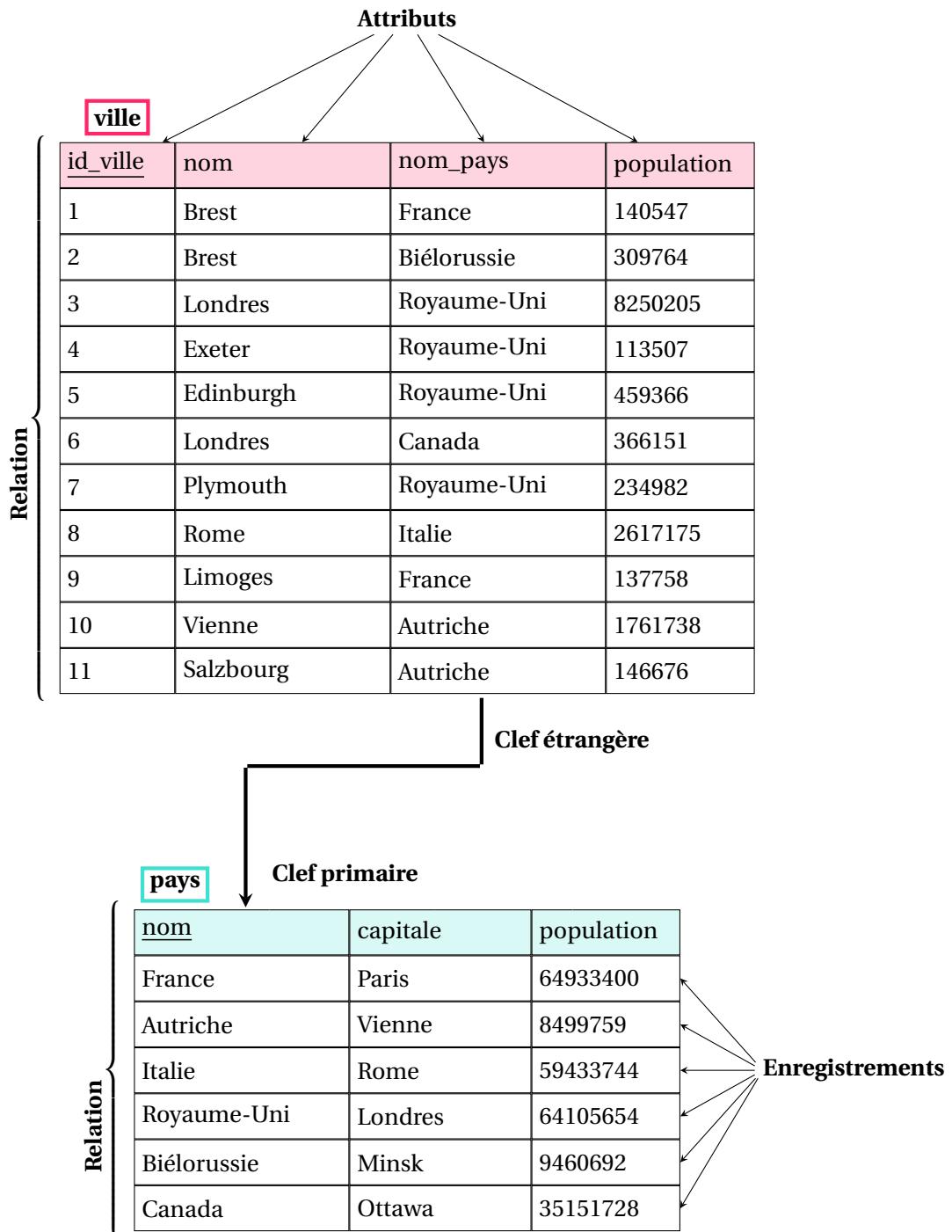


FIGURE 22.1 – Modèle relationnel utilisé pour les exemples de projection et de sélection.

B Projection : SELECT ...FROM

■ **Définition 158 — Projection.** Une projection est une opération qui consiste à sélectionner des colonnes (attributs) dans une table. En langage SQL, la projection est effectuée grâce aux mots clefs **SELECT** et **FROM**.

Code 22.2 – Exemples de projections

```

1  SELECT nom      — la colonne nom
2  FROM ville;    — de la table ville
3  — SORTIE DU SGBD :
4  — Brest
5  — Brest
6  — Londres
7  — Exeter
8  — Edinburgh
9  — Londres
10 — Plymouth
11 — Rome
12 — Limoges
13 — Vienne
14 — Salzbourg
15
16 SELECT *      — toutes les colonnes
17 FROM ville;  — de la table ville
18 — SORTIE DU SGBD :
19 — 1 Brest France 140547
20 — 2 Brest Bielorussie 309764
21 — 3 Londres Royaume-Uni 8250205
22 — 4 Exeter Royaume-Uni 113507
23 — 5 Edinburgh Royaume-Uni 459366
24 — 6 Londres Canada 366151
25 — 7 Plymouth Royaume-Uni 234982
26 — 8 Rome Italie 2617175
27 — 9 Limoges France 137758
28 — 10 Vienne Autriche 1761738
29 — 11 Salzbourg Autriche 146676
30
31 SELECT nom, nom_pays — la colonne nom et la colonne nom_pays
32 FROM ville; — de la table ville
33 — SORTIE DU SGBD :
34 — Brest France
35 — Brest Bielorussie
36 — Londres Royaume-Uni
37 — Exeter Royaume-Uni
38 — Edinburgh Royaume-Uni
39 — Londres Canada
40 — Plymouth Royaume-Uni
41 — Rome Italie
42 — Limoges France
43 — Vienne Autriche
44 — Salzbourg Autriche

```

C Sélection : WHERE ...

■ **Définition 159 — Sélection.** Une sélection est une opération qui consiste à sélectionner des lignes (enregistrements) dans une table. En langage SQL, la sélection est effectuée grâce au mot clef `WHERE` suivi d'une condition.

Code 22.3 – Sélections

```

1  SELECT nom      — la colonne nom
2  FROM ville      — de la table ville
3  WHERE nom_pays = "France";
4  — condition de selection : les lignes dont la colonne nom_pays vaut "France"
5  — SORTIE DU SGBD :
6  — Brest
7  — Limoges
8
9  SELECT nom      — la colonne nom
10 FROM ville      — de la table ville
11 WHERE nom_pays = "France" OR nom_pays = "Royaume-Uni";
12 — condition de selection : les lignes dont la colonne nom_pays vaut "France" ou du "
13 — "Royaume Uni"
14 — SORTIE DU SGBD :
15 — Brest
16 — Londres
17 — Exeter
18 — Edinburgh
19 — Plymouth
20 — Limoges

```

Une fois la projection et la sélection effectuée, on peut préciser la manière dont les résultats s'affichent : on peut ordonner les résultats selon un attribut en particulier, c'est la vocation des mots-clefs `ORDER BY`. L'ordre suivi par le SGBD sera l'ordre alphabétique ou numérique en fonction du type de données par rapport auquel on souhaite ordonner. Si on souhaite ordonner par un attribut de type texte, c'est l'ordre lexicographique qui est utilisé. Si on souhaite ordonner par un attribut du type entier, c'est l'ordre numérique qui sera choisi.

On peut également indiquer qu'on veut une projection sans redondance, c'est à dire ne garder qu'un seul exemplaire de chaque résultat (cf. code 22.4) grâce aux mots-clefs `SELECT DISTINCT`.

L'opérateur `LIMIT n` permet de limiter le nombre de réponses aux `n` premières. L'opérateur `OFFSET n` permet d'écarter les `n` premières réponses. En combinant les deux, on peut sélectionner n'importe quelle plage de la réponse.

Code 22.4 – Sélections - ordre des résultats - projection non redondante

```

1  SELECT nom      — la colonne nom
2  FROM ville      — de la table ville
3  WHERE nom_pays = "France" OR nom_pays = "Royaume-Uni"
4  ORDER BY nom_pays;
5  — condition de selection : colonne nom_pays vaut "France" ou du "Royaume Uni"
6  — ordonner par nom_pays
7  — SORTIE DU SGBD :
8  — Brest
9  — Londres
10 — Exeter
11 — Edinburgh
12 — Plymouth
13 — Limoges
14
15 SELECT DISTINCT nom_pays — la colonne nom_pays
16 FROM ville      — de la table ville
17 ORDER BY nom_pays;
18 — projection : les pays des villes sans redondance DISTINCT
19 — ordonner par nom_pays
20 — SORTIE DU SGBD :
21 — Autriche
22 — Bielorussie
23 — Canada
24 — France
25 — Italie
26 — Royaume-Uni
27
28 SELECT nom, population — la colonne population
29 FROM ville      — de la table ville
30 ORDER BY nom_pays — regrouper par pays
31 LIMIT 3;          — les trois premiers resultats
32 — SORTIE DU SGBD :
33 — Vienne 1761738
34 — Salzbourg 146676
35 — Brest 309764
36
37 SELECT nom, population
38 FROM pays
39 ORDER BY population
40 LIMIT 3          — les trois premiers resultats
41 OFFSET 2;        — on ecarte les 5 premieres
42 — SORTIE DU SGBD :
43 — Canada 35151728
44 — Italie 59433744
45 — Royaume-Uni 64105654

```

R Il est possible qu'il existe une ambiguïté dans le nom des attributs. Par exemple, la table *ville* possède un champ nom, tout comme la table *pays*. Parfois, cette ambiguïté n'a pas de conséquence, comme dans le code 22.10 : on a deux commandes `SELECT nom` qui désignent deux attributs différents, mais cela reste intelligible pour le SGBD, car ces commandes font parties de deux projections différentes, les `FROM` sont différents.

Cependant, il arrive que le SGBD ne puisse pas lever l'ambiguïté des noms utilisés. Dans ce cas, on peut soit :

- utiliser le préfixe du nom de la table suivi d'un point pour désigner un attribut. Par exemple, pour ne pas confondre la population d'une ville et la population d'un pays, on peut écrire les deux expressions `ville.population` et `pays.population`.
- renommer les tables, les attributs ou les champs calculés grâce à l'opérateur de renommage `AS` comme dans le code 22.5.

D Jointure : JOIN ...ON ...

Il est possible d'effectuer une requête SQL sur plusieurs tables simultanément, soit en utilisant le produit cartésien de tables soit en effectuant un jointure interne. Celle-ci est la syntaxe à privilégier pour cette opération.

■ **Définition 160 — Jointure.** En langage SQL, une jointure est un moyen d'utiliser plusieurs tables dans une même requête en créant une liaison entre les tables. Pour associer deux tables dans une jointure, il est nécessaire d'utiliser les mots-clefs `JOIN` et `ON`. Après `JOIN` on précise quelles tables sont à joindre, après `ON` on explicite la condition qui crée la liaison entre les tables.

■ **Exemple 63 — Exemples de jointures.** Le code 22.5 présente deux jointures de type différent.

La première jointure est une jointure entre deux tables différentes. La condition de jointure est que les lignes des deux tables qui sont à joindre doivent porter les mêmes noms de pays. La figure 22.2 détaille les étapes de cette requête.

La seconde est une autojointure, c'est à dire qu'on relie la table à elle même pour faire une requête. Dans cet exemple, on joint deux à deux les lignes des villes qui font parties d'un même pays, puis on sélectionne les lignes dont la première ville possède une population plus grande que la seconde.

Code 22.5 – Exemples de jointures

```

1  SELECT ville.nom, pays.population
2  FROM ville
3  JOIN pays
4  ON ville.nom_pays = pays.nom
5  WHERE pays.population < 60000000;
6  — Jointure de ville et pays d'apres les noms des pays
7  — Condition : les pays comportent moins de 60 millions d'habitants
8  — Sortie du SGBD
9  — Brest 9460692
10 — Londres 35151728
11 — Rome 59433744
12 — Vienne 8499759
13 — Salzbourg 8499759
14
15
16 SELECT v.nom, w.nom, v.population-w.population
17 FROM ville as v
18 JOIN ville as w
19 ON v.nom_pays = w.nom_pays
20 WHERE v.population > w.population;
21 — Projection avec renommage v
22 — Autojointure avec renommage w
23 — La condition de jointure est que le nom du pays des deux villes doit etre le meme.
24 — La selection ne retient que les couples de villes tels que la premiere ville est
   plus peuplee
25 — On calcule la difference de population entre les deux villes
26 — SORTIE DU SGBD :
27 — Exeter Edinburgh 345859
28 — Exeter Londres 8136698
29 — Exeter Plymouth 121475
30 — Edinburgh Londres 7790839
31 — Plymouth Edinburgh 224384
32 — Plymouth Londres 8015223
33 — Limoges Brest 2789
34 — Salzbourg Vienne 1615062

```

a

Étape 1 : jointure des tables ville et pays d'après les noms des pays.

<u>id_ville</u>	nom	nom_pays	population	<u>nom</u>	capitale	population
1	Brest	France	140547	France	Paris	64933400
2	Brest	Biélorussie	309764	Biélorussie	Minsk	9460692
3	Londres	Royaume-Uni	8250205	Royaume-Uni	Londres	64105654
4	Exeter	Royaume-Uni	113507	Royaume-Uni	Londres	64105654
5	Edinburgh	Royaume-Uni	459366	Royaume-Uni	Londres	64105654
6	Londres	Canada	366151	Canada	Ottawa	35151728
7	Plymouth	Royaume-Uni	234982	Royaume-Uni	Londres	64105654
8	Rome	Italie	2617175	Italie	Rome	59433744
9	Limoges	France	137758	France	Paris	64933400
10	Vienne	Autriche	1761738	Autriche	Vienne	8499759
11	Salzbourg	Autriche	146676	Autriche	Vienne	8499759

Étape 2 : sélection des pays de moins de 60 millions d'habitants.

<u>id_ville</u>	nom	nom_pays	population	<u>nom</u>	capitale	population
2	Brest	Biélorussie	309764	Biélorussie	Minsk	9460692
6	Londres	Canada	366151	Canada	Ottawa	3515172
8	Rome	Italie	2617175	Italie	Rome	59433744
10	Vienne	Autriche	1761738	Autriche	Vienne	8499759
11	Salzbourg	Autriche	146676	Autriche	Vienne	8499759

Étape 3 : projection du nom de la ville et de la population du pays.

nom	population
Brest	9460692
Londres	3515172
Rome	59433744
Vienne	8499759
Salzbourg	8499759

FIGURE 22.2 – Représentation des étapes de la requête `SELECT ville.nom, pays.population FROM ville JOIN pays ON ville.nom_pays = pays.nom WHERE pays.population < 60000000` ; du code 22.5 qui opère une jointure entre les tables ville et pays. Étape 1 : les lignes des deux tables sont mises côte à côte d'après le critère de jointure. Étape 2 : seules les lignes qui respecte la condition de sélection sont gardées. Étape 3 : la projection sélectionne les colonnes résultats. à

E Agréger, grouper et filtrer des résultats

a Fonctions d'agrégation : SUM, COUNT, AVG, MAX, MIN

■ **Définition 161 — Fonction d'agrégation.** Une fonction d'agrégation effectue des opérations statistiques sur un ensemble de registres. Elle s'applique à plusieurs lignes en même temps et permet :

- **COUNT** : de compter les éléments projetés et sélectionnés.
- **MIN** et **MAX** : de trouver le minimum ou le maximum des éléments,
- **SUM** : de calculer la somme des éléments,
- **AVG** : de calculer la moyenne des éléments.

R On peut utiliser les fonctions d'agrégation avec l'opérateur de projection **SELECT** ou l'opérateur **HAVING**. Par contre, **on n'utilise jamais** de fonctions d'agrégation avec l'opérateur de sélection **WHERE**. La raison est que pour pouvoir agréger des lignes, il faut d'abord sélectionner ces lignes des tables.



