Algorithmes et Programmation

Cours de Licence 2, Mathématiques et MIASHS, Université Paris Cité

Najib Idrissi

Version du 30 juin 2026

Table des matières

1 Algorithmes, programmes et conventions **5**
1.1 Qu'est-ce qu'un algorithme ? 6
1.2 Conventions du cours 7

2 Correction d'un algorithme **11**
2.1 Invariants de boucle 11
2.2 Un invariant pour une boucle `while` 14

3 Complexité **17**
3.1 Ordres de grandeur : O, Θ, Ω 18
3.2 Complexité d'un algorithme 21
3.3 Optimalité d'une borne de complexité 24

4 Recherche et tris naïfs **27**
4.1 Recherche dans une liste 28
4.2 Tri à bulles 31
4.3 Tri par sélection 34
4.4 Tri par insertion 36

5 La récursivité **41**
5.1 Principe et exemples 42
5.2 La méthode de mémoïsation 47
5.3 De la récursivité à l'itération : la récursion terminale 50
5.4 Récursivité mutuelle ou croisée 54
5.5 Objets définis récursivement : les arbres 57

6 Diviser pour régner **63**
6.1 Fusion de deux listes triées 64
6.2 Tri fusion 67
6.3 Tri rapide 71
6.4 Optimalité : une borne inférieure pour les tris par comparaison 76
6.5 Tri par comptage 79
6.6 Produit rapide de polynômes 82
6.7 Sélection en temps linéaire ★ 87

7 Programmation dynamique **93**
7.1 Une suite définie par une récurrence sur deux indices 94
7.2 Plus longue sous-suite commune : formulation 98

7.3 Plus longue sous-suite commune : algorithme et reconstruction 102
7.4 Le problème du sac à dos 106

8 Algorithmes sur les graphes 111
8.1 Définitions et représentations 112
8.2 Parcours en profondeur 116
8.3 Parcours en largeur et distances 120
8.4 Connexité et composantes connexes 123
8.5 Détection d'un circuit 127
8.6 Tri topologique 130

9 Algorithmes de plus court chemin 135
9.1 Chemins pondérés et lemme de relâchement 136
9.2 L'algorithme de Bellman–Ford 140
9.3 Cas général et détection d'un circuit négatif 144
9.4 L'algorithme de Dijkstra 148
9.5 Une application : les contraintes de différences 152

10 Algorithmes de recherche de motif 157
10.1 Algorithme naïf de recherche de motif 158
10.2 L'algorithme de Karp–Rabin 161
10.3 Recherche par automate 165

11 Les tas ★ 171
11.1 Files de priorité et propriété de tas 172
11.2 Construction d'un tas en temps linéaire 176
11.3 Extraction du maximum et insertion 180
11.4 Tri par tas 185

Index 189

1 Algorithmes, programmes et conventions

Ces notes sont une réécriture d'un cours hérité de Hervé Fournier, François Le Maître et Adrien Brochier, que nous remercions ; les choix de présentation, les erreurs éventuelles et les ajouts sont les nôtres.

Résumé

Ce premier chapitre pose le décor. Nous y précisons ce qu'est un *algorithme*, en quoi il diffère du *programme* qui l'implémente, et quelles sont les deux grandes questions que l'on se posera à son sujet tout au long du cours : est-il *correct*, et quelle est sa *complexité* ? Nous fixons ensuite, une fois pour toutes, les conventions d'écriture du cours — indexation, pseudocode et code Python, opérations de base sur les listes — afin que toutes les sections suivantes parlent le même langage.

Prérequis

Aucune connaissance préalable en algorithmique n'est supposée. On utilisera la notion de fonction et un peu de manipulation de listes, au niveau d'une première initiation à la programmation ; les rappels nécessaires sont faits au fil de l'eau.

Objectifs

À l'issue de ce chapitre, vous saurez distinguer un algorithme de son implémentation, énoncer les deux questions — correction et complexité — qui guident toute l'analyse d'un algorithme, et lire sans ambiguïté le pseudocode et le code Python employés dans la suite du cours, en particulier la convention d'indexation.

Un *algorithme*, c'est d'abord une recette. Pour préparer une mayonnaise, réaliser une division posée, ou retrouver un mot dans un dictionnaire, on suit une suite d'instructions précises : faites ceci, puis cela, et si telle condition est remplie, faites encore autre chose. La recette ne dépend ni de la cuisine où on l'exécute, ni de la personne qui la suit ; elle est, en un sens, une idée indépendante de sa réalisation. C'est exactement le statut d'un algorithme vis-à-vis de l'ordinateur qui le fera tourner.

Mais une recette de cuisine tolère le flou — « salez à votre goût » — là où un ordinateur, lui, ne tolère rien : chaque instruction doit être parfaitement définie, et l'on doit pouvoir affirmer, *avant* même de lancer la machine, que la recette atteindra son but. D'où les deux questions qui parcourent ce cours : comment être *sûr* qu'un algorithme fait bien ce qu'on attend de lui, et combien de temps lui faudra-t-il ? Ce chapitre répond à la première en posant le vocabulaire ; les chapitres suivants outilleront les deux.

1.1 Qu'est-ce qu'un algorithme ?

Lorsqu'on veut résoudre un problème à l'aide d'un ordinateur, il est essentiel de distinguer deux choses que le langage courant confond souvent.

Définition 1.1.1 (Algorithme et programme)**.** *Un* algorithme *est une suite finie d'instructions précises résolvant un problème donné, indépendamment de tout langage de programmation et de toute machine. Un* programme *(ou* implémentation*) est l'écriture d'un algorithme dans un langage particulier, en vue de le faire effectivement exécuter par un ordinateur.*

En d'autres termes, l'algorithme est l'*idée* — la stratégie de résolution — tandis que le programme en est une *réalisation* concrète. Un même algorithme peut être implémenté dans des dizaines de langages différents, sur des machines différentes ; il reste le même algorithme. Cette indépendance n'a rien d'anecdotique : c'est elle qui permet d'*étudier* un algorithme de façon mathématique, sans jamais allumer un ordinateur.

Remarque 1.1.2. *La notion d'algorithme précède l'ordinateur de très loin. L'algorithme d'Euclide pour calculer le plus grand commun diviseur de deux entiers date du* IIIe *siècle avant notre ère. Le mot lui-même vient du nom du mathématicien perse al-Khwârizmî (*IXe *siècle), dont le traité a aussi donné le mot « algèbre » ; on calculait donc, et l'on raisonnait sur les calculs, bien avant d'avoir des machines pour les exécuter.*

Dans ce cours, nous nous intéressons aux algorithmes eux-mêmes, c'est-à-dire au versant mathématique de la question. Les algorithmes seront implémentés — en Python — et l'on pourra les exécuter ; mais le code est là pour *illustrer* et *vérifier* les idées, non pour viser l'efficacité maximale d'un programmeur professionnel. Nous ignorerons délibérément les nombreuses subtilités d'une implémentation optimisée (gestion fine de la mémoire, particularités du langage), qui relèvent d'un autre métier.

1.1.1 Les deux questions

Sur un algorithme donné, deux questions se posent invariablement, et ce sont elles qui structurent tout le cours.

1. **Est-il correct ?** Autrement dit : fait-il réellement ce qu'on attend de lui, sur *toutes* les entrées possibles ? Se convaincre qu'un algorithme est correct ne se fait pas en le testant sur quelques exemples — il y en a en général une infinité — mais en le *démontrant*. C'est l'objet du chapitre 2, où nous introduirons l'outil central des *invariants de boucle*.
2. **Quelle est sa complexité ?** Autrement dit : de combien d'étapes élémentaires a-t-il besoin pour répondre, en fonction de la taille de l'entrée ? Cette quantité gouverne le temps de calcul. C'est l'objet du chapitre 3, consacré à la *complexité*, où nous apprendrons à mesurer et à comparer les coûts.

Ces deux questions sont *purement mathématiques* : elles ont une réponse qui ne dépend ni de la marque de l'ordinateur, ni de la rapidité du langage, ni de la quantité de mémoire

disponible. Le temps réellement mis par une machine dépend, lui, de tous ces facteurs ; mais la complexité, telle que nous la définirons, en capture l'essentiel — la façon dont le coût *croît* quand l'entrée grandit — indépendamment de la machine.

Avertissement 1.1.3. *Tester un programme sur quelques exemples ne prouve pas qu'il est correct ! Un algorithme peut donner le bon résultat sur mille cas et se tromper sur le mille-et-unième. Les tests* augmentent la confiance *; seule une démonstration établit la* correction. *Nous reviendrons longuement sur cette distinction au chapitre 2.*

1.1.2 Exercices

Exercice 1.1.4. *Décrire, sous forme d'une suite d'instructions précises, l'algorithme que vous suivez pour chercher un mot dans un dictionnaire papier. Votre description dépend-elle de la langue du dictionnaire ? du fait qu'il soit en papier ou électronique ? Qu'est-ce qui, dans votre description, relève de l'*algorithme *et qu'est-ce qui relèverait d'une* implémentation *?*

Exercice 1.1.5. *Pour chacune des phrases suivantes, dire si elle décrit plutôt un algorithme ou plutôt un programme, et justifier : (a) « parcourir la liste de gauche à droite en retenant le plus grand élément vu jusque-là » ; (b) « la fonction Python* `max(L)` *» ; (c) « la méthode de la division euclidienne apprise à l'école ».*

Exercice 1.1.6. *Donner un exemple, tiré de la vie courante, d'une méthode qui résout correctement un problème mais qu'on jugerait trop lente pour être utilisable en pratique. Cet exemple illustre-t-il que correction et complexité sont deux questions* indépendantes *?*

1.2 Conventions du cours

Avant d'entrer dans le vif du sujet, fixons une fois pour toutes la façon dont les algorithmes seront écrits et les listes manipulées dans ce cours. Ces conventions peuvent sembler des détails, mais une indexation mal fixée ou une primitive ambiguë est une source classique d'erreurs ; les arrêter maintenant nous évitera bien des confusions par la suite.

1.2.1 Listes et indexation

La plupart des algorithmes du cours manipulent des *listes* (on dit aussi *tableaux* ; voir la remarque 1.2.2 sur la nuance entre ces deux mots). Une liste L de *longueur* n est une suite finie de n valeurs. La grande question est de savoir comment numéroter ses cases.

Convention 1.2.1 (Indexation à partir de zéro)**.** *Dans tout le cours, les listes sont indexées* à partir de 0*. Une liste L de longueur n a donc ses cases numérotées de* 0 *à* $n-1$ *: ses éléments sont*

$$L[0],\ L[1],\ \ldots,\ L[n-1].$$

Le premier élément est $L[0]$, le dernier est $L[n-1]$, et $L[n]$ n'existe pas*.*

Concrètement, pour parcourir tous les éléments de L, l'indice i va de 0 *inclus* à n *exclu*. C'est la convention du langage Python, que nous emploierons ; une partie de la littérature mathématique numérote plutôt à partir de 1, et nous signalerons cette divergence le cas échéant, mais nous ne nous en servirons *jamais* dans ce cours. Fixer ce choix une fois pour toutes élimine une source d'erreurs récurrente, les fameux décalages d'indice (« à un près »).

On notera $n = \texttt{len}(L)$ la longueur de la liste, et l'on accède à la case d'indice i par $L[i]$. On utilisera aussi les *tranches* : pour $0 \leqslant a \leqslant b \leqslant n$, la tranche $L[a:b]$ est la sous-liste formée des éléments d'indices $a, a+1, \ldots, b-1$ — l'indice de gauche est inclus, celui de droite est exclu. Ainsi $L[a:b]$ a exactement $b-a$ éléments ; en particulier $L[:m]$ désigne les m premiers éléments (indices 0 à $m-1$) et $L[m:]$ les suivants (indices m à $n-1$).

1.2.2 Du pseudocode au code Python

Traditionnellement, un algorithme se décrit en *pseudocode* : un mélange de langage courant et de notation mathématique, assez précis pour être sans ambiguïté, mais sans les contraintes d'un vrai langage. Nous prenons ici un parti légèrement différent, conforme à l'esprit du cours : nos algorithmes sont écrits directement en *Python*, mais dans un style volontairement épuré, aussi proche que possible du pseudocode. Le lecteur n'a donc pas à apprendre deux notations, et tout algorithme du cours est un programme que l'on peut réellement exécuter.

Voici, à titre d'exemple, la fonction la plus simple qui soit : la somme des éléments d'une liste.

```
def somme(L):
    """Renvoie la somme des éléments de la liste ``L``.

    >>> somme([3, 1, 4, 1, 5])
    14
    >>> somme([])
    0
    >>> somme([42])
    42
    """
    s = 0
    for i in range(len(L)):
        s = s + L[i]
    return s
```

Détaillons ce que cet exemple fixe, car tous les algorithmes du cours suivront le même moule :

- un algorithme est une *fonction*, introduite par `def`, prenant ses données en arguments et renvoyant son résultat par `return` ;
- l'affectation se note avec le signe = : `s = s + L[i]` *donne* à `s` la valeur `s + L[i]` (ce n'est pas une égalité mathématique !) ;

— la boucle `for i in range(len(L))` fait parcourir à i les entiers de 0 à $n-1$, conformément à la convention 1.2.1 ;
— le texte entre triples guillemets est la *spécification* de la fonction, suivie d'exemples d'exécution (après les chevrons `>>>`). [1]

1.2.3 Opérations de base sur les listes

Nous nous autoriserons quelques *opérations élémentaires* sur les listes, que nous considérerons comme primitives — c'est-à-dire admises sans les réimplémenter. Les voici, avec la notation *unique* retenue pour tout le cours :

Opération	Notation
longueur de L	`len(L)`
accès à l'élément d'indice i	`L[i]`
modification de l'élément d'indice i	`L[i] = v`
ajout d'un élément x à la fin	`L.append(x)`
tranche d'indices a (inclus) à b (exclu)	`L[a:b]`
échange de deux éléments	`L[i], L[j] = L[j], L[i]`
liste de n zéros	`[0] * n`

Chacune de ces opérations s'effectue en temps constant, à l'exception de la création d'une liste de n éléments et de la tranche $L[a:b]$, qui coûtent un temps proportionnel au nombre d'éléments produits — nous y reviendrons quand nous analyserons les complexités. Toute autre opération un peu élaborée sera, elle, écrite explicitement comme un algorithme.

Remarque 1.2.2 (« Liste » ou « tableau » ?)**.** *En toute rigueur, « liste » et « tableau » ne désignent pas la même chose en informatique. Un* tableau *(*array*) est une zone de mémoire de taille* fixe *aux cases contiguës : on y accède à n'importe quel élément $L[i]$ immédiatement, en temps constant, mais on ne peut pas l'agrandir. Une* liste *au sens strict — typiquement une* liste chaînée *— relie au contraire ses éléments les uns aux autres : on peut l'allonger à volonté, mais atteindre son i-ième élément oblige à la parcourir depuis le début. Les deux structures n'ont donc pas les mêmes coûts.*

Le type `list` de Python réalise un compromis entre les deux : c'est un tableau de taille dynamique*, qui offre* à la fois *l'accès indexé immédiat d'un tableau* et *la possibilité d'ajouter des éléments à la fin (`L.append`) comme une liste. C'est cette structure-là que nous utiliserons partout ; dans tout le cours, nous dirons « liste » pour la désigner et « tableau » comme synonyme commode, sans plus les distinguer. Lorsqu'une complexité dépendra finement de la structure réellement employée, nous le signalerons.*

1.2.4 Exercices

Exercice 1.2.3. *Soit L une liste de longueur 7, indexée selon la convention 1.2.1. Quel est l'indice de son premier élément ? de son dernier ? Combien d'éléments contient la tranche $L[2:5]$, et lesquels ? Que vaut $L[:3]$ suivi de $L[3:]$?*

1. Le module Python `doctest` permet de vérifier que les exemples sont corrects automatiquement.

Exercice 1.2.4. *Réécrire la fonction* `somme` *en parcourant directement les* éléments *de la liste (boucle* `for x in L`*) plutôt que ses indices. Vérifier qu'elle donne le même résultat. Laquelle des deux versions vous semble la plus proche du pseudocode ?*

Exercice 1.2.5. *On veut une fonction qui renvoie l'*avant-dernier *élément d'une liste L de longueur $n \geqslant 2$. Un étudiant propose* `return L[n-1]`*. Pourquoi est-ce faux, et quelle est la bonne expression ? Cet exercice illustre l'erreur « à un près » que la convention 1.2.1 vise à dissiper.*

2 Correction d’un algorithme

Résumé

Comment être *sûr* qu’un algorithme fait ce qu’on attend de lui ? Ce chapitre répond à cette question — celle de la *correction* — au moyen d’un outil central, l’*invariant de boucle* : une propriété vraie à chaque tour de boucle, qu’on établit par récurrence et qu’on exploite à la sortie de la boucle pour conclure. Nous le mettons en œuvre sur deux exemples, une boucle `for` (recherche du minimum) et une boucle `while` (calcul sur une suite récurrente), ce second cas soulevant en prime la question de la *terminaison*.

Prérequis

Le raisonnement par récurrence, au niveau de la première année. Les conventions du chapitre 1 (indexation à partir de 0, lecture du code Python).

Objectifs

À l’issue de ce chapitre, vous saurez énoncer un invariant de boucle adapté à un algorithme donné, le démontrer par récurrence sur le nombre d’itérations, et vous en servir pour prouver qu’une boucle produit le résultat voulu ; vous saurez aussi distinguer correction et terminaison pour une boucle `while`.

Au chapitre précédent, nous avons dégagé les deux questions que pose tout algorithme : est-il correct, et quelle est sa complexité ? Ce chapitre s’attaque à la première. Démontrer qu’un algorithme est *correct*, c’est établir qu’il renvoie le bon résultat sur *toutes* les entrées — pas seulement sur celles qu’on a essayées. Or un algorithme est essentiellement fait de boucles, et c’est là que réside la difficulté : une boucle peut tourner un nombre arbitraire de fois, et il faut un argument qui vaille quel que soit ce nombre. L’outil taillé pour cela est l’invariant de boucle, et le raisonnement qui le sous-tend est une vieille connaissance : la récurrence.

2.1 Invariants de boucle

Prenons un exemple que vous avez sans doute déjà rencontré : la recherche du plus petit élément d’une liste non vide. On parcourt la liste en gardant à tout instant, dans une variable m, le plus petit élément vu jusque-là.

```
def minimum(L):
    """Renvoie le plus petit élément de la liste non vide ``L``.
```

```
    >>> minimum([4, 2, 7, 1, 9])
    1
    >>> minimum([42])
    42
    >>> minimum([-3, -1, -7])
    -7
    >>> import random
    >>> all(minimum(L) == min(L)
    ...     for L in ([random.randint(-20, 20) for _ in range(n)]
    ...               for n in range(1, 60)))
    True
    """
    n = len(L)
    m = L[0]
    for i in range(1, n):
        if L[i] < m:
            m = L[i]
    return m
```

La fonction renvoie la variable m. Pour prouver qu'elle est correcte, il faut donc montrer qu'à la sortie de la boucle, m vaut bien le minimum de la liste L entière. La variable m n'est modifiée qu'à deux endroits : avant la boucle, où elle reçoit $L[0]$, et à l'intérieur de la boucle, où elle peut recevoir $L[i]$. Comme cette seconde affectation a lieu dans une boucle exécutée un nombre variable de fois, c'est une récurrence qu'il nous faut.

Définition 2.1.1 (Invariant de boucle)**.** *Un* invariant de boucle *est une propriété qui est vraie après chaque itération de la boucle (et déjà avant la première).*

En d'autres termes, c'est une propriété que la boucle *préserve* : si elle est vraie au début d'une itération, elle l'est encore à la fin. La méthode est alors toujours la même, en trois temps :

1. *énoncer* un invariant — une propriété qui capture « où en est » le calcul après chaque tour ;
2. le *démontrer*, par récurrence sur le nombre d'itérations (initialisation : il est vrai avant la boucle ; hérédité : chaque itération le préserve) ;
3. l'*exploiter* à la sortie de la boucle : en spécialisant l'invariant à l'état où la boucle s'arrête, on lit le résultat cherché.

C'est cette dernière étape qui justifie tout l'intérêt de la démarche ; tout l'art consiste à choisir un invariant à la fois *vrai* et *assez fort* pour qu'à la sortie il donne exactement ce qu'on veut. Illustrons-la sur la recherche du minimum, en présentant *deux* rédactions du même invariant : la première, plus proche des récurrences auxquelles vous êtes habitués, fait apparaître un compteur d'itérations explicite ; la seconde, plus économe, ne parle que des variables du programme. Les deux sont correctes et nous les conserverons toutes les deux, car comparer leurs styles est instructif.

2.1.1 Première version : avec un compteur d'itérations

Numérotons les itérations : l'itération k (pour k de 1 à $n-1$) est celle qui traite l'indice $i = k$. Convenons que « après l'itération 0 » désigne l'état juste avant d'entrer dans la boucle.

Invariant 2.1.2 (de la recherche du minimum, première version)**.** *Pour tout $k \in \{0, 1, \ldots, n-1\}$, après l'itération k de la boucle, la variable m est égale au minimum de $\{L[0], L[1], \ldots, L[k]\}$.*

Démonstration. Récurrence sur k.

Initialisation ($k = 0$). Avant d'entrer dans la boucle, on a exécuté l'affectation $m = L[0]$. Donc $m = L[0]$, qui est bien le minimum de l'ensemble à un seul élément $\{L[0]\}$.

Hérédité. Soit $k \in \{0, \ldots, n-2\}$; supposons l'invariant vrai après l'itération k, c'est-à-dire $m = \min\{L[0], \ldots, L[k]\}$, et considérons l'itération $k+1$, qui traite l'indice $i = k+1$. Deux cas selon le test $L[k+1] < m$:

— si $L[k+1] < m$, l'affectation $m = L[k+1]$ a lieu ; comme on avait $m = \min\{L[0], \ldots, L[k]\}$ et que $L[k+1]$ est plus petit que ce minimum, $L[k+1]$ est le minimum de $\{L[0], \ldots, L[k+1]\}$, et c'est bien la nouvelle valeur de m ;

— si $L[k+1] \geqslant m$, la variable m n'est pas modifiée ; or $m = \min\{L[0], \ldots, L[k]\}$ est alors $\leqslant L[k+1]$, donc reste le minimum de $\{L[0], \ldots, L[k+1]\}$.

Dans les deux cas, après l'itération $k+1$, on a $m = \min\{L[0], \ldots, L[k+1]\}$. L'invariant est donc vrai pour tout $k \in \{0, \ldots, n-1\}$. □

Exploitation à la sortie. La boucle s'arrête après avoir traité le dernier indice, c'est-à-dire après l'itération $k = n-1$. L'invariant donne alors $m = \min\{L[0], \ldots, L[n-1]\}$: m est le minimum de la liste L tout entière, et c'est cette valeur que la fonction renvoie. L'algorithme est correct.

2.1.2 Seconde version : sans compteur

On peut éviter le compteur k et formuler l'invariant directement avec la variable de boucle i. C'est la rédaction que nous privilégierons en général : plus légère, elle parle le langage du programme.

Invariant 2.1.3 (de la recherche du minimum, seconde version)**.** *Après chaque itération de la boucle, la variable m est égale au minimum de $\{L[0], L[1], \ldots, L[i]\}$, où i désigne la valeur courante de la variable de boucle.*

Démonstration. Récurrence sur le numéro de l'itération.

Initialisation. À la première itération, i vaut 1. Avant elle, on avait $m = L[0]$; l'itération teste $L[1] < m$ et remplace m par $L[1]$ si c'est le cas. Dans les deux cas, m devient le plus petit de $L[0]$ et $L[1]$, soit $m = \min\{L[0], L[1]\}$ — l'invariant pour $i = 1$.

Hérédité. Supposons qu'après une certaine itération on ait $m = \min\{L[0], \ldots, L[i]\}$, et que la boucle effectue une itération de plus. La variable de boucle passe alors de i à $i+1$, et l'itération teste $L[i+1] < m$:

— si $L[i+1] < m$, alors m reçoit $L[i+1]$, qui est plus petit que $\min\{L[0], \ldots, L[i]\}$, donc égal à $\min\{L[0], \ldots, L[i+1]\}$;
— sinon $L[i+1] \geqslant m = \min\{L[0], \ldots, L[i]\}$, et m, inchangé, reste le minimum de $\{L[0], \ldots, L[i+1]\}$.

Après cette itération, on a donc $m = \min\{L[0], \ldots, L[i+1]\}$: l'invariant est préservé. □

Exploitation à la sortie. La boucle se termine une fois la dernière valeur $i = n-1$ traitée ; l'invariant donne $m = \min\{L[0], \ldots, L[n-1]\}$, le minimum de L. On conclut comme précédemment.