Le paragraphe précédent est important pour l'épreuve d'informatique!

Code 22.6 – Exemples d'utilisation des fonctions d'agrégation

```

1  SELECT COUNT(nom_pays)
2  FROM ville
3  WHERE population > 1000000;
4  — On compte les pays qui ont des villes de plus de 5 millions d'habitants.
5  — SORTIE DU SGBD :
6  — — 3
7
8  SELECT MAX(population), nom
9  FROM ville
10 WHERE nom_pays = "Royaume-Uni";
11 — On détermine le nom de la ville du Royaume-Uni qui possède le plus grand nombre d'
12 — habitants.
13 — SORTIE DU SGBD :
14 — — 8250205 Londres
15
16 SELECT MIN(nom)
17 FROM pays;
18 — On détermine le nom du pays le premier dans l'ordre lexicographique.
19 — SORTIE DU SGBD :
20 — — Autriche
21
22 SELECT AVG(population), MIN(population), MAX(population)
23 FROM ville;
24 — On détermine la moyenne, le minimum et le maximum de la population des villes.
25 — SORTIE DU SGBD :
26 — — 1321624.45454545 113507 8250205

```

b Regrouper : GROUP BY ...

L'opérateur **GROUP BY** permet de créer des groupes dans la réponse. L'intérêt principal est que les fonctions d'agrégation peuvent calculer leurs statistiques non plus sur l'ensemble des réponses mais sur chacun de ces groupes.

Code 22.7 – Exemple d'utilisation de GROUP BY

```

1  SELECT nom_pays, SUM(population)
2  FROM ville
3  WHERE population > 200000
4  GROUP BY nom_pays;
5  — On fait la somme des populations des villes de plus de 200000 habitants par pays.
6  — SORTIE DU SGBD :
7  — Autriche 1761738
8  — Bielorussie 309764
9  — anada 366151
10 — Italie 2617175
11 — Royaume-Uni 8944553

```

c Filtrer des résultats regroupés : GROUP BY ... HAVING...

SQL permet de filtrer les groupes créés par **GROUP BY**. En effet, l'opérateur **WHERE** a déjà sélectionné les lignes, les groupes ont été créés à partir de ces lignes. Si on souhaite filtrer les groupes, il faut donc un autre opérateur. C'est le rôle de **HAVING**.

Code 22.8 – Exemple d'utilisation de GROUP BY ... HAVING

```

1  SELECT nom_pays, SUM(population)
2  FROM ville
3  WHERE population > 200000
4  GROUP BY nom_pays
5  HAVING COUNT(nom) > 2;
6  — On fait la somme des populations des villes de plus de 200000 habitants par pays
7  — si seulement le pays en possède au moins 3.
8  — SORTIE DU SGBD :
9  — Royaume-Uni 8944553

```

F Opérations ensemblistes

SQL définit également des opérateurs ensemblistes : un opérateur d'union, d'intersection et de différence. Ils sont principalement utiles lorsqu'on dispose de plusieurs tables cohérentes, voire qui représente la même entité, et qu'on souhaite extraire des informations de ces deux tables simultanément. Il est nécessaire que les projections génèrent le même nombre de colonnes pour pouvoir mener une opération ensembliste. Les opérateurs sont **UNION**, **INTER** et **EXCEPT**.

Code 22.9 – Exemple d'opération ensembliste

```
1  SELECT nom, population
2  FROM ville
3  WHERE nom = "Londres"
4  UNION
5  SELECT nom, population
6  FROM ville
7  WHERE population < 200000;
8  —— SORTIE DU SGBD :
9  —— Brest 140547
10 —— Exeter 113507
11 —— Limoges 137758
12 —— Londres 366151
13 —— Londres 8250205
14 —— Salzbourg 146676
```

G Requêtes imbriquées → HORS PROGRAMME

- **Définition 162 — Sous-requête ou requête imbriquée.** Une requête imbriquée est une requête incluse dans une requête. Son résultat devient le champ de recherche d'une autre requête.



Vocabulary 22 — Nested request ↗ Requête imbriquée

Comme le montre le code 22.10, il est possible d'utiliser les opérateurs de comparaison `=`, `<`, `<=`, `>`, `>=` pour créer une condition qui dépend de la requête imbriquée, si celle-ci ne renvoie qu'une seule valeur. Mais ce n'est pas toujours le cas. Dans le cas contraire, la sous-requête résulte en une liste de valeur, alors il est nécessaire d'utiliser les mots-clefs `IN` ou `NOT IN` pour tester l'appartenance du critère à la liste de valeurs retournée par la requête imbriquée.

Code 22.10 – Requêtes imbriquées

```

16   — Requete imbriquee : la liste des noms des pays qui comportent moins de 60 millions d
     'habitants
17   — Sortie du SGBD
18   — Brest
19   — Londres
20   — Rome
21   — Vienne
22   — Salzbourg

```

H De belles requêtes SQL

Comme vous l'aurez observé tout au long de ce chapitre, on n'écrit pas n'importe comment les requêtes SQL. Les codes qui vont sont présentés respectent certaines conventions.

M Méthode 13 — Écriture de requêtes SQL et conventions Afin de garantir une bonne lisibilité des requêtes SQL, il est nécessaire de bien les écrire. Même s'il ne s'agit de convention, il est important :

- d'écrire en majuscules pour les opérateurs SQL,
- d'écrire en minuscules les relations (tables) et les attributs (champs),
- de revenir à la ligne après chaque opération : projection, sélection, regroupement, agrégation, ordonnancement.

 Le paragraphe précédent est important pour l'épreuve d'informatique!

Opérateurs	Action
<code>SELECT ... FROM ...</code>	Projection des colonnes d'une table
<code>SELECT DISTINCT ... FROM ...</code>	Idem mais sans redondance, sans doublons
<code>WHERE ...</code>	Condition de sélection des lignes
<code>GROUP BY ...</code>	Créer des regroupements des résultats
<code>HAVING ...</code>	Filtrer les regroupements de résultats
<code>ORDER BY ... ASC/DESC</code>	Ordonner les résultats
<code>LIMIT n</code>	Limiter le nombre de résultats aux n premiers
<code>OFFSET n</code>	Écarter les n premiers résultats
<code>UNION, INTERSECT, EXCEPT</code>	Opérations ensemblistes

TABLE 22.1 – Synthèse des opérateurs SQL au programme

23

ET LA MACHINE APPRIT

L'intelligence artificielle n'existe pas.

Luc Julia [15]

À la fin de ce chapitre, je sais :

- ☒ définir les termes intelligence artificielle et apprentissage automatique
- ☒ expliquer et utiliser l'algorithme d'apprentissage supervisé des k plus proches voisins (KNN)
- ☒ expliquer et utiliser l'algorithme d'apprentissage non supervisé des k moyennes (Kmeans)

A Principles

■ **Définition 163 — Intelligence artificielle.** L'intelligence artificielle est un mot valise qui désigne des systèmes électroniques et informatiques capables de percevoir, de comprendre, d'apprendre et d'agir pour atteindre un but précis.

Cette définition pose de nombreux problèmes car elle repose sur le concept d'intelligence qu'il est bien difficile de définir. D'après le Larousse, l'intelligence est un *ensemble des fonctions mentales ayant pour objet la connaissance conceptuelle et rationnelle*. La machine aurait-elle des fonctions mentales? C'est à dire des fonctions cognitives, exécutives et langagières? De nombreuses personnes [15] insiste sur l'absurdité du mot intelligence artificielle qui a fait naître de nombreux fantasmes depuis son invention en 1955 par John McCarthy[18]. C'est pourquoi, dans ce chapitre, on s'intéressera davantage aux algorithmes d'**apprentissage** automatique, terme plus précis, plus pragmatique, qui pose moins de problèmes métaphysiques.

**Vocabulary 23 — Machine Learning** ⇔ Apprentissage automatique

■ **Définition 164 — Apprentissage automatique.** L'apprentissage automatique est un ensemble de techniques qui ont pour but d'automatiser les prises de décision en créant des systèmes capables, à partir d'un jeu de données, de découvrir, d'identifier et de classifier des motifs récurrents. Ces systèmes peuvent en plus être capables d'améliorer leurs performances au fur et à mesure de leur utilisation.

Les données d'entrée peuvent être de nature très différente : langage, nombres, images, sons... Les motifs récurrents identifiés par la machine ne sont généralement pas perceptibles à l'être humain ce qui donne à ces techniques une apparence de boîte noire. Cependant ces motifs identifiés le sont grâce à une analyse rationnelle du problème de décision considéré : rien d'ésotérique là dedans.

L'apprentissage automatique est considéré comme un sous-ensemble de l'intelligence artificielle.

Les données jouent un rôle central dans l'apprentissage automatique. Avant de se lancer dans l'apprentissage automatique, il est nécessaire d'extraire, de sélectionner et de transformer les données pour constituer un jeu de données utilisable. C'est ce jeu de données qui constitue l'entrée d'un algorithme.

Les données peuvent être **étiquetées** : on peut indiquer ainsi au système les caractéristiques qu'il va devoir apprendre à identifier. Elles peuvent aussi être **non étiquetées** et le système devra alors repérer et extraire les motifs récurrents sans indications.

Il existe deux grands objectifs à l'apprentissage automatique : la régression et la classification. Parfois un même algorithme peut permettre d'effectuer une régression ou une classification.

■ **Définition 165 — Régression.** Un algorithme dont la sortie est une régression génère un résultat sous la forme d'une valeur continue. Par exemple un prix, une température ou le cours d'une action.

■ **Définition 166 — Classification.** Un algorithme dont la sortie est une classification génère un résultat sous la forme d'une valeur discrète qui permet de trier un ensemble de données selon plusieurs catégories. Par exemple un masculin ou féminin, vrai ou faux, spam ou pas spam, carotte, navet ou pomme de terre si on considère des légumes à trier, pierre, sable, végétal ou eau si l'on considère la nature d'un sol à identifier.

On distingue plusieurs types d'algorithmes d'apprentissage automatique : l'apprentissage supervisé, non supervisé et l'apprentissage par renforcement. Seules deux approches sont au programme : une supervisée (KNN), l'autre non supervisée (K-means).

■ **Définition 167 — Apprentissage supervisé.** Un algorithme d'apprentissage supervisé s'appuie sur un jeu de données étiquetées pour s'entraîner à produire des résultats corrects. Au fur et à mesure que les données sont lues, il ajuste les poids du modèle afin de coller au mieux aux étiquettes (phase d'entraînement). On réserve généralement une partie du jeu

de données pour vérifier la validité du modèle ainsi obtenu (phase de test).

■ **Définition 168 — Matrice de confusion.** La matrice de confusion est un outil qui permet de visualiser les performances d'un algorithme de classification. Chaque case contient le nombre *a* de cas d'une classe *donnée* qui a été **prédit** par l'algorithme comme appartenant à une certaine classe.

La matrice de confusion est donc organisée comme suit :

- les lignes représentent les classes données (par le jeu de données),
- les colonnes les classes estimées par l'algorithme.

Les résultats peuvent être classés en quatre catégories :

- les vrais positifs : la prédiction et la classe donnée coïncident et c'est vrai.
- les vrais négatifs : la prédiction et la classe donnée ne coïncident pas et c'est vrai.
- les faux positifs : la prédiction et la classe donnée coïncident et c'est faux,
- les faux négatifs : la prédiction et la classe donnée ne coïncident pas et c'est faux.

a. ou le pourcentage

■ **Exemple 64 — Matrice de confusion.** On considère un algorithme de traitement automatique des messages instantanés : on cherche à classer les messages en différentes catégories (peur (P), joie (J), amour (A), tristesse (T), violence (V)) en fonction de leur contenu. On dispose d'un jeu de données étiquetées dont on utilise une partie pour l'apprentissage de l'algorithme et une partie pour le test de l'algorithme *a*. La matrice ci-dessous représente les résultats obtenus par l'algorithme sur le jeu de données de test.

Donnée	P	J	A	T	V	
	55	6	4	30	5	
J	10	75	2	11	2	
A	3	1	91	5	1	
T	25	15	1	58	1	
V	4	1	2	0	93	
Prédiction						

On peut lire la première ligne de la matrice ainsi : sur la centaine de messages de test étiquetés P (peur), 55 ont été effectivement prédisits comme tels par l'algorithme. Mais, 6 ont été prédisits joie, 4 amour, 30 tristesse et 5 violence. Notre algorithme engendre donc une

certaine confusion entre la peur et la tristesse ^b, confusion qui est explicite lorsqu'on affiche cette matrice! Lorsqu'un algorithme fonctionne correctement, les valeurs sur la diagonale sont maximales et les éléments non diagonaux mimimaux.

- a. Généralement on sélectionne 70% des données pour l'apprentissage et 30% pour le test.
- b. à moins que ce ne soit notre jeu de données et les classes choisies qui ne sont pas pertinentes!

R Il existe de nombreux algorithmes pour la régression et/ou la classification en apprentissage automatique. Le TP associé à ce cours est l'occasion d'illustrer les performances de plusieurs algorithmes confrontés à des jeux de données différents. Le choix d'un algorithme plutôt qu'un autre en apprentissage automatique est un compromis issu de l'expérience.

Parmi les algorithmes les plus utilisés on note :

1. l'analyse en composantes principales (Principal Component Analysis) pour réduire les dimensions du jeu de données,
2. les k-moyennes et les k plus proches voisins,
3. les arbres de décisions (Decision Trees),
4. les séparateurs à vaste marge (Support Machine Vector),
5. les réseaux de neurones (ANN),
6. les réseaux de neurones convolutionnels (CNN) pour l'apprentissage profond.

R La visualisation des caractéristiques d'un jeu de données et des résultats de l'apprentissage est un élément déterminant pour l'analyse des performances des algorithmes. Un exemple de visualisation des données brutes produit par les bibliothèques Scikit-Learn et Seaborn Python est donné sur la figure 23.1 : il s'agit des paramètres d'un jeu de données sur les manchots. Un exemple de projection des résultats de l'algorithme KNN sur le jeu de données iris selon toutes les dimensions est donné sur la figure 23.2.

B Apprentissage supervisé : k plus proches voisins

 **Vocabulary 24 — K-Nearest Neighbours (KNN) Algorithm** ⇔ Algorithme des k plus proches voisins

L'algorithme des k plus proches voisins [9] est un algorithme relativement simple, supervisé et encore très utilisé aujourd'hui pour effectuer des régressions ou des classifications. Il ne présente pas à proprement parlé de phase d'entraînement qui est confondue avec la phase de prédiction : il utilise toutes les données à chaque calcul¹. Plus le jeu de données grandit, plus KNN devient inefficace². Il est cependant couramment utilisé pour les systèmes de recommandation simples, la reconnaissance de modèle, la fouille de données, les prévisions des

-
- 1. On dit que l'algorithme est paresseux.
 - 2. Ce problème est souvent désigné en IA par la malédiction de la dimension d'un problème.