Remarque 2.1.4. *Les deux versions énoncent* la même *propriété de la boucle ; seule change la façon de la repérer (par un compteur k, ou par la variable i elle-même). La première colle au schéma habituel d'une récurrence sur un entier ; la seconde évite d'introduire une variable supplémentaire et se lit plus directement sur le code. Choisir l'une ou l'autre est affaire de goût et de commodité.*

Avertissement 2.1.5. *Cet algorithme suppose la liste* non vide *: la toute première affectation $m = L[0]$ n'a pas de sens si L est vide, et la notion même de minimum d'un ensemble vide n'en a pas non plus. C'est une* précondition *de la fonction : une hypothèse sur l'entrée, sans laquelle l'énoncé de correction ne tient pas.*

2.1.3 Exercices

Exercice 2.1.6. *Dérouler l'algorithme sur la liste $L = [4, 2, 7, 1, 9]$ en donnant, après chaque itération, les valeurs de i et de m. Vérifier que l'invariant 2.1.3 est satisfait à chaque tour.*

Exercice 2.1.7. *Modifier l'algorithme pour qu'il renvoie le* maximum *d'une liste non vide. Énoncer l'invariant correspondant (sur le modèle de la version 2.1.3) et adapter sa preuve.*

Exercice 2.1.8. *On veut maintenant la* position *d'un plus petit élément, c'est-à-dire un indice j tel que $L[j]$ soit minimal. Écrire la fonction correspondante. Quel invariant satisfait la variable j après chaque itération ? (On prendra garde au cas où le minimum est atteint plusieurs fois.)*

Exercice 2.1.9. *Un étudiant initialise m à 0 au lieu de $L[0]$ (puis parcourt toute la liste, de l'indice 0 à $n-1$). Sur quelles listes sa fonction renvoie-t-elle malgré tout le bon résultat, et sur lesquelles se trompe-t-elle ? Quel invariant est mis en défaut ?*

2.2 Un invariant pour une boucle `while`

Dans la section précédente, la boucle `for` parcourait une plage d'indices connue d'avance : on savait qu'elle ferait exactement $n-1$ tours. Une boucle `while`, elle, tourne *tant qu'*une condition reste vraie, sans qu'on sache à l'avance combien de fois. Cela ne

change rien à la méthode de l'invariant, mais soulève une question nouvelle, que la boucle `for` réglait toute seule : celle de la *terminaison*.

Considérons la suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$u_0 = 0 \qquad \text{et} \qquad u_{n+1} = 1 + u_n^2 \quad (n \in \mathbb{N}),$$

dont les premiers termes sont $0, 1, 2, 5, 26, 677, \ldots$ Étant donné un entier A, on cherche le premier indice n tel que $u_n \geqslant A$. L'algorithme calcule les termes de la suite l'un après l'autre jusqu'à atteindre le seuil.

```
def premier_n(A):
    """Renvoie le premier entier ``n`` tel que ``u_n >= A``.

    où ``u_0 = 0`` et ``u_{n+1} = 1 + u_n ** 2`` (donc
    ``u = 0, 1, 2, 5, 26, 677, ...``).

    >>> premier_n(0)
    0
    >>> premier_n(1)
    1
    >>> premier_n(2)
    2
    >>> premier_n(5)
    3
    >>> premier_n(6)
    4
    >>> premier_n(677)
    5
    """
    n = 0
    u = 0
    while u < A:
        n = n + 1
        u = 1 + u * u
    return n
```

À chaque tour, n compte les itérations et u contient le terme courant de la suite. Énonçons-le comme un invariant.

Invariant 2.2.1 (de la boucle de calcul de la suite)**.** *Pour tout $k \in \mathbb{N}$, après l'itération k de la boucle `while`, on a $u = u_k$ et $n = k$ (l'itération 0 désignant l'état juste avant la boucle).*

Démonstration. Récurrence sur k.

Initialisation ($k = 0$). Avant d'entrer dans la boucle, les affectations $n = 0$ et $u = 0$ ont eu lieu. On a donc $n = 0$ et $u = 0 = u_0$.

Hérédité. Supposons qu'après l'itération k on ait $u = u_k$ et $n = k$, et que la boucle effectue une itération de plus. Celle-ci exécute $n = n + 1$, donc n devient $k + 1$; puis $u = 1 + u^2$, donc u devient $1 + u_k^2 = u_{k+1}$. Après l'itération $k + 1$, on a bien $u = u_{k+1}$ et $n = k + 1$. □

Cet invariant nous dit *ce que valent* les variables à chaque tour, mais pas que la boucle s'arrête un jour. Il faut l'établir séparément.

Proposition 2.2.2 (Terminaison)**.** *Pour toute entrée A, la boucle* `while` *s'arrête après un nombre fini d'itérations.*

Démonstration. Tous les termes u_k sont des entiers positifs (récurrence immédiate à partir de $u_0 = 0$), et la suite est strictement croissante : pour un entier $u_k \geqslant 0$, on a $u_k^2 \geqslant u_k$, donc $u_{k+1} = 1 + u_k^2 \geqslant 1 + u_k > u_k$. Comme elle croît ainsi d'au moins 1 à chaque pas, on a $u_k \geqslant k$ pour tout k ; en particulier u_k finit par dépasser n'importe quel seuil A. Il existe donc un entier k tel que $u_k \geqslant A$, et la condition $u < A$ de la boucle devient fausse : la boucle s'arrête. □

Exploitation à la sortie. La boucle s'arrête à la première itération au terme de laquelle $u \geqslant A$. D'après l'invariant 2.2.1, si cela se produit à l'itération k, alors $u = u_k \geqslant A$ et $n = k$, et k est le *premier* indice tel que $u_k \geqslant A$ — car aux itérations précédentes la condition $u < A$ était encore vraie, c'est-à-dire $u_j < A$ pour $j < k$. La fonction renvoie $n = k$: c'est bien le premier indice cherché. L'algorithme est correct.

Remarque 2.2.3. *Nous avons détaillé ici la preuve par récurrence de l'invariant pour fixer la méthode. Dans les cas simples comme celui-ci, on se contentera souvent, par la suite, d'*énoncer *l'invariant sans rédiger sa récurrence — pourvu qu'elle soit immédiate. La terminaison, en revanche, mérite toujours qu'on s'assure qu'elle a bien lieu : une boucle* `while` *mal pensée peut tourner indéfiniment.*

2.2.1 Exercices

Exercice 2.2.4. *En s'inspirant de la seconde version de l'invariant de la section 2.1, démontrer la correction du même algorithme à l'aide de l'invariant suivant, qui ne mentionne que la variable u :* après chaque itération de la boucle, $u = u_n$ *(où n est la valeur courante de la variable n).*

Exercice 2.2.5. *Modifier la suite en $u_0 = 5$ et $u_{n+1} = u_n$ (suite constante). Pour quelles valeurs de A la boucle* `while` *de* `premier_n` *ne s'arrête-t-elle* jamais *? Cet exemple montre que la terminaison n'a rien d'automatique et doit se démontrer.*

Exercice 2.2.6. *Écrire, avec une boucle* `while`*, une fonction qui prend un entier A et renvoie le plus petit entier n tel que $2^0 + 2^1 + \cdots + 2^n \geqslant A$. Énoncer un invariant reliant, après chaque itération, la valeur d'une variable accumulant la somme à l'indice courant, et justifier la terminaison.*

3 Complexité

Résumé

Ce chapitre répond à la seconde grande question que l'on se pose sur un algorithme : combien de temps prend-il ? Nous mettons d'abord en place le langage qui sert à comparer la vitesse de croissance de deux fonctions sans s'encombrer des détails de la machine — les notations O, Θ et Ω. Nous définissons ensuite la *complexité* d'un algorithme comme son nombre d'opérations dans le *pire cas*, en fonction de la taille de l'entrée, et nous l'analysons sur quelques exemples. Le chapitre se termine sur une mise au point : qu'affirme-t-on *exactement* lorsqu'on dit qu'une borne de complexité est optimale ?

Prérequis

La notion de fonction de $\mathbb{N}$ dans $\mathbb{R}$, la comparaison asymptotique de suites et de fonctions au voisinage de $+\infty$, et les fonctions usuelles — logarithme, exponentielle, puissances — vues en première année. La notion d'algorithme et de boucle (chapitres 1 et 2).

Objectifs

À l'issue de ce chapitre, vous saurez manier les notations O, Θ et Ω et comparer entre eux les ordres de grandeur courants ; vous saurez analyser un algorithme simple pour majorer son nombre d'opérations dans le pire cas ; et vous saurez ce que signifie — et ce que ne signifie pas — l'affirmation qu'une telle borne est optimale.

Au chapitre précédent, nous avons appris à nous convaincre qu'un algorithme est *correct*, c'est-à-dire qu'il calcule bien ce qu'on attend de lui. Reste la seconde question du cours, tout aussi importante en pratique : *combien de temps lui faut-il ?* Un algorithme correct mais trop lent pour terminer avant la fin de l'univers n'est d'aucune utilité.

Comment comparer deux algorithmes de ce point de vue ? On pourrait les programmer, les lancer sur la même machine et chronométrer ; mais le résultat dépendrait alors de mille détails — le langage, le processeur, la qualité du compilateur — qui n'ont rien à voir avec l'algorithme lui-même. Pour comparer deux *itinéraires*, on ne regarde pas la marque de la voiture : on compte les kilomètres. De même, pour comparer deux algorithmes, on compte le nombre d'*opérations élémentaires* qu'ils effectuent, en fonction de la taille de l'entrée.

Et ce qui compte vraiment, c'est la *manière dont ce nombre croît* quand l'entrée grandit. Un algorithme qui traite une liste de n éléments en n^2 opérations et un autre qui la traite en $100\,n$ opérations : pour $n = 10$, le second est dix fois plus lent ; mais dès $n = 10^6$, le

premier demande 10^{12} opérations contre 10^8 pour le second — dix mille fois plus. Passé une certaine taille, c'est toujours la croissance qui l'emporte, jamais les constantes. C'est elle que l'on cherche à capturer, et c'est l'objet de la première section.

3.1 Ordres de grandeur : O, Θ, Ω

La complexité d'un algorithme s'exprimera comme une fonction de la taille n de l'entrée : le nombre d'opérations qu'il effectue. Avant même de parler d'algorithmes, donnons-nous le vocabulaire qui permet de comparer la croissance de deux telles fonctions en négligeant les constantes et le comportement sur les petites entrées. Ces fonctions comptent des opérations : elles sont définies sur $\mathbb{N}$ et à valeurs positives. C'est dans ce cadre que nous travaillons.

Définition 3.1.1 (Domination : la notation O)**.** *Soient $f, g : \mathbb{N} \to \mathbb{R}_+$ deux fonctions à valeurs positives. On dit que f est* dominée *par g, et l'on écrit*

$$f = O(g)$$

(lire « f est un grand O de g »), s'il existe une constante $C > 0$ et un rang $n_0 \in \mathbb{N}$ tels que

$$f(n) \leqslant C\, g(n) \qquad \text{pour tout } n \geqslant n_0.$$

En d'autres termes, $f = O(g)$ signifie qu'*à partir d'un certain rang*, f ne dépasse pas g *à une constante multiplicative près*. Les deux libertés sont essentielles : on s'autorise à ignorer ce qui se passe sur les petites entrées (le rang n_0) et à gonfler g d'un facteur constant (la constante C). Par exemple, écrire $f = O(n)$ — où n désigne ici la fonction $n \mapsto n$ — veut dire qu'il existe $C > 0$ et n_0 tels que $f(n) \leqslant C\, n$ dès que $n \geqslant n_0$; et $f = O(n^2)$, qu'il existe $C > 0$ et n_0 tels que $f(n) \leqslant C\, n^2$ dès que $n \geqslant n_0$.[1]

Avertissement 3.1.2 ($f = O(g)$ n'est pas une égalité)**.** *Le signe « = » dans $f = O(g)$ est un* abus de notation *consacré par l'usage : ce n'est pas une égalité, et on ne peut pas la « retourner ». On écrit $f = O(g)$, jamais $O(g) = f$, et de $f = O(g)$ on ne déduit* nullement *$g = O(f)$. Il faut lire $O(g)$ comme « l'une des fonctions dominées par g », et le symbole « = » comme « est ». Nous verrons juste après quelle notation exprime, elle, une domination dans les deux sens.*

Cette dissymétrie invite à introduire les deux notations qui complètent le tableau : l'une pour minorer, l'autre pour encadrer.

Définition 3.1.3 (Minoration Ω et encadrement Θ)**.** *Soient $f, g : \mathbb{N} \to \mathbb{R}_+$. On dit que :*

— *$f = \Omega(g)$ (« f est un grand* oméga *de g ») s'il existe $c > 0$ et $n_0 \in \mathbb{N}$ tels que $f(n) \geqslant c\, g(n)$ pour tout $n \geqslant n_0$; autrement dit, f minore g à une constante près, à partir d'un certain rang ;*

1. Il existe une notation voisine, le « petit o » $f = o(g)$, plus exigeante (elle demande que le rapport $f(n)/g(n)$ tende vers 0). Nous ne l'utiliserons pas dans ce cours ; ne confondez pas les deux.

— $f = \Theta(g)$ *(« f est un grand* thêta *de g ») si l'on a* à la fois $f = O(g)$ *et* $f = \Omega(g)$.

Concrètement, $f = \Omega(g)$ est le pendant inférieur de $f = O(g)$: on encadre f par en dessous au lieu de l'encadrer par au-dessus. Et $f = \Theta(g)$ dit que f et g croissent *au même rythme* : il existe deux constantes $0 < c \leqslant C$ et un rang n_0 tels que

$$c\,g(n) \leqslant f(n) \leqslant C\,g(n) \qquad \text{pour tout } n \geqslant n_0.$$

La notation Θ est la bonne lorsqu'on veut dire que l'on connaît la croissance de f *exactement* (à constante près), et pas seulement une borne supérieure.

Les trois notations sont reliées par une dualité simple, qu'il faut avoir en tête.

Proposition 3.1.4 (Dualité O/Ω)**.** *Pour toutes fonctions $f, g : \mathbb{N} \to \mathbb{R}_+$, on a $f = \Omega(g)$ si et seulement si $g = O(f)$.*

Démonstration. Supposons $f = \Omega(g)$: il existe $c > 0$ et n_0 tels que $f(n) \geqslant c\,g(n)$ pour $n \geqslant n_0$. En divisant par $c > 0$, on obtient $g(n) \leqslant (1/c)\,f(n)$ pour $n \geqslant n_0$, c'est-à-dire $g = O(f)$ avec la constante $C = 1/c$. La réciproque s'obtient de la même façon en échangeant les rôles de f et g. □

En particulier $f = \Theta(g)$ équivaut à $g = \Theta(f)$: *la relation Θ, elle, est symétrique*, contrairement à O et Ω. C'est exactement la différence que pointait l'avertissement 3.1.2.

Exemple 3.1.5 (O peut être large, Θ est serré)**.** *Prenons $f(n) = 3n + 5$. D'une part $f = \Theta(n)$: pour tout $n \geqslant 1$ on a $3n \leqslant 3n + 5 \leqslant 8n$, ce qui donne l'encadrement avec $c = 3$, $C = 8$ et $n_0 = 1$. D'autre part $f = O(n^2)$: dès $n \geqslant 1$, $3n + 5 \leqslant 8n \leqslant 8n^2$. Mais $f \neq \Theta(n^2)$: aucune constante $c > 0$ ne vérifie $3n + 5 \geqslant c\,n^2$ pour n grand, car alors $c \leqslant (3n + 5)/n^2 \to 0$. Ainsi $O(n^2)$ est une borne supérieure* correcte mais large *pour f, tandis que $\Theta(n)$ en capture la croissance exacte. La morale : O majore, Θ situe.*

3.1.1 Quelques règles de calcul

La notation O se manipule à l'aide de quelques règles qui permettront, plus loin, de déduire la complexité d'un algorithme de celle de ses parties.

Proposition 3.1.6 (Transitivité)**.** *Soient $f, g, h : \mathbb{N} \to \mathbb{R}_+$. Si $f = O(g)$ et $g = O(h)$, alors $f = O(h)$.*

Démonstration. Soient $C_1 > 0$ et n_1 tels que $f(n) \leqslant C_1\,g(n)$ pour $n \geqslant n_1$, et $C_2 > 0$ et n_2 tels que $g(n) \leqslant C_2\,h(n)$ pour $n \geqslant n_2$. Alors, pour tout $n \geqslant \max(n_1, n_2)$, on a $f(n) \leqslant C_1\,g(n) \leqslant C_1 C_2\,h(n)$, donc $f = O(h)$ avec la constante $C_1 C_2$. □

Proposition 3.1.7 (Somme et produit)**.** *Soient $f, g, h, \ell : \mathbb{N} \to \mathbb{R}_+$.*

— *Si $f = O(h)$ et $g = O(h)$, alors $f + g = O(h)$.*

— *Si $f = O(h)$ et $g = O(\ell)$, alors $fg = O(h\ell)$.*

Démonstration. Soient $C_1 > 0$ et n_1 tels que $f(n) \leqslant C_1\, h(n)$ pour $n \geqslant n_1$, et $C_2 > 0$ et n_2 tels que $g(n) \leqslant C_2\, h(n)$ (premier point) ou $g(n) \leqslant C_2\, \ell(n)$ (second point) pour $n \geqslant n_2$. Pour tout $n \geqslant \max(n_1, n_2)$, on a d'une part $f(n) + g(n) \leqslant (C_1 + C_2)\, h(n)$, d'autre part $f(n)\, g(n) \leqslant C_1 C_2\, h(n)\, \ell(n)$ (toutes les fonctions étant positives, le produit des inégalités est licite). D'où les deux résultats. □

Cette proposition, bien que simple, est l'outil de tous les calculs de complexité à venir : pour majorer le coût d'un algorithme par $O(n^k)$, il suffira de majorer chacune de ses parties par $O(n^k)$. En voici une première illustration, qui justifie qu'en complexité on ne retient d'un polynôme que son terme de plus haut degré.

Exemple 3.1.8 (Un polynôme est un Θ de son terme dominant)**.** *Soit* $P(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ *un polynôme à coefficients positifs, avec* $a_k > 0$*. Alors* $P = \Theta(n^k)$*, et en particulier* $P = O(n^k)$*.*

Majoration. *Pour* $\ell \leqslant k$ *et* $n \geqslant 1$*, on a* $n^\ell \leqslant n^k$*, donc* $a_\ell n^\ell \leqslant a_\ell n^k$*. En sommant,* $P(n) \leqslant (a_0 + a_1 + \cdots + a_k)\, n^k$ *pour* $n \geqslant 1$*, soit* $P = O(n^k)$ *avec* $C = a_0 + \cdots + a_k$*.*

Minoration. *Tous les coefficients étant positifs,* $P(n) \geqslant a_k n^k$ *pour tout* n*, soit* $P = \Omega(n^k)$ *avec* $c = a_k$*. Les deux ensemble donnent* $P = \Theta(n^k)$*.* [2]

3.1.2 La hiérarchie des ordres de grandeur

En pratique, presque toutes les complexités que nous rencontrerons se rangent dans une même échelle, de la croissance la plus lente (la meilleure, pour un coût) à la plus rapide (la pire) :

$$1 \;\prec\; \log n \;\prec\; n \;\prec\; n\log n \;\prec\; n^2 \;\prec\; n^3 \;\prec\; \cdots \;\prec\; 2^n \;\prec\; n!\,,$$

où $f \prec g$ signifie ici $f = O(g)$ *sans* que $f = \Theta(g)$: g croît strictement plus vite que f. Quelques repères pour mémoriser cette échelle : le logarithme croît moins vite que toute puissance n^ε ($\varepsilon > 0$), si petite soit cette puissance ; et toute puissance n^α croît moins vite que n'importe quelle exponentielle c^n ($c > 1$), si proche de 1 soit cette base. C'est ce qui sépare un algorithme *polynomial* (utilisable) d'un algorithme *exponentiel* (vite hors de portée) : pour $n = 60$, n^3 vaut environ $2 \cdot 10^5$ tandis que 2^n dépasse 10^{18}. L'exercice 3.1.9 demande de vérifier les comparaisons sur lesquelles repose cette échelle.

3.1.3 Exercices

Exercice 3.1.9. *Vérifier que* $f = O(g)$ *dans chacun des cas suivants (on pourra utiliser les fonctions usuelles et leurs croissances comparées vues en première année) :*

1. $f(n) = n^c$ *et* $g(n) = n^d$ *avec* $0 \leqslant c \leqslant d$ *;*
2. $f(n) = \log n$ *et* $g(n) = n^\varepsilon$ *pour* $\varepsilon > 0$ *;*
3. $f(n) = n \log n$ *et* $g(n) = n^{1+\varepsilon}$ *pour* $\varepsilon > 0$ *;*

2. Si l'on autorise des coefficients de signe quelconque, la fonction P peut prendre des valeurs négatives et sort du cadre de la définition 3.1.1 ; on a néanmoins toujours $|P(n)| \leqslant (|a_0| + \cdots + |a_k|)\, n^k$, donc $|P| = O(n^k)$.

4. $f(n) = n^\alpha$ *et* $g(n) = c^n$ *pour* $\alpha \geqslant 0$ *et* $c > 1$.

Dans lesquels de ces cas a-t-on de plus $f = \Theta(g)$ *?*

Exercice 3.1.10. *Montrer que la relation* Θ *est* réflexive *(*$f = \Theta(f)$*),* symétrique *(*$f = \Theta(g)$ *entraîne* $g = \Theta(f)$*) et* transitive*. La relation* O *est, elle, réflexive et transitive ; est-elle symétrique ? Donner un contre-exemple.*

Exercice 3.1.11. *Soient* $f, g : \mathbb{N} \to \mathbb{R}_+$*. Montrer que* $\max(f, g) = \Theta(f + g)$*, où* $\max(f, g)$ *désigne la fonction* $n \mapsto \max\big(f(n), g(n)\big)$*.* (C'est la raison pour laquelle, dans une somme de coûts, seul le plus grand terme compte.)

Exercice 3.1.12. *Un premier algorithme traite une entrée de taille* n *en* $100\,n$ *opérations, un second en* n^2 *opérations. À partir de quelle taille* n *le second devient-il plus lent que le premier ? Si chaque opération prend une nanoseconde, combien de temps chacun met-il pour* $n = 10^6$ *?*

3.2 Complexité d'un algorithme

Nous disposons maintenant du langage des ordres de grandeur. Appliquons-le aux algorithmes : définissons précisément ce qu'on appelle la *complexité* d'un algorithme, puis analysons-en quelques-uns.

Mesurer le temps d'un algorithme, ce n'est pas le chronométrer sur une machine donnée — nous l'avons dit — mais compter le nombre d'*opérations élémentaires* qu'il effectue : comparaisons, affectations, additions, multiplications, accès à une case de liste, et ainsi de suite. On parlera par commodité de *temps de calcul*, même si ce qu'on mesure est en réalité ce décompte d'opérations. [3]

3.2.1 La taille de l'entrée

Un algorithme n'est pas conçu pour une entrée particulière, mais pour toute une famille d'entrées : un algorithme de tri doit fonctionner sur *n'importe quelle* liste, pas sur une liste fixée. Son temps de calcul dépend donc de l'entrée, et la première chose à fixer est ce que l'on appelle la *taille* de cette entrée. Selon la nature des données, on choisit la mesure la plus naturelle :

- pour un entier : le nombre de chiffres de son écriture binaire ;
- pour une liste : son nombre d'éléments (ou, si l'on veut être plus fin, la somme des tailles de ses éléments) ;
- pour un graphe : son nombre de sommets et d'arêtes ;
- pour une matrice : sa dimension $n \times m$ (ou la somme des tailles de ses coefficients).

Chaque fois que l'on parle de complexité, il faut donc préciser *quelle* taille sert de référence. Pour une entrée x, on note $|x|$ sa taille, et — lorsque le calcul s'arrête — $T(x)$ le nombre d'opérations élémentaires effectuées sur cette entrée.

3. Une analyse plus fine pèserait chaque opération selon la taille de ses arguments — combien de chiffres binaires pour additionner deux entiers, par exemple. Nous nous en tenons au décompte grossier, qui suffit à classer les algorithmes par ordre de grandeur.

3.2.2 Le pire cas

Pour une taille n donnée, il reste en général une infinité d'entrées possibles, et $T(x)$ peut varier de l'une à l'autre. Lequel de ces temps retient-on ? Le parti pris de ce cours — et le plus répandu — est de retenir le *plus grand* : on se prémunit contre le cas le plus défavorable.

Définition 3.2.1 (Complexité dans le pire cas)**.** *La* complexité *(dans le pire cas) d'un algorithme A est la fonction $t : \mathbb{N} \to \mathbb{N}$ qui, à une taille n, associe le nombre maximal d'opérations effectuées sur une entrée de cette taille :*

$$t(n) = \max \{ T(x) \ : \ |x| = n \}.$$

En d'autres termes, $t(n)$ est le temps de calcul sur les entrées de taille n *qui prennent le plus de temps*. Annoncer que la complexité d'un algorithme est $O(g(n))$, c'est donc garantir que, *quelle que soit* l'entrée de taille n, le calcul s'achève en au plus $C\,g(n)$ opérations à partir d'un certain rang.