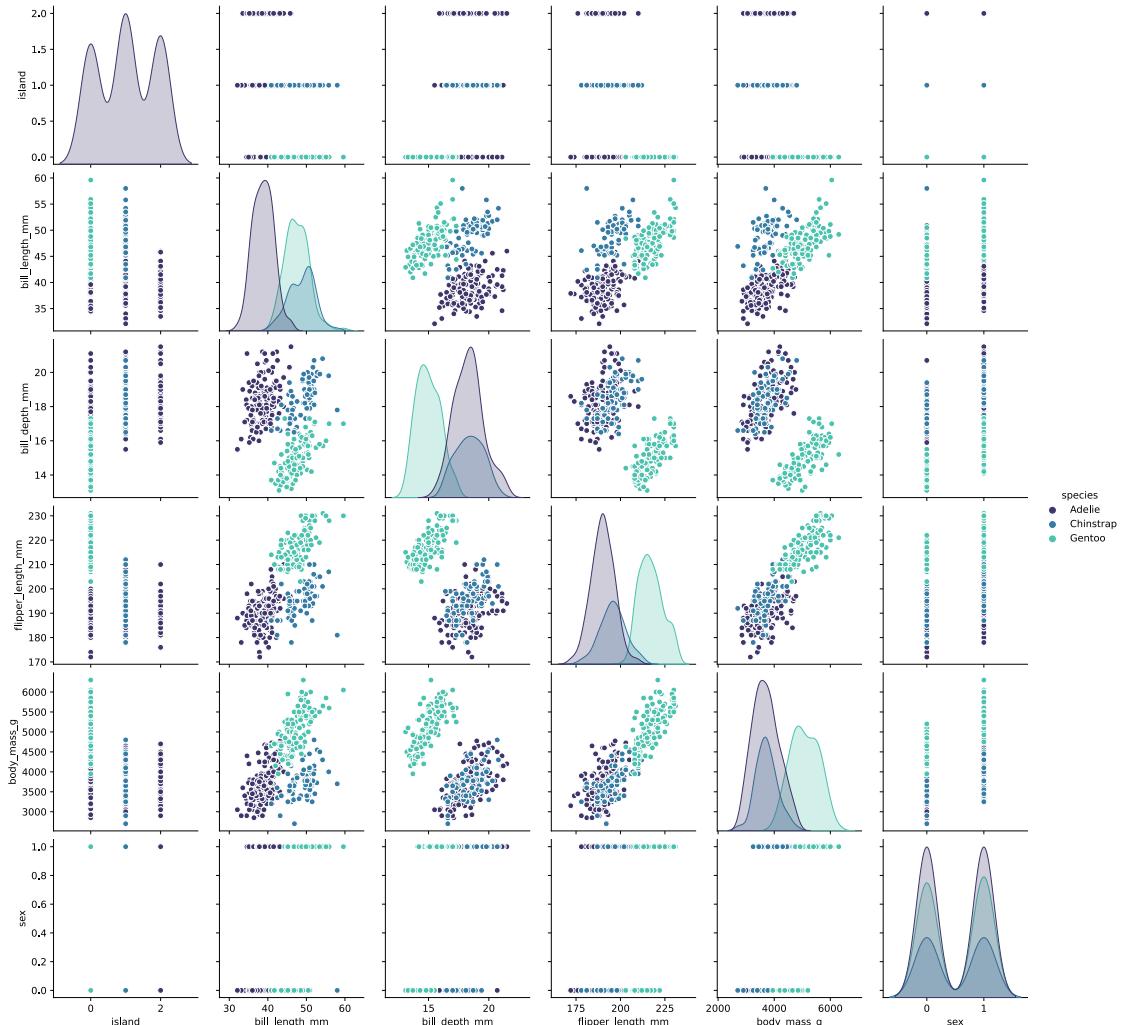


FIGURE 23.1 – Visualisation de la dispersion des paramètres sur le jeu de données des manchots répartis sur trois îles et en trois espèces

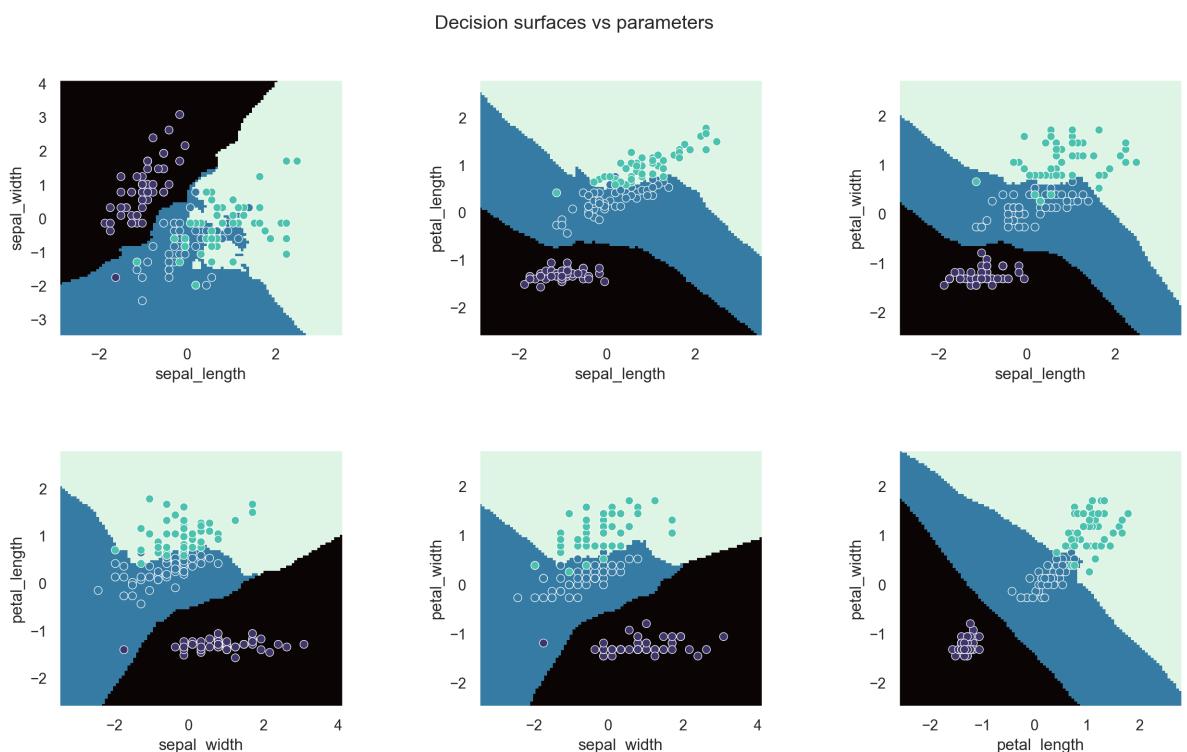


FIGURE 23.2 – Résultats de l'apprentissage de l'algorithme KNN sur le jeu de données de l'iris. Les régions de décisions du classificateur sont superposées au valeurs vraies des espèces d'iris.

marchés financiers ou la détection d'intrusion.

Pour les problèmes de classification, l'étiquette de classe est affectée à l'échantillon analysé sur la base d'un vote à la majorité : on utilise l'étiquette de la classe la plus fréquemment représentée autour de l'échantillon. Pour les problèmes de régression et une certaine fonction f à valeur continue, on procède de la même manière mais on calcule la prédiction sur la valeur de la moyenne de f sur les k plus proches voisins.

Pour expliquer cet algorithme simplement, on considère un problème de classification dans \mathbb{R}^d . C'est à dire qu'on dispose d'un ensemble d'échantillons étiquetés \mathcal{E} dont les éléments sont des couples composés d'un vecteur de \mathbb{R}^d et d'une étiquette désignant la classe à laquelle appartient l'échantillon. On note l'ensemble des classes possibles \mathcal{C} . On cherche donc à déterminer la classe d'autres échantillons non étiquetés.

a Principe de KNN pour la classification

Pour un échantillon $x \in \mathbb{R}^d$, l'algorithme KNN détermine dans un premier temps les k voisins les plus proches dans les données étiquetées \mathcal{E} . Puis, dans un deuxième temps, il décide de la classe de l'échantillon de e à partir d'un vote majoritaire : il choisit l'étiquette de la classe la plus fréquemment représentée autour de l'échantillon.

b Distances

Pour savoir si un voisin est proche, il faut être capable de mesurer sa distance. L'algorithme KNN utilise donc une distance dont la définition varie selon le problème considéré : distance euclidienne, de manhattan, de levenshtein ou de hamming...

c Comment choisir k ?

Si n est la dimension du problème, c'est à dire que le jeu de données est constitué de n échantillons, alors il faut nécessairement choisir k dans $\llbracket 1, n \rrbracket$. Choisir un k trop grand amènera à classer la majorité des échantillons dans la classe dominante. Choisir un k trop petit ne permettra pas de conclure précisément.

d Algorithmes des n plus proches voisins

On note $\mathcal{E} = \{(e_i, c_i), i \in \llbracket 1, n \rrbracket, e_i \in \mathbb{R}^d, c_i \in \mathcal{C}\}$, l'ensemble des données étiquetées dont on dispose. On cherche à trouver la classe de $e \in \mathbb{R}^d$. On dispose d'une distance δ sur \mathbb{R}^d .

La figure 23.3 illustre le résultat de l'algorithme 60 sur le très célèbre jeu de données de l'iris [8]. Les résultats montrent clairement que l'algorithme est capable de discriminer plusieurs variétés d'iris en connaissant uniquement les paramètres morphologiques des pétales et des sépals. On remarque essentiellement une légère confusion entre Versicolor et Virginica. Certaines Virginica sont prises pour des Versicolor. Cet apprentissage est effectué en TP.

Les étapes de l'algorithme 60 des lignes 4 à 6 nécessitent un tri de l'ensemble Δ . On peut imaginer effectuer un tri en ligne au fur et à mesure par insertion, un tri fusion ou utiliser un tas min.

Algorithme 60 k plus proches voisins (KNN)

```

1: Fonction KNN( $D, x, k, \delta$ )
2:    $n \leftarrow |D|$ 
3:    $\Delta \leftarrow \emptyset$                                  $\triangleright$  Distances à calculer
4:   pour chaque échantillon étiqueté  $e \in \mathcal{E}$  répéter
5:     Ajouter  $\delta(x, e)$  à  $\Delta$ 
6:   Sélectionner les  $k$  voisins les plus proches de  $x$  en utilisant  $\Delta$ 
7:   Compter le nombre d'occurrences de chaque classe des  $k$  voisins de  $x$ 
8:   renvoyer la classe  $c$  la plus représentée parmi les  $k$  plus proches voisins

```

Pour détecter la classe majoritaire, il est possible d'utiliser les fonctionnalités de base de Python3, Numpy ou l'algorithme de Boyer-Moore.

En cas d'égalité, plusieurs solutions sont possibles : prendre un voisin de plus ou bien tirer au sort la classe renvoyée. Certaines variantes de l'algorithme pondèrent le vote majoritaire à l'aide de la distance.

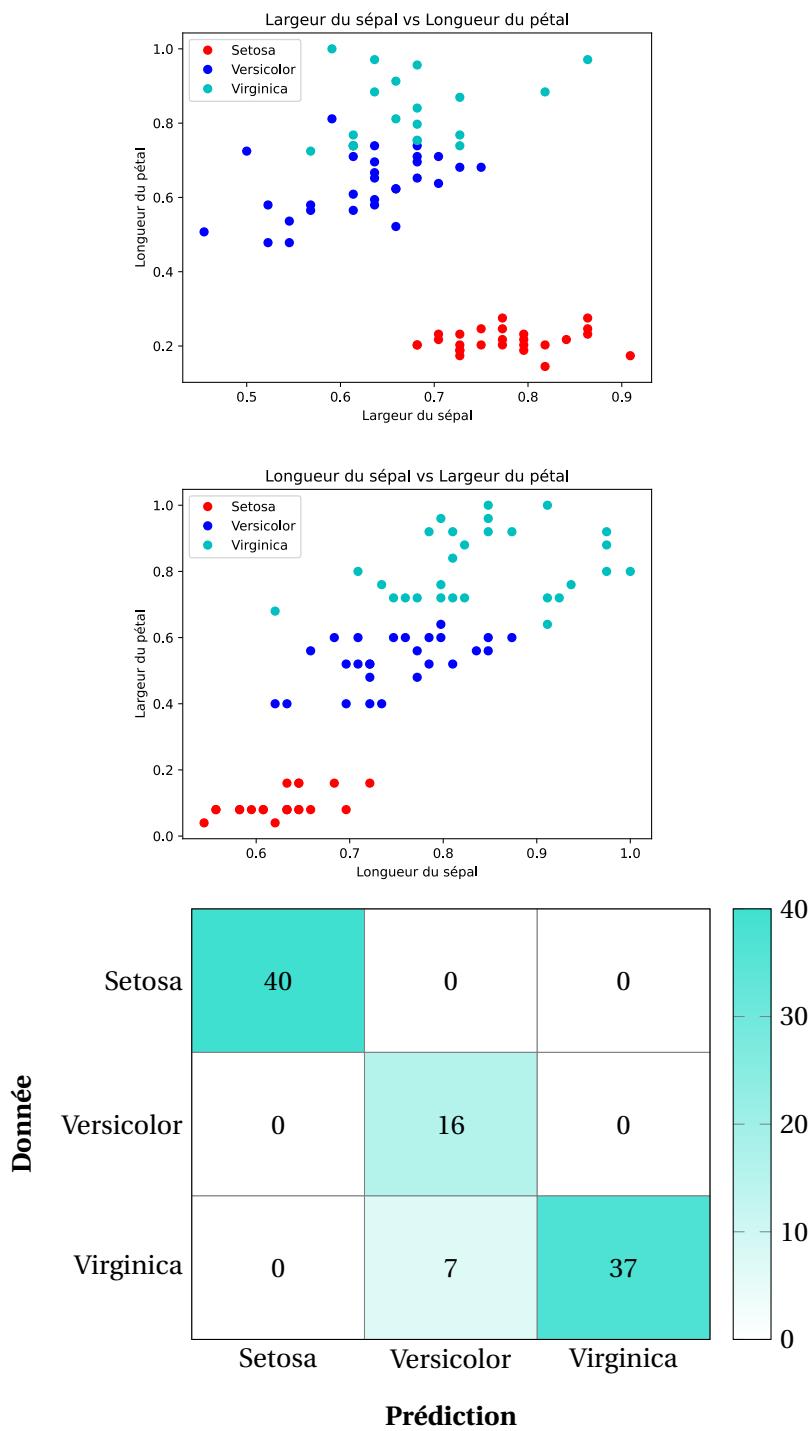


FIGURE 23.3 – Illustration de l'algorithme KNN appliqué à la distinction de trois variétés d'iris : Setosa, Versicolor et Virginica. Les paramètres d'entrées sont les largeurs et longueurs des pétales et sépals. Les figures représentent certains paramètres du jeu de données l'un en fonction de l'autre. Le résultat de l'algorithme se lit au travers des couleurs. Le résultat est correct dans plus de 90% des cas comme le montre la matrice de confusion.

C Apprentissage non supervisé : k moyennes



Vocabulary 25 — k-means \rightsquigarrow Algorithmes des k-moyennes

On ne dispose pas toujours d'un jeu de données étiquetées, c'est à dire des échantillons dont on connaît la classe. C'est pourquoi certaines algorithmes d'apprentissage automatique développent une approche non supervisée. L'absence d'échantillons de vérification fait qu'il est parfois plus difficile d'évaluer la performance de ces algorithmes.

On considère de nouveau un problème de **classification**. On suppose qu'on dispose d'un ensemble d'échantillons $\mathcal{E} = \{e_i, i \in \llbracket 1, n \rrbracket, e_i \in \mathbb{R}^d\}$. Il s'agit de créer une partition de \mathcal{E} selon k classes.