Ce n'est pas le seul choix possible. On pourrait s'intéresser au *meilleur cas* (le minimum de $T(x)$), rarement instructif, ou au *cas moyen* (la moyenne des $T(x)$). Mais le cas moyen suppose qu'on se donne une *loi de probabilité* sur les entrées de taille n, ce qui est à la fois plus délicat à manier et discutable — quelle loi ? Sauf mention explicite du contraire, *toutes les complexités de ce cours sont des complexités dans le pire cas.*[4]

3.2.3 Premier exemple : la recherche du minimum

Reprenons l'algorithme `minimum` de la section 2.1, qui parcourt une liste L de longueur n en maintenant le plus petit élément rencontré. La boucle effectue $n - 1$ tours ; chaque tour fait un nombre constant d'opérations (une comparaison $L[i] < m$ et, le cas échéant, une affectation). Le nombre total d'opérations est donc de la forme $a(n-1) + b$ pour des constantes a, b, soit, d'après l'exemple 3.1.8,

$$O(n).$$

Remarquons que ce coût ne dépend même pas du contenu de la liste : il faut de toute façon examiner les n éléments. La complexité dans le pire cas coïncide ici avec celle de n'importe quel cas. Nous verrons à la section 3.3 que cette borne $O(n)$ ne peut pas être améliorée.

3.2.4 Second exemple : le produit de deux matrices

Soient A et B deux matrices carrées de taille $n \times n$. Les coefficients du produit $C = AB$ sont donnés, pour $0 \leqslant i, j \leqslant n - 1$, par

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k]\, B[k][j].$$

4. Il arrive qu'on exprime la complexité en fonction de plusieurs paramètres liés à la taille plutôt que de la seule taille : on chiffre le produit de deux matrices $n \times n$ en fonction de n, bien que l'entrée soit en réalité de taille $2n^2$.

On traduit directement cette formule en trois boucles imbriquées :

```
def produit_matrices(A, B):
    """Produit C = A·B de deux matrices carrées n×n.

    Les matrices sont représentées par des listes de n listes de n entiers ;
    A[i][j] est le coefficient ligne i, colonne j (indices à partir de 0).
    Trois boucles imbriquées de n tours, corps en temps constant : O(n^3).

    >>> produit_matrices([[1, 2], [3, 4]], [[5, 6], [7, 8]])
    [[19, 22], [43, 50]]
    >>> produit_matrices([[1, 0], [0, 1]], [[2, 3], [4, 5]])
    [[2, 3], [4, 5]]
    """
    n = len(A)
    C = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
    return C
```

Conformément à notre convention, une matrice est une liste de listes, indexée à partir de 0, et $A[i][j]$ désigne le coefficient situé ligne i, colonne j. Analysons le coût. L'initialisation de la liste C remplit n^2 cases, soit $O(n^2)$ opérations. Viennent ensuite les trois boucles imbriquées : pour chacune des n valeurs de i, et chacune des n valeurs de j, et chacune des n valeurs de k, on exécute le corps le plus interne — une multiplication, une addition, une affectation, soit un nombre constant d'opérations. Le corps est donc exécuté $n \times n \times n = n^3$ fois, ce qui donne $O(n^3)$ opérations. En ajoutant l'initialisation, négligeable devant ce terme, la complexité de cet algorithme est

$$O(n^3).$$

Remarque 3.2.2 (Le produit matriciel admet des algorithmes plus rapides)**.** *Il s'agit là de la complexité de* cet *algorithme. Le* problème *du produit de deux matrices, lui, se résout plus vite : l'algorithme de Strassen (1969) descend à $O(n^{\log_2 7})$ opérations, où* $\log_2 7 \approx 2{,}807$*, en se ramenant à sept produits de sous-matrices au lieu de huit. Des algorithmes plus rapides encore ont suivi ; à ce jour (2024), le meilleur connu tourne autour de $O(n^{2{,}37})$, et l'on ignore toujours jusqu'où l'on peut descendre. Il ne faut donc pas confondre la complexité d'un* algorithme *donné avec celle du* problème *qu'il résout : nous y revenons à la section 3.3.*

3.2.5 Exercices

Exercice 3.2.3. *Pour chacun des extraits suivants, opérant sur une liste L de longueur n, donner la complexité dans le pire cas (en justifiant) :*

1. *une seule boucle* `for i in range(n)` *dont le corps est en temps constant ;*
2. *deux boucles imbriquées* `for i in range(n)` *puis* `for j in range(n)`*, corps en temps constant ;*
3. *deux boucles imbriquées* `for i in range(n)` *puis* `for j in range(i, n)`*, corps en temps constant.* (Combien de fois le corps est-il exécuté en tout ?)

Exercice 3.2.4. *Écrire une fonction qui calcule le produit Av d'une matrice $n \times n$ par un vecteur de taille n, représentés comme à la section 3.2.4. Quelle est sa complexité ? En déduire que calculer $A(Bv)$ est moins coûteux que calculer $(AB)v$.*

Exercice 3.2.5. *On cherche l'indice du premier élément nul d'une liste L de longueur n (ou n s'il n'y en a aucun), en la parcourant de gauche à droite et en s'arrêtant au premier zéro rencontré. Écrire la fonction. Combien d'opérations effectue-t-elle dans le meilleur cas ? dans le pire cas ? Quelle est sa complexité au sens de la définition 3.2.1 ?*

3.3 Optimalité d'une borne de complexité

Nous savons maintenant majorer le coût d'un algorithme par un $O(f(n))$. Mais une majoration peut être trop large : dire que la complexité est $O(n^2)$ n'interdit pas qu'elle soit en réalité $O(n)$. Dans cette section, nous précisons ce que l'on affirme au juste quand on dit qu'une borne est *optimale* — c'est, nous allons le voir, une affirmation en Ω — et nous mettons en garde contre une confusion fréquente.

3.3.1 Une borne optimale est une borne en Θ

Quand on a établi que la complexité $t(n)$ d'un algorithme vérifie $t(n) = O(f(n))$, on tient une *borne supérieure* sur son nombre de pas de calcul. Rien ne dit que cette borne soit serrée : il se peut que l'analyse ait majoré trop généreusement, et que l'algorithme soit en fait plus rapide. Pour avoir le cœur net, il faut une minoration qui vienne épouser la majoration.

Définition 3.3.1 (Borne optimale)**.** *On dit que la borne $t(n) = O(f(n))$ sur la complexité d'un algorithme est* optimale *(on dit aussi qu'elle est* atteinte*) si l'on a de plus $t(n) = \Omega(f(n))$, c'est-à-dire si*

$$t(n) = \Theta(f(n)).$$

Dit autrement, une borne est optimale lorsqu'elle capture la croissance *exacte* du coût, et pas seulement un plafond. C'est ici que la notation Ω introduite à la définition 3.1.3 prend tout son sens : les discussions d'optimalité sont des énoncés de minoration déguisés.

Comment établit-on, concrètement, une minoration $t(n) = \Omega(f(n))$? La réponse tient en une remarque : $t(n)$ est un *maximum*, $t(n) = \max\{T(x) : |x| = n\}$, et pour minorer un maximum il suffit d'exhiber un seul élément qui le rende grand.

Proposition 3.3.2 (Méthode pour minorer une complexité)**.** *Soit $f : \mathbb{N} \to \mathbb{R}_+$. S'il existe une constante $c > 0$ et un rang n_1 tels que, pour tout $n \geqslant n_1$, il existe une entrée de taille n sur laquelle l'algorithme effectue au moins $c\,f(n)$ opérations, alors $t(n) = \Omega(f(n))$.*

Démonstration. Par définition, $t(n) = \max\{T(x) : |x| = n\}$ est supérieur ou égal à $T(x)$ pour n'importe quelle entrée x de taille n. En particulier, pour $n \geqslant n_1$, en prenant pour x l'entrée fournie par l'hypothèse, on obtient $t(n) \geqslant c\,f(n)$, ce qui est exactement la définition de $t(n) = \Omega(f(n))$. □

Une seule mauvaise entrée par taille suffit donc : le pire cas ne retient que la plus coûteuse.

Exemple 3.3.3 (La recherche du minimum est en $\Theta(n)$)**.** *Reprenons l'algorithme* `minimum` *analysé à la section 3.2.3 : nous avions trouvé $t(n) = O(n)$. Cette borne est optimale. En effet,* sur toute *liste de longueur n, la boucle parcourt les n éléments et effectue $n - 1$ comparaisons : aucune entrée n'y échappe. On a donc $t(n) \geqslant n - 1$, soit $t(n) = \Omega(n)$, et finalement $t(n) = \Theta(n)$. La borne $O(n)$ ne peut pas être améliorée.*

3.3.2 Optimalité d'une borne ≠ optimalité de l'algorithme

Il faut prendre garde à une confusion fréquente, que la définition 3.3.1 rend facile à commettre.

Avertissement 3.3.4 (Deux questions à ne pas confondre)**.** *L'optimalité d'une* borne de complexité pour un algorithme donné *— la borne $O(f(n))$ ne peut pas être resserrée pour* cet *algorithme — n'a rien à voir avec l'optimalité d'un* algorithme pour un problème donné *—* aucun autre *algorithme ne résout le problème plus vite. La première question se règle en analysant un seul algorithme ; la seconde, infiniment plus délicate, demande de raisonner sur* tous *les algorithmes imaginables.*

L'algorithme du produit de matrices de la section 3.2.4 illustre parfaitement l'écart. Sa complexité est $\Theta(n^3)$: il exécute ses trois boucles imbriquées *quelle que soit* l'entrée, donc $t(n) \geqslant n^3$ et la borne $O(n^3)$ est optimale *pour cet algorithme*. Et pourtant cet algorithme n'est pas le meilleur pour le *problème* du produit matriciel : l'algorithme de Strassen évoqué à la remarque 3.2.2 résout le même problème en $O(n^{\log_2 7})$ opérations, strictement moins. Une borne optimale pour un algorithme médiocre reste la borne d'un algorithme médiocre.

Établir qu'aucun algorithme ne peut descendre sous un certain coût — une *borne inférieure pour le problème* — est une tout autre affaire : il faut un argument qui vaille pour n'importe quelle méthode. Nous en verrons un exemple complet au chapitre 6, où l'on démontre qu'*aucun* tri procédant par comparaisons ne peut trier en moins de $\Omega(n \log n)$ opérations.

3.3.3 Exercices

Exercice 3.3.5. *Montrer en détail que la borne $O(n^3)$ du produit de matrices (section 3.2.4) est optimale, c'est-à-dire que $t(n) = \Theta(n^3)$.* (On précisera pourquoi le nombre d'opérations ne dépend pas de l'entrée.)

Exercice 3.3.6. *On reprend l'algorithme de l'exercice 3.2.5 (indice du premier élément nul d'une liste de longueur n), dont la complexité est $O(n)$. Exhiber, pour chaque n, une*

entrée de taille n sur laquelle l'algorithme effectue exactement n comparaisons, et en déduire, à l'aide de la proposition 3.3.2, que la borne $O(n)$ est optimale.

Exercice 3.3.7. *Vrai ou faux, en justifiant : « si la complexité d'un algorithme est $O(n^2)$, alors il existe une entrée de taille n sur laquelle il effectue de l'ordre de n^2 opérations ».* (Penser à ce que garantit, et ne garantit pas, une borne en O ; un algorithme en $\Theta(n)$ a aussi une complexité $O(n^2)$.)

Exercice 3.3.8. *Pour chacune des deux affirmations suivantes, dire de laquelle des deux questions de l'avertissement 3.3.4 elle relève : (a) « le tri à bulles effectue $\Theta(n^2)$ comparaisons dans le pire cas » ; (b) « aucun tri par comparaisons ne fait mieux que $\Theta(n \log n)$ ». Laquelle des deux est, selon vous, la plus difficile à établir ?*

4 Recherche et tris naïfs

Résumé

Ce chapitre aborde deux problèmes omniprésents : *chercher* un élément dans une liste, et *trier* une liste. On verra d'abord que chercher dans une liste déjà triée est remarquablement rapide — la *recherche dichotomique* — ce qui donne une première raison de savoir trier. On étudiera ensuite trois tris dits *naïfs*, tous de complexité $O(n^2)$: le tri à bulles, le tri par sélection et le tri par insertion. Chacun met en pratique la méthode des invariants de boucle du chapitre 2 et l'analyse de complexité du chapitre 3.

Prérequis

Les invariants de boucle (chapitre 2) ; les notations O, Θ, Ω et la complexité dans le pire cas (chapitre 3) ; la manipulation des listes et la convention d'indexation à partir de 0 (chapitre 1).

Objectifs

À l'issue de ce chapitre, vous saurez écrire et prouver les algorithmes de recherche séquentielle et dichotomique et analyser leur coût ; vous connaîtrez trois tris de complexité $O(n^2)$, saurez démontrer leur correction par un invariant de boucle et établir leur complexité ; et vous saurez pourquoi l'on cherche, aux chapitres suivants, à faire mieux.

Ouvrez un dictionnaire pour y trouver le mot *algorithme* : vous tombez dessus en quelques secondes, sans lire la moindre page de A à Z. Cette rapidité ne tient pas au dictionnaire — épais de plusieurs milliers de pages — mais au fait qu'il est *trié*. Imaginez le même dictionnaire dont les mots auraient été jetés dans le désordre : le retrouver supposerait de parcourir l'ouvrage entier, mot après mot. Tout le bénéfice est dans le tri.

C'est la première leçon de ce chapitre, et nous la rendrons quantitative. Dans une liste quelconque de n éléments, retrouver une valeur peut exiger d'examiner les n cases. Dans une liste *triée*, on peut couper l'intervalle de recherche en deux à chaque comparaison, et n'en faire ainsi qu'une poignée : pour un annuaire de 10^5 noms, là où la recherche aveugle demande jusqu'à 10^5 comparaisons, la recherche *dichotomique* n'en demande qu'environ dix-sept. *Voilà pourquoi l'on trie !*

Nous commençons donc par la recherche — séquentielle, puis dichotomique — avant d'étudier, dans les sections suivantes, trois manières de trier une liste. Ces trois tris sont *naïfs* : simples à écrire et à prouver, mais coûteux ($O(n^2)$). Les méthodes plus rapides attendront les chapitres consacrés au paradigme « diviser pour régner » et à la programmation dynamique.

4.1 Recherche dans une liste

Commençons par le problème qui motive tout le chapitre. On dispose d'une liste L et d'une valeur x, et l'on veut savoir si x figure dans L. Précisément, l'algorithme doit renvoyer un indice i tel que $L[i] = x$ si x apparaît dans L, et la valeur conventionnelle -1 sinon (aucun indice valide n'étant négatif, -1 signale sans ambiguïté l'absence).

4.1.1 Valeurs comparables : l'ordre total

Chercher — et plus encore trier — suppose que l'on sache *comparer* les éléments de la liste. Les valeurs que nous manipulons (des entiers, le plus souvent) sont *totalement ordonnées* : deux valeurs quelconques sont toujours comparables.

Définition 4.1.1 (Ordre total strict)**.** *Une relation $<$ sur un ensemble E est un* ordre total strict *si elle vérifie les deux propriétés suivantes :*

1. (trichotomie) *pour tous $x, y \in E$,* un et un seul *des trois cas se produit : $x < y$, ou $x = y$, ou $y < x$;*
2. (transitivité) *pour tous $x, y, z \in E$, si $x < y$ et $y < z$, alors $x < z$.*

En d'autres termes, deux éléments distincts sont toujours rangeables l'un par rapport à l'autre (trichotomie), et l'ordre se propage de proche en proche (transitivité). Les entiers et les réels, munis de leur $<$ habituel, sont totalement ordonnés ; mais aussi les mots, rangés par l'ordre alphabétique (dit *lexicographique*). En revanche, l'inclusion $\subseteq$ entre parties d'un ensemble n'est *pas* un ordre total : les parties $\{1\}$ et $\{2\}$ ne sont pas comparables (ni $\{1\} \subseteq \{2\}$, ni $\{2\} \subseteq \{1\}$), de sorte que la trichotomie échoue. C'est précisément ce genre de situation que la totalité exclut, et c'est ce qui permet de trier.

Les algorithmes de ce chapitre n'utilisent les données qu'à travers de telles comparaisons : jamais nous n'additionnerons ni ne multiplierons les éléments de la liste. De ce fait, leur déroulement ne dépend que de l'*ordre relatif* des éléments, pas de leur valeur : trier $[1, 5, 3]$ ne diffère en rien de trier $[-1, 100, 20]$. On parlera de *tri par comparaison*, une notion sur laquelle nous reviendrons au chapitre 6, où elle livre une borne inférieure sur le coût de tout tri.

4.1.2 Recherche séquentielle

Si l'on ne sait rien de l'organisation de la liste, on ne peut pas faire mieux que de passer les éléments en revue, un par un, jusqu'à rencontrer x. C'est la *recherche séquentielle* ; elle fonctionne sur une liste quelconque, triée ou non.

```
def recherche_sequentielle(L, x):
    """Renvoie un indice i tel que L[i] == x, ou -1 si x n'est pas dans L.

    Parcourt L de gauche à droite ; aucune hypothèse sur l'ordre de L.
    Complexité O(n) dans le pire cas (x absent).
```

```
    >>> recherche_sequentielle([5, 2, 8, 1], 8)
    2
    >>> recherche_sequentielle([5, 2, 8, 1], 7)
    -1
    >>> recherche_sequentielle([], 3)
    -1
    """
    n = len(L)
    for i in range(n):
        if L[i] == x:
            return i
    return -1
```

La correction est immédiate à partir de l'invariant suivant : *au début de l'itération d'indice i, aucune des cases $L[0], \dots, L[i-1]$ ne contient x*. En effet, l'itération précédente n'a pas renvoyé, donc $L[i-1] \neq x$, et l'invariant se propage. Si la boucle s'achève sans rien renvoyer, c'est qu'*aucune* case ne contient x, et l'on renvoie -1 à bon droit ; sinon, on renvoie un indice i tel que $L[i] = x$.

Quant au coût : dans le pire cas — lorsque x est absent, ou en dernière position — la boucle effectue n comparaisons, chacune en temps constant. La complexité de la recherche séquentielle est donc

$$O(n), \qquad n = \texttt{len}(L).$$

Exercice 4.1.2. *Montrer que cette borne $O(n)$ est optimale, au sens de la définition 3.3.1 : exhiber, pour chaque n, une entrée de taille n sur laquelle la recherche séquentielle effectue n comparaisons.*

4.1.3 Recherche dichotomique

Si la liste est *triée* en ordre croissant, on peut faire bien mieux. L'idée — notre premier exemple du principe « diviser pour régner » — est de comparer x à l'élément du *milieu* de la liste : s'il est égal à x, c'est gagné ; s'il est plus petit que x, alors x ne peut se trouver que dans la moitié droite ; s'il est plus grand, que dans la moitié gauche. Dans tous les cas, une seule comparaison élimine la moitié des candidats. On répète sur la moitié retenue, jusqu'à trouver x ou épuiser l'intervalle.

On maintient pour cela deux indices, d (début) et f (fin), délimitant la portion $L[d], \dots, L[f]$ où x peut encore se trouver ; le milieu est $m = \lfloor (d+f)/2 \rfloor$.

```
def recherche_dichotomique(L, x):
    """Renvoie un indice i tel que L[i] == x, ou -1 si x n'est pas dans L.

    La liste L doit être triée en ordre croissant. À chaque tour, l'intervalle
    de recherche [d, f] est coupé en deux : complexité O(log n).

    >>> recherche_dichotomique([1, 2, 5, 8, 10], 8)
```

```
    3
    >>> recherche_dichotomique([1, 2, 5, 8, 10], 1)
    0
    >>> recherche_dichotomique([1, 2, 5, 8, 10], 3)
    -1
    >>> recherche_dichotomique([], 3)
    -1
    """
    d = 0
    f = len(L) - 1
    while d <= f:
        m = (d + f) // 2
        if L[m] == x:
            return m
        if L[m] < x:
            d = m + 1
        else:
            f = m - 1
    return -1
```

Correction. Elle repose sur l'invariant de boucle suivant.

Invariant 4.1.3 (de la recherche dichotomique)**.** *Au début de chaque tour de la boucle* `while`*, si x figure dans L, alors sa position p vérifie $d \leqslant p \leqslant f$.*

Démonstration. Récurrence sur le nombre de tours. *Initialisation :* au premier tour, $d = 0$ et $f = n - 1$, donc l'encadrement $0 \leqslant p \leqslant n - 1$ est vrai pour toute position p. *Hérédité :* supposons l'invariant vrai au début d'un tour, et supposons x dans L, en position p, avec $d \leqslant p \leqslant f$. On calcule $m = \lfloor (d + f)/2 \rfloor$, qui vérifie $d \leqslant m \leqslant f$. Si $L[m] = x$, on renvoie et la boucle s'arrête. Sinon, deux cas :

— si $L[m] < x$, alors, L étant triée, toute position contenant x est $> m$; comme $p \geqslant d$ par hypothèse, on a $m + 1 \leqslant p \leqslant f$. Or le tour remplace d par $m + 1$ sans toucher à f : l'encadrement $d \leqslant p \leqslant f$ est rétabli ;

— si $L[m] > x$, alors toute position contenant x est $< m$, donc $d \leqslant p \leqslant m - 1$. Le tour remplace f par $m - 1$ sans toucher à d : l'encadrement est de nouveau rétabli.

Dans les deux cas, l'invariant vaut au début du tour suivant. □

On sort de la boucle de deux façons. Soit par un `return` m après avoir constaté $L[m] = x$: on renvoie alors une position correcte. Soit parce que la condition $d \leqslant f$ devient fausse, c'est-à-dire $d > f$: l'intervalle $[d, f]$ est vide, et l'invariant affirme que si x était dans L, sa position serait dans cet intervalle vide — ce qui est impossible. C'est donc que x n'est pas dans L, et l'on renvoie -1 à bon droit.

Terminaison. La boucle `while` mérite, comme toujours, une preuve de terminaison séparée. À chaque tour qui ne renvoie pas, la largeur $f - d$ de l'intervalle diminue strictement : d augmente ou f diminue d'au moins 1. Partant d'une valeur finie, elle finit par rendre $d > f$; la boucle s'arrête donc.

Complexité. Mesurons la largeur de l'intervalle de recherche par le nombre $f - d + 1$ de positions encore en lice. Il vaut n au départ. Comme m est le milieu, chacune des deux moitiés contient au plus $\lceil n/2 \rceil$ positions ; après k tours, il en reste donc au plus $n/2^k$. La boucle s'arrête dès que l'intervalle est vide, c'est-à-dire dès que $n/2^k < 1$, soit $k > \log_2 n$. Le nombre de tours est donc au plus $\lfloor \log_2 n \rfloor + 1$, et chaque tour coûte un temps constant. La complexité de la recherche dichotomique est

$$O(\log n).$$

C'est spectaculairement mieux que le $O(n)$ de la recherche séquentielle : pour $n = 10^6$, on passe d'un million de comparaisons à une vingtaine.

Remarque 4.1.4 ($\log_2 n$ et le nombre de chiffres binaires)**.** *Le logarithme en base* 2 *apparaît naturellement parce qu'on divise par* 2 *à chaque tour. Il a une interprétation concrète : un entier* $n \geqslant 1$ *s'écrit avec exactement* $\lfloor \log_2 n \rfloor + 1$ *chiffres binaires (ses* bits*). La recherche dichotomique effectue donc, à une unité près, autant de comparaisons que* n *a de bits — soit précisément la « taille » de* n *au sens du chapitre 3.*

Exercice 4.1.5. *Montrer que la borne* $O(\log n)$ *de la recherche dichotomique est optimale.* (On pourra remarquer qu'avec une comparaison, on distingue au plus deux issues, donc qu'avec k comparaisons on distingue au plus 2^k positions possibles pour x.)

Exercice 4.1.6. *Réécrire la recherche dichotomique sous forme* récursive *: une fonction qui prend en plus les bornes* d *et* f *et s'appelle elle-même sur la moitié retenue. Vérifier qu'elle renvoie les mêmes résultats que la version itérative.*

Exercice 4.1.7. *Lorsque* x *apparaît plusieurs fois dans* L*, la recherche dichotomique renvoie l'*une *de ses positions, pas nécessairement la première. Modifier l'algorithme pour qu'il renvoie l'indice de la* première *occurrence de* x *(la plus petite position), toujours en* $O(\log n)$*.* (Indication : au lieu de s'arrêter dès que $L[m] = x$, mémoriser cette position et continuer à chercher dans la moitié gauche.)

4.2 Tri à bulles

Nous avons appris à chercher dans une liste (section 4.1) ; apprenons maintenant à la trier. Commençons par préciser le problème.

Définition 4.2.1 (Trier une liste)**.** Trier *une liste* L *de longueur* n *en ordre croissant, c'est en réorganiser les éléments — les* permuter *— de façon que*

$$L[0] \leqslant L[1] \leqslant \cdots \leqslant L[n-1].$$

Autrement dit, on ne change ni n'ajoute aucune valeur : on se contente de les ranger. Par exemple, la liste de gauche ci-dessous, une fois triée, devient celle de droite :

indice	0	1	2	3	4	5	6
valeur	5	2	-3	1	5	8	0

$\longrightarrow$

indice	0	1	2	3	4	5	6
valeur	-3	0	1	2	5	5	8

Nous présentons dans ce chapitre trois tris qui résolvent ce problème : le tri à bulles, le tri par sélection et le tri par insertion. Tous les trois trient *en place*, c'est-à-dire en réorganisant la liste donnée sans en allouer de copie, et tous les trois coûtent $O(n^2)$ opérations dans le pire cas. Ils se distinguent par la *manière* dont ils placent les éléments. Voici le premier.