Algorithme 61 k moyennes (k-means)

```

1: Fonction KMEANS( $\mathcal{E}, k, \delta$ )
2:    $\mathcal{P} \leftarrow \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_k\}$  une partition quelconque de  $\mathcal{E}$  en  $k$  classes
3:   tant que des échantillons changent de partition répéter
4:      $(b_1, b_2, \dots, b_k) \leftarrow \text{BARYCENTRE}(\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_k)$ 
5:     pour chaque échantillon  $e$  de  $\mathcal{E}$  répéter
6:       Trouver la partition  $\mathcal{P}_i$  la plus proche de  $e$ ,  $\|e - b_i\|^2$  est minimale sur  $\mathcal{P}$ 
7:       Ajouter  $e$  à la partition  $\mathcal{P}_i$  trouvée
8:   renvoyer  $(\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_k)$ 

```

La cœur de cet algorithme est la construction des partitions de l'ensemble du jeu de données. L'algorithme crée donc des catégories, autant qu'on lui en demande. Si l'utilisateur n'a pas d'idée précise du nombre de catégories à créer, il est nécessaire de tâtonner. Par ailleurs, le lien sémantique entre les données et le résultat est à la charge de l'utilisateur de l'algorithme : K-means se contente de créer des partitions, il ne leur attribue aucun sens particulier. On est loin de l'intelligence artificielle qui fait fantasmer les foules!



L'algorithme des k-moyennes minimise la coût défini par la fonction :

$$f(\mathcal{E}, k) = \sum_{i=1}^k \sum_{x \in \mathcal{P}_i} \|x - m_i\|^2 \quad (23.1)$$

L'algorithme consiste à minimiser la variance des partitions (clusters). La fonction BARYCENTRE utilise donc la norme **euclidienne**. D'autres algorithmes comme k-médiannes ou k-médiodes existe et permettent d'utiliser des distances.



La condition de convergence (ligne 3 de l'algorithme 61) doit être adaptée selon la nature des données.

R → HORS PROGRAMME On peut démontrer que cet algorithme termine en considérant la fonction de coût $f(\mathcal{E}, k) = \sum_{i=1}^k \sum_{x \in \mathcal{P}_i} \|x - m_i\|^2$.

Comme il y a k^n façons de répartir n points en k partitions, l'ensemble des partitions possible est un ensemble fini. L'algorithme itère sur cet ensemble fini. S'il ne termine pas, c'est donc qu'il opère un cycle dans cet ensemble fini. Si c'était le cas, comme la fonction f est strictement décroissante d'après notre choix des partitions à chaque itération, cela signifierait qu'une même partition pourrait présenter un coût plus faible que son propre coût, ce qui est absurde. Donc, l'algorithme n'itère pas sur un cycle et nécessairement termine de parcourir l'ensemble des partitions en minimisant la fonction de coût.

24

THÉORIE DES JEUX

Si deux ont proposé entre eux, de dire chacun l'un après l'autre alternativement un nombre à plaisir, qui toutefois ne surpasse pas un certain nombre précis, pour voir ajoutant ensemble les nombres qu'ils diront qui arrivera plutôt à quelque nombre prescrit; faire si bien qu'on arrive toujours le premier au nombre destiné.

Claude-Gaspar Bachet de Méziriac, 1612 [2]

À la fin de ce chapitre, je sais :

- expliquer l'intérêt de la théorie des jeux
- expliquer le concept de jeu d'accessibilité
- coder le calcul des attracteurs
- expliquer la notion d'heuristique
- appliquer les algorithmes A* et minimax

La théorie des jeux a été initié par John Von Neumann pendant et après la seconde guerre mondiale. Dans un ouvrage resté célèbre[24], de nombreux problèmes très généraux sont abordés sous la perspective du jeu et de l'économie. Tout comme les algorithmes d'IA développés aujourd'hui, cette théorie s'inscrit dans l'objectif d'aider à la décision lorsque l'environnement est incertain, c'est-à-dire complexe et imprévisible. Elle fait intervenir des joueurs considérés comme des individus rationnels, des règles et des contextes d'évolution du jeu.

Le programme de classe préparatoire n'aborde que les jeux d'accessibilité à deux joueurs que l'on peut modéliser avec un graphe orienté biparti. Néanmoins, cela permet de lever le voile sur une théorie puissante et fascinante.

A Introduction à la théorie des jeux

■ **Définition 169 — Jeu.** Dans le cadre de cette théorie, on considère qu'un jeu est une activité humaine définie dans le cadre d'un contexte et dont les participants doivent suivre les règles énoncées et faire des choix pour gagner en s'opposant ou résoudre un problème ensemble. Cette activité nécessite des compétences intellectuelles, des savoirs et incorpore le hasard.

Les jeux ainsi définis englobent donc la plupart des activités humaines : l'économie, la guerre, l'étude du vivant ou même la physique peuvent être le cadre de jeux qui servent alors de modèles pour découvrir, établir des stratégies ou simuler une réalité.

■ **Définition 170 — Jeux coopératifs.** Un jeu coopératif permet la construction de coalitions entre joueurs. Cela suppose une concertation sur la stratégie à adopter et un engagement à coopérer.

■ **Définition 171 — Jeu à somme nulle.** Les jeux à somme nulle sont des jeux à deux joueurs pour lesquels les gains de l'un sont strictement les pertes de l'autre. Si on utilise une fonction de gain pour évaluer les perspectives de gain de chaque joueur, alors la somme des deux fonctions de gain est nulle.

■ **Exemple 65 — Jouer à somme nulle.** Parmi les jeux les plus connus à somme nulle, on trouve :

- les échecs,
- les jeux de carte comme la belote, le tarot ou le poker,
- shi-fu-mi.

R La plupart des situations de la vie quotidienne engendre des jeux à somme non nulle. Par exemple, le commerce est un jeu à somme non nulle plutôt positive : un marchand de voiture est gagnant lors qu'il vend une voiture à un client et son gain n'est pas égal à l'inverse du crédit qu'a souscrit l'acheteur... Néanmoins, cela ne signifie pas qu'ils ont perdu pour autant, les situations commerciales peuvent être gagnant-gagnant : si vous avez faim, vous serez content d'acheter de la nourriture qu'un marchand voudra bien vous vendre.

■ **Définition 172 — Dilemme du prisonnier.** Le dilemme du prisonnier est un exemple fondamental ^a de la théorie des jeux. Il a été formalisé par Tucker en 1950 [22] pour pointer une insuffisance de la théorie des jeux de l'époque : deux individus rationnels ne coopèrent pas nécessairement ^b. Le principe est le suivant :

Deux membres d'un même gang criminel sont arrêtés et emprisonnés. Chaque prisonnier est mis à l'isolement : il ne peut pas communiquer avec l'autre. La police ne dispose pas de suffisamment de preuves pour les accuser formellement tous les deux et il est envisagé

de les condamner à un an de prison tous les deux pour des charges moindre. Pour l'instant les deux prisonniers gardent le silence.

Néanmoins, la police propose à chaque prisonnier A et B un marché diabolique. En voici les termes :

1. Si A et B se dénoncent mutuellement, il seront condamnés à deux ans de prisons.
 2. Si A trahit B et que B demeure silencieux, A sera libéré et B sera condamné à trois ans.
 3. Symétriquement, si A demeure silencieux et que B le dénonce, alors A fera trois ans et B sera libéré.
 4. Enfin, si A et B demeurent silencieux, les deux feront un an de prison.
-
- a. un paradigme
 - b. On trouve ici [1] une fabuleuse introduction à ce dilemme dans la série Voyages au pays des maths d'Arte.
- À regarder absolument!

R Le dilemme du prisonnier illustre bien des situations (guerre commerciale par exemple) dans lesquelles les acteurs peuvent agir rationnellement, ne pas coopérer spontanément et perdre simultanément. L'incitation à tricher est naturellement au cœur du dilemme.

La répétition du jeu peut cependant amener à considérer d'autres stratégies : chaque joueur peut adapter son comportement par rapport à l'expérience passée et choisir de coopérer ou au contraire de se venger. Lorsque l'incitation à tricher est moins forte que les représailles potentielles, la coopération peut alors s'imposer et le jeu peut atteindre un équilibre de Nash.

■ **Définition 173 — Jeu séquentiel.** Un jeu séquentiel est un jeu au cours duquel les joueurs décident de leur stratégie les uns après les autres et peuvent donc tenir compte des actions des joueurs précédents.

■ **Définition 174 — Jeu à information parfaite.** Un jeu est à information parfaite si chaque joueur est parfaitement informé des actions passées des autres joueurs avant de prendre sa décision : aucune action du jeu n'a été cachée. On se rappelle de tous les coups joués précédemment. Un jeu à information parfaite est un jeu séquentiel.

■ **Définition 175 — Jeu à information complète.** Un jeu à information est à information complète si tous les joueurs ont une connaissance totale des données du jeu : règles, pièces, actions possibles, fonction de gain, objectifs des autres joueurs.

R Les jeux à information incomplète sont appelés jeux bayésiens. Dans ce cas, les joueurs n'ont pas une connaissance commune du jeu : chacun n'a qu'une vision partielle des données du jeu.

■ **Exemple 66 — Jouer à information (in)complète et (im)parfaite.** Aux échecs, s'il s'agit

d'une partie d'échec classique, les joueurs évoluent dans un contexte d'information parfaite : chaque joueur a pu voir tous les coups joués précédemment au cours de la partie. De plus, les règles sont connues, toutes les pièces sont toutes visibles, le chronomètre aussi : alors l'information est complète également. Par contre, si une partie est pris en cours de route et que le joueur n'a pas connaissance des coups passés, l'information est imparfaite.

Les jeux de cartes comme le bridge ou le poker sont des jeux à information imparfaite la distribution est inconnue (aléatoire et personne n'en a connaissance puisque les cartes sont retournées lors de la distribution) et incomplète car on ne connaît pas la main des adversaires lors du jeu.

Les aventuriers du rail est un exemple de jeu de plateau à information incomplète car on ne connaît pas les objectifs des autres joueurs mais parfaite car toutes les actions passées sont connues.

■ **Définition 176 — Arbre de jeu ou forme extensive.** La représentation d'un jeu séquentiel sous la forme d'un arbre est appelée forme extensive ou arbre de jeu. Les nœuds représentent les positions du jeu. Les nœuds d'un même niveau sont contrôlés par un même joueur.

Un exemple d'arbre de jeu pour une partie de morpion est donné sur la figure 24.1.

La représentation arborescente est fondamentale pour la plupart des raisonnements sur les jeux et en général pour l'exploration d'un ensemble de possibilités.

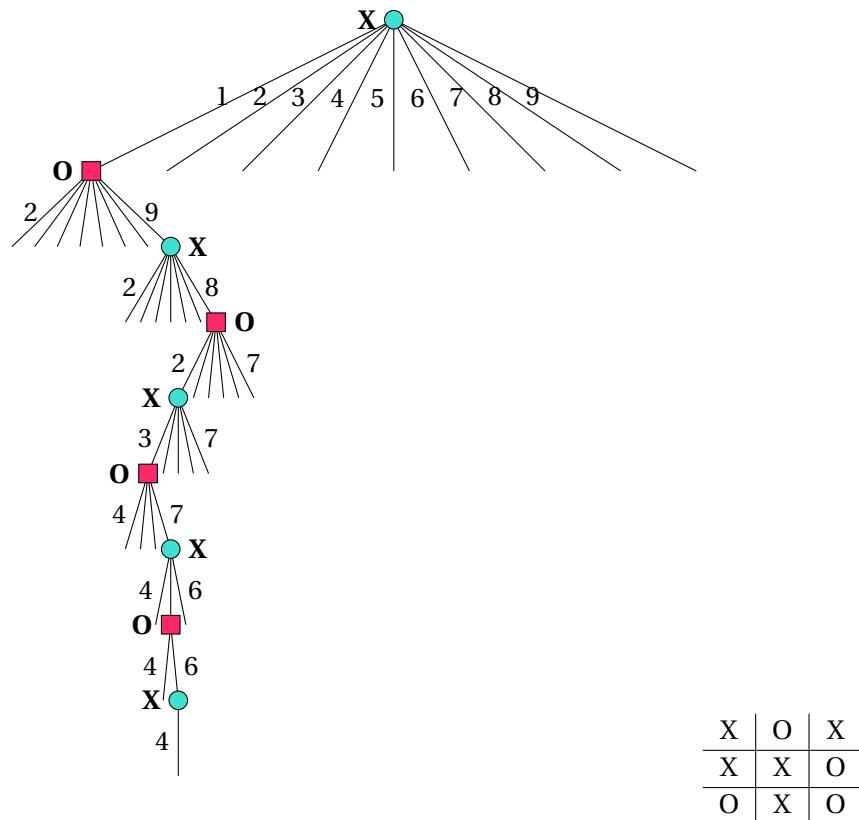


FIGURE 24.1 – Arbre de jeu d'une partie de morpion, partie nulle. On considère que les cases sont numérotées de 1 à 9 en ligne et en partant du haut. La position finale est donnée sous l'arbre. Le joueur à la croix X joue en premier car il contrôle la racine de l'arbre.

B Jeux d'accessibilité, l'exemple des jeux de Nim

■ **Définition 177 — Jeu d'accessibilité.** Un jeu d'accessibilité est un jeu à deux joueurs, à information complète et parfaite, séquentiel et pour lequel il n'y a pas de hasard. Ces jeux peuvent être modélisés par un graphe orienté biparti.

🇬🇧 **Vocabulary 26 — Impartial Game** ⇔ Jeu d'accessibilité. Un jeu d'accessibilité peut être qualifié d'impartial car les coups possibles ne dépendent que de la position dans le jeu et pas du joueur.

■ **Définition 178 — Jeu de Nim.** Un jeu de Nim est un jeu d'accessibilité dont il existe de nombreuses variantes. Il s'agit de déplacer, de poser ou de retirer un certain nombre d'objets simples (pièces, allumettes, graines, des billes...). Le dernier à jouer gagne ou perd (variante misère). Le jeu de Nim fait donc nécessairement un perdant et un gagnant.

🇬🇧 **Vocabulary 27 — Soustraction game** ⇔ Jeu de Nim ou jeu de la soustraction.

■ **Exemple 67 — Variantes du jeu de Nim.** Parmi les variantes les plus célèbres, on peut citer :

- le jeu de Marienbad (avec des cartes ou des allumettes)[13],
- le jeu des bâtonnets ^a [12],
- le jeu de Grundy.

La figure 24.2 donne un exemple de jeu de Marienbad tel qu'il est présenté dans le film d'Alain Resnais. Chaque joueur peut retirer autant d'allumettes qu'il le veut sur une ligne seulement. Le perdant est celui qui retire la dernière allumette.

Ce jeu est modélisable par un graphe orienté comme l'indique la figure 24.3. Sur cette figure, on considère que les joueurs jouent alternativement en se déplaçant sur le graphe : un des joueurs est initialement sur la position start, quatre allumettes sont reparties sur deux rangées.

Une modélisation plus exacte peut se faire en utilisant un graphe biparti comme le montre la figure 24.4. Ces deux figures illustrent la même position de départ. Le graphe biparti distingue en couleur les sommets des joueurs. Sur ce graphe, on peut jouer tous les cas : le joueur rouge est le premier ou le joueur cyan est premier.

^a. type Fort Boyard

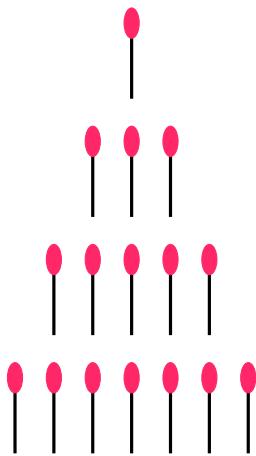


FIGURE 24.2 – Jeu de Marienbad avec des allumettes

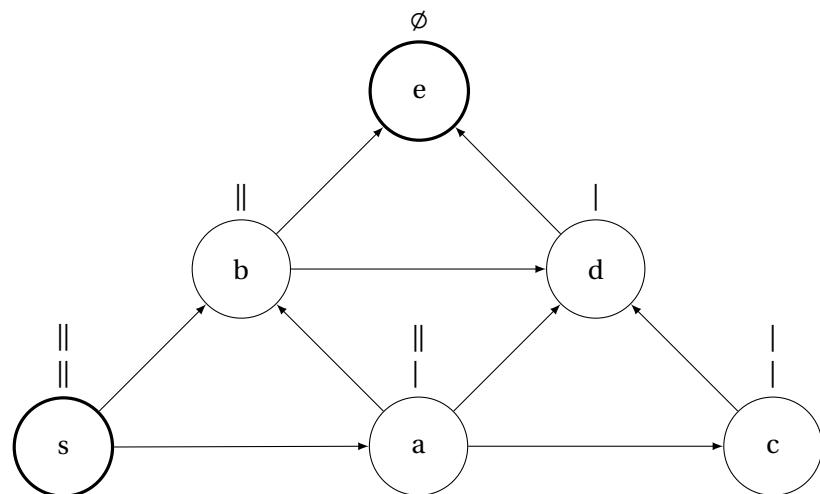


FIGURE 24.3 – Modélisation par graphe orienté d'une partie de jeu de Nim avec deux rangées de deux allumettes au départ. On peut jouer dessus avec un pion placé en s au départ. Puis chaque joueur fait avancer le pion d'un saut sur le graphe en sélectionnant un successeur en suivant les arcs. Le joueur qui se trouve en position e a gagné ou perdu dans la variante misère.

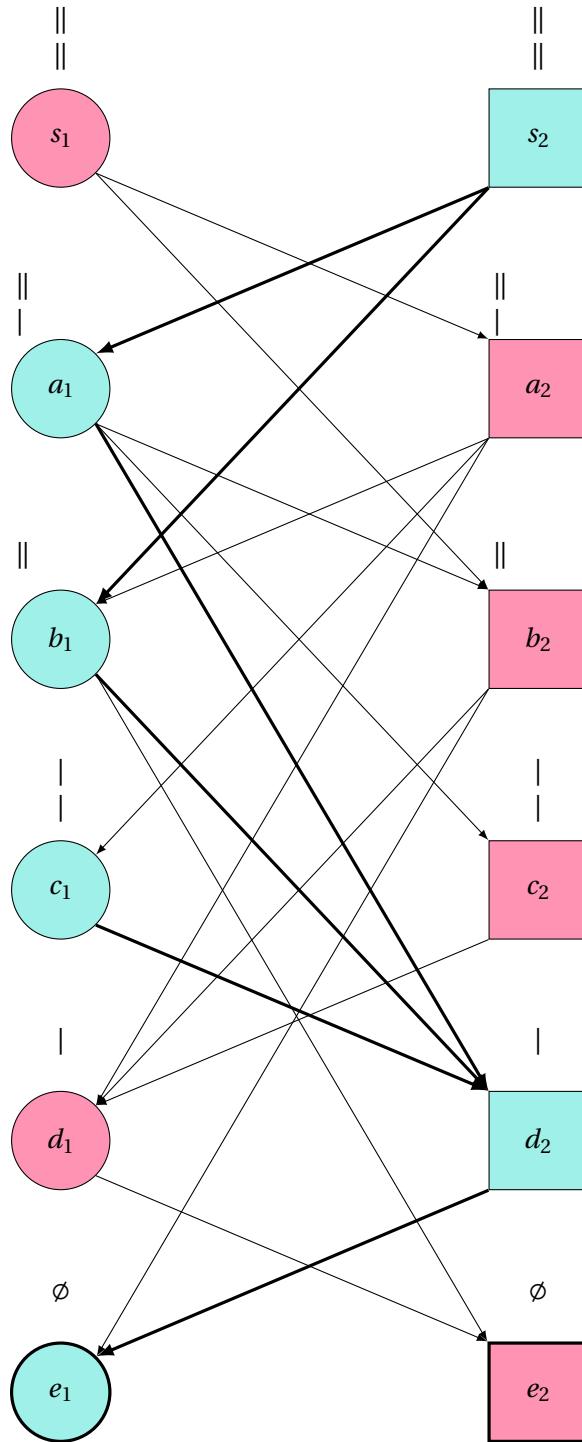


FIGURE 24.4 – Modélisation par graphe orienté biparti d'un jeu de Nim, variante misère : le dernier à jouer a perdu. Les sommets des joueurs 1 et 2 sont distingués par des cercles (1) et des carrés (2). La couleur cyan représente l'attracteur du joueur $J_1 : \mathcal{A}_1 = \{s_2, a_1, b_1, c_1, d_2, e_1\}$. On voit donc que 1 n'a pas intérêt à commencer à jouer dans cette configuration. Il en est de même pour le joueur 2 dont l'attracteur en rouge. C'est normal car la somme de Nim de la configuration initiale est nulle.

C Modélisation d'un jeu d'accessibilité

■ **Définition 179 — Arène de jeu.** Le graphe $G = (V_1, V_2, E)$ est nommé arène de jeu si est biparti si $G = (V = V_1 \cup V_2, E)$ est un graphe orienté biparti et $V_1 \cap V_2 = \emptyset$. Sur cette arène, le joueurs se répartissent dont les sommets : le joueur J_1 contrôle V_1 , le joueur J_2 V_2 .

■ **Définition 180 — Partie.** Une partie est un chemin sur l'arène de jeu : à chaque tour, le joueur J_1 en $v_i \in V_1$ choisit une arête de E dont le premier sommet est v_i et le second un sommet $v_j \in V_2$. J_2 choisit ensuite à partir de v_j le sommet suivant dans V_1 . Une partie en n coups s'écrit donc $(v_0, \dots, v_i, \dots, v_n)$.

■ **Définition 181 — Condition de gain.** Une condition de gain pour un joueur J_i sur une arène de jeu $G = (V, E)$ est un sous-ensemble C_i^g de V_i . La partie est remportée par le joueur J_i si celui-ci visite un sommet de C_i^g en premier.

■ **Définition 182 — Condition de victoire.** Une condition de victoire d'un joueur J_i est un sous-ensemble de toutes les parties possibles \mathcal{P} remportées par ce joueur. On la note :

$$C_i^v = \{\mathcal{P}, \mathcal{P} \text{ visite un sommet de } C_i^g\} \quad (24.1)$$

■ **Exemple 68 — Condition de gain et de victoire pour le jeu de Nim.** Pour le jeu de Nim de la figure 24.4 en choisissant la variante misère et J_1 comme premier joueur, alors $C_1^g = \{c_1\}$ est une condition de gain pour J_1 . De plus, $C_1^v = \{(s_1, a_2, c_1)\}$ est la condition de victoire de J_1 . J_1 n'a donc guère le choix...

D Stratégies et positions

■ **Définition 183 — Stratégie sans mémoire.** Soit $G = (V, E)$ une arène de jeu. On note $V_i^{>0}$ l'ensemble des sommets contrôlés par le joueur $i \in \{1, 2\}$ de degré sortant non nul. Une stratégie est une application $\phi : V_i^{>0} \rightarrow V$ telle que :

$$\forall v \in V_i^{>0}, (v, \phi(v)) \in E \quad (24.2)$$

Cette stratégie est sans mémoire car elle ne dépend que du sommet courant et pas des sommets précédents de la partie.

R Une stratégie permet donc de calculer le coup à jouer. Le joueur J_i suit la stratégie ϕ lors d'une partie $\mathcal{P} = (v_0, v_1, \dots, v_n)$ si $\forall j \in \llbracket 0, n \rrbracket, v_j \in V_i^{>0} \implies v_{j+1} = \phi(v_j)$

■ **Définition 184 — Stratégie gagnante.** Une stratégie ϕ est gagnante pour le joueur J_i

depuis le sommet $v_0 \in V_i$ si toute partie jouée depuis v_0 par J_i en suivant ϕ est gagnante pour J_i .

■ **Définition 185 — Position gagnante.** Soit $G = (V = V_1 \cup V_2, E)$ un jeu d'accessibilité à deux joueurs. Un sommet $v \in V_i$ est appelé position gagnante pour le joueur J_i si celui-ci possède une stratégie gagnante depuis v .

■ **Exemple 69 — Position gagnante du jeu de Nim.** Sur le jeu de la figure 24.4, le sommet a_1 est une position gagnante pour J_1 . Reste à trouver la stratégie $\phi \dots$ Le sommet s_1 n'est pas une position gagnante pour J_1 .

E Attracteurs

Pour gagner une partie d'un jeu à deux joueurs, il semble donc logique de chercher les positions gagnantes et une stratégie associée. La notion d'attracteur a été développée pour construire l'ensemble des positions gagnantes. L'idée est de construire cet ensemble en partant de la condition de gain, en remontant les arcs du graphe à l'envers et en ne conservant que les positions gagnantes.

■ **Définition 186 — Suite des ensembles attracteurs.** Soit $G = (V = V_1 \cup V_2, E)$ une arène d'un jeu d'accessibilité. On définit par induction la suite des attracteurs $(\mathcal{A}_j^1)_{j \in \mathbb{N}}$ du joueur J_1 , c'est à dire des ensembles des sommets de V à partir desquels le joueur J_1 peut forcer la partie à arriver en C_1^g , de la manière suivante :

$$\mathcal{A}_0^1 = C_1^g \quad \text{si } j = 0 \quad (24.3)$$

$$\mathcal{A}_{j+1}^1 = \mathcal{A}_j^1 \cup \{v \in V_1, \exists v' \in \mathcal{A}_j^1, (v, v') \in E\} \cup \{v \in V_2, \forall v' \in V, (v, v') \in E \Rightarrow v' \in \mathcal{A}_j^1\} \quad \forall j \geq 0 \quad (24.4)$$

$$(24.5)$$

Formulé simplement, le premier terme de cette suite est la condition de gain du joueur, c'est à dire les sommets qui lui donnent la victoire. Puis, le terme $j + 1$ de la suite est l'union :

- du terme \mathcal{A}_j^1 ,
- des sommets de V_1 qu'un arc peut mener à une position gagnante de \mathcal{A}_j^1 de V_2 ,
- des sommets de V_2 qui font obligatoirement aboutir à une position gagnante de V_1 .

■ **Définition 187 — Attracteur du joueur J_i .** L'attracteur du joueur i est l'ensemble des sommets d'une arène de jeu défini par :

$$\mathcal{A}^i = \bigcup_0^{+\infty} \mathcal{A}_j^i. \quad (24.6)$$

Théorème 19 — L'attracteur du joueur J_i contient exactement toutes les positions gagnantes de J_i .

Démonstration. On procède par récurrence sur le rang d'un sommet de G , une fonction $r : V \rightarrow \mathbb{N}$ définie comme suit :

$$\forall v \in V, r(v) = \min\{j, v \in \mathcal{A}_j^i\} \quad (24.7)$$

Pour un sommet n'appartenant pas à l'attracteur \mathcal{A} , le rang est infini. Cette définition est possible car la suite $(\mathcal{A}_j^i)_{j \in \mathbb{N}}$ est croissante au sens de l'inclusion.

L'hypothèse de récurrence est la suivante : \mathcal{H}_j : Pour tout $j \in \mathbb{N}$, les sommets de rang j sont des positions gagnantes du joueur J_1 .

- Initialisation \mathcal{H}_0 : pour $j = 0$, $\mathcal{A}_0^1 = C_1^g$, donc tous les sommets de \mathcal{A}_0^1 sont des positions gagnantes.
- Hérédité : on suppose que, pour un certain entier naturel j , l'ensemble \mathcal{A}_j^1 ne contient que des positions gagnantes de J_1 (\mathcal{H}_j est vraie). Considérons maintenant un élément v de l'ensemble \mathcal{A}_{j+1} de rang $j+1$. Supposons de plus¹ que v n'appartient pas à \mathcal{A}_j^1 . Il reste alors deux possibilités :
 1. Si $v \in V_1$, alors par définition de l'ensemble, il existe un arc qui amène à une position gagnante de \mathcal{A}_j^1 . Donc, v est une position gagnante.
 2. Si $v \in V_2$, alors par définition de l'ensemble, tous les arcs de l'arène l'amène vers une position gagnante de \mathcal{A}_j^1 . C'est donc une position gagnante.
- Conclusion : on peut donc conclure que les ensembles \mathcal{A}_j^1 ne contiennent que des positions gagnantes. L'attracteur \mathcal{A} ne possède donc que des positions gagnantes.

■

R On peut maintenant construire une stratégie **gagnante** : la stratégie sans mémoire ϕ qui, au fur et à mesure de la partie, fait diminuer le rang de la position courante :

$$\forall v_j \in \mathcal{A}_j^1 \cap V_1, v_{j+1} = \phi(v_j), r(v_{j+1}) < r(v_j) \quad (24.8)$$

est gagnante. En effet, en choisissant de diminuer le rang de la position suivante, on se rapproche de la victoire.

F Algorithme de calcul de l'attracteur

Pour trouver l'attracteur d'un joueur, il faut parcourir l'arène de jeu en inversant les arcs, c'est à dire en transposant le graphe. Ainsi, en partant de la conditions de gain C_i^g et en remontant les arcs, on parvient à trouver \mathcal{A} .

L'algorithme 62 détaille la procédure à suivre. Cet algorithme possède une sous-fonction récursive. Comme le graphe est fini, on est cependant sûr de la terminaison. Sa correction a

1. sinon c'est trivial

été donnée par la démonstration précédente : on ne fait que faire croître les ensembles \mathcal{A}_j et appliquer leur définition.

Par ailleurs, l'algorithme utilise les degrés entrants du graphe transposé, mais il est tout à fait possible de raisonner sur les degrés sortant du graphe. Il m'apparaît juste plus naturel de remonter le graphe transposé en se demandant si on peut arriver à un sommet de V_2 autrement que par un sommet de l'attracteur, plutôt qu'en se demandant si les chemins sortant d'un sommet de V_2 mènent à des sommets n'appartenant pas à l'attracteur.

Algorithme 62 Calcul de l'attracteur du joueur J_1

Entrée : g le graphe biparti de l'arène de jeu

Entrée : V_1 l'ensemble de sommets du joueur J_1

Entrée : C_g^1 la condition de gain du joueur J_1

```

1: Fonction ATTRACTEUR( $g, V_1, C_g^1$ )
2:    $\mathcal{A} \leftarrow \emptyset$                                       $\triangleright$  l'attracteur
3:    $g^t \leftarrow$  le transposé du graphe  $g$                    $\triangleright$  Pour remonter le graphe
4:    $d_{in} \leftarrow$  tableau des degrés entrants du graphe transposé     $\triangleright$  Pour compter ce qui entre
5:   pour chaque sommet  $v \in C_g^1$  répéter            $\triangleright$  On part de la condition de gain
6:     AUGMENTER_ATTRACTEUR( $v, \mathcal{A}, g^t, d_{in}, V_1$ )
7:   renvoyer  $\mathcal{A}$ 
8: Fonction AUGMENTER_ATTRACTEUR( $v, \mathcal{A}, g^t, d_{in}, V_1$ )
9:   si  $v \notin \mathcal{A}$  alors
10:     $\mathcal{A} \leftarrow \mathcal{A} \cup \{v\}$ 
11:    pour chaque voisin  $u$  de  $v$  répéter            $\triangleright$  Pour remonter le graphe
12:       $d_{in}[u] \leftarrow d_{in}[u] - 1$             $\triangleright$  On passe par ce sommet une fois depuis  $\mathcal{A}$ 
13:      si  $u \in V_1$  ou  $d_{in}[u] = 0$  alors  $\triangleright$  Soit  $u \in V_1$  soit tous ses arcs entrant viennent de  $\mathcal{A}$ 
14:        AUGMENTER_ATTRACTEUR( $u, \mathcal{A}, g^t, d_{in}, V_1$ )

```

★ G Solution des jeux de Nim et impartialiaux --> HORS PROGRAMME

S'il est possible de calculer les attracteurs d'un joueur, il reste néanmoins un problème de taille : comment disposer de l'arène ? En effet, sur un exemple simple comme le jeu de Marienbad décrit sur la figure 24.4, il est relativement facile de créer le graphe associé à une arène de jeux. Cependant, dès que les dimensions augmentent, par exemple le nombre de rangées et le nombre de bâtonnets, même sur un jeu fini, il devient difficile de construire le graphe en entier. Il faut donc envisager d'autres méthodes pour trouver des stratégies gagnantes. Le nombre de Grundy permet de calculer la stratégie à adopter sans construire le graphe en entier, en connaissant uniquement la position courante du jeu, c'est à dire les nombres de bâtonnets des n piles (x_1, \dots, x_n).