4.2.1 Principe

Le tri à bulles balaie la liste de gauche à droite et compare chaque élément à son voisin de droite : s'ils sont mal ordonnés, il les échange. En un balayage complet, le plus grand élément se trouve poussé, de proche en proche, jusqu'à la dernière case — il « remonte » comme une bulle, d'où le nom. Une fois ce maximum à sa place, on recommence sur le reste de la liste, puis encore, jusqu'à ce que tout soit trié.

```python
def tri_bulles(L):
    """Trie la liste L en ordre croissant, en place, et la renvoie.

    Méthode des bulles : on balaie la liste de gauche à droite en échangeant
    deux voisins mal ordonnés ; à chaque balayage, le plus grand élément non
    encore placé « remonte » jusqu'à sa position finale, à droite. Le balayage
    numéro k ne parcourt que les n-1-k premières paires, le suffixe étant déjà
    trié. Complexité O(n^2).

    >>> tri_bulles([5, 2, -3, 1, 5, 8, 0])
    [-3, 0, 1, 2, 5, 5, 8]
    >>> tri_bulles([3, 1, 2])
    [1, 2, 3]
    >>> tri_bulles([])
    []
    """
    n = len(L)
    for k in range(n - 1):
        for j in range(n - 1 - k):
            if L[j] > L[j + 1]:
                L[j], L[j + 1] = L[j + 1], L[j]
    return L
```

Comme le plus grand élément est définitivement placé après le premier balayage, le deuxième n'a plus besoin de visiter la dernière case ; le troisième peut s'arrêter deux cases

avant la fin, et ainsi de suite. C'est pourquoi le balayage numéro k ne parcourt que les premières paires, jusqu'à l'indice $n-1-k$ exclu.

4.2.2 Correction

La correction repose sur l'invariant suivant, qui décrit le suffixe déjà trié.

Invariant 4.2.2 (du tri à bulles)**.** *Au début du balayage d'indice k (pour $0 \leqslant k \leqslant n-1$), les k plus grands éléments de la liste occupent, rangés en ordre croissant, les k dernières cases $L[n-k], \ldots, L[n-1]$.*

Pour l'établir, nous aurons besoin de comprendre l'effet d'un seul balayage. C'est l'objet du lemme suivant, qui formalise la « remontée » de la bulle.

Lemme 4.2.3 (Effet d'un balayage)**.** *Après l'exécution complète de la boucle interne pour une valeur de k donnée, la case $L[n-1-k]$ contient le plus grand des éléments de $L[0], \ldots, L[n-1-k]$.*

Démonstration. Notons $M = n-1-k$. La boucle interne fait varier j de 0 à $M-1$, et l'on démontre par récurrence sur j la propriété : *après le test d'indice j, la case $L[j+1]$ contient* $\max(L[0], \ldots, L[j+1])$. *Initialisation ($j=0$) :* le test compare $L[0]$ et $L[1]$ et place le plus grand des deux en $L[1]$, qui vaut donc $\max(L[0], L[1])$. *Hérédité :* si après le test d'indice j la case $L[j+1]$ contient $\max(L[0], \ldots, L[j+1])$, alors le test d'indice $j+1$ compare $L[j+1]$ et $L[j+2]$ et place le plus grand en $L[j+2]$, qui vaut donc $\max\big(\max(L[0], \ldots, L[j+1]), L[j+2]\big) = \max(L[0], \ldots, L[j+2])$. Après le dernier test ($j = M-1$), la case $L[M]$ contient $\max(L[0], \ldots, L[M])$. □

Démonstration de l'invariant 4.2.2. Récurrence sur k. *Initialisation ($k=0$) :* l'énoncé porte sur les 0 plus grands éléments et un suffixe vide ; il est vrai sans rien dire. *Hérédité :* supposons l'invariant vrai au début du balayage d'indice k : les cases $L[n-k], \ldots, L[n-1]$ contiennent les k plus grands éléments, triés. Ces éléments sont en particulier supérieurs ou égaux à tous ceux de $L[0], \ldots, L[n-1-k]$. D'après le lemme 4.2.3, le balayage d'indice k amène en position $n-1-k$ le maximum de $L[0], \ldots, L[n-1-k]$: c'est le $(k+1)$-ième plus grand élément de la liste, et il est inférieur ou égal à ceux du suffixe déjà trié. Les cases $L[n-1-k], \ldots, L[n-1]$ contiennent donc maintenant les $k+1$ plus grands éléments, triés : c'est l'invariant au début du balayage d'indice $k+1$. □

À la sortie de la boucle externe, le balayage d'indice $k = n-1$ vient d'être préparé : les $n-1$ plus grands éléments occupent, triés, les cases $L[1], \ldots, L[n-1]$. La case restante $L[0]$ contient alors le plus petit élément, et la liste entière est triée.

4.2.3 Complexité

Le balayage d'indice k effectue $n-1-k$ comparaisons, chacune suivie d'au plus un échange — un nombre constant d'opérations. Le nombre total de comparaisons est donc

$$\sum_{k=0}^{n-2} (n-1-k) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

soit $O(n^2)$. La complexité du tri à bulles est $O(n^2)$. C'est simple, c'est correct — mais, vous le verrez, c'est lent : nous saurons bientôt trier bien plus vite.

4.2.4 Exercices

Exercice 4.2.4. *Dans sa forme ci-dessus, le tri à bulles effectue tous ses balayages même si la liste est déjà triée. Modifier l'algorithme pour qu'il* s'arrête *dès qu'un balayage complet n'effectue aucun échange. Quelle est la complexité de cette variante dans le meilleur cas (liste déjà triée) ? Et dans le pire cas ?*

Exercice 4.2.5. *Montrer que la borne $O(n^2)$ du tri à bulles est optimale (au sens de la définition 3.3.1) : exhiber, pour chaque n, une liste de taille n sur laquelle l'algorithme effectue $n(n-1)/2$ échanges.* (Penser à une liste rangée en ordre décroissant.)

Exercice 4.2.6. *Adapter le tri à bulles pour qu'il trie en ordre* décroissant*. Quelle ligne suffit-il de modifier ?*

4.3 Tri par sélection

Le tri à bulles plaçait les grands éléments à droite, de proche en proche. Le *tri par sélection* procède plus directement : à chaque étape, il *sélectionne* le plus petit élément qui reste à ranger et le met à sa place définitive.

4.3.1 Principe

On range la liste de la gauche vers la droite. À l'étape i, les cases $L[0], \ldots, L[i-1]$ contiennent déjà, triés, les i plus petits éléments ; il reste à traiter la portion $L[i], \ldots, L[n-1]$. On y cherche le plus petit élément, et on l'échange avec celui de la position i : la portion triée gagne une case, et l'on passe à l'étape $i+1$.

```
def tri_selection(L):
    """Trie la liste L en ordre croissant, en place, et la renvoie.

    Tri par sélection : à l'étape i, on cherche le plus petit élément de la
    portion non encore triée L[i..n-1] et on l'amène en position i par un
    échange. Après l'étape i, les cases L[0..i] contiennent les i+1 plus petits
    éléments, triés. Complexité O(n^2), mais seulement n-1 échanges.

    >>> tri_selection([5, 2, -3, 1, 5, 8, 0])
    [-3, 0, 1, 2, 5, 5, 8]
    >>> tri_selection([3, 1, 2])
    [1, 2, 3]
    >>> tri_selection([])
    []
    """
```

```
n = len(L)
for i in range(n - 1):
    p = i
    for j in range(i + 1, n):
        if L[j] < L[p]:
            p = j
    L[i], L[p] = L[p], L[i]
return L
```

La boucle interne n'est autre que la recherche de la *position* du minimum de $L[i], \dots, L[n-1]$, déjà rencontrée à la section 2.1 : elle parcourt la portion non triée en retenant dans p l'indice du plus petit élément vu jusque-là. À sa sortie, p est donc l'indice d'un plus petit élément de $L[i], \dots, L[n-1]$.

4.3.2 Correction

Invariant 4.3.1 (du tri par sélection)**.** *Au début de l'itération d'indice i (pour $0 \leqslant i \leqslant n-1$), les cases $L[0], \dots, L[i-1]$ contiennent les i plus petits éléments de la liste, rangés en ordre croissant.*

Remarquons d'emblée une conséquence de cet énoncé : si $L[0], \dots, L[i-1]$ sont les i plus petits éléments, alors tous les éléments restants, dans $L[i], \dots, L[n-1]$, leur sont supérieurs ou égaux.

Démonstration. Récurrence sur i. *Initialisation ($i = 0$) :* l'énoncé porte sur un préfixe vide, il est vrai sans rien dire. *Hérédité :* supposons l'invariant vrai au début de l'itération i. La boucle interne place dans p l'indice d'un plus petit élément de $L[i], \dots, L[n-1]$; l'échange amène cet élément en position i. C'est le minimum des éléments restants, donc — par la remarque ci-dessus — le $(i+1)$-ième plus petit élément de la liste entière, et il est supérieur ou égal à chacun des i plus petits qui le précèdent dans $L[0], \dots, L[i-1]$. Ainsi $L[0], \dots, L[i]$ contiennent les $i+1$ plus petits éléments, triés : c'est l'invariant au début de l'itération $i+1$. □

À la sortie de la boucle, l'itération d'indice $n-1$ vient d'être préparée : les cases $L[0], \dots, L[n-2]$ contiennent les $n-1$ plus petits éléments, triés. La dernière case $L[n-1]$ contient donc le plus grand élément, et la liste est entièrement triée.

4.3.3 Complexité

À l'itération i, la boucle interne parcourt $L[i+1], \dots, L[n-1]$, soit $n-1-i$ comparaisons, suivies d'un *unique* échange. Le nombre total de comparaisons est

$$\sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2},$$

soit $O(n^2)$. La complexité du tri par sélection est donc $O(n^2)$, comme celle du tri à bulles. Mais une différence mérite d'être notée : le tri par sélection n'effectue que $n-1$ échanges

en tout — un par étape — là où le tri à bulles peut en faire de l'ordre de n^2. C'est un atout lorsque déplacer un élément coûte cher.

4.3.4 Exercices

Exercice 4.3.2. *Écrire la variante « tri du maximum » du tri par sélection : à chaque étape, on sélectionne le plus* grand *élément de la portion non triée et on le place à la* fin*. Énoncer l'invariant de boucle correspondant.*

Exercice 4.3.3. *Contrairement au tri à bulles optimisé (exercice 4.2.4), le tri par sélection effectue* toujours *$n(n-1)/2$ comparaisons, que la liste soit déjà triée ou non. Le justifier, et en déduire que sa complexité est $\Theta(n^2)$ — y compris dans le meilleur cas.*

Exercice 4.3.4. *Montrer que le tri par sélection effectue au plus $n-1$ échanges sur une liste de taille n. Construire une liste sur laquelle il en effectue exactement $n-1$, et une autre sur laquelle il en effectue zéro.*

4.4 Tri par insertion

Après le tri à bulles et le tri par sélection, voici un troisième tri, peut-être le plus naturel de tous : c'est celui qu'on emploie spontanément pour ranger une main de cartes. On prend les cartes une à une et l'on glisse chacune à sa place parmi celles déjà en main, qu'on tient triées. C'est le *tri par insertion*.

4.4.1 Principe

On parcourt la liste de gauche à droite. À l'étape i, le préfixe $L[0], \ldots, L[i-1]$ est déjà trié — mais, contrairement au tri par sélection, ce ne sont pas forcément les i plus petits éléments, seulement les i premiers, rangés. On prend alors la valeur $x = L[i]$ et on l'*insère* à sa place dans ce préfixe : on décale d'une case vers la droite tous les éléments du préfixe qui sont plus grands que x, ce qui libère le bon emplacement, où l'on dépose x.

```
def tri_insertion(L):
    """Trie la liste L en ordre croissant, en place, et la renvoie.

    Tri par insertion : on insère successivement L[1], L[2], ..., L[n-1] à leur
    place dans le préfixe déjà trié, en décalant d'une case vers la droite les
    éléments plus grands que la valeur insérée. Complexité O(n^2) dans le pire
    cas, mais O(n) si L est déjà (presque) triée : le tri est « adaptatif ».

    >>> tri_insertion([5, 2, -3, 1, 5, 8, 0])
    [-3, 0, 1, 2, 5, 5, 8]
    >>> tri_insertion([3, 1, 2])
    [1, 2, 3]
    >>> tri_insertion([])
```

```
    []
    """
    n = len(L)
    for i in range(1, n):
        x = L[i]
        j = i - 1
        while j >= 0 and L[j] > x:
            L[j + 1] = L[j]
            j = j - 1
        L[j + 1] = x
    return L
```

La boucle `while` fait précisément ce décalage : partant de la position $i-1$, elle recule tant qu'elle rencontre un élément $L[j] > x$, en le poussant d'une case à droite ; elle s'arrête au premier élément $\leqslant x$ (ou au bord gauche de la liste), et x se loge dans la case ainsi libérée.

4.4.2 Correction

Invariant 4.4.1 (du tri par insertion)**.** *Au début de l'itération d'indice i (pour $1 \leqslant i \leqslant n$), les cases $L[0], \dots, L[i-1]$ sont triées en ordre croissant et contiennent les mêmes valeurs que les i premières cases de la liste initiale (à permutation près).*

C'est bien la différence avec le tri par sélection : ici le préfixe trié n'est pas constitué des plus petits éléments, mais simplement des premiers, remis dans l'ordre.

Démonstration. Récurrence sur i. *Initialisation ($i = 1$) :* le préfixe $L[0]$ ne comporte qu'un élément, il est trié et inchangé. *Hérédité :* supposons l'invariant vrai au début de l'itération i, et posons $x = L[i]$. La boucle `while` décale vers la droite, un à un, les éléments du préfixe strictement supérieurs à x. Plus précisément, elle maintient la propriété : *au moment d'un test, les éléments initialement en positions $j+1, \dots, i-1$ qui étaient $> x$ ont été recopiés une case plus à droite (en $j+2, \dots, i$), la case $j+1$ est disponible, et $L[0], \dots, L[j]$ sont inchangées.* La boucle s'arrête de deux façons : soit j devient négatif — alors x est plus petit que tout le préfixe et se place en tête, à l'indice 0 ; soit $L[j] \leqslant x$ — alors, le préfixe étant trié, on a $L[0], \dots, L[j] \leqslant x$, et l'affectation place x en position $j+1$, juste après des valeurs qui lui sont inférieures ou égales et juste avant des valeurs qui lui sont supérieures. Dans les deux cas, $L[0], \dots, L[i]$ est trié, et c'est une permutation des $i+1$ premières valeurs initiales : l'invariant vaut au début de l'itération $i+1$. □

À la sortie de la boucle, l'itération d'indice n vient d'être préparée : la liste $L[0], \dots, L[n-1]$ tout entière est triée, et c'est une permutation de la liste de départ.

4.4.3 Complexité

À l'étape i, la boucle `while` effectue au plus i comparaisons et décalages. Dans le pire cas, le nombre d'opérations est donc au plus

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2},$$

soit $O(n^2)$. Ce pire cas est atteint sur une liste rangée en ordre *décroissant* : chaque nouvel élément doit traverser tout le préfixe.

Mais — et c'est la particularité du tri par insertion — le *meilleur* cas est bien meilleur. Si la liste est déjà triée, le test $L[j] > x$ est faux dès le premier essai à chaque étape : la boucle `while` ne s'exécute jamais, et le coût tombe à $O(n)$. Le tri par insertion est *adaptatif* : il est d'autant plus rapide que la liste est proche d'être triée. C'est ce qui en fait, en pratique, le meilleur des trois tris naïfs sur des données presque ordonnées.

4.4.4 Synthèse : les trois tris naïfs

Récapitulons. Les trois tris de ce chapitre résolvent le même problème et partagent la même complexité $O(n^2)$ dans le pire cas, mais leurs profils diffèrent :

	Comparaisons (pire cas)	Échanges/décalages	Meilleur cas
Tri à bulles	$\Theta(n^2)$	$O(n^2)$	$O(n)$ (avec arrêt anticipé)
Tri par sélection	$\Theta(n^2)$	$n-1$	$\Theta(n^2)$
Tri par insertion	$O(n^2)$	$O(n^2)$	$O(n)$

Le tri par sélection se distingue par son nombre d'échanges minimal $(n-1)$, au prix d'un nombre de comparaisons toujours quadratique. Le tri par insertion est le seul à être réellement *adaptatif*. Tous trois ont une complexité $O(n^2)$ optimale au sens du chapitre 3 — ils prennent effectivement de l'ordre de n^2 opérations dans le pire cas, comme le montrent les exercices 4.2.5 et 4.4.3.

Ce plafond de n^2 n'est pourtant pas une fatalité. En changeant de stratégie — le paradigme « diviser pour régner » — nous trierons au chapitre 6 en $O(n \log n)$ opérations, un gain spectaculaire : pour $n = 10^6$, on passe de 10^{12} à quelque $2 \cdot 10^7$ opérations.

4.4.5 Exercices

Exercice 4.4.2. *Dérouler le tri par insertion sur la liste* $[5, 2, 4, 1, 3]$ *: indiquer l'état de la liste au début de chaque itération de la boucle externe.*

Exercice 4.4.3. *Décrire une liste de taille n sur laquelle le tri par insertion effectue exactement $n(n-1)/2$ comparaisons, et une autre sur laquelle il n'en effectue que $n-1$. En déduire que sa complexité est $O(n^2)$ dans le pire cas et $O(n)$ dans le meilleur.*

Exercice 4.4.4 (★ Tri par insertion dichotomique)**.** *Puisque le préfixe* $L[0], \dots, L[i-1]$ *est trié, on peut trouver la position d'insertion de* $x = L[i]$ *par une* recherche dichotomique *(section 4.1.3) au lieu d'un balayage. Écrire cette variante. Montrer qu'elle ramène le nombre de* comparaisons *à* $O(n \log n)$*, mais que le nombre de* décalages *reste* $O(n^2)$ *: la complexité globale demeure donc* $O(n^2)$*.* (Pourquoi les décalages, eux, ne profitent-ils pas de la dichotomie ?)

5 La récursivité

Résumé

Ce chapitre présente un style de programmation où une fonction, pour résoudre un problème, s'appelle elle-même sur des données plus petites : la *récursivité*. C'est la traduction directe en algorithme des définitions *par récurrence* familières en mathématiques. Nous en dégageons le principe sur des exemples (factorielle, suite de Fibonacci), nous expliquons ce qui se passe « en coulisse » lors de l'exécution (la pile d'appels, l'arbre des appels récursifs), puis nous étudions deux phénomènes importants : une fonction récursive peut être catastrophiquement lente quand elle recalcule sans cesse les mêmes valeurs — d'où la *mémoïsation* — et une fonction récursive peut toujours, en principe, être réécrite sans récursion — la *dérécursivation*, particulièrement simple dans le cas terminal. Nous terminons par les objets qui sont eux-mêmes définis récursivement, comme les arbres, sur lesquels les algorithmes récursifs deviennent naturels.

Prérequis

Le raisonnement par récurrence, simple et forte (utilisé dès le chapitre 2) ; les suites définies par une relation de récurrence (cours de L1) ; la division euclidienne et le calcul de pgcd ; les notations de complexité, en particulier $O(\cdot)$ (chapitre 3) ; les listes Python et leur manipulation.

Objectifs

À l'issue de ce chapitre, vous saurez écrire une fonction récursive à partir d'une définition par récurrence, repérer son ou ses cas de base, prouver sa correction par récurrence et analyser son coût en dénombrant les appels ; vous saurez reconnaître quand la récursion naïve est inefficace et y remédier ; vous saurez transformer une récursion terminale en boucle ; et vous saurez écrire des algorithmes récursifs sur des objets définis récursivement.

En mathématiques, on définit volontiers un objet *en fonction de lui-même*, ou plutôt en fonction de versions plus petites de lui-même. La factorielle est le produit $n! = n \cdot (n-1) \cdots 2 \cdot 1$, mais on peut tout aussi bien dire : $n!$ vaut n fois $(n-1)!$, et $0! = 1$. La suite de Fibonacci se définit en disant que chaque terme est la somme des deux précédents. Dans les deux cas, on ramène le calcul d'un cas au calcul d'un cas plus petit, jusqu'à un point de départ connu d'avance. *Une fonction informatique peut-elle se définir de la même manière, en s'appelant elle-même ?*

La réponse est oui, et c'est tout l'objet de ce chapitre. Une telle fonction est dite *récursive*. L'image qui revient le plus souvent est celle des poupées russes : pour savoir

combien de poupées contient une matriochka, on l'ouvre, on compte celle qu'on tient plus tout ce que contient la poupée du dedans — laquelle se traite exactement de la même façon — et l'on s'arrête sur la plus petite, qui ne contient rien. Le programme récursif suit le même mouvement : il décrit comment résoudre un problème à partir de la solution du « même problème, en plus petit », et prévoit un *cas de base* où la réponse est immédiate.

5.1 Principe et exemples

5.1.1 Un premier exemple : la factorielle

Rappelons la définition de la factorielle d'un entier $n \geqslant 0$:

$$n! = \prod_{i=1}^{n} i = n \cdot (n-1) \cdots 2 \cdot 1, \qquad \text{avec la convention } 0! = 1.$$

Cette définition « à plat » se reformule de façon *récurrente*, ce qui en rend l'aspect auto-référent manifeste :

$$\begin{cases} 0! = 1, \\ n! = n \cdot (n-1)! \quad \text{pour } n > 0. \end{cases} \tag{5.1.1}$$

La ligne du bas exprime $n!$ à l'aide de $(n-1)!$, c'est-à-dire de la factorielle d'un entier *plus petit* ; la ligne du haut fournit une valeur connue d'avance, sur laquelle l'enchaînement s'arrête. Cette relation (5.1.1) se traduit *mot pour mot* en une fonction qui s'appelle elle-même.

```python
def factorielle(n):
    """Renvoie n! = 1 * 2 * ... * n, calculé récursivement.

    Le cas de base est n == 0, où l'on renvoie 1 ; sinon on se ramène
    au calcul de (n-1)! par un appel récursif.

    >>> factorielle(0)
    1
    >>> factorielle(1)
    1
    >>> factorielle(5)
    120
    >>> import math
    >>> all(factorielle(n) == math.factorial(n) for n in range(13))
    True
    """
    if n == 0:
        return 1
    return n * factorielle(n - 1)
```

Le test `n == 0` est le *cas de base* : la réponse y est donnée sans calcul. Dans le cas contraire, la fonction renvoie `n` fois le résultat de son propre appel sur `n - 1`. Ainsi `factorielle(3)` réclame `factorielle(2)`, qui réclame `factorielle(1)`, qui réclame `factorielle(0)` — lequel répond 1 sans rien réclamer, et la chaîne se « rembobine ».

Une fonction si proche de sa définition mathématique a un avantage : sa correction se démontre presque mécaniquement, par récurrence.

Proposition 5.1.2 (Correction de `factorielle`)**.** *Pour tout entier $n \geqslant 0$, l'appel `factorielle(`n`)` se termine et renvoie $n!$.*

C'est notre première preuve de correction d'une fonction *récursive* ; elle suit fidèlement la forme de la définition (5.1.1), et c'est ce qui en fait un modèle.

Démonstration. Nous raisonnons par récurrence sur n. Notons P_n la propriété « `factorielle(`n`)` se termine et renvoie $n!$ ».

Initialisation. Pour $n = 0$, le test `n == 0` est vrai et la fonction renvoie 1, sans aucun appel récursif : elle se termine, et $1 = 0!$. Donc P_0 est vraie.

Hérédité. Soit $n \geqslant 0$; supposons P_n vraie et montrons P_{n+1}. Comme $n + 1 > 0$, l'appel `factorielle(`$n+1$`)` exécute la dernière ligne : il évalue `(n+1) * factorielle(`n`)`. Par hypothèse de récurrence, l'appel `factorielle(`n`)` se termine et renvoie $n!$; donc `factorielle(`$n+1$`)` se termine à son tour et renvoie $(n+1) \cdot n! = (n+1)!$. Donc P_{n+1} est vraie.

Par récurrence, P_n est vraie pour tout $n \geqslant 0$. □

L'argument décalque la définition récursive : le cas de base de la fonction fournit l'initialisation, l'appel récursif fournit l'hérédité. C'est la règle générale pour une fonction récursive : *on prouve sa correction par récurrence sur le paramètre qui décroît à chaque appel.*

Quant au coût, comptons les multiplications. Soit $M(n)$ leur nombre lors de l'appel `factorielle(`n`)`. Le cas de base n'en fait aucune, et chaque cas récursif en ajoute une à celles de l'appel sur $n - 1$:

$$M(0) = 0, \qquad M(n) = 1 + M(n-1) \text{ pour } n > 0.$$

On en tire immédiatement $M(n) = n$: la fonction effectue exactement n multiplications, soit un coût en $O(n)$.