L'idée ingénieuse des mathématiciens pour résoudre les jeux d'accessibilité est la suivante :

- ramener un jeu d'accessibilité à un jeu de Nim dans une position donnée, (x_1, \dots, x_n) où les x_i sont les tailles des piles,

- décomposer ce jeu en n jeux de Nim à une seule pile G_1, G_2, \dots, G_n et définir une addition sur jeux pour faire en sorte que le jeu initial soit somme des jeux à une pile.

L'addition a été trouvé par Bouton en 1901, c'est le ou exclusif.

Théorème 20 — Bouton^[5]. Un position donnée (x_1, \dots, x_n) d'un jeu de Nim est une position gagnante si et seulement si $x_1 \oplus x_2 \oplus \dots \oplus x_n = 0$, où \oplus est l'opérateur du ou exclusif bit à bit ^a.

a. Par exemple, $5 \oplus 3 = 6$

■ **Définition 188 — Nombre de Grundy d'un jeu de Nim.** Le nombre de Grundy est la somme trouvée dans le théorème 20 en utilisant le ou exclusif sur la position du jeu : $x_1 \oplus x_2 \oplus \dots \oplus x_n = 0$.

Mais on peut également la définir récursivement :

- si la pile du jeu de Nim est en position finale, le nombre de Grundy vaut 0,
- sinon, le nombre de Grundy d'une position donnée (x_1, \dots, x_n) est le plus petit entier positif ou nul qui n'apparaît pas dans la liste des nombres de Grundy des positions qui suivent immédiatement la position donnée.

Ceci s'écrit parfois :

$$\gamma = \text{mex}(x_1, \dots, x_n) \quad (24.9)$$

où la fonction *mex* est le plus petit entier positif non trouvé dans un ensemble.

Le théorème 21 décrit alors précisément la stratégie gagnante.

Théorème 21 — Sprague et Grundy . Tout jeu d'accessibilité \mathcal{I} est équivalent à un jeu de Nim \mathcal{N} .

Pour une position de \mathcal{I} donnée, il existe une position de \mathcal{N} dont le nombre de Grundy est γ . Cette position est équivalente à celle d'un jeu de Nim à un seul tas comportant γ allumettes.

La stratégie gagnante est celle qui consiste à choisir la position suivante de telle manière à ce que son nombre de Grundy soit nul.

■ **Exemple 70 — Utilisation du nombre de Grundy.** Ces théorèmes permettent d'affirmer que la position de départ de la figure 24.4 est une position perdante, car le nombre de Grundy est nul : $10_2 \oplus 10_2 = 00_2$. Il ne reste plus qu'à vous entraîner au calcul en binaire.

H Au-delà des jeux d'accessibilité, les heuristiques

Une stratégie est connue pour les jeux d'accessibilité, mais, s'ils permettent de briller dans les salons, une fois la stratégie connue, la lassitude s'installe. De nombreux autres jeux existent, comme les échecs ou le go. Pour ces jeux, il est impensable de dessiner l'arbre de jeu, la combinatoire nous indique que le nombre de parties possibles est bien trop grand. Le nombre de

Shannon est une tentative pour estimer le nombre de parties différentes² possibles. Il vaut 10^{123} pour les échecs ce qui est plus grand que le nombre d'atomes dans l'univers observable... S'il nous faut donc renoncer à explorer ces arbres de jeux d'une manière exhaustive, rien ne nous empêche néanmoins de les explorer localement.

Si on utilise un arbre de jeu exhaustif, la partie se finit lorsque la position est une feuille à laquelle est associé un gain ou un score. Lorsqu'on explore localement un arbre de jeu, les feuilles sont parfois absentes voire encore très loin de la position. C'est pourquoi le développement de ces algorithmes s'appuie sur des heuristiques capables d'estimer le gain d'une position sans disposer de l'intégralité de l'arbre de jeu.

■ **Définition 189 — Heuristique.** Une heuristique^a est une approche de résolution de problème à partir de connaissances incomplètes. Une heuristique doit permettre d'aboutir en un temps limité à des solutions néanmoins acceptables même si elles ne sont pas nécessairement optimales.

a. Je trouve en grec ancien.

I Minimax et les heuristiques

L'algorithme Minimax associé à une heuristique permet de développer des stratégies sans avoir à explorer tout l'arbre de jeu. Il découle naturellement du théorème du même nom démontré par Von Neumann en 1926.

L'algorithme 63 détaille la procédure Minimax. Le principe est le suivant : on construit un arbre de jeu incomplet comme celui-de la figure 24.5. La hauteur de l'arbre est un entier fixé qui limite la profondeur de l'exploration de l'algorithme. La complexité s'en trouve ainsi améliorée.

Chaque niveau de l'arbre est contrôlé par un seul joueur. Comme, pour les jeux d'accessibilité considérés, les gains de l'un sont les pertes de l'autre, on choisit de nommer les joueurs J_{max} en rouge et J_{min} en cyan. Le but du jeu est de maximiser le gain pour J_{max} , son score est donc positif, et, symétriquement, minimiser le gain pour J_{min} dont le score est donc négatif.

■ **Définition 190 — Définition du score des joueurs pour une position donnée.** Le jeu associe à chaque feuille de l'arbre minimax un gain qui devient le score du joueur qui atteint cette feuille. Si la feuille est en position p alors on choisit de noter ce score $s(p) \in \mathbb{R}$. À partir du score associé aux feuilles, on peut définir récursivement le score maximal ou minimal associé à un nœuds internes p de l'arbre minimax comme suit :

$$s(p) = \begin{cases} s(p) & \text{si } p \text{ est une feuille} \\ \max \{s(f), f \text{ fils de } p\} & \text{si } p \text{ est contrôlé par } J_{max} \\ \min \{s(f), f \text{ fils de } p\} & \text{si } p \text{ est contrôlé par } J_{min} \end{cases} \quad (24.10)$$

Ainsi, à la fin de la partie, si J_{max} en position p effectue les choix décrit ci-dessus, son score final sera au moins $s(p)$. De même, le score de J_{min} en position p sera au plus $s(p)$. Ceci peut se démontrer par récurrence.

2. qui ont un sens

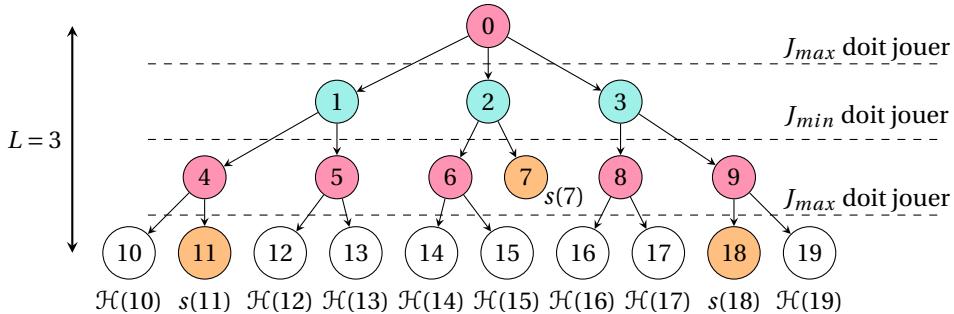


FIGURE 24.5 – Exemples d’arbre minimax. Chaque niveau est contrôlé par un seul joueur, J_{max} en rouge et J_{min} en cyan. La hauteur de l’arbre L est telle que seule une partie de l’arbre de jeu est accessible. Certaines feuilles sont visibles (en orange, c’est l’automne). L’arbre minimax s’achève donc parfois sur des nœuds internes pour lesquels on donne une estimation du gain (non coloré).

Le score d’un joueur ne peut se calculer tel que décrit ci-dessus puisqu’on ne connaît pas l’intégralité de l’arbre de jeu. L’algorithme 63 suppose donc qu’on connaît une heuristique \mathcal{H} pour estimer le score d’une position d’un nœud intermédiaire. Cette heuristique est une fonction de la position p dans l’arbre et sa valeur est un nombre réel $\mathcal{H}(p)$.

■ **Exemple 71 — Calcul des scores sur un arbre Minimax.** La figure 24.6 superpose les scores calculés par l’algorithme Minimax sur chaque nœud. Le joueur J_{max} peut donc espérer au plus un score de 6 s’il se trouve en position 0.

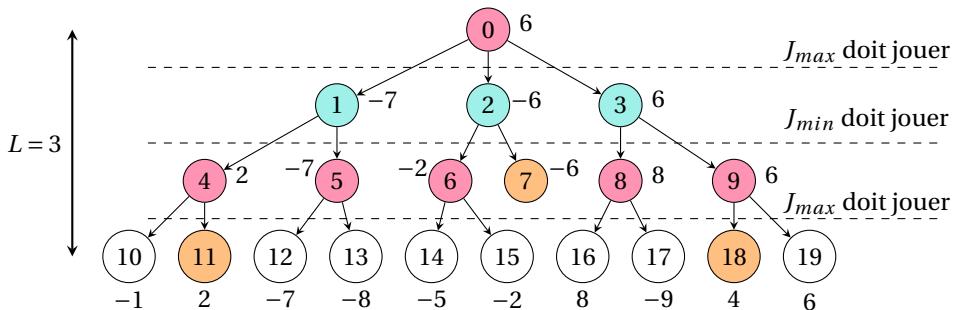


FIGURE 24.6 – Exemples d’arbre minimax complété avec les scores. Chaque niveau est contrôlé par un seul joueur, J_{max} en rouge et J_{min} en cyan.

Algorithme 63 Minimax

Entrée : p un position dans l'arbre de jeu (un nœud de l'arbre)**Entrée :** s la fonction de score sur les feuilles**Entrée :** \mathcal{H} l'heuristique de calcul du score pour un nœud interne**Entrée :** L la profondeur maximale de l'arbre Minimax

```

1: Fonction MINIMAX( $p, s, \mathcal{H}, L$ )
2:   si  $p$  est une feuille alors
3:     renvoyer  $s(p)$ 
4:   si  $L = 0$  alors
5:     renvoyer  $\mathcal{H}(p)$                                  $\triangleright$  On arrête d'explorer, on estime
6:   si  $p$  est contrôlé par  $J_{max}$  alors
7:      $M \leftarrow -\infty$ 
8:      $p_M$  un nœud vide
9:     pour chaque fils  $f$  de  $p$  répéter
10:     $v \leftarrow \text{MINIMAX}(f, s, \mathcal{H}, L - 1)$            $\triangleright v$  est un score de  $J_{min}$ 
11:    si  $v > M$  alors
12:       $M \leftarrow v$ 
13:       $p_M = f$ 
14:    renvoyer  $M, p_M$                                  $\triangleright$  Valeur maximale trouvée et la racine de cette solution
15:  sinon
16:     $m \leftarrow +\infty$ 
17:     $p_m$  un nœud vide
18:    pour chaque fils  $f$  de  $p$  répéter
19:       $v \leftarrow \text{MINIMAX}(f, s, \mathcal{H}, L - 1)$            $\triangleright v$  est un score de  $J_{max}$ 
20:      si  $v < m$  alors
21:         $m \leftarrow v$ 
22:         $p_m = f$ 
23:    renvoyer  $m, p_m$                                  $\triangleright$  Valeur minimale trouvée et la racine de cette solution

```

J Élagage $\alpha\beta$ sur un arbre Minimax

--- HORS PROGRAMME

Selon les situations considérées, limiter la profondeur d'exploration de l'arbre de jeux peut s'avérer être insuffisant pour réduire la complexité du problème. L'élagage $\alpha\beta$ est une technique pour ne pas explorer certaines branches de l'arbre qui n'ont pas besoin de l'être. Le principe est détaillé sur la figure 24.7.

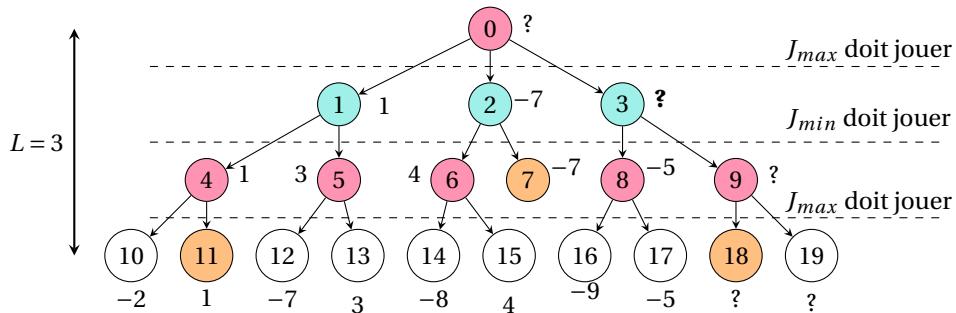


FIGURE 24.7 – L'élagage $\alpha\beta$ permet sur cet exemple d'éviter l'exploration complète du sous-arbre du nœud numéro 3. En effet, comme c'est un nœud de J_{min} , que le premier nœud du niveau a un score de 1 et que le score calculé du nœud 8 vaut -5, alors on comprend que J_{min} remontera au moins -5 et que J_{max} ne choisira pas ce nœud numéro 3 car ce n'est pas le maximum du niveau.

On distingue deux types d'élagage possible, les types α et β comme le montre les figures 24.8 et 24.9. Pour améliorer la complexité de l'algorithme Minimax, on peut choisir d'élaguer comme le montre l'algorithme 64.

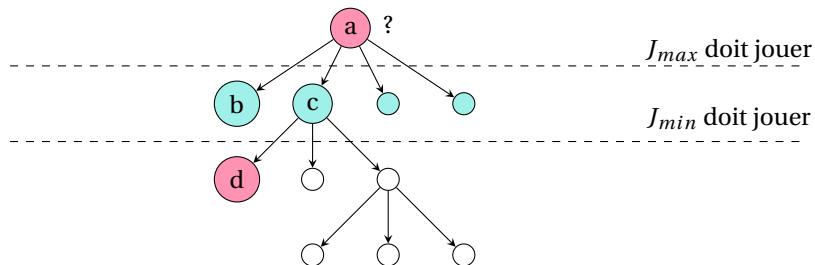


FIGURE 24.8 – L'élagage α : supposons que $S(b)$ ait déjà été calculé par l'algorithme Minimax. Si $S(b) \geq S(d)$, alors il est inutile d'explorer le sous-arbre c : J_{max} ne choisira pas le nœud c , il lui préfèrera b .

Algorithme 64 Minimax avec élagage $\alpha\beta$

Entrée : p un position dans l'arbre de jeu (un nœud de l'arbre)**Entrée :** s la fonction de score sur les feuilles**Entrée :** \mathcal{H} l'heuristique de calcul du score pour un nœud interne**Entrée :** L la profondeur maximale de l'arbre Minimax**Entrée :** α le niveau de coupure α ▷ $-\infty$ au démarrage**Entrée :** β le niveau de coupure β ▷ $+\infty$ au démarrage1: **Fonction** $\text{MINIMAX_}\alpha\beta(p, s, \mathcal{H}, L, \alpha, \beta)$ 2: **si** p est une feuille **alors**3: **renvoyer** $s(p)$ 4: **si** $L = 0$ **alors**5: **renvoyer** $\mathcal{H}(p)$

▷ On arrête d'explorer, on estime

6: **si** p est contrôlé par J_{max} **alors**7: $M \leftarrow -\infty$ 8: p_M un nœud vide9: **pour** chaque fils f de p **répéter**10: $v \leftarrow \text{MINIMAX_}\alpha\beta(f, s, \mathcal{H}, L - 1, \alpha, \beta)$ ▷ v est un score de J_{min} 11: **si** $v > M$ **alors**12: $M \leftarrow v$ 13: $p_M = f$ 14: **si** $v \geq \beta$ **alors**15: **renvoyer** M ▷ Élagage de type β 16: $\alpha \leftarrow \max(\alpha, M)$ ▷ Mise à jour du niveau de l'élagage17: **renvoyer** M, p_M ▷ Valeur maximale trouvée et la racine de cette solution18: **sinon**19: $m \leftarrow +\infty$ 20: p_m un nœud vide21: **pour** chaque fils f de p **répéter**22: $v \leftarrow \text{MINIMAX_}\alpha\beta(f, s, \mathcal{H}, L - 1, \alpha, \beta)$ ▷ v est un score de J_{max} 23: **si** $v < m$ **alors**24: $m \leftarrow v$ 25: $p_m = f$ 26: **si** $v \leq \alpha$ **alors**27: **renvoyer** m ▷ Élagage de type α 28: $\beta \leftarrow \min(\beta, m)$ ▷ Mise à jour du niveau de l'élagage29: **renvoyer** m, p_m ▷ Valeur minimale trouvée et la racine de cette solution

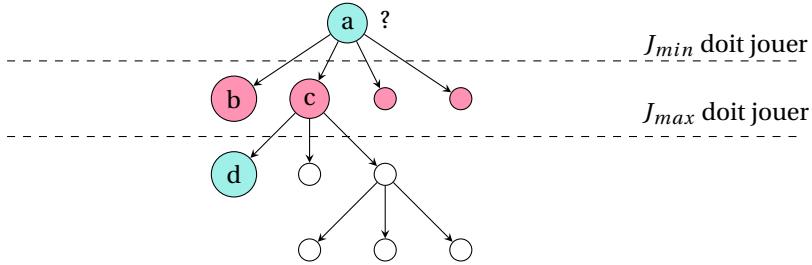


FIGURE 24.9 – L’élagage β : supposons que $\mathcal{S}(b)$ ait déjà été calculé par l’algorithme Minimax. Si $\mathcal{S}(b) \leq \mathcal{S}(d)$, alors il est inutile d’explorer le sous-arbre c : J_{min} ne choisira pas le nœud c , il lui préférera b .

K A* pour trouver un chemin

Si le jeu que l’on considère ne peut pas être modélisé selon un arbre Minimax, alors la situation n’est pas désespérée car il nous reste encore les graphes. Il est toujours possible de modéliser l’évolution d’un jeu comme une succession d’état, chaque coup joué permettant de passer d’un état à un autre. Développer une stratégie dans un jeu modélisé par un graphe à état revient donc à trouver un chemin d’un sommet à un autre dans un graphe. Or, nous sommes déjà bien outillés pour faire cela.

L’algorithme A*³ est un algorithme couteau suisse qui peut être considéré comme un algorithme de Dijkstra muni d’une heuristique : là où Dijkstra ne tient compte que du coût du chemin déjà parcouru, A* considère ce coût et une heuristique qui l’informe sur le reste du chemin à parcourir. Il faut bien remarquer que le chemin qu’il reste à parcourir n’est pas nécessairement déjà exploré : il est rarement possible d’explorer tout le graphe à état d’un jeu. Si l’heuristique pour évaluer le reste du chemin à parcourir est bien choisie, alors A* converge aussi vite voire plus vite que Dijkstra[23].

■ **Définition 191 — Heuristique admissible.** Une heuristique \mathcal{H} est admissible si pour tout sommet du graphe d’état, $\mathcal{H}(s)$ est une borne inférieure de la plus courte distance séparant le sommet de départ du sommet d’arrivée.

■ **Définition 192 — Heuristique cohérente.** Une heuristique \mathcal{H} est cohérente si pour tout arc (s, p) du graphe d’état $G = (V, E, w)$, $\mathcal{H}(s) \leq \mathcal{H}(p) + w(s, p)$.

■ **Définition 193 — Heuristique monotone.** Une heuristique \mathcal{H} est monotone si l’estimation du coût **total** du chemin ne décroît pas lors du passage d’un sommet à ses successeurs. Pour un chemin (s_0, s_1, \dots, s_n) , on $\forall 0 \leq i < j \leq n, c(s_j) \geq c(s_i)$.

Soit $G = (V, E, w)$ un graphe orienté pondéré. Soit d la fonction de distance utilisée par

3. prononcer *A étoile* ou *A star*

l'algorithme de Dijkstra (cf. algorithme 51). A*, muni d'une fonction h permettant d'évaluer l'heuristique, calcule alors le coût total pour aller jusqu'à un sommet p comme suit :

$$c(p) = d(p) + h(p) \quad (24.11)$$

Le coût obtenu n'est pas nécessairement optimal, il dépend de l'heuristique.

Supposons que l'on cherche le chemin le plus court entre les sommets s_0 et p . Supposons que l'on connaisse un chemin optimal entre s_0 et un sommet s . Alors on peut écrire que le coût total vers le sommet p vaut :

$$c(p) = d(p) + h(p) \quad (24.12)$$

$$= d(s) + w(s, p) + h(p) \quad (24.13)$$

$$= d(s) + h(s) + w(s, p) - h(s) + h(p) \quad (24.14)$$

$$= c(s) + w(s, p) - h(s) + h(p) \quad (24.15)$$

Ainsi, on peut voir l'algorithme A* comme un algorithme de Dijkstra muni :

- de la distance $\tilde{d} = c$,
- et de la pondération $\tilde{w}(s, p) = w(s, p) - h(s) + h(p)$.

L'algorithme 65 donne le détail de la procédure à suivre.

Algorithme 65 A*

```

1: Fonction ASTAR( $G = (V, E, w)$ ,  $a$ ) ▷ Sommet de départ  $a$ 
2:    $\Delta \leftarrow a$ 
3:    $\Pi \leftarrow$ 
4:    $\tilde{d} \leftarrow$  l'ensemble des distances au sommet  $a$ 
5:    $\forall s \in V, \tilde{d}[s] \leftarrow \tilde{w}(a, s)$  ▷ Le graphe est partiel, l'heuristique fait le reste
6:   tant que  $\bar{\Delta}$  n'est pas vide répéter ▷  $\bar{\Delta}$  : sommets dont la distance n'est pas connue
7:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $\tilde{d}[u] = \min(\tilde{d}[v], v \in \bar{\Delta})$ 
8:      $\Delta = \Delta \cup \{u\}$ 
9:     pour  $x \in \bar{\Delta}$  répéter ▷ Ou bien  $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$ , pour tous les voisins de  $u$  dans  $\bar{\Delta}$ 
10:    si  $\tilde{d}[x] > \tilde{d}[u] + \tilde{w}(u, x)$  alors
11:       $\tilde{d}[x] \leftarrow \tilde{d}[u] + \tilde{w}(u, x)$ 
12:       $\Pi[x] \leftarrow u$  ▷ Pour garder la tracer du chemin le plus court
13:    renvoyer  $\tilde{d}, \Pi$ 

```

Cinquième partie

Annexes

BIBLIOGRAPHIE

Articles

- [3] Richard BELLMAN. "The theory of dynamic programming". In : *Bulletin of the American Mathematical Society* 60.6 (1954), pages 503-515 (cf. page 185).
- [4] Richard BELLMAN. "On a routing problem". In : *Quarterly of applied mathematics* 16.1 (1958), pages 87-90 (cf. page 155).
- [5] Charles L BOUTON. "Nim, a game with a complete mathematical theory". In : *The Annals of Mathematics* 3.1 (1901). Publisher : JSTOR, pages 35-39 (cf. page 251).
- [6] E. W. DIJKSTRA. "A note on two problems in connexion with graphs". In : *Numerische Mathematik* 1.1 (1^{er} déc. 1959), pages 269-271. ISSN : 0945-3245. DOI : [10.1007/BF01386390](https://doi.org/10.1007/BF01386390). URL : <https://doi.org/10.1007/BF01386390> (visité le 27/07/2022) (cf. page 150).
- [8] Ronald A FISHER. "The use of multiple measurements in taxonomic problems". In : *Annals of eugenics* 7.2 (1936). Publisher : Wiley Online Library, pages 179-188 (cf. page 233).
- [10] Robert W FLOYD. "Algorithm 97 : shortest path". In : *Communications of the ACM* 5.6 (1962). Publisher : ACM New York, NY, USA, page 345 (cf. page 156).
- [14] Donald B JOHNSON. "Efficient algorithms for shortest paths in sparse networks". In : *Journal of the ACM (JACM)* 24.1 (1977). Publisher : ACM New York, NY, USA, pages 1-13 (cf. page 154).
- [16] Casimir KURATOWSKI. "Sur le probleme des courbes gauches en topologie". In : *Fundamenta mathematicae* 15.1 (1930), pages 271-283 (cf. page 123).
- [18] John MCCARTHY et al. "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955". In : *AI magazine* 27.4 (2006), pages 12-12 (cf. page 227).
- [20] Henry Gordon RICE. "Classes of recursively enumerable sets and their decision problems". In : *Transactions of the American Mathematical Society* 74.2 (1953). Publisher : JSTOR, pages 358-366 (cf. page 86).
- [21] Bernard ROY. "Transitivité et connexité". In : *Comptes Rendus Hebdomadaires Des Séances De L'Academie Des Sciences* 249.2 (1959). Publisher : GAUTHIER-VILLARS/EDITIONS ELSEVIER 23 RUE LINOIS, 75015 PARIS, FRANCE, pages 216-218 (cf. page 156).

- [22] Geoffrey TWEEDALE. "William Poundstone, Prisoner's Dilemma : John von Neumann, Game Theory, and the Puzzle of the Bomb. Oxford : Oxford University Press, 1992. Pp. xi+ 290. ISBN 0-19-286162-X.\pounds 7.99 (paperback edition)." In : *The British Journal for the History of Science* 26.3 (1993). Publisher : Cambridge University Press, pages 375-376 (cf. page 240).
- [25] Stephen MARSHALL. "A theorem on boolean matrices". In : *Journal of the ACM (JACM)* 9.1 (1962). Publisher : ACM New York, NY, USA, pages 11-12 (cf. page 156).

Livres

- [2] Claude Gaspar BACHET. *Problèmes plaisans et delectables, qui se font par les nombres.* chez Pierre Rigaud, 1612. 200 pages (cf. page 239).
- [15] Luc JULIA. *L'intelligence artificielle n'existe pas.* First, 2019 (cf. page 227).
- [17] Édouard LUCAS. *Récréations mathématiques.* Tome 2. Gauthier-Villars et fils, 1883 (cf. page 131).
- [24] John VON NEUMANN et Oscar MORGENSTERN. *Theory of Games and Economic Behaviour.* Press, Princeton, 1944 (cf. page 239).

Vidéos

- [1] ARTE. *Le dilemme du prisonnier | Voyages au pays des maths | ARTE.* 9 oct. 2021. URL : <https://www.youtube.com/watch?v=G9ER5bLxQEU> (visité le 23/08/2022) (cf. page 241).
- [12] FORT-BOYARD.FR. *Bâtonnets.* 5 juill. 2011. URL : <https://www.youtube.com/watch?v=10CUpulWxww> (visité le 23/08/2022) (cf. page 244).
- [13] ITEMPRODUCTIONS. *Nim game from "last year at marienbad".* 7 juill. 2010. URL : <https://www.youtube.com/watch?v=8218FtL60g4> (visité le 23/08/2022) (cf. page 244).
- [23] UNSWMECHATRONICS. *Dijkstra's Algorithm vs. A* Search vs. Concurrent Dijkstra's Algorithm.* 24 juin 2013. URL : <https://www.youtube.com/watch?v=cSxn0m5aceA> (visité le 26/08/2022) (cf. pages 159, 257).

Sites web

TABLE DES FIGURES

4.1	Comparaison des chaînes d'exécution des langages compilés (à gauche) et des langages interprétés (à droite)	17
4.2	Bytecode Python du code 4.1 de la fonction produit.	18
4.3	Paradigmes des langages de programmation	19
4.4	Positionnement du système d'exploitation et des bibliothèques logicielles entre les éléments logiciels et le matériel électronique	21
4.5	Arborescence typique d'un système de fichier Unix/Linux/Mac OS X	23
5.1	Types de données composés du langage Python : les conteneurs	29
5.2	Variable de type immuable, affectation et référencement en mémoire (à gauche) et programme Python pour la visualisation des adresses en mémoire (à droite) . .	34
5.3	Variable de type mutable, affectation et référencement en mémoire (à gauche) et programme Python pour la visualisation des adresses en mémoire (à droite) . .	34
8.1	Illustration de la recherche dichotomique de la valeur 7 dans un tableau trié (Source : Wikimedia Commons)	62
9.1	Vision de la pile d'exécution de l'algorithme factoriel récursif 12 pour le calcul de $3!$	70
12.1	Comparaison des croissances des complexités usuelles	98
12.2	Structure d'arbre et appels récursifs pour la récurrence de la recherche dichotomique : $T(n) = T(n/2) + c$ et $\frac{n}{2^k} = 1$. Hors appel récursif, la fonction opère un nombre constant d'opérations c	103
12.3	Structure d'arbre et appels récursifs pour le tri fusion : $T(n) = 2T(n/2) + f(n)$ et $\frac{n}{2^k} = 1$. La fonction FUSION opère un nombre d'opérations $f(n)$	107
13.1	Principe de la décomposition d'un problème en sous-problèmes indépendants pour un algorithme de type diviser pour régner.	110
13.2	Structure d'arbre et appels récursifs pour la récurrence : $T(n) = rT(n/d) + f(n)$ ET $n/d^k = s$. On a choisi $r = 3$ pour l'illustration, c'est-à-dire chaque nœud possède trois enfants au maximum : on opère trois appels récursifs à chaque étape de l'algorithme.	111
13.3	Structure d'arbre et appels récursifs pour la récurrence de la recherche dichotomique : $T(n) = T(n/2) + c$ et $\frac{n}{2^k} = 1$. Hors appel récursif, la fonction opère un nombre constant d'opérations c	113