5.1.2 La pile d'exécution

L'algorithme ci-dessus est limpide, mais il escamote une question : comment la machine s'y prend-elle pour suspendre un calcul (« je multiplierai par `n` *après* avoir obtenu `factorielle(n-1)` »), en lancer un autre, puis reprendre le premier là où il en était ? Le mécanisme mérite d'être décrit, car il vaut pour *toute* fonction, récursive ou non.

Remarque 5.1.3 (La pile d'appels)**.** *À chaque appel de fonction, le système met de côté un* contexte *(la valeur des variables locales, et l'endroit du programme où reprendre*

une fois l'appel terminé). Ces contextes sont rangés dans une pile *: une structure où l'on ajoute et retire toujours par le* sommet*, comme une pile d'assiettes. Lancer un appel, c'est* empiler *un nouveau contexte ; terminer un appel et renvoyer une valeur, c'est* dépiler *le contexte courant et rendre la main à celui qui se trouve juste en dessous — l'appelant. Pour `factorielle(3)`, la pile grandit jusqu'à contenir les quatre contextes de `factorielle(3)`, `(2)`, `(1)`, `(0)`, puis se vide en sens inverse, chaque appel multipliant par sa propre valeur de `n` le résultat que lui rend l'appel du dessous. Chaque contexte garde sa* propre *copie des variables : c'est pourquoi les différents `n` ne se mélangent pas.*

Cette pile a une conséquence pratique : elle occupe de la mémoire, proportionnellement à la profondeur des appels imbriqués. Une récursion trop profonde peut la faire déborder [1] ; nous y reviendrons en transformant certaines récursions en boucles (section 5.3).

On visualise l'enchaînement des appels par un schéma appelé *arbre des appels récursifs* : chaque appel est un nœud, et ses appels récursifs sont ses « fils ». Pour la factorielle, cet arbre est tout en hauteur — chaque appel n'en déclenche qu'un seul :

```
factorielle(3)
      |
factorielle(2)
      |
factorielle(1)
      |
factorielle(0)
```

5.1.3 Un second exemple : la suite de Fibonacci

Passons à un exemple où un appel en déclenche *deux*. La suite de Fibonacci est définie par

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geqslant 2. \end{cases} \tag{5.1.4}$$

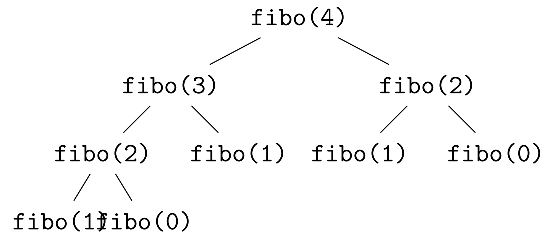
Comme chaque terme dépend des *deux* précédents, il faut ici *deux* cas de base ($n = 0$ et $n = 1$) pour amorcer le calcul. La traduction récursive est de nouveau directe.

```python
def fibonacci_recursif(n):
    """Renvoie le n-ième terme de la suite de Fibonacci (F_0 = 0, F_1 = 1).

    Traduction directe de la récurrence F_n = F_{n-1} + F_{n-2}.
    Cette version est correcte mais catastrophiquement lente : elle
    recalcule un nombre exponentiel de fois les mêmes termes.

    >>> fibonacci_recursif(0)
```

1. En Python, une récursion trop profonde lève l'erreur `RecursionError`, le langage imposant une profondeur maximale d'appels (de l'ordre du millier par défaut) précisément pour éviter ce débordement.

```
    0
    >>> fibonacci_recursif(1)
    1
    >>> [fibonacci_recursif(n) for n in range(10)]
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    """
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci_recursif(n - 1) + fibonacci_recursif(n - 2)
```

Cette fonction est correcte — la preuve par récurrence (forte cette fois, car on invoque deux rangs précédents) suit le même schéma qu'à la proposition 5.1.2, et nous la laissons en exercice. Mais elle souffre d'un défaut spectaculaire. Pour le voir, dessinons son arbre des appels sur l'entrée $n = 4$:



Le défaut saute aux yeux : `fibo(2)` est calculé *deux* fois, et plus n grandit, plus les répétitions se multiplient. Quantifions ce gâchis. Notons a_n le nombre d'additions effectuées par `fibonacci_recursif(`n`)`. Les deux cas de base n'en font aucune, et un appel sur $n \geqslant 2$ fait une addition de plus que ses deux sous-appels :

$$a_0 = a_1 = 0, \qquad a_n = a_{n-1} + a_{n-2} + 1 \text{ pour } n \geqslant 2.$$

Cette suite ressemble à s'y méprendre à celle de Fibonacci ; on montre d'ailleurs que $a_n = F_{n+1} - 1$, et l'on sait que F_n croît comme φ^n, où $\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618$ est le *nombre d'or*. Le nombre d'additions est donc *exponentiel* en n !

C'est d'autant plus frustrant qu'on calcule facilement F_n avec seulement $n-1$ additions, en partant de F_0, F_1 et en avançant terme à terme par une simple boucle. La traduction récursive naïve, elle, est inutilisable dès que n dépasse quelques dizaines. Nous verrons à la section 5.2 comment garder l'écriture récursive tout en supprimant les recalculs, grâce à la *mémoïsation*.

5.1.4 Anatomie d'une fonction récursive

Ces deux exemples dégagent la structure commune à toute fonction récursive bien formée. Elle comporte :

- un ou plusieurs *cas de base* (aussi appelés *conditions d'arrêt*), où la réponse est fournie directement, sans appel récursif ;
- un ou plusieurs *cas récursifs*, où la fonction s'appelle elle-même sur des données *plus petites*, puis combine les résultats obtenus.

La condition « sur des données plus petites » n'a rien d'une formalité : c'est elle qui garantit la *terminaison*. Si les appels récursifs ne se rapprochaient pas des cas de base, la fonction s'appellerait indéfiniment.

Avertissement 5.1.5. *Une fonction récursive dont les appels ne décroissent pas vers un cas de base* ne termine pas. *Par exemple, écrire `factorielle(n - 1)` sans le test `n == 0`, ou appeler `factorielle(n)` sur un entier négatif, provoque une suite infinie d'appels — en pratique, un débordement de la pile. Vérifier qu'un cas de base est bien atteint fait partie intégrante de l'écriture d'une fonction récursive.*

Concrètement, prouver la terminaison revient à exhiber une quantité entière positive, fonction des arguments, qui décroît strictement à chaque appel récursif : elle ne peut décroître indéfiniment, donc un cas de base finit par être atteint. Pour `factorielle(`n`)` comme pour `fibonacci_recursif(`n`)`, cette quantité est simplement n.

5.1.5 Exercices

Exercice 5.1.6. *Écrire une fonction récursive `somme(L)` qui renvoie la somme des éléments d'une liste `L` d'entiers, sans utiliser de boucle.* Indication *: le cas de base est la liste vide (de somme* 0*) ; sinon, la somme de `L` est son premier élément ajouté à la somme du reste, `L[1:]`. Prouver par récurrence sur la longueur de `L` que la fonction est correcte.*

Exercice 5.1.7. *On définit la suite* (u_n) *par* $u_0 = u_1 = u_2 = 0$ *et, pour* $n \geqslant 3$, $u_n = u_{n-1} + u_{n-2} \cdot u_{n-3} + 1$.

1. *Écrire la fonction récursive qui traduit directement cette définition. Dessiner son arbre des appels pour `f(3)` puis `f(4)` ; combien d'appels chacun déclenche-t-il en tout ?*
2. *En notant* a_n *le nombre total d'appels déclenchés par `f(n)` (l'appel principal compris, donc* $a_0 = a_1 = a_2 = 1$*), justifier que pour* $n \geqslant 3$, $a_n = 1 + a_{n-1} + a_{n-2} + a_{n-3}$.
3. *Montrer par récurrence que* $a_{3k} \geqslant 3^k$*, et en déduire que* a_n *croît au moins exponentiellement, donc que la fonction est inutilisable pour* n *grand.*
4. *Proposer une fonction* itérative *calculant* u_n *en temps* $O(n)$*, qui ne conserve à chaque tour que les trois derniers termes.*

Exercice 5.1.8. *Écrire une fonction récursive `puissance(a, n)` qui calcule* a^n *pour* $n \geqslant 0$*, en s'appuyant sur la relation* $a^n = a \cdot a^{n-1}$ *et le cas de base* $a^0 = 1$*. Combien de multiplications effectue-t-elle ?*

Exercice 5.1.9 (★). (Exponentiation rapide.) *Soit* q *et* r *le quotient et le reste de la division de* n *par* 2*. De* $n = 2q + r$ *on tire* $a^n = (a^2)^q \cdot a^r$*, ou encore* $a^n = (a^q)^2 \cdot a^r$.

1. *En déduire une fonction récursive calculant* a^n *avec un* seul *appel récursif, portant sur l'exposant* $\lfloor n/2 \rfloor$.
2. *Montrer que le nombre de multiplications est en* $O(\log n)$. *Comparer à l'exercice 5.1.8 pour* $n = 1000$.

Exercice 5.1.10. (Exponentiation en base m.) *On reprend l'idée de l'exercice 5.1.9 avec un entier* $m \geqslant 2$ *fixé à la place de 2. Soit* q *et* r *le quotient et le reste de la division de* n *par* m *; de* $n = mq + r$ *on tire* $a^n = (a^m)^q \cdot a^r$.

1. *En déduire une fonction récursive calculant* a^n *avec un* seul *appel récursif, portant sur l'exposant* $\lfloor n/m \rfloor$.
2. *Combien de multiplications chaque appel coûte-t-il pour former* a^m *puis multiplier par* a^r *? Comparer le nombre total de multiplications à celui de l'exercice 5.1.9 : augmenter* m *améliore-t-il le cas* $m = 2$ *?*

Exercice 5.1.11 (★). (Fibonacci en $O(\log n)$ opérations.) *On rappelle la suite de Fibonacci :* $F_0 = 0$, $F_1 = 1$ *et* $F_{n+2} = F_{n+1} + F_n$. *La récursion directe demande un nombre exponentiel d'additions, la version itérative un nombre linéaire ; on va descendre à* $O(\log n)$ *opérations arithmétiques grâce à l'exponentiation rapide. Le point de départ est la relation, valable pour* $n \geqslant 1$,

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}.$$

1. *En notant* $M = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, *en déduire que* $\left(\begin{smallmatrix} F_{n+1} \\ F_n \end{smallmatrix}\right) = M^n \left(\begin{smallmatrix} F_1 \\ F_0 \end{smallmatrix}\right)$ *pour tout* $n \geqslant 0$. *En particulier,* F_n *se lit comme un coefficient de* M^n.
2. *Adapter l'exponentiation rapide de l'exercice 5.1.9 pour calculer la puissance* M^n *d'une matrice* 2×2, *en supposant disposer d'une fonction qui multiplie deux telles matrices.* (L'élément neutre est ici la matrice identité.)
3. *En déduire une fonction calculant* F_n.
4. *Combien d'opérations arithmétiques effectue-t-elle ? Comparer aux versions récursive et itérative.*

5.2 La méthode de mémoïsation

Nous avons vu à la section précédente qu'en traduisant directement la définition de la suite de Fibonacci, on obtient une fonction *exponentielle*, parce qu'elle recalcule sans cesse les mêmes termes. Dans cette section, nous corrigeons ce défaut sans renoncer à l'écriture récursive, grâce à une technique générale : la *mémoïsation*.

L'idée tient en une phrase : *ne jamais calculer deux fois la même chose*. La première fois qu'on évalue f sur un argument, on range le résultat dans une table ; aux appels suivants sur le même argument, on se contente de relire la table au lieu de relancer le

calcul. Le terme consacré pour cette mémorisation des résultats d'une fonction est la *mémoïsation*[2].

Reprenons Fibonacci. Comme argument, n est un entier ; une table indexée par n fait l'affaire. Nous utilisons un *dictionnaire* Python, noté `memo`, qui à un entier déjà rencontré associe la valeur F_n correspondante.

```
def fibonacci_memo(n, memo=None):
    """Renvoie le n-ième terme de Fibonacci par mémoïsation.

    Le dictionnaire ``memo`` retient les termes déjà calculés ; avant de
    lancer un calcul, on regarde s'il n'y figure pas déjà. Chaque terme
    n'est ainsi calculé qu'une seule fois, ce qui ramène le coût à O(n).

    >>> fibonacci_memo(0)
    0
    >>> fibonacci_memo(10)
    55
    >>> [fibonacci_memo(n) for n in range(10)]
    [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
    >>> fibonacci_memo(50)
    12586269025
    """
    if memo is None:
        memo = {}
    if n == 0:
        return 0
    if n == 1:
        return 1
    if n in memo:
        return memo[n]
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]
```

Comparée à la version naïve de la section 5.1.3, la fonction ne change qu'en deux points. Avant de calculer F_n, elle teste si n figure déjà dans `memo` : si oui, elle renvoie la valeur stockée sans aucun appel récursif ; sinon, elle calcule la valeur *une fois*, la range dans `memo`, et la renvoie. Le dictionnaire est passé en argument à chaque appel récursif, de sorte que tous les appels partagent la *même* table[3].

Regardons ce que devient l'arbre des appels. La première branche descend jusqu'à F_1 et F_0 en remplissant `memo` au passage ; toutes les demandes ultérieures portant sur un n

2. Avec un « i » tréma : le mot vient de l'anglais *memoization*, lui-même forgé sur le latin *memorandum*, « ce dont il faut se souvenir ». À ne pas confondre avec la *mémorisation* ordinaire, dont c'est une forme particulière, appliquée aux valeurs d'une fonction.

3. Le paramètre `memo=None` puis l'affectation `memo = {}` créent un dictionnaire neuf au premier appel seulement. On évite ainsi un piège classique de Python : un dictionnaire donné directement comme valeur par défaut serait partagé entre des appels *indépendants* de `fibonacci_memo`, ce qui n'est pas voulu.

déjà calculé sont satisfaites immédiatement, sans redescendre dans l'arbre. Les nœuds qui, dans la section 5.1.3, étaient recalculés encore et encore, sont maintenant « coupés » : il ne reste qu'un petit nombre d'appels effectifs.

Proposition 5.2.1 (Coût de la version mémoïsée)**.** *Pour calculer F_n, la fonction `fibonacci_memo` effectue $O(n)$ additions.*

Démonstration. Chaque valeur F_k, pour $2 \leqslant k \leqslant n$, est calculée — donc rangée dans `memo` — au plus une fois : dès qu'elle y figure, tout appel ultérieur sur k renvoie immédiatement. Or il n'y a qu'une addition par valeur ainsi calculée. Le nombre total d'additions est donc majoré par $n-1$, soit $O(n)$. □

On retrouve le coût linéaire de la version par boucle, mais en gardant la lisibilité de l'écriture récursive. Le prix à payer est l'espace mémoire de la table, qui retient $O(n)$ valeurs : comme souvent, on échange du *temps* contre de l'*espace*.

Remarque 5.2.2. *La mémoïsation n'a rien de propre à Fibonacci. Elle s'applique à* toute *fonction récursive qui rappelle plusieurs fois les mêmes sous-problèmes : il suffit que ses arguments puissent servir de clefs dans une table. C'est une première forme de* programmation dynamique*, le grand paradigme du chapitre 7 ; on y distinguera cette approche* descendante *(on part du problème et l'on mémorise les sous-problèmes au fur et à mesure qu'ils se présentent) d'une approche* montante *qui remplit la table directement, dans le bon ordre, sans récursion.*

5.2.1 Exercices

Exercice 5.2.3. *Le coefficient binomial $\binom{n}{k}$ se calcule par la* règle de Pascal *: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, avec les cas de base $\binom{n}{0} = \binom{n}{n} = 1$.*

1. *Écrire la fonction récursive directe traduisant cette règle, et expliquer pourquoi elle recalcule de nombreuses fois les mêmes coefficients.*
2. *En mémoïser le calcul à l'aide d'un dictionnaire indexé par le couple (n, k). Quel est, à une constante près, le nombre de coefficients distincts calculés pour obtenir $\binom{n}{k}$?*

Exercice 5.2.4. *Modifier la version* naïve *`fibonacci_recursif` pour qu'elle renvoie, en plus de F_n, le nombre d'additions qu'elle a effectuées. Vérifier sur quelques valeurs que ce nombre vaut $F_{n+1} - 1$, et le comparer aux $n-1$ additions de la version linéaire. Pour quelle valeur de n dépasse-t-on le million d'additions ?*

Exercice 5.2.5. *Reprendre la suite (u_n) de l'exercice 5.1.7 ($u_0 = u_1 = u_2 = 0$, puis $u_n = u_{n-1} + u_{n-2} \cdot u_{n-3} + 1$). Mémoïser sa traduction récursive directe, et constater que le coût retombe de exponentiel à linéaire. Comparer avec la version itérative à trois variables demandée à l'exercice 5.1.7.*

5.3 De la récursivité à l'itération : la récursion terminale

Nous avons vu (section 5.1) que la récursivité et l'itération expriment la même idée de répétition. Dans cette section, nous isolons un cas — la *récursion terminale* — où l'on passe mécaniquement de l'une à l'autre. En toute généralité, *tout* programme récursif peut être réécrit sous forme *itérative*, en gérant « à la main » la pile des appels (section 5.1.2) ; mais cette transformation est lourde, là où le cas terminal est immédiat.

5.3.1 Récursion terminale

Définition 5.3.1 (Récursion terminale)**.** *Une fonction récursive est dite* récursive terminale *lorsqu'elle ne contient qu'un seul appel récursif, et que cet appel est la* dernière *action effectuée : son résultat est renvoyé tel quel, sans aucun calcul après le retour.*

Dit autrement, dans une récursion terminale il n'y a rien « en attente » au moment de l'appel récursif : le résultat de l'appel *est* le résultat de la fonction. C'est ce qui la distingue, par exemple, de la factorielle de la section 5.1, où après l'appel `factorielle(n - 1)` il reste encore à multiplier par `n`.

Une telle fonction a toujours la même allure. Notons c le test d'arrêt, g la valeur renvoyée dans le cas de base, et s la transformation qui fait « décroître » l'argument :

```
def schema_recursif(x, c, g, s):
    """Forme générale d'une fonction récursive terminale.

    ``c`` est le test d'arrêt, ``g`` donne le résultat dans le cas de base,
    et ``s`` fait décroître l'argument. Le seul appel récursif est la
    dernière action de la fonction : c'est ce qui la rend terminale.

    Sur l'exemple de l'algorithme d'Euclide, où x = (a, b) :

    >>> arret = lambda ab: ab[1] == 0
    >>> base = lambda ab: ab[0]
    >>> pas = lambda ab: (ab[1], ab[0] % ab[1])
    >>> schema_recursif((12, 8), arret, base, pas)
    4
    """
    if c(x):
        return g(x)
    return schema_recursif(s(x), c, g, s)


def schema_iteratif(x, c, g, s):
    """Version itérative équivalente, obtenue par dérécursivation.

    On rejoue x <- s(x) tant que le test d'arrêt n'est pas vérifié, puis on
```

```
    renvoie g(x). Le résultat est le même que ``schema_recursif`` :

    >>> arret = lambda ab: ab[1] == 0
    >>> base = lambda ab: ab[0]
    >>> pas = lambda ab: (ab[1], ab[0] % ab[1])
    >>> schema_iteratif((12, 8), arret, base, pas)
    4
    >>> import math
    >>> all(schema_iteratif((a, b), arret, base, pas)
    ...     == schema_recursif((a, b), arret, base, pas)
    ...     == math.gcd(a, b)
    ...     for a in range(1, 30) for b in range(0, 30))
    True
    """
    while not c(x):
        x = s(x)
    return g(x)
```

La première fonction ci-dessus, `schema_recursif`, est le patron de toute récursion terminale[4] ; la seconde, `schema_iteratif`, en est la traduction itérative. Le passage de l'une à l'autre se lit directement : *on remplace la récursion par une boucle « tant que le cas de base n'est pas atteint »*.

Proposition 5.3.2 (Dérécursivation d'une récursion terminale)**.** *Les deux fonctions `schema_recursif` et `schema_iteratif` calculent la même valeur sur toute entrée pour laquelle elles terminent.*

Démonstration. Suivons ce que produit `schema_recursif(`x`)`. Si $c(x)$ est vrai, elle renvoie $g(x)$. Sinon elle renvoie `schema_recursif(`$s(x)$`)`, et l'on recommence le même raisonnement sur $s(x)$. La suite des arguments rencontrés est donc x, $s(x)$, $s(s(x))$, $\ldots$, et le calcul s'arrête au premier de ces arguments — appelons-le $x^\star$ — pour lequel $c(x^\star)$ est vrai ; la valeur renvoyée est alors $g(x^\star)$.

La boucle de `schema_iteratif(`x`)` parcourt exactement la même suite : partant de x, elle remplace x par $s(x)$ tant que $c(x)$ est faux, c'est-à-dire jusqu'à atteindre ce même $x^\star$; elle renvoie alors $g(x^\star)$. Les deux fonctions renvoient la même valeur. (Elles terminent dans les mêmes cas, puisque c'est la même suite d'arguments qui doit atteindre le test d'arrêt.) □

5.3.2 L'algorithme d'Euclide

L'exemple emblématique de récursion terminale est l'algorithme d'Euclide, qui calcule le pgcd de deux entiers. Il repose sur l'identité $\mathrm{pgcd}(a, b) = \mathrm{pgcd}(b,\ a \bmod b)$ pour $b \neq 0$, et sur le cas de base $\mathrm{pgcd}(a, 0) = a$.

4. Les arguments c, g, s y sont des fonctions passées en paramètre, ce qui permet d'écrire le schéma une fois pour toutes et de l'instancier sur n'importe quel cas particulier — ici l'algorithme d'Euclide, dans les doctests.

```python
def euclide_recursif(a, b):
    """Renvoie le pgcd de a et b (entiers >= 0, non tous deux nuls).

    Version récursive terminale de l'algorithme d'Euclide : le seul appel
    récursif est aussi la dernière chose que fait la fonction. Le cas de
    base est b == 0, où le pgcd vaut a.

    >>> euclide_recursif(12, 8)
    4
    >>> euclide_recursif(8, 12)
    4
    >>> euclide_recursif(17, 5)
    1
    >>> euclide_recursif(42, 0)
    42
    >>> import math
    >>> all(euclide_recursif(a, b) == math.gcd(a, b)
    ...     for a in range(1, 40) for b in range(0, 40))
    True
    """
    if b == 0:
        return a
    return euclide_recursif(b, a % b)
```

C'est bien une récursion terminale : l'unique appel récursif, `euclide_recursif(b, a % b)`, est renvoyé tel quel. On y reconnaît le schéma précédent, avec pour argument le couple $x = (a, b)$, pour test d'arrêt $c(x) =$ « $b = 0$ », pour valeur de base $g(x) = a$, et pour pas $s(x) = (b,\ a \bmod b)$. La proposition 5.3.2 fournit donc aussitôt la version itérative.

```python
def euclide_iteratif(a, b):
    """Renvoie le pgcd de a et b (entiers >= 0, non tous deux nuls).

    Version itérative, obtenue en dérécursivant la récursion terminale
    de ``euclide_recursif`` : la boucle ``while`` rejoue les appels
    ``(a, b) <- (b, a % b)`` tant que le cas de base ``b == 0`` n'est pas
    atteint. Le test en tête de boucle traite correctement le cas b == 0
    (le pgcd vaut alors a), là où une division placée avant le test
    échouerait.

    >>> euclide_iteratif(12, 8)
    4
    >>> euclide_iteratif(42, 0)
    42
    >>> euclide_iteratif(0, 42)
    42
```

```
    >>> import math
    >>> all(euclide_iteratif(a, b) == math.gcd(a, b)
    ...     for a in range(0, 40) for b in range(0, 40) if (a, b) != (0, 0))
    True
    """
    while b != 0:
        a, b = b, a % b
    return a
```

Avertissement 5.3.3. *L'ordre des instructions dans la boucle n'est pas indifférent. Le test `b != 0` doit précéder la division : si l'on calculait `a % b` avant de tester b, l'appel `euclide_iteratif(42, 0)` provoquerait une division par zéro, alors que la version récursive, elle, traite correctement ce cas en renvoyant* 42. *Une dérécursivation négligente peut ainsi introduire un bug à la frontière ; en plaçant le test en tête de boucle, comme le fait le schéma général, les deux versions coïncident sur* toutes *les entrées,* $b = 0$ *compris.*

5.3.3 Complexité de l'algorithme d'Euclide

Combien de divisions l'algorithme effectue-t-il ? Plutôt que de compter les itérations une à une, observons à quelle *vitesse* décroissent les restes successifs. Posons $r_0 = a$, $r_1 = b$, et notons $r_2, r_3, \dots$ les restes calculés : à chaque étape, r_{i+1} est le reste de la division de r_{i-1} par r_i, jusqu'à tomber sur un reste nul. Le point clef est le suivant.

Lemme 5.3.4 (Les restes sont divisés par deux tous les deux pas)**.** *Pour tout* $i \geqslant 0$ *tel que* r_{i+2} *est défini, on a* $r_{i+2} < \frac{1}{2}\, r_i$.

Démonstration. Par définition du reste, $r_{i+2} < r_{i+1} \leqslant r_i$. Distinguons deux cas selon la taille de r_{i+1}.

Cas $r_{i+1} \leqslant \frac{1}{2} r_i$. Alors $r_{i+2} < r_{i+1} \leqslant \frac{1}{2} r_i$, et c'est gagné.

Cas $r_{i+1} > \frac{1}{2} r_i$. Alors $r_i < 2\, r_{i+1}$, donc le quotient de la division de r_i par r_{i+1} vaut 1 : on a $r_i = 1 \cdot r_{i+1} + r_{i+2}$, d'où $r_{i+2} = r_i - r_{i+1} < r_i - \frac{1}{2} r_i = \frac{1}{2} r_i$.

Dans les deux cas, $r_{i+2} < \frac{1}{2} r_i$. □

Appliquons le lemme en partant de $r_1 = b$. Une récurrence sur k donne alors $r_{1+2k} < \left(\frac{1}{2}\right)^k b$ pour tout k tel que ce reste existe : le cas $k = 0$ est l'égalité $r_1 = b$, et le passage de k à $k+1$ combine le lemme et l'hypothèse de récurrence, $r_{1+2(k+1)} < \frac{1}{2} r_{1+2k} < \left(\frac{1}{2}\right)^{k+1} b$. Or les restes sont des entiers $\geqslant 1$ tant qu'on n'a pas atteint le reste nul final ; dès que $\left(\frac{1}{2}\right)^k b \leqslant 1$, soit $k \geqslant \log_2 b$, la suite des restes non nuls est donc épuisée. Le nombre d'étapes est ainsi majoré par environ $2 \log_2 b$: l'algorithme d'Euclide effectue

$$O(\log b)$$

divisions. C'est remarquablement peu : le coût est proportionnel au nombre de *chiffres* de b, et non à b lui-même.

Remarque 5.3.5. *Cette borne est optimale, au sens du chapitre 3 : elle est atteinte. Les entrées les plus coûteuses sont précisément deux nombres de Fibonacci consécutifs,* $a = F_{n+1}$ *et* $b = F_n$ *: la division y donne toujours un quotient* 1*, les restes parcourent toute la suite de Fibonacci en descendant, et l'algorithme prend de l'ordre de* n *étapes — soit bien* $\Theta(\log b)$*, puisque* F_n *croît comme* φ^n*, où* $\varphi = \frac{1+\sqrt{5}}{2}$ *est le nombre d'or rencontré à la section 5.1.3.*

5.3.4 Un cas non terminal : la factorielle

Toutes les fonctions récursives ne sont pas terminales. La factorielle de la section 5.1 ne l'est pas : après l'appel `factorielle(n - 1)`, il reste à multiplier le résultat par `n`. On ne peut donc pas lui appliquer directement la proposition 5.3.2.

Il existe toutefois une astuce classique pour *rendre* terminale une telle fonction : transporter le travail « en attente » dans un argument supplémentaire, l'*accumulateur*, qui contient le résultat partiel déjà calculé. Pour la factorielle, on accumule le produit au fur et à mesure de la descente ; quand on atteint le cas de base, l'accumulateur contient déjà $n!$ et il n'y a plus qu'à le renvoyer. La fonction devient terminale, donc dérécursivable en la boucle attendue. Le détail fait l'objet de l'exercice 5.3.6.

5.3.5 Exercices

Exercice 5.3.6. (Factorielle terminale.)

1. *Écrire une fonction récursive* `factorielle(n, accumulateur)` *qui soit* terminale, *en utilisant un accumulateur initialisé à* 1 *: au lieu de multiplier* après *l'appel récursif, on passe à l'appel le produit déjà constitué. Vérifier qu'elle renvoie bien* $n!$.
2. *En appliquant la proposition 5.3.2, en déduire une fonction itérative calculant* $n!$ *par une simple boucle.*

Exercice 5.3.7. *On exécute l'algorithme d'Euclide sur* $a = 21$*,* $b = 13$ *(deux nombres de Fibonacci consécutifs). Écrire la suite des restes* $r_0, r_1, r_2, \ldots$ *jusqu'au reste nul, et compter les divisions effectuées. Vérifier que tous les quotients valent* 1*, conformément à la remarque 5.3.5.*

Exercice 5.3.8. *Notons* $S(n)$ *la somme des chiffres (en base* 10*) d'un entier* $n \geqslant 0$ *; elle se calcule récursivement par* $S(n) = (n \bmod 10) + S(\lfloor n/10 \rfloor)$*, avec* $S(0) = 0$*.*

1. *Cette récursion est-elle terminale ? Justifier.*
2. *La rendre terminale à l'aide d'un accumulateur, puis la dérécursiver en une boucle* `while`*.*

5.4 Récursivité mutuelle ou croisée

Jusqu'ici, une fonction récursive s'appelait *elle-même*. Mais la récursivité peut aussi être *partagée* entre plusieurs fonctions : deux fonctions dont chacune appelle l'autre dans

sa définition. On parle alors de *récursivité mutuelle* (ou *croisée*). Aucune des deux ne s'appelle directement, et pourtant le comportement est bien récursif.

L'exemple classique teste la parité d'un entier sans aucune opération arithmétique autre que le passage au prédécesseur. L'idée repose sur une évidence : n est pair si et seulement si $n-1$ est impair, et réciproquement ; et 0 est pair.

```
def est_pair(n):
    """Teste si l'entier n >= 0 est pair, sans arithmétique autre que -1.

    Définie par récursivité mutuelle avec ``est_impair`` : 0 est pair, et
    n est pair si et seulement si n-1 est impair.

    >>> est_pair(0)
    True
    >>> est_pair(7)
    False
    >>> [est_pair(n) for n in range(6)]
    [True, False, True, False, True, False]
    >>> all(est_pair(n) == (n % 2 == 0) for n in range(50))
    True
    """
    if n == 0:
        return True
    return est_impair(n - 1)


def est_impair(n):
    """Teste si l'entier n >= 0 est impair, par récursivité mutuelle.

    0 n'est pas impair, et n est impair si et seulement si n-1 est pair.

    >>> est_impair(0)
    False
    >>> est_impair(7)
    True
    >>> all(est_impair(n) == (n % 2 == 1) for n in range(50))
    True
    """
    if n == 0:
        return False
    return est_pair(n - 1)
```

La fonction `est_pair` appelle `est_impair`, qui appelle `est_pair`, et ainsi de suite, chaque appel portant sur un entier *plus petit* d'une unité. C'est ce décrément qui garantit qu'on atteint le cas de base 0 : les deux fonctions terminent donc sur toute entrée $n \geqslant 0$. Quant

à leur correction, elle se démontre, comme on pouvait s'y attendre, par une récurrence — mais une récurrence qui porte sur les *deux* fonctions à la fois.

Proposition 5.4.1 (Correction de `est_pair` et `est_impair`)**.** *Pour tout entier $n \geqslant 0$, `est_pair(n)` renvoie `True` si n est pair et `False` sinon, et `est_impair(n)` a le comportement opposé.*

Démonstration. On démontre par récurrence sur n la propriété conjointe P_n : « `est_pair(`n`)` renvoie le bon résultat *et* `est_impair(`n`)` renvoie le bon résultat ». Énoncer les deux ensemble est essentiel, puisque chaque fonction s'appuie sur l'autre.

Initialisation. Pour $n = 0$: `est_pair(0)` renvoie `True` et 0 est pair ; `est_impair(0)` renvoie `False` et 0 n'est pas impair. Donc P_0 est vraie.

Hérédité. Soit $n \geqslant 1$; supposons P_{n-1} vraie. L'appel `est_pair(`n`)` renvoie `est_impair(`$n-1$`)`, qui par P_{n-1} vaut `True` si et seulement si $n-1$ est impair, c'est-à-dire si et seulement si n est pair : correct. Symétriquement, `est_impair(`n`)` renvoie `est_pair(`$n-1$`)`, qui vaut `True` si et seulement si $n-1$ est pair, c'est-à-dire si et seulement si n est impair : correct également. Donc P_n est vraie.

Par récurrence, P_n est vraie pour tout $n \geqslant 0$. □

La leçon dépasse cet exemple un peu artificiel : *quand des fonctions sont définies par récursivité mutuelle, leur correction se démontre par une seule récurrence portant simultanément sur toutes les fonctions du groupe.* On ne peut pas raisonner sur l'une en supposant l'autre déjà connue : il faut les traiter d'un bloc.

Remarque 5.4.2. *La récursivité mutuelle n'est pas toujours indispensable. Ici, on peut tester la parité par une seule fonction — un drapeau booléen qui bascule à chaque décrément (exercice 5.4.3) — ou, plus directement, par `n % 2 == 0`. Mais certains objets se prêtent si naturellement à une description croisée qu'il serait artificiel de l'éviter ; nous en rencontrerons en analysant des expressions arithmétiques à la section 5.5, et l'exercice étoilé 5.4.5 en donne un exemple où les deux fonctions sont réellement inséparables.*

5.4.1 Exercices

Exercice 5.4.3. *Réécrire `est_pair` sans récursivité mutuelle, sous la forme d'une unique boucle : partir de la réponse `True` et la faire basculer à chaque fois qu'on retranche 1 à n, jusqu'à atteindre 0. Justifier la correction par un invariant de boucle.*

Exercice 5.4.4. *On veut tester si $n \geqslant 0$ est un multiple de 3 en n'utilisant que le prédécesseur. Définir trois fonctions mutuellement récursives `reste0`, `reste1`, `reste2` telles que `restek(n)` renvoie `True` si et seulement si $n \equiv k \pmod 3$. Quelle est la relation de récursion croisée entre elles ?*

Exercice 5.4.5 (★). (Suites de Hofstadter.) *On définit deux suites par récursivité mutuelle :*

$$F(0) = 1, \quad F(n) = n - M(F(n-1)) \quad \textit{et} \quad M(0) = 0, \quad M(n) = n - F(M(n-1)).$$

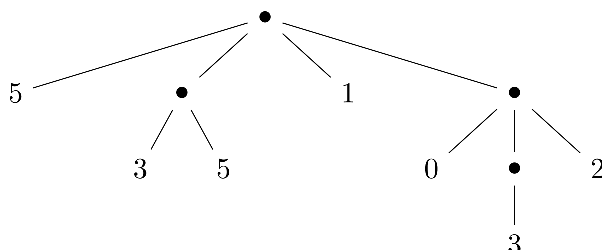
1. *Implémenter F et M par deux fonctions mutuellement récursives, et calculer leurs onze premiers termes.*
2. *Contrairement à `est_pair`, on ne peut pas « découpler » ces deux suites. Expliquer pourquoi, puis expliquer pourquoi la version naïve devient rapidement lente. Comment la mémoïsation (section 5.2) s'applique-t-elle ici ?*

5.5 Objets définis récursivement : les arbres

Jusqu'à présent, nos fonctions récursives opéraient sur des entiers. Dans cette section, nous appliquons la récursivité à des objets eux-mêmes définis récursivement : les *arbres*, où un arbre est fait de sous-arbres, qui sont eux-mêmes des arbres. Sur de tels objets, une fonction récursive décalque simplement la définition de l'objet.

5.5.1 Les arbres

Convenons d'appeler *arbre* l'un des deux objets suivants : soit une *feuille*, étiquetée par un entier naturel ; soit un *nœud interne* muni d'une liste ordonnée de fils $(t_1, \ldots, t_p)$, avec $p \geqslant 1$, chaque t_i étant lui-même un arbre. Cette description est *récursive* : elle définit les arbres à partir d'arbres plus petits, en s'arrêtant sur les feuilles. On peut la rendre rigoureuse en construisant les arbres par « étages », selon leur hauteur [5].
Voici un arbre, dessiné à la manière habituelle (racine en haut) ; les nœuds internes sont marqués •, les feuilles portent leur étiquette :



L'ordre des fils *compte* : deux nœuds dont les fils sont les mêmes mais rangés différemment forment des arbres différents.

Pour manipuler les arbres en machine, il faut les *représenter*. Une représentation naturelle en Python suit la définition récursive elle-même : une feuille étiquetée n est représentée par l'entier `n` ; un nœud interne de fils $t_1, \ldots, t_p$ est représenté par la *liste* $[\mathtt{R}_1, \ldots, \mathtt{R}_p]$ des représentations de ses fils. L'arbre ci-dessus s'écrit ainsi `[5, [3, 5], 1, [0, [3], 2]]`. Distinguer une feuille d'un nœud interne revient alors à distinguer un entier d'une liste, ce que fait la fonction `est_feuille`.

5. Formellement : les arbres de hauteur 0 sont les entiers, $\mathcal{A}_0 = \mathbb{N}$; pour $h > 0$, $\mathcal{A}_h = \mathcal{A}_{h-1} \cup \{(t_1, \ldots, t_p) \mid p \geqslant 1,\ t_i \in \mathcal{A}_{h-1}\}$; et $\mathcal{A} = \bigcup_{h \in \mathbb{N}} \mathcal{A}_h$. Tout arbre apparaît à un étage fini.

5.5.2 Des algorithmes récursifs sur les arbres

Tout algorithme sur les arbres suit le même moule, dicté par la définition de l'objet : *traiter à part le cas d'une feuille, et, pour un nœud interne, combiner les résultats des appels récursifs sur ses fils.* Prenons le décompte des feuilles.

```
def est_feuille(a):
    """Teste si l'arbre a est réduit à une feuille (un entier).

    Convention de représentation : une feuille étiquetée n est l'entier n ;
    un noeud interne de fils t_1, ..., t_p est la liste [t_1, ..., t_p]
    (avec p >= 1).

    >>> est_feuille(3)
    True
    >>> est_feuille([3, 5])
    False
    """
    return isinstance(a, int)


def nombre_feuilles(a):
    """Renvoie le nombre de feuilles de l'arbre a.

    Une feuille en a une (elle-même) ; un noeud interne en a autant que la
    somme de celles de ses fils.

    >>> nombre_feuilles(3)
    1
    >>> nombre_feuilles([3, 5])
    2
    >>> nombre_feuilles([5, [3, 5], 1, [0, [3], 2]])
    7
    """
    if est_feuille(a):
        return 1
    return sum(nombre_feuilles(fils) for fils in a)


def hauteur(a):
    """Renvoie la hauteur de l'arbre a (longueur max. d'un chemin racine-feuille).

    Une feuille est de hauteur 0 ; un noeud interne a pour hauteur 1 de plus
    que la plus grande des hauteurs de ses fils.
```

```
    >>> hauteur(3)
    0
    >>> hauteur([3, 5])
    1
    >>> hauteur([5, [3, 5], 1, [0, [3], 2]])
    3
    """
    if est_feuille(a):
        return 0
    return 1 + max(hauteur(fils) for fils in a)
```

La fonction `nombre_feuilles` dit exactement ceci : une feuille compte pour une feuille ; un nœud interne en compte autant que la somme de celles de ses fils. Sa correction se prouve par récurrence — mais une récurrence qui porte sur l'arbre, à travers sa *taille*.

Proposition 5.5.1 (Correction de `nombre_feuilles`). *Pour tout arbre a, l'appel* `nombre_feuilles(a)` *se termine et renvoie le nombre de feuilles de a.*

Démonstration. Notons $|a|$ la *taille* de a, c'est-à-dire son nombre total de nœuds (feuilles comprises). Nous raisonnons par récurrence forte sur $|a|$; le point qui fait marcher l'argument est que chaque fils d'un nœud interne est un arbre *strictement plus petit* que l'arbre entier.

Cas de base. Si $|a| = 1$, l'arbre a est une feuille. Le test `est_feuille(`a`)` est vrai et la fonction renvoie 1, qui est bien le nombre de feuilles d'une feuille.

Hérédité. Soit a un arbre de taille $t \geqslant 2$, et supposons la proposition acquise pour tout arbre de taille $< t$. Alors a est un nœud interne, de fils $t_1, \ldots, t_p$. Chaque t_i a une taille strictement inférieure à t (il lui manque au moins la racine de a) ; par hypothèse de récurrence, `nombre_feuilles(`t_i`)` renvoie le nombre de feuilles de t_i. Or les feuilles de a sont exactement celles de ses fils, réparties sans chevauchement entre $t_1, \ldots, t_p$: leur nombre est la somme des nombres de feuilles des t_i. C'est précisément ce que renvoie la fonction. La terminaison suit de la même décroissance stricte des tailles. □

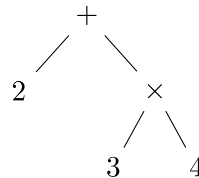
Quant au coût, il est linéaire en la taille de l'arbre : la fonction est appelée une fois par nœud, et le travail propre à chaque nœud (le test, et l'addition des résultats des fils) est proportionnel à son nombre de fils. La somme des nombres de fils sur tous les nœuds vaut le nombre d'arêtes, soit $|a| - 1$; le coût total est donc $O(|a|)$.

La fonction `hauteur`, donnée dans le même fichier, suit exactement le même moule : 0 pour une feuille, et 1 de plus que la plus grande hauteur des fils pour un nœud interne. On y reconnaît la traduction littérale de la définition récursive de la hauteur.

5.5.3 Une application : évaluer une expression arithmétique

Les arbres servent notamment à représenter des *expressions arithmétiques*. Considérons les expressions complètement parenthésées formées d'entiers et des opérations binaires $+$ et $\times$, comme $(2 + (3 \times 4))$. À une telle expression on associe naturellement un arbre :

les feuilles portent les entiers, et chaque nœud interne porte l'opération qui combine ses deux sous-expressions. L'expression $(2 + (3 \times 4))$ donne :



Nous représentons un tel arbre binaire par un entier (feuille) ou par une liste `[operateur, gauche, droite]`, où `operateur` vaut `'+'` ou `'*'`. L'expression ci-dessus s'écrit alors `['+', 2, ['*', 3, 4]]`. L'évaluation se lit directement sur la structure de l'arbre : la valeur d'une feuille est son entier ; la valeur d'un nœud interne s'obtient en évaluant ses deux fils, puis en leur appliquant l'opération.

```
def evaluation(a):
    """Évalue une expression arithmétique représentée par un arbre binaire.

    Représentation : un entier est une feuille (valeur immédiate) ; un noeud
    interne est une liste [operateur, gauche, droite] où operateur vaut '+'
    ou '*'. On évalue récursivement les deux sous-arbres, puis on combine.

    >>> evaluation(2)
    2
    >>> evaluation(['+', 2, ['*', 3, 4]])
    14
    >>> evaluation(['+', 2, ['*', ['+', 1, 4], 3]])
    17
    """
    if isinstance(a, int):
        return a
    operateur, gauche, droite = a
    if operateur == '+':
        return evaluation(gauche) + evaluation(droite)
    return evaluation(gauche) * evaluation(droite)
```

La correction se démontre par récurrence sur la taille de l'expression, selon le schéma désormais familier. Pour une feuille (taille 1), la fonction renvoie l'entier, qui est bien sa valeur. Pour un nœud interne $(F_1 \odot F_2)$, où $\odot \in \{+, \times\}$, les deux sous-expressions F_1 et F_2 sont de taille strictement plus petite ; par hypothèse de récurrence, leurs évaluations récursives renvoient leurs valeurs respectives, et la fonction renvoie leur somme ou leur produit — la valeur cherchée. (Le cas $\odot = +$ et le cas $\odot = \times$ se traitent de la même façon.) Le coût est, là encore, linéaire en la taille de l'expression.

5.5.4 Exercices

Exercice 5.5.2. *Sur le modèle de* `nombre_feuilles`*, écrire une fonction* `somme_feuilles(a)` *qui renvoie la somme des étiquettes des feuilles d'un arbre. La tester sur l'arbre* `[5, [3, 5], 1, [0, [3], 2]]` *(on doit trouver* 19*).*

Exercice 5.5.3. *Le* symétrique *d'un arbre s'obtient en inversant l'ordre des fils de chaque nœud interne. Écrire une fonction* `symetrique(a)` *qui le renvoie.* Indication *: une feuille est son propre symétrique ; pour un nœud interne, inverser l'ordre des fils* et *symétriser chacun d'eux. Vérifier que l'appliquer deux fois redonne l'arbre de départ.*

Exercice 5.5.4. *On reprend les expressions arithmétiques et leur fonction* `evaluation`*.*

1. *Dessiner l'arbre de l'expression* $(2 + ((1 + 4) \times 3))$ *et donner sa représentation en liste. Vérifier que* `evaluation` *renvoie* 17*.*
2. *Écrire une fonction* `taille(a)` *qui compte le nombre de nœuds de l'arbre d'une expression, et confirmer la borne de coût* $O(taille)$ *annoncée.*

Exercice 5.5.5 (★). *Deux arbres sont* égaux à rotation près *si l'on passe de l'un à l'autre en permutant l'ordre des fils de certains nœuds internes. Par exemple,* `[1, [1, 3, 2]]`*,* `[[1, 3, 2], 1]` *et* `[[1, 2, 3], 1]` *sont tous égaux à rotation près. Écrire une fonction* `egaux_rotation(a, b)` *qui teste cette égalité.* Indication *: associer à chaque arbre une* forme canonique *en triant récursivement les formes de ses fils, de sorte que deux arbres égaux à rotation près aient la même forme canonique ; il reste à comparer les deux formes.*

6 Diviser pour régner

Résumé

Ce chapitre présente l'un des grands paradigmes de l'algorithmique, le « diviser pour régner » : résoudre un problème en le découpant en sous-problèmes de même nature mais plus petits, puis en recombinant leurs solutions. Nous l'appliquons d'abord au tri — fusion de deux listes triées, *tri fusion*, puis *tri rapide* — avant d'établir qu'aucun tri par comparaisons ne peut descendre sous $O(n \log n)$ opérations, une borne que le *tri par comptage* contourne en changeant les règles du jeu. Le chapitre se clôt sur une autre application du paradigme, le produit rapide de polynômes, et, en complément (★), sur la sélection en temps linéaire.

Prérequis

La récursivité et le raisonnement par récurrence (chapitre 5) ; les notations de complexité, en particulier $O(\cdot)$ (chapitre 3) ; les tris naïfs et la recherche dichotomique (chapitre 4).

Objectifs

À l'issue de ce chapitre, vous saurez reconnaître un problème qui se prête au découpage récursif, écrire un algorithme « diviser pour régner » et prouver sa correction par récurrence, enfin établir puis résoudre la récurrence qui gouverne sa complexité.

Les tris naïfs du chapitre précédent — tri à bulles, par sélection, par insertion — trient une liste de n éléments au prix de $O(n^2)$ comparaisons. Tant que n reste petit, peu importe ; mais pour $n = 10^6$, c'est de l'ordre de 10^{12} opérations, soit plusieurs minutes de calcul là où l'on attend une fraction de seconde. *Peut-on trier plus vite ?*

La réponse passe par une idée que l'on retrouve partout en informatique : « diviser pour régner ». Plutôt que d'attaquer le problème de front, on le découpe en sous-problèmes de même nature mais de taille moitié, on les résout récursivement, puis on recombine leurs solutions. Nous en avons déjà croisé un exemple avec la recherche dichotomique, qui élimine la moitié des candidats à chaque test. Toute la subtilité d'un algorithme « diviser pour régner » tient dans l'étape de recombinaison : c'est là que se joue son efficacité.

Pour le tri, cette recombinaison repose sur une opération étonnamment simple. Imaginez deux piles de copies, chacune déjà rangée par ordre alphabétique ; pour les réunir en une seule pile triée, nul besoin de tout recommencer : il suffit de comparer les deux copies du dessus et de prélever la plus petite, encore et encore. C'est cette opération — la *fusion*

de deux listes triées — que nous étudions dans cette première section, avant d'en faire le moteur du tri fusion.

6.1 Fusion de deux listes triées

On dispose de deux listes A et B, de longueurs respectives n et m, *déjà triées* en ordre croissant, et l'on veut construire une liste C de longueur $n + m$ contenant tous leurs éléments, elle aussi triée.

Une première idée serait de recopier les éléments de A puis ceux de B dans une même liste, et d'appliquer à celle-ci un tri du chapitre précédent. Mais ce serait gâcher l'information dont on dispose : A et B sont *déjà* triées, et un tri général n'en tiendrait aucun compte. Exploitons-la plutôt.

L'idée est de parcourir A et B de gauche à droite en parallèle. À tout instant, on garde un indice i dans A et un indice j dans B ; les éléments « courants » $A[i]$ et $B[j]$ sont les premiers de chaque liste à ne pas avoir encore trouvé leur place dans C. Comme A et B sont triées, le prochain élément à placer dans C est nécessairement le plus petit des deux : on le recopie, et on avance d'un cran dans la liste dont il provient. Lorsque l'une des listes est épuisée, il ne reste qu'à recopier la fin de l'autre.

L'algorithme ci-dessous met en œuvre ce principe. On y maintient un troisième indice k, la position courante dans C.[1]

```python
def fusion(A, B):
    """Fusionne deux listes croissantes ``A`` et ``B`` en une seule.

    La liste renvoyée contient tous les éléments de ``A`` et de ``B``,
    rangés en ordre croissant.

    >>> fusion([1, 4, 7, 8, 10], [2, 3, 5, 6, 9])
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    >>> fusion([], [1, 2])
    [1, 2]
    >>> fusion([1, 2], [])
    [1, 2]
    >>> fusion([], [])
    []
    >>> fusion([1, 1, 2], [1, 3])
    [1, 1, 1, 2, 3]
    """
    n = len(A)
    m = len(B)
    C = [None] * (n + m)
```

1. On l'a écrit avec une seule boucle, gouvernée par k, plutôt qu'avec une boucle « tant que $i < n$ et $j < m$ » suivie d'une phase de recopie : c'est un peu moins naturel, mais cela donne un *unique* invariant de boucle, ce qui simplifie la preuve de correction.

```
i = 0
j = 0
for k in range(n + m):
    if i >= n:
        C[k] = B[j]
        j = j + 1
    elif j >= m:
        C[k] = A[i]
        i = i + 1
    elif A[i] < B[j]:
        C[k] = A[i]
        i = i + 1
    else:
        C[k] = B[j]
        j = j + 1
return C
```

Conformément à notre convention, les listes sont indexées à partir de 0 : A va de $A[0]$ à $A[n-1]$, et de même pour B et C. Les deux premiers tests traitent le cas où l'une des listes est épuisée ($i \geqslant n$ ou $j \geqslant m$) ; sinon, on compare les éléments courants et l'on prélève le plus petit.

Exemple 6.1.1. *Fusionnons $A = [1, 4, 7, 8, 10]$ et $B = [2, 3, 5, 6, 9]$. À chaque étape, on compare l'élément courant de A à celui de B et l'on retient le plus petit ; le tableau ci-dessous indique, pour chaque case de C, de quelle liste provient son contenu.*

C	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
provenance	*A*	*B*	*B*	*A*	*B*	*B*	*A*	*A*	*B*	*A*

Les deux derniers éléments, 9 et 10, illustrent le cas d'épuisement : une fois A entièrement parcourue (après avoir placé 8), il ne reste qu'à recopier la fin de B, puis inversement.

6.1.1 Correction

Pour prouver que la fusion est correcte, nous dégageons une propriété vraie après chaque tour de boucle — un *invariant de boucle* — puis nous en déduisons le résultat en sortie de boucle. C'est la méthode standard pour raisonner sur une boucle, et nous y reviendrons à de nombreuses reprises.

Invariant 6.1.2 (de la boucle de fusion)**.** *Supposons A et B non vides. À la fin de l'itération de la boucle d'indice k (pour $0 \leqslant k \leqslant n+m-1$) :*

(a) on a $i+j = k+1$, et les cases $C[0], \ldots, C[k]$ contiennent, rangés en ordre croissant, les éléments $A[0], \ldots, A[i-1]$ et $B[0], \ldots, B[j-1]$;

(b) tout élément de A d'indice $\geqslant i$ et tout élément de B d'indice $\geqslant j$ est supérieur ou égal à $C[k]$.

En d'autres termes, après le tour numéro k, la portion déjà remplie de C est exactement la fusion triée des préfixes déjà consommés de A et B (propriété (a)), et tout ce qui reste à traiter est plus grand que le dernier élément placé (propriété (b)).

Démonstration. Récurrence sur k.

Initialisation ($k = 0$). Comme A et B sont non vides, les deux premiers tests sont faux et l'on compare $A[0]$ et $B[0]$. Deux cas se présentent ; traitons $A[0] < B[0]$, le cas $A[0] \geqslant B[0]$ étant symétrique. On place alors $C[0] = A[0]$ et l'on passe à $i = 1$, $j = 0$. On a bien $i + j = 1 = k + 1$, et $C[0]$ contient le seul élément $A[0]$: la propriété (a) est vérifiée. Pour (b) : comme A est triée, $A[i'] \geqslant A[0] = C[0]$ pour tout $i' \geqslant 1$; et comme B est triée avec $B[0] > A[0]$, on a $B[j'] \geqslant B[0] > C[0]$ pour tout $j' \geqslant 0$. La propriété (b) est donc vérifiée.

Hérédité. Supposons les propriétés (a) et (b) vraies à la fin de l'itération $k = \ell$, et considérons l'itération $\ell + 1$. Notons i, j les valeurs des indices au début de cette itération ; par l'hypothèse de récurrence, $i + j = \ell + 1$.

Premier cas : une des listes est épuisée, disons A (le cas où B est épuisée est identique en échangeant les rôles). L'itération place en position $\ell + 1$ l'élément courant $B[j]$ de la liste restante. C'est le plus petit des éléments de B non encore placés ; par l'hypothèse de récurrence (b), il est supérieur ou égal à $C[\ell]$. Donc $C[0], \dots, C[\ell + 1]$ est encore trié : on a (a). Comme B est triée, les éléments de B restant après celui-ci sont tous $\geqslant C[\ell + 1]$, d'où (b).

Second cas : aucune des deux listes n'est épuisée. L'alternative compare $A[i]$ et $B[j]$ et place en position $\ell + 1$ le plus petit des deux. Par l'hypothèse de récurrence (b), $A[i]$ et $B[j]$ sont tous deux $\geqslant C[\ell]$; leur minimum $C[\ell + 1]$ l'est donc aussi, et $C[0], \dots, C[\ell + 1]$ reste trié : on a (a). Enfin $C[\ell + 1]$ est, par construction, le plus petit des éléments encore présents dans A et B ; comme ces listes sont triées, tous les éléments restants lui sont supérieurs ou égaux, d'où (b). Dans les deux cas, on a aussi $i + j = \ell + 2 = (\ell + 1) + 1$, puisqu'on a incrémenté exactement un des deux indices. □

À la sortie de la boucle, on a effectué les $n + m$ itérations, donc $k = n + m - 1$ et $i + j = n + m$; comme $i \leqslant n$ et $j \leqslant m$, cela force $i = n$ et $j = m$. La propriété (a) affirme alors que $C[0], \dots, C[n + m - 1]$ contient tous les éléments de A et de B rangés en ordre croissant : la fusion est correcte. (Si A est vide, la condition $i \geqslant n$ est vraie dès le premier tour, de sorte que la boucle recopie $B[0], \dots, B[m - 1]$ dans l'ordre dans C ; comme B est triée, C l'est aussi. Le cas où B est vide se traite de la même manière.)

6.1.2 Complexité

La boucle effectue exactement $n + m$ itérations, et chaque itération ne fait qu'un nombre constant d'opérations (un ou deux tests, une affectation, une incrémentation). La fusion de deux listes triées de longueurs n et m demande donc

$$O(n + m)$$

opérations. C'est ce coût linéaire — bien meilleur que celui d'un tri appliqué à la concaténation — qui fera tout l'intérêt du tri fusion à la section suivante.

6.1.3 Exercices

Exercice 6.1.3. *Sur le modèle de l'exemple 6.1.1, dérouler la fusion de* $A = [2, 5, 5, 11]$ *et* $B = [1, 5, 8]$ *en indiquant, à chaque tour de boucle, les valeurs de* i*,* j*,* k *et de* $C[k]$*. Que devient l'élément* 5 *présent dans les deux listes : dans quel ordre les trois exemplaires apparaissent-ils dans* C *?*

Exercice 6.1.4. *Modifier la fonction* `fusion` *pour qu'elle fusionne deux listes triées en ordre* décroissant *en une liste triée en ordre décroissant, sans inverser les listes au préalable ni renverser le résultat.*

Exercice 6.1.5. *On veut fusionner* A *et* B *en* supprimant les doublons*, c'est-à-dire en ne gardant qu'un exemplaire de chaque valeur (on suppose* A *et* B *chacune sans doublon). Écrire la fonction correspondante, toujours en temps* $O(n + m)$*. Quel est le lien avec l'union de deux ensembles ?*

Exercice 6.1.6. *Soit* k *listes triées, de longueur totale* N*. Pour les fusionner en une seule, on peut les fusionner deux à deux en cascade (*A_1 *avec* A_2*, le résultat avec* A_3*, etc.). Montrer que cette stratégie coûte* $O(kN)$ *dans le pire des cas.* (On verra au chapitre 11 comment descendre à $O(N \log k)$.)

6.2 Tri fusion

Nous avons construit à la section précédente une fonction `fusion` qui, à partir de deux listes *déjà triées* de longueurs n et m, produit une liste triée contenant tous leurs éléments en temps $O(n + m)$. Cette opération est peu coûteuse ; nous allons en tirer une méthode de tri bien plus économe que les tris naïfs du chapitre précédent, qui demandaient $O(n^2)$ opérations.

L'idée tient en une phrase : pour trier une liste, on la coupe en deux moitiés, on trie chacune *récursivement*, puis on fusionne les deux moitiés triées. C'est une illustration du principe « diviser pour régner » : on ramène le problème à deux sous-problèmes de même nature mais de taille moitié, et toute la difficulté de la recombinaison a déjà été réglée par la fusion. Une liste de longueur 0 ou 1 est déjà triée : c'est le cas de base, qui arrête la récursion.

L'algorithme s'écrit alors de façon très compacte ; la fonction `fusion` qui y apparaît est celle de la section précédente.

```python
from fusion import fusion


def tri_fusion(L):
    """Renvoie une liste contenant les éléments de ``L`` en ordre croissant.

    Diviser pour régner : on coupe ``L`` en deux moitiés, on les trie
    récursivement, puis on les fusionne.
```

```
    >>> tri_fusion([5, 3, 8, 1, 9, 2, 7])
    [1, 2, 3, 5, 7, 8, 9]
    >>> tri_fusion([])
    []
    >>> tri_fusion([42])
    [42]
    >>> tri_fusion([2, 1])
    [1, 2]
    >>> import random
    >>> all(tri_fusion(L) == sorted(L)
    ...     for L in ([random.randint(0, 50) for _ in range(n)]
    ...               for n in range(60)))
    True
    """
    n = len(L)
    if n <= 1:
        return L
    m = n // 2
    L1 = L[:m]
    L2 = L[m:]
    L1 = tri_fusion(L1)
    L2 = tri_fusion(L2)
    return fusion(L1, L2)
```

Conformément à notre convention d'indexation, les listes sont indexées à partir de 0 ; la tranche `L[:m]` contient les m premiers éléments (indices 0 à $m-1$) et `L[m:]` les suivants (indices m à $n-1$).

Regardons de plus près la taille des deux moitiés selon la parité de n, car ce point servira pour l'analyse de complexité. On pose $m = \lfloor n/2 \rfloor$. Si $n = 2k$ est pair, alors $m = k$ et les deux moitiés `L[:m]` et `L[m:]` ont chacune $k = \lfloor n/2 \rfloor$ éléments. Si $n = 2k+1$ est impair, alors $m = k$: la première moitié a $k = \lfloor n/2 \rfloor$ éléments et la seconde en a $k+1 = \lceil n/2 \rceil$. Dans les deux cas, les moitiés ont pour tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$.

Exemple 6.2.1. *Déroulons l'algorithme sur la liste* $[5, 3, 8, 1, 9, 2, 7]$. *La phase de découpe (la descente dans l'arbre des appels) coupe la liste en deux jusqu'à atteindre des listes à un seul élément ; la phase de fusion (la remontée) recombine les moitiés triées. La figure 6.1 montre, à chaque nœud, la sous-liste reçue puis, après la flèche, la sous-liste triée renvoyée.*

6.2.1 Correction

Montrons que l'algorithme *termine* et *trie correctement* toute liste. La fonction s'appelant elle-même sur des listes plus courtes, c'est une récurrence forte sur la longueur n de la liste qui convient.

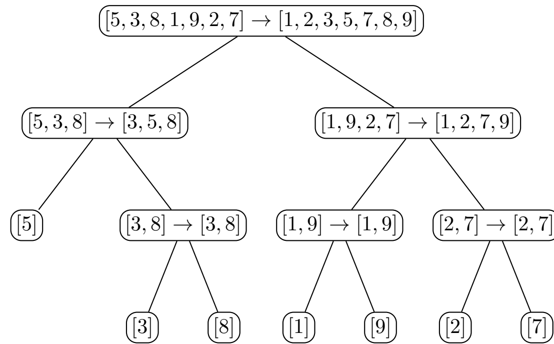


FIGURE 6.1 – Arbre des appels du tri fusion sur $[5, 3, 8, 1, 9, 2, 7]$. À chaque nœud : la sous-liste reçue → la sous-liste triée renvoyée.

Démonstration. Raisonnons par récurrence forte sur $n = |L|$.

Cas de base. Si $n \leqslant 1$, la liste est déjà triée et l'algorithme la renvoie sans faire d'appel récursif : il termine et le résultat est correct.

Hérédité. Soit $n \geqslant 2$, et supposons l'énoncé établi pour toute liste de longueur strictement inférieure à n. Sur une liste L de longueur n, l'algorithme forme les deux moitiés `L[:m]` et `L[m:]`, de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$. Comme $n \geqslant 2$, ces deux tailles sont au moins 1 et au plus $n - 1$: elles sont strictement inférieures à n. L'hypothèse de récurrence s'applique donc aux deux appels récursifs, qui terminent et renvoient les deux moitiés triées. Enfin `fusion` reçoit deux listes triées et, d'après la section précédente, termine en renvoyant une liste triée contenant exactement leurs éléments, c'est-à-dire tous les éléments de L. L'algorithme termine donc, et son résultat est la liste L triée. □

6.2.2 Complexité

Théorème 6.2.2 (Le tri fusion s'exécute en temps $O(n \log n)$)**.** *Pour trier une liste de longueur n, le tri fusion effectue $O(n \log n)$ opérations dans le pire des cas.*

Concrètement, doubler la taille de la liste ne fait pas quadrupler le temps de calcul (comme pour un tri en $O(n^2)$) mais le multiplie à peine plus que par deux : c'est un progrès décisif sur les tris naïfs.

Démonstration. Notons $t(n)$ le nombre maximal d'opérations effectuées par le tri fusion sur une liste de longueur n. Nous allons montrer qu'il existe une constante $C > 0$ telle que

$$t(n) \leqslant C\, n \log_2 n \qquad \text{pour tout } n \geqslant 2,$$

ce qui établira $t(n) = O(n \log n)$.

Décomposons le coût d'un appel sur une liste de longueur $n \geqslant 2$ en trois contributions : la découpe de la liste en deux moitiés, les deux appels récursifs, et la fusion finale. La découpe et la fusion sont toutes deux linéaires (section précédente) : il existe donc une constante $a > 0$ telle que, le coût des deux appels récursifs mis à part, un appel sur une

liste de longueur n coûte au plus $a\,n$. On obtient ainsi la majoration

$$t(n) \leqslant t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + a\,n.$$

Choisissons d'abord C assez grand pour que l'inégalité $t(n) \leqslant C\,n \log_2 n$ soit vraie pour $n \in \{2, 3\}$; nous préciserons à la fin la contrainte exacte sur C. Soit $n \geqslant 4$, et supposons l'inégalité établie pour tout entier de $\{2, 3, \dots, n-1\}$. Comme $n \geqslant 4$, on a

$$2 \leqslant \lfloor n/2 \rfloor \leqslant \lceil n/2 \rceil \leqslant n - 1,$$

si bien que l'hypothèse de récurrence s'applique aux deux moitiés :

$$t(\lfloor n/2 \rfloor) \leqslant C \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor \qquad \text{et} \qquad t(\lceil n/2 \rceil) \leqslant C \lceil n/2 \rceil \log_2 \lceil n/2 \rceil.$$

Majorons les deux logarithmes par $\log_2 \lceil n/2 \rceil$ et la somme des deux planchers/plafonds par n ; comme $\lceil n/2 \rceil \leqslant (n+1)/2$, il vient

$$t(n) \leqslant a\,n + C\,n \log_2 \lceil n/2 \rceil \leqslant a\,n + C\,n \log_2 \frac{n+1}{2}.$$

Il reste à comparer ce majorant à $C\,n \log_2 n$. On a

$$a\,n + C\,n \log_2 \frac{n+1}{2} \leqslant C\,n \log_2 n \quad \Longleftrightarrow \quad a \leqslant C \log_2 \frac{2n}{n+1}.$$

La fonction $x \mapsto \log_2 \frac{2x}{x+1}$ est croissante sur $\mathbb{R}^+$, donc pour $n \geqslant 4$ on a $\log_2 \frac{2n}{n+1} \geqslant \log_2 \frac{8}{5}$. Il suffit donc de prendre

$$C \geqslant \frac{a}{\log_2(8/5)},$$

et, quitte à agrandir C pour couvrir aussi les cas $n \in \{2, 3\}$, l'inégalité $t(n) \leqslant C\,n \log_2 n$ est établie pour tout $n \geqslant 2$. C'est bien ce que nous voulions démontrer. □

Nous montrerons plus loin qu'*aucun* tri par comparaison ne peut faire asymptotiquement mieux : le tri fusion est optimal pour cette classe d'algorithmes.

6.2.3 Exercices

Exercice 6.2.3. *Modifier le tri fusion pour qu'il renvoie, en plus de la liste triée, le nombre total de comparaisons d'éléments (les tests de la forme* `A[i] < B[j]`*) qu'il a effectuées. Vérifier sur l'exemple 6.2.1 que ce nombre vaut* 14.

Exercice 6.2.4. *Écrire une version* itérative *(sans récursivité) du tri fusion, dite* ascendante *: fusionner d'abord les blocs voisins de taille* 1*, puis les blocs de taille* 2*, puis* 4*, etc. Quel est l'invariant satisfait après l'étape qui fusionne les blocs de taille w ?*

Exercice 6.2.5. *Montrer par récurrence que le tri fusion d'une liste de longueur n effectue au plus $n\lceil \log_2 n \rceil$ comparaisons d'éléments. En déduire que, pour $n = 2^k$, ce nombre est au plus $n \log_2 n$. Corroborer la borne expérimentalement à l'aide de la fonction de l'exercice 6.2.3.*

Exercice 6.2.6 (★). *Une* inversion *d'une liste L est un couple d'indices (i, j) avec $i < j$ et $L[i] > L[j]$. Une liste triée n'a aucune inversion ; une liste rangée en ordre décroissant en a $\binom{n}{2}$. Adapter le tri fusion pour compter le nombre d'inversions de L en temps $O(n \log n)$.* Indication : *lors de la fusion, lorsqu'on prélève un élément de la moitié droite alors qu'il reste r éléments dans la moitié gauche, ces r éléments forment chacun une inversion avec lui.*

6.3 Tri rapide

Nous avons vu (section 6.2) que le tri fusion concentre tout son travail dans la recombinaison — la fusion. Dans cette section, nous étudions le *tri rapide*, qui au contraire investit l'effort dans la découpe : en plaçant d'emblée chaque élément du bon côté d'une valeur de référence, il fait de la recombinaison une simple concaténation. C'est encore un algorithme « diviser pour régner », mais qui travaille « à la descente » plutôt qu'« à la remontée ».

6.3.1 Partitionner autour d'un pivot

Le cœur de la méthode est l'opération de *partition*. On se donne une liste L et une valeur x, appelée *pivot* ; on veut répartir les éléments de L en deux listes : une liste A rassemblant ceux qui sont *inférieurs ou égaux* à x, et une liste B rassemblant ceux qui sont *strictement supérieurs* à x. Il suffit pour cela de parcourir L une fois en aiguillant chaque élément vers A ou vers B.

```
def partition(L, x):
    """Répartit les éléments de ``L`` selon le pivot ``x``.

    Renvoie un couple de listes ``(A, B)`` où ``A`` contient les éléments
    de ``L`` inférieurs ou égaux à ``x`` et ``B`` ceux strictement
    supérieurs à ``x``, chacun dans l'ordre où ils apparaissent dans ``L``.

    >>> partition([3, 1, 4, 1, 5, 9, 2, 6], 4)
    ([3, 1, 4, 1, 2], [5, 9, 6])
    >>> partition([7, 7, 7], 7)
    ([7, 7, 7], [])
    >>> partition([], 0)
    ([], [])
    """
    A = []
    B = []
    for e in L:
        if e <= x:
            A.append(e)
        else:
```

```
            B.append(e)
    return A, B
```

Exemple 6.3.1. *Partitionnons* $L = [3, 1, 4, 1, 5, 9, 2, 6]$ *autour du pivot* $x = 4$. *En parcourant* L *de gauche à droite, on envoie dans* A *les éléments* $\leqslant 4$ *et dans* B *ceux* > 4 *:*

$$A = [3, 1, 4, 1, 2], \qquad B = [5, 9, 6].$$

On remarquera que le pivot 4*, présent dans* L*, se retrouve dans* A *(puisque* $4 \leqslant 4$*), et qu'un élément égal au pivot va toujours dans* A*.*

Le parcours effectue une comparaison et une insertion par élément : la partition d'une liste de longueur n coûte donc $O(n)$ opérations.

6.3.2 L'algorithme de tri

L'idée du tri rapide est maintenant la suivante. Pour trier une liste L de longueur $n \geqslant 2$, on choisit l'un de ses éléments comme pivot ; nous prenons l'élément du milieu, $x = L[m]$ avec $m = \lfloor n/2 \rfloor$. On partitionne *le reste* de la liste autour de x, ce qui donne deux listes A (les éléments $\leqslant x$) et B (les éléments $> x$). On trie chacune *récursivement*, puis on recolle le tout en intercalant le pivot :

$$\texttt{tri_rapide}(A) +\!\!+ [x] +\!\!+ \texttt{tri_rapide}(B).$$

Comme tous les éléments de A sont $\leqslant x$ et tous ceux de B sont $> x$, le pivot est exactement à sa place entre les deux. Une liste de longueur 0 ou 1 est déjà triée : c'est le cas de base.

```
from partition import partition


def tri_rapide(L):
    """Renvoie une liste contenant les éléments de ``L`` en ordre croissant.

    Diviser pour régner : on choisit un pivot, on répartit les autres
    éléments de part et d'autre de lui (fonction ``partition``), on trie
    récursivement chaque part, puis on recolle le tout en plaçant le pivot
    entre les deux.

    >>> tri_rapide([5, 3, 8, 1, 9, 2, 7])
    [1, 2, 3, 5, 7, 8, 9]
    >>> tri_rapide([])
    []
    >>> tri_rapide([42])
    [42]
    >>> tri_rapide([3, 1, 2, 1, 3])
```

```
    [1, 1, 2, 3, 3]
    >>> import random
    >>> all(tri_rapide(L) == sorted(L)
    ...     for L in ([random.randint(0, 50) for _ in range(n)]
    ...               for n in range(60)))
    True
    """
    n = len(L)
    if n <= 1:
        return L
    m = n // 2
    x = L[m]
    reste = L[:m] + L[m + 1:]
    A, B = partition(reste, x)
    return tri_rapide(A) + [x] + tri_rapide(B)
```

On notera que l'on partitionne `reste = L[:m] + L[m+1:]`, c'est-à-dire L *privée de son pivot*, et non L tout entière. C'est essentiel : en retirant le pivot, on garantit que les deux sous-listes A et B ont chacune une longueur *strictement* inférieure à n, ce qui assure que la récursion progresse vers le cas de base. [2]

Exemple 6.3.2. *Déroulons l'algorithme sur* $L = [6, 2, 8, 5, 1, 7, 4]$. *Le pivot initial est* $L[3] = 5$ *; la partition du reste donne* $A = [2, 1, 4]$ *et* $B = [6, 8, 7]$*, que l'on trie récursivement. La figure 6.2 montre l'arbre des appels : à chaque nœud, le pivot choisi est mis en évidence, et ses deux enfants sont les sous-listes A (à gauche) et B (à droite) renvoyées par la partition. Les sous-listes vides — produites lorsque le pivot est le plus petit ou le plus grand de sa sous-liste — arrêtent immédiatement la récursion.*

6.3.3 Correction

Comme pour le tri fusion, la fonction s'appelle elle-même sur des listes plus courtes : c'est donc une récurrence forte sur la longueur n de la liste qui convient. Le point clef est que la concaténation de deux listes triées rangées *de part et d'autre* du pivot est elle-même triée.

Démonstration. Raisonnons par récurrence forte sur $n = |L|$, en montrant que `tri_rapide` termine et renvoie une liste triée contenant exactement les éléments de L.

Cas de base. Si $n \leqslant 1$, la liste est déjà triée et l'algorithme la renvoie sans appel récursif : il termine, et le résultat est correct.

Hérédité. Soit $n \geqslant 2$, et supposons l'énoncé établi pour toute liste de longueur strictement inférieure à n. Sur une liste L de longueur n, l'algorithme choisit le pivot $x = L[m]$,

2. Si l'on partitionnait L entière, le pivot lui-même tomberait dans A ; et si le pivot était le plus grand élément, on aurait $A = L$ et $B = []$, donc un appel récursif sur une liste *aussi longue* que celle de départ — la récursion ne se terminerait jamais.

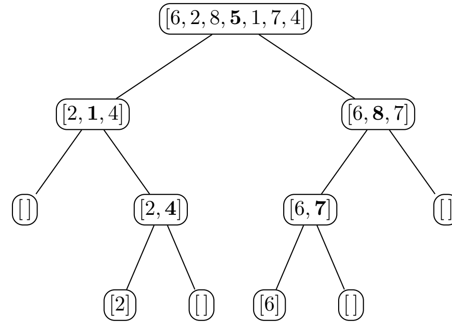


FIGURE 6.2 – Arbre des appels du tri rapide sur $[6, 2, 8, 5, 1, 7, 4]$. À chaque nœud, le pivot est en gras et placé entre les deux sous-listes A (gauche, éléments $\leqslant$ pivot) et B (droite, éléments $>$ pivot) qu'engendre la partition. La liste triée se lit en parcourant l'arbre de gauche à droite : $[1, 2, 4, 5, 6, 7, 8]$.

puis partitionne la liste `reste`, de longueur $n - 1$, en

$$A = \{\, e \in \texttt{reste} : e \leqslant x \,\}, \qquad B = \{\, e \in \texttt{reste} : e > x \,\}$$

(au sens des listes : A et B conservent les répétitions). Comme A et B se partagent les $n - 1$ éléments de `reste`, on a $|A| \leqslant n - 1 < n$ et $|B| \leqslant n - 1 < n$. L'hypothèse de récurrence s'applique donc aux deux appels récursifs : ils terminent et renvoient respectivement A et B triées. L'algorithme termine donc.

Reste à voir que le résultat

$$R = \texttt{tri_rapide}(A) \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \texttt{tri_rapide}(B)$$

est trié. D'abord, R contient bien tous les éléments de L : les éléments de `reste` se répartissent entre A et B, auxquels on rajoute le pivot x, et $L = \texttt{reste} \cup \{x\}$ (au sens des listes). Ensuite, montrons que R est croissante. Par hypothèse de récurrence, $\texttt{tri_rapide}(A)$ et $\texttt{tri_rapide}(B)$ sont triées ; il suffit donc de vérifier les « raccords » avec le pivot. Tout élément de A est $\leqslant x$ par définition de la partition : le bloc de gauche est donc suivi d'un élément, x, qui lui est supérieur ou égal. De même, tout élément de B est $> x$: le pivot est donc suivi d'éléments qui lui sont strictement supérieurs. Les trois blocs $\texttt{tri_rapide}(A)$, $[x]$ et $\texttt{tri_rapide}(B)$ se succèdent donc en ordre croissant, et R est triée. C'est ce que nous voulions démontrer. □

6.3.4 Complexité

Le coût du tri rapide dépend, de façon cruciale, de la qualité des partitions — autrement dit de l'équilibre entre les tailles de A et de B.

Notons $t(n)$ le nombre maximal d'opérations sur une liste de longueur n. La partition coûte $O(n)$ (un parcours), et la concaténation finale aussi ; le reste du travail est dans les deux appels récursifs, sur des listes de tailles $|A|$ et $|B|$ avec $|A| + |B| = n - 1$.

Cas favorable. Si à chaque étape le pivot tombe au milieu, A et B ont des tailles voisines de $n/2$, et l'on retrouve exactement la récurrence du tri fusion :

$$t(n) \leqslant t(\lfloor (n-1)/2 \rfloor) + t(\lceil (n-1)/2 \rceil) + a\,n,$$

qui se résout, comme à la section précédente, en $t(n) = O(n \log n)$.

Cas défavorable. À l'opposé, si à chaque étape le pivot est le plus petit (ou le plus grand) élément de sa sous-liste, l'une des deux parts est vide et l'autre contient les $n-1$ éléments restants. La récurrence devient alors

$$t(n) = t(n-1) + a\,n,$$

d'où, en sommant, $t(n) = a\,(n + (n-1) + \cdots + 2) = O(n^2)$. Dans le pire des cas, le tri rapide n'est donc *pas* meilleur que les tris naïfs !

Avertissement 6.3.3. *Le pire cas n'est pas qu'une curiosité théorique : il dépend du choix du pivot. Si l'on choisissait toujours comme pivot le* premier *élément $L[0]$, alors une liste* déjà triée *fournirait exactement ce pire cas — le pivot y est à chaque fois le minimum, A est vide et B contient tout le reste. Notre choix de l'élément du milieu évite ce piège-là, mais il existe pour lui aussi des entrées défavorables. C'est pourquoi, en pratique, on choisit souvent le pivot* au hasard *: aucune entrée n'est alors systématiquement défavorable, et l'on peut montrer que le coût* moyen *reste $O(n \log n)$. Le tri rapide se comporte ainsi en $O(n \log n)$ sur la quasi-totalité des entrées, ce qui, joint à sa simplicité, explique son nom et sa popularité.*

6.3.5 Exercices

Exercice 6.3.4. *Sur le modèle de l'exemple 6.3.2, dérouler le tri rapide sur la liste* $[4, 8, 2, 6, 1, 5, 3]$ *: indiquer le pivot choisi à chaque appel et dessiner l'arbre des appels. Combien d'appels récursifs aboutissent à une sous-liste vide ?*

Exercice 6.3.5. *Modifier le tri rapide pour qu'il renvoie, en plus de la liste triée, le nombre total de comparaisons* élément contre pivot *(les tests* `e <= x`*) qu'il effectue. À l'aide de cette fonction, comparer le nombre de comparaisons sur une liste aléatoire de longueur n et sur la liste* $[0, 1, 2, \ldots, n-1]$ *déjà triée, pour* $n = 1000$. *Qu'observe-t-on, et est-ce cohérent avec l'avertissement 6.3.3 ?*

Exercice 6.3.6. *Que fait le tri rapide sur une liste dont tous les éléments sont* égaux *? Préciser, dans ce cas, les tailles de A et de B à chaque appel, et en déduire le nombre d'opérations effectuées en fonction de n.*

Exercice 6.3.7 (★). *La version ci-dessus construit, à chaque appel, deux nouvelles listes A et B. On peut s'en passer et trier* en place, *par échanges à l'intérieur de la liste. Écrire une fonction* `partition_en_place(L, g, d)` *qui réorganise la tranche $L[g..d]$ autour du pivot $L[d]$ et renvoie la position finale p du pivot, de sorte que $L[g..p-1] \leqslant L[p] < L[p+1..d]$; en déduire un tri rapide en place.* Indication : *balayer la tranche avec un indice j en maintenant un indice i tel que $L[g..i-1]$ regroupe les éléments déjà vus qui sont $\leqslant$ au pivot.*

6.4 Optimalité : une borne inférieure pour les tris par comparaison

Le tri fusion (section 6.2) trie en $O(n \log n)$ opérations, et nous avons annoncé qu'aucun tri ne peut faire asymptotiquement mieux. Le moment est venu de le démontrer — du moins pour une grande famille de tris, ceux qui n'accèdent aux données qu'en *comparant* leurs éléments. Toute la différence avec les sections précédentes est que nous allons borner non pas le coût d'*un* algorithme, mais celui de *tous* les algorithmes d'une certaine classe à la fois : c'est une borne *inférieure*, et c'est d'une autre nature.

Définition 6.4.1 (Tri par comparaison)**.** *Un* tri par comparaison *est un algorithme de tri dont le déroulement ne dépend des éléments de la liste L qu'à travers des tests portant sur l'ordre relatif de deux d'entre eux, c'est-à-dire des tests de la forme « $L[i] < L[j]$ ».*

Concrètement, un tel algorithme a le droit de déplacer les éléments et de comparer deux d'entre eux, mais jamais d'inspecter leur valeur ni de faire de l'arithmétique dessus. Le tri par sélection, le tri à bulles, le tri par insertion et le tri fusion sont tous des tris par comparaison ; nous verrons à la section suivante un tri qui, lui, n'en est pas un.

6.4.1 Trier, c'est choisir une permutation

Fixons une liste L de longueur n dont les n éléments sont *deux à deux distincts*. Trier L, c'est en réarranger les éléments en ordre croissant ; autrement dit, c'est déterminer la *permutation* σ de $\{0, 1, \ldots, n-1\}$ telle que

$$L[\sigma(0)] < L[\sigma(1)] < \cdots < L[\sigma(n-1)].$$

Comme les éléments sont distincts, il existe une et une seule telle permutation : $\sigma(k)$ est la position, dans la liste de départ, du k-ième plus petit élément. Trouver l'ordre trié et trouver cette permutation, c'est exactement la même chose.

Or il y a $n!$ permutations de $\{0, 1, \ldots, n-1\}$, et chacune est la bonne réponse pour au moins une liste d'entrée. C'est ce simple décompte qui va imposer sa borne au problème : un algorithme correct doit être capable de distinguer ces $n!$ réponses possibles, et chaque comparaison ne lui apporte qu'un seul bit d'information (le test est vrai, ou il est faux).

6.4.2 L'arbre de décision d'un tri

Rendons cette idée précise. Fixons un tri par comparaison et une longueur n, et suivons son exécution sur les listes de longueur n. Le premier test qu'il effectue est toujours le même — disons « $L[i] < L[j]$? » — puisque, tant qu'aucune comparaison n'a été faite, l'algorithme ne peut rien distinguer entre deux listes. Selon que ce test est vrai ou faux, il enchaîne sur l'un de deux tests suivants, et ainsi de suite. On représente ce déroulement par un *arbre de décision* : un arbre binaire dont chaque nœud interne est étiqueté par un test « $L[i] < L[j]$? », le sous-arbre gauche décrivant la suite de l'exécution quand le test est vrai, le sous-arbre droit quand il est faux. Une *feuille* est atteinte lorsque l'algorithme s'arrête : il a alors rangé les éléments, donc déterminé la permutation σ qui trie la liste.

Exemple 6.4.2. *La figure 6.3 montre l'arbre de décision d'un tri de trois éléments* $L[0], L[1], L[2]$*. Trois comparaisons suffisent dans le pire des cas, et les six feuilles correspondent aux* $3! = 6$ *ordres possibles. Par exemple, la branche la plus à gauche teste « $L[0] < L[1]$? » (vrai), puis « $L[1] < L[2]$? » (vrai), et conclut* $L[0] < L[1] < L[2]$*.*

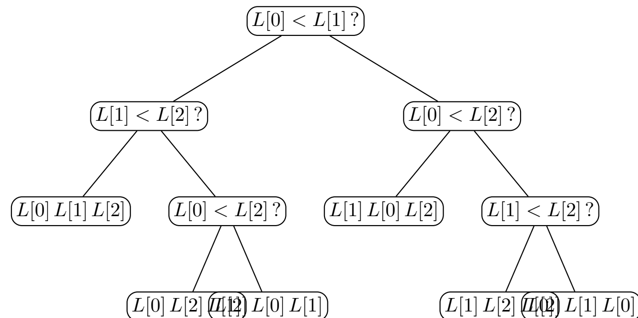


FIGURE 6.3 – Arbre de décision d'un tri par comparaison de trois éléments. À chaque nœud interne, la branche de gauche correspond au test vrai, celle de droite au test faux. Chaque feuille donne l'ordre croissant des trois positions (p.ex. $L[2]\,L[0]\,L[1]$ signifie $L[2] < L[0] < L[1]$).

Le point essentiel est que ces comparaisons étiquetant les nœuds parlent toujours des *valeurs de la liste initiale*, même après que l'algorithme a déplacé des éléments : l'arbre ne dépend que de l'algorithme et de n, pas de la liste particulière qu'on lui donne. Chaque liste d'entrée suit un unique chemin de la racine à une feuille, dicté par les réponses à ses comparaisons.

6.4.3 La borne inférieure

Théorème 6.4.3 (Tout tri par comparaison est en $\Omega(n \log n)$)**.** *Il existe une constante* $C > 0$ *telle que, pour tout* n*, tout tri par comparaison effectue au moins* $C\,n \log_2 n$ *comparaisons sur au moins une liste de longueur* n*. Autrement dit, son coût dans le pire des cas est en* $\Omega(n \log n)$*.*

Rappelons (chapitre 3, définition 3.1.3) que la notation Ω est le pendant inférieur de O : on écrit $t(n) = \Omega(g(n))$ lorsqu'il existe une constante $c > 0$ telle que $t(n) \geqslant c\,g(n)$ pour n assez grand — une *minoration* asymptotique, là où O est une majoration — et qu'une quantité à la fois en $O(g)$ et en $\Omega(g)$ est dite en $\Theta(g)$. Le théorème dit donc que le tri fusion, qui est en $O(n \log n)$, ne peut pas être battu en ordre de grandeur : il est en $\Theta(n \log n)$, et *optimal* parmi les tris par comparaison.

Démonstration. Fixons n et un tri par comparaison, et considérons son arbre de décision sur les listes de longueur n. Notons $t(n)$ son coût dans le pire des cas, c'est-à-dire le nombre maximal de comparaisons effectuées sur une liste de longueur n : c'est la hauteur de l'arbre, puisqu'un chemin de la racine à une feuille compte une comparaison par nœud interne traversé.

L'arbre a au moins $n!$ feuilles. Considérons les listes à éléments deux à deux distincts. Chacune est triée par une unique permutation σ, et deux listes appelant des permutations différentes doivent aboutir à des feuilles différentes. En effet, une feuille fixe le réarrangement final que produit l'algorithme ; si deux listes nécessitant des permutations distinctes atteignaient la même feuille, le même réarrangement serait appliqué aux deux, et ne pourrait pas les trier toutes les deux. Comme les $n!$ permutations sont toutes nécessaires, l'arbre possède au moins $n!$ feuilles.

Un arbre binaire de hauteur h a au plus 2^h feuilles. Cela se montre par récurrence sur h : un arbre de hauteur 0 est réduit à une feuille ($1 = 2^0$) ; un arbre de hauteur $h \geqslant 1$ se compose d'une racine et de deux sous-arbres de hauteur au plus $h-1$, donc d'au plus $2^{h-1} + 2^{h-1} = 2^h$ feuilles.

En combinant les deux, $2^{t(n)} \geqslant n!$, d'où

$$t(n) \geqslant \log_2(n!).$$

Minoration de $\log_2(n!)$. Il reste à voir que $\log_2(n!)$ est de l'ordre de $n \log_2 n$. Dans le produit $n! = 1 \cdot 2 \cdots n$, gardons seulement les facteurs d'indice $\geqslant n/2$: ils sont au moins $n/2$ et valent chacun au moins $n/2$, donc

$$n! \;\geqslant\; \left(\frac{n}{2}\right)^{n/2}.$$

En passant au logarithme en base 2,

$$\log_2(n!) \;\geqslant\; \frac{n}{2}\log_2\frac{n}{2} \;=\; \frac{n}{2}(\log_2 n - 1).$$

Pour $n \geqslant 4$ on a $\log_2 n \geqslant 2$, donc $\log_2 n - 1 \geqslant \frac{1}{2}\log_2 n$, et finalement

$$t(n) \;\geqslant\; \log_2(n!) \;\geqslant\; \frac{n}{4}\log_2 n.$$

On a donc bien $t(n) = \Omega(n \log n)$, avec $C = 1/4$ qui convient pour $n \geqslant 4$. C'est ce que nous voulions établir. ☐

Remarque 6.4.4. *Insistons sur la portée de l'énoncé. Les bornes des sections précédentes majoraient le coût d'un algorithme* donné. *Ici, c'est une propriété du* problème *du tri lui-même :* quel que soit *l'ingéniosité qu'on y mette, tant qu'on se restreint aux comparaisons, on ne descend pas sous $n \log_2 n$. Une borne inférieure ferme une porte ; encore faut-il, pour la contourner, changer les règles du jeu — c'est ce que fera le tri par comptage à la section suivante, qui* n'est pas *un tri par comparaison et trie en temps $O(n)$.*

6.4.4 Exercices

Exercice 6.4.5. *Sur l'arbre de la figure 6.3, vérifier que les six feuilles donnent bien les six ordres possibles de trois éléments distincts. Pourquoi un tri par comparaison de trois éléments ne peut-il pas garantir moins de trois comparaisons dans le pire des cas ?* (On pourra invoquer la hauteur minimale d'un arbre binaire à six feuilles.)

Exercice 6.4.6. *Écrire une fonction qui, pour un entier n, calcule le nombre minimal de comparaisons garanti par le théorème 6.4.3, $\lceil \log_2(n!) \rceil$, et le compare au nombre $n\lceil \log_2 n \rceil$ de comparaisons du tri fusion (exercice 6.2.5). Tabuler les deux quantités pour n de 1 à 16 et constater qu'elles sont du même ordre de grandeur.*

Exercice 6.4.7. *Dans la preuve du théorème 6.4.3, où a-t-on utilisé que les éléments de la liste sont deux à deux distincts ? La borne inférieure resterait-elle valable si l'on n'autorisait en entrée que des listes comportant des répétitions ?*

Exercice 6.4.8 (★). *Montrer de même que $n! \leqslant n^n$, et en déduire $\log_2(n!) \leqslant n \log_2 n$. Conclure que $\log_2(n!) = \Theta(n \log n)$.* (Ainsi la borne inférieure du théorème est, à une constante près, la meilleure possible : le tri fusion l'atteint.)

6.5 Tri par comptage

La section précédente a fermé une porte : aucun tri par comparaison ne descend sous $\Omega(n \log n)$. Mais cette borne ne vaut que pour les algorithmes qui n'accèdent aux données qu'en comparant leurs éléments. Que se passe-t-il si l'on s'autorise à *regarder* les valeurs elles-mêmes ? Dans cette section, nous trions des entiers en temps $O(n)$ — en violation apparente du théorème, mais sans le contredire, car l'algorithme que voici n'est pas un tri par comparaison.

6.5.1 L'idée : compter plutôt que comparer

Plaçons-nous dans une situation particulière mais fréquente : la liste L à trier est formée de n entiers, tous compris entre 0 et $k-1$ pour un entier k *connu à l'avance* (on note k cette borne). Par exemple, des notes sur 20 ($k = 21$), des âges, des codes postaux : à chaque fois, un grand nombre de valeurs prises dans un petit intervalle.

Plutôt que de comparer les éléments entre eux, on peut alors se contenter de les *compter*. Si l'on sait que la valeur 0 apparaît deux fois, la valeur 1 trois fois, la valeur 2 pas du tout, et ainsi de suite, alors on connaît entièrement la liste triée : il suffit d'écrire deux fois 0, puis trois fois 1, et de continuer dans l'ordre. Aucune comparaison n'est nécessaire ; toute l'information est dans le *décompte des occurrences*.

6.5.2 Le comptage

La première étape construit la liste des occurrences. On parcourt L et, pour chaque valeur v rencontrée, on incrémente un compteur dédié à v. Comme les valeurs sont des entiers de $\{0, \dots, k-1\}$, on les utilise directement comme indices dans une liste C de longueur k : c'est là, et nulle part ailleurs, qu'on exploite le fait que les éléments sont des entiers bornés.

```
def comptage(L, k):
    """Renvoie la liste ``C`` des occurrences des valeurs ``0..k-1`` dans ``L``.
```

```
    On suppose que tous les éléments de ``L`` sont des entiers de
    l'intervalle ``{0, ..., k-1}``. Pour chaque ``v`` de cet intervalle,
    ``C[v]`` est le nombre d'indices ``j`` tels que ``L[j] == v``.

    >>> comptage([3, 1, 0, 1, 0, 1], 6)
    [2, 3, 0, 1, 0, 0]
    >>> comptage([], 4)
    [0, 0, 0, 0]
    >>> comptage([2, 0, 0, 1, 0, 1], 3)
    [3, 2, 1]
    """
    C = [0] * k
    for v in L:
        C[v] = C[v] + 1
    return C
```

Exemple 6.5.1. *Prenons* $L = [3, 1, 0, 1, 0, 1]$, *avec* $k = 6$. *En parcourant* L, *on dénombre deux* 0, *trois* 1, *aucun* 2, *un seul* 3, *et aucun* 4 *ni* 5 *:*

$$C = [\,2,\ 3,\ 0,\ 1,\ 0,\ 0\,].$$

Cette seule liste C *détermine la version triée de* L *: deux* 0, *suivis de trois* 1, *suivis d'un* 3, *soit* $[0, 0, 1, 1, 1, 3]$.

6.5.3 La reconstruction

La seconde étape relit le décompte C et réécrit les valeurs dans l'ordre croissant : pour chaque valeur v de 0 à $k - 1$, on écrit v exactement $C[v]$ fois.

```
from comptage import comptage


def tri_comptage(L, k):
    """Renvoie la liste ``L`` triée en ordre croissant.

    On suppose que tous les éléments de ``L`` sont des entiers de
    l'intervalle ``{0, ..., k-1}``. On compte les occurrences de chaque
    valeur, puis on réécrit les valeurs dans l'ordre, chacune répétée
    autant de fois qu'elle apparaît.

    >>> tri_comptage([3, 1, 0, 1, 0, 1], 6)
    [0, 0, 1, 1, 1, 3]
    >>> tri_comptage([], 5)
    []
    >>> import random
```

```
    >>> all(tri_comptage(L, 10) == sorted(L)
    ...     for L in ([random.randint(0, 9) for _ in range(n)]
    ...               for n in range(60)))
    True
    """
    C = comptage(L, k)
    T = []
    for v in range(k):
        for _ in range(C[v]):
            T.append(v)
    return T
```

La correction de l'algorithme ci-dessus est immédiate : la liste renvoyée contient, pour chaque valeur v, exactement $C[v]$ copies de v, c'est-à-dire autant que L en contient (par définition de C) ; et elle les énumère par valeurs croissantes de v. C'est donc bien une liste triée contenant exactement les éléments de L.

6.5.4 Complexité

Théorème 6.5.2 (Le tri par comptage s'exécute en temps $O(n+k)$)**.** *Trier par comptage une liste de n entiers de $\{0,\dots,k-1\}$ demande $O(n+k)$ opérations. En particulier, si k est de l'ordre de n (p.ex. $k \leqslant c\,n$ pour une constante c), le tri s'exécute en temps $O(n)$.*

Démonstration. La création de la liste C coûte $O(k)$ (on l'initialise à k zéros), et le comptage parcourt une fois L, soit $O(n)$: la fonction `comptage` est en $O(n+k)$. Pour la reconstruction, l'astuce est de bien compter les deux boucles imbriquées : la boucle externe parcourt les k valeurs, et pour chaque valeur v la boucle interne effectue $C[v]$ écritures. Le nombre total d'écritures est donc

$$\sum_{v=0}^{k-1} C[v] = n,$$

puisque les occurrences de toutes les valeurs se somment au nombre total d'éléments. En ajoutant les k tours de la boucle externe (qui s'exécutent même lorsque $C[v] = 0$), la reconstruction coûte $O(n+k)$. Au total, le tri par comptage est en $O(n+k)$. □

Lorsque k reste proportionnel à n, on obtient un tri en temps *linéaire* — strictement meilleur que le $O(n \log n)$ du tri fusion. En revanche, si k est gigantesque devant n (trier dix entiers pris parmi le milliard), le terme $O(k)$ domine et la méthode devient inefficace : tout l'enjeu est que les valeurs soient *bornées* et l'intervalle *petit*.

6.5.5 Pourquoi le théorème de la borne inférieure n'est pas contredit

Le tri par comptage trie en $O(n)$, et pourtant le théorème 6.4.3 affirmait qu'aucun tri ne descend sous $\Omega(n \log n)$. Où est le piège ? Il n'y en a pas : ce théorème ne concerne que les tris *par comparaison* (définition 6.4.1), ceux qui n'accèdent aux données qu'à travers

des tests « $L[i] < L[j]$ ». Or le tri par comptage ne compare jamais deux éléments ! Il *lit* la valeur d'un élément pour s'en servir comme indice dans la liste C — exactement l'opération que la définition d'un tri par comparaison interdit. L'hypothèse du théorème n'étant pas satisfaite, sa conclusion ne s'applique pas.

Remarque 6.5.3. *Il n'y a là aucune magie, et surtout aucune contradiction. Le tri par comptage* achète *sa vitesse en faisant une hypothèse forte sur les données — des entiers dans un petit intervalle connu — dont un tri par comparaison se passe complètement. Chaque algorithme paie quelque part : ici, c'est en généralité.*

6.5.6 Exercices

Exercice 6.5.4. *Sur le modèle de l'exemple 6.5.1, dérouler le tri par comptage sur* $L = [2, 5, 2, 0, 5, 2, 1]$ *avec* $k = 6$ *: donner la liste* C *des occurrences, puis la liste triée reconstruite.*

Exercice 6.5.5. *Reprendre la fonction* `tri_comptage` *et identifier précisément la (ou les) instruction(s) où une valeur de* L *est utilisée. Vérifier qu'aucune n'est une comparaison entre deux éléments de* L*, et expliquer en une phrase pourquoi cela suffit à échapper au théorème 6.4.3.*

Exercice 6.5.6. *On veut trier* $n = 100$ *entiers dont on sait seulement qu'ils sont compris entre* 0 *et* $10^6 - 1$*. Quel est, en fonction de ces nombres, le coût du tri par comptage ? Le comparer à celui du tri fusion. Lequel choisiriez-vous, et pourquoi la réponse s'inverserait-elle si l'on devait trier* $n = 10^6$ *entiers de ce même intervalle ?*

Exercice 6.5.7. *Adapter le tri par comptage à une liste d'entiers tous compris entre deux bornes* a *et* b *(avec* a *éventuellement négatif), sans gaspiller de mémoire pour les valeurs inférieures à* a*.* Indication : *la valeur* v *se range à l'indice* $v - a$ *d'un compteur de longueur* $b - a + 1$*. Quel est le coût en fonction de* n *et de* $b - a$ *?*

6.6 Produit