14.1 Étape de résolution d'un problème par décomposition en sous-problèmes et approche gloutonne.	116
14.2 Illustration du problème du sac à dos (d'après Wikipedia). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 . Le poids total admissible dans le sac est 15kg.	119
15.1 Graphe simple	122
15.2 Multigraphe à deux boucle et deux arêtes parallèles → HORS PROGRAMME	122
15.3 Graphe pondéré	123
15.4 Graphe orienté	123
15.5 Graphe complet K_5	124
15.6 Graphe biparti	124
15.7 Graphe biparti complet K_{34}	124
15.8 Graphe d'ordre cinq, de taille quatre et de séquence $[0, 1, 2, 2, 3]$. Le sommet d est isolé. Ce graphe n'est ni complet ni connexe.	127
15.9 Graphe complémentaire du graphe de la figure 15.8	127
15.10 Graphe d'ordre trois, de taille deux et de séquence $[1, 1, 2]$	128
15.11 Graphe d'ordre trois, de taille deux et de séquence $[1, 1, 2]$	128
15.12 Graphe d'exemple $G = (S = \{a, b, c, d\}, A = \{ab, ac, bc, bd, dc\})$	129
15.13 Graphe d'exemple $G' = (S' = \{1, 2, 3, 4\}, A' = \{12, 14, 24, 23, 43\})$	129
15.14 Exemple de chaîne simple reliant a à d en rouge	130
15.15 Exemple de cycle en turquoise	131
15.16 Saurez-vous trouver le cycle eulérien de ce graphe?	131
15.17 Graphe du jeu icosien et du dodécaèdre (solide régulier à 12 faces pentagonales). C'est un graphe cubique car chaque sommet possède trois voisins. Ce graphe possède un cycle hamiltonien. Saurez-vous le trouver?	132
15.18 Graphe K_5 : saurez-vous trouver des cycles hamiltonien et eulérien de ce graphe?	132
15.19 Exemple de sous-graphe couvrant G en rouge : $G = (S = \{a, b, c, d\}, A = \{ab, ac, cd\})$	133
15.20 Exemple de sous-graphe induit par les sommets $S = \{a, c, d\}$ en turquoise. $G[S] = (S = \{a, c, d\}, A = \{ac, cd\})$	133
15.21 Exemple de 4-coloration valide d'un graphe. Cette coloration n'est pas optimale.	134
15.22 Exemple de 3-coloration valide d'un graphe. Cette coloration est optimale.	134
15.23 Graphe de Petersen : saurez-vous proposer une coloration optimale de ce graphe sachant que son nombre chromatique vaut trois?	135
15.24 Exemples d'arbres enracinés. Les racines des arbres sont en rouge, les feuilles en turquoise. Le tout forme une forêt.	137
15.25 Arbre binaire	137
15.26 Arbre binaire parfait	137
16.1 Sur ces graphes, on peut vérifier les théorèmes de caractérisation des chaînes eulériennes et des cycles eulériens	140

17.1 Exemple de parcours en largeur au départ de a : a → b → c → d → e → g → f → h.	145
17.2 Exemple de parcours en profondeur au départ de a : a → b → c → d → g → h → e → f	147
17.3 Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijkstra.	152
17.4 Graphe pondéré et orienté à valeurs positives et négatives pour l'application de l'algorithme de Bellman-Ford.	156
17.5 Exemple de graphe orienté et pondéré pour expliquer le concept de matrice d'adjacence.	157
18.1 Représentation d'un tableau statique en mémoire. Il peut représenter un tableau t de cinq entiers codés sur huit bits. On accède directement à l'élément i en écrivant $t[i]$	166
18.2 Représentation d'une liste simplement chaînée d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.	168
18.3 Représentation d'une liste doublement chaînée d'entiers L. On conserve un pointeur sur le premier élément et un autre sur le dernier élément de la liste.	169
19.1 Illustration du concept de dictionnaire	172
19.2 Illustration du concept de dictionnaire, ensembles concrets	172
19.3 Illustration de l'implémentation d'un dictionnaire par table de hachage. La fonction de hachage h permet de calculer les indices du tableau. La valeur a associée à α se trouve à la case $h(\alpha)$ du tableau. Toutes les clefs n'ont pas forcément de valeur associée à un moment donnée de l'algorithme. Dans ce cas, à l'indice associé à cette clef, le tableau est vide.	178
19.4 Illustration de l'implémentation d'un dictionnaire par table de hachage avec chaînage. Les clefs β et ϕ , engendrent des collisions. On stocke donc les valeurs possibles pour un même code (indice) dans une liste.	181
20.1 Schématisation des différentes approches des algorithmes de décomposition d'un problème en sous-problèmes : diviser pour régner, approche gloutonne et programmation dynamique itérative.	184
20.2 Illustration du principe d'optimalité et de sous-structure optimale : trouver le plus court chemin dans ce graphe. Les nombres représentent la longueur d'un chemin. Les lignes droites sont des arrêtes. Les lignes ondulées indiquent les plus courts chemins dans le graphe : il faut imaginer qu'on n'a pas représenté tous les sommets.	186
20.3 Programmation dynamique du calcul de $\binom{6}{3}$	187
20.4 Illustration du problème du sac à dos (d'après Wikipedia). On a cinq objets de poids 9, 12, 2, 7 et 5 kg et de valeur 10, 7, 1, 3 et 2 €. Le poids total admissible dans le sac est 15kg.	189
20.5 Formulation récursive du problème du sac à dos.	191
20.6 Schéma de remplissage du tableau pour le problème KP(n, π). Pour calculer $S(i, P)$ on a besoin de $S(i - 1, P)$ et de $S(i - 1, P - p_i)$	192

21.1 Un modèle conceptuel de type entité-association associé à la réalité de l'exemple 59. Les entités sont inscrites dans des rectangles, les attributs des entités dans des ovales et les associations entre les entités dans des losanges. Des cardinalités sont précisées sur les arcs qui relient les entités aux associations	200
21.2 Modèle relationnel construit à partir du modèle conceptuel 21.1. Seules les en-têtes de colonne des relations sont représentées. Les associations du modèle conceptuel ont été intégrées aux tables en ajoutant des colonnes qui contiennent les clefs primaires et les clefs étrangères nécessaires ou en créant une nouvelle table (direction). Les flèches mettent en évidence ces associations.	204
21.3 Traduction d'une association de un à un dans un modèle relationnel. Modèle relationnel spécifiant les tables résidence et orchestre du modèle relationnel de la figure 21.2.	206
21.4 Traduction d'une association de un à plusieurs dans un modèle relationnel. Modèle relationnel spécifiant les tables orchestre et style du modèle relationnel de la figure 21.2.	207
21.5 Traduction d'une association de plusieurs à plusieurs dans un modèle relationnel. Modèle relationnel spécifiant les tables orchestre, direction et chef qui implémente l'association <i>dirige</i> du modèle conceptuel de la figure 21.1.	209
21.6 Modèle physique de la base de données du modèle relationnel 21.2.	210
22.1 Modèle relationnel utilisé pour les exemples de projection et de sélection.	215
22.2 Représentation des étapes de la requête <code>SELECT ville.nom, pays.population FROM ville JOIN pays ON ville.nom_pays = pays.nom WHERE pays.population < 60000000</code> ; du code 22.5 qui opère une jointure entre les tables ville et pays. Étape 1 : les lignes des deux tables sont mises côte à côte d'après le critère de jointure. Étape 2 : seules les lignes qui respecte la condition de sélection sont gardées. Étape 3 : la projection sélectionne les colonnes résultats. à	221
23.1 Visualisation de la dispersion des paramètres sur le jeu de données des manchots répartis sur trois îles et en trois espèces	231
23.2 Résultats de l'apprentissage de l'algorithme KNN sur le jeu de données de l'iris. Les régions de décisions du classificateur sont superposées au valeurs vraies des espèces d'iris.	232
23.3 Illustration de l'algorithme KNN appliqué à la distinction de trois variété d'iris : Setosa, Versicolor et Virginica. Les paramètres d'entrées sont les largeurs et longueurs des pétales et sépals. Les figures représentent certains paramètres du jeu de données l'un en fonction de l'autre. Le résultat de l'algorithme se lit au travers des couleurs. Le résultat est correct dans plus de 90% des cas comme le montre la matrice de confusion.	235
24.1 Arbre de jeu d'une partie de morpion, partie nulle. On considère que les cases sont numérotées de 1 à 9 en ligne et en partant du haut. La position finale est donnée sous l'arbre. Le joueur à la croix X joue en premier car il contrôle la racine de l'arbre.	243

24.2 Jeu de Marienbad avec des allumettes	245
24.3 Modélisation par graphe orienté d'une partie de jeu de Nim avec deux rangées de deux allumettes au départ. On peut jouer dessus avec un pion placé en s au départ. Puis chaque joueur fait avancer le pion d'un saut sur le graphe en sélectionnant un successeur en suivant les arcs. Le joueur qui se trouve en position e a gagné ou perdu dans la variante misère.	245
24.4 Modélisation par graphe orienté biparti d'un jeu de Nim, variante misère : le dernier à jouer a perdu. Les sommets des joueurs 1 et 2 sont distingués par des cercles (1) et des carrés (2). La couleur cyan représente l'attracteur du joueur J_1 : $\mathcal{A}_1 = \{s_2, a_1, b_1, c_1, d_2, e_1\}$. On voit donc que 1 n'a pas intérêt à commencer à jouer dans cette configuration. Il en est de même pour le joueur 2 dont l'attracteur en rouge. C'est normal car la somme de Nim de la configuration initiale est nulle.	246
24.5 Exemples d'arbre minimax. Chaque niveau est contrôlé par un seul joueur, J_{max} en rouge et J_{min} en cyan . La hauteur de l'arbre L est telle que seule une partie de l'arbre de jeu est accessible. Certaines feuilles sont visibles (en orange , c'est l'automne). L'arbre minimax s'achève donc parfois sur des nœuds internes pour lesquels on donne une estimation du gain (non coloré).	253
24.6 Exemples d'arbre minimax complété avec les scores. Chaque niveau est contrôlé par un seul joueur, J_{max} en rouge et J_{min} en cyan	253
24.7 L'élagage $\alpha\beta$ permet sur cet exemple d'éviter l'exploration complète du sous-arbre du nœud numéro 3. En effet, comme c'est un nœud de J_{min} , que le premier nœud du niveau a un score de 1 et que le score calculé du nœud 8 vaut -5, alors on comprend que J_{min} remontera au moins -5 et que J_{max} ne choisira pas ce nœud numéro 3 car ce n'est pas le maximum du niveau.	255
24.8 L'élagage α : supposons que $\mathcal{S}(b)$ ait déjà été calculé par l'algorithme Minimax. Si $\mathcal{S}(b) \geq \mathcal{S}(d)$, alors il est inutile d'explorer le sous-arbre c : J_{max} ne choisira pas le nœud c , il lui préférera b	255
24.9 L'élagage β : supposons que $\mathcal{S}(b)$ ait déjà été calculé par l'algorithme Minimax. Si $\mathcal{S}(b) \leq \mathcal{S}(d)$, alors il est inutile d'explorer le sous-arbre c : J_{min} ne choisira pas le nœud c , il lui préférera b	257

LISTE DES TABLEAUX

5.1 Priorités des opérateurs en Python : dans l'ordre d'apparition, du plus prioritaire (1) au moins prioritaire (13)	31
5.2 Mots-clefs du langage Python	35
8.1 Comparatif des caractéristiques des algorithmes de tri. Ils existent de nombreuses variantes de ces algorithmes. Les informations de ce tableau concernent les versions implémentées en cours ou en TP.	61
8.2 Complexité des différents algorithmes de tri.	61
12.1 Hiérarchie des complexités temporelles de la moins complexe à la plus complexe.	97
12.2 Sur une machine cadencée à 2 Ghz, quelle est la durée prévisible d'exécution d'un algorithme en fonction de la taille des données d'entrée et de sa complexité? On suppose qu'une seule période d'horloge est nécessaire au traitement d'une donnée.	97
12.3 Récurrences et complexités associées utiles et à connaître	107
17.1 Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Dijkstra appliqué au graphe de la figure 17.3	152
17.2 Tableau d des distances au sommet a successivement trouvées au cours de l'algorithme de Bellman-Ford appliqué au graphe de la figure 17.4	155
18.1 Complexité des opérations associées à l'utilisation d'un tableau statique.	166
18.2 Complexité des opérations associées à l'utilisation d'un tableau dynamique.	167
18.3 Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.	168
18.4 Complexité des opérations associées à l'utilisation d'une liste doublement chaînée.	169
18.5 Complexité des opérations associées à l'utilisation des listes et des tableaux.	170
19.1 Complexité des opérations associées à l'utilisation des tables de hachage ou des arbres pour implémenter un TAD dictionnaire. Les coûts indiqués sont dans le pire des cas ou en moyenne.	177
19.2 Probabilité de collision p_c dans l'hypothèse d'une répartition uniforme des valeurs dans le tableau d'une table de hachage. Même dans le cas d'un tableau à un million d'éléments, la probabilité de collision est quasi-certaine dès que la taille des clefs utilisées est supérieure à 2500. C'est le paradoxe des anniversaires.	179

20.1 Triangle de Pascal : valeurs de $\binom{n}{k}$. En couleur apparaissent les éléments nécessaires pour calculer $\binom{6}{3}$ qu'il faut mettre en parallèle de la figure 20.3. On a représenter le triangle du bas vers le haut afin de montrer le lien avec la programmation dynamique.	187
20.2 Synthèse des informations relatives au problème de la figure 20.4.	191
20.3 Tableau de résolution du sac à dos dans le cas de la figure 20.4 donnant les valeurs de $S(i, P)$, avec $i \in \llbracket 0, 5 \rrbracket$ et $P \in \llbracket 0, 15 \rrbracket$	192
22.1 Synthèse des opérateurs SQL au programme	225

LISTE DES CODES

4.1	Un exemple de programme en Python - traduction de l'algorithme produit	15
4.2	Importer un module et l'utiliser	24
4.3	Importer toutes les fonctions d'un module et en utiliser certaines	24
4.4	Importer quelques fonctions d'un module et les utiliser	24
5.1	Types simples	28
5.2	Affectations de type immuable	34
5.3	Affectations de type mutable	34
6.1	Anatomie d'un programme Python (variables globales, fonctions, indentation, blocs, programme principal)	37
6.2	blocs conditionnels	39
6.3	Expression conditionnelle	40
6.4	Boucle inconditionnelle for	40
6.5	Utilisation de range	41
6.6	Boucle conditionnelle while	41
6.7	Fonction Python à un paramètre renvoyant un entier	44
6.8	Procédure Python	44
6.9	Fonction avec paramètre optionnel et plusieurs chemin d'exécution avec valeur retour	44
7.1	Algorithmes simples et incontournables	51
7.2	Passage en paramètre d'un type mutable et d'un type immuable à une fonction .	54
10.1	Créer des tableaux Numpy	76
10.2	Accéder aux éléments d'un tableau numpy	78
10.3	Opérer élément par élément	79
10.4	Calcul matriciel	79
10.5	Types de données numpy	80
21.1	Création du modèle physique d'une base de données	211
22.1	Structure de base d'une requête SQL : projection et sélection	214
22.2	Exemples de projections	216
22.3	Sélections	217
22.4	Sélections - ordre des résultats - projection non redondante	218
22.5	Exemples de jointures	220
22.6	Exemples d'utilisation des fonctions d'agrégation	222
22.7	Exemple d'utilisation de GROUP BY	223
22.8	Exemple d'utilisation de GROUP BY ... HAVING	223
22.9	Exemple d'opération ensembliste	224

22.10 Requêtes imbriquées	224
-------------------------------------	-----

INDEX

Clef primaire, 202

Clef étrangère, 202

correction, 90

invariant de boucle, 91

list

constructeurs, 47

indexing, 50

indicable, 50

itération, 50

opération, 49

slicing, 51

tronçonnage, 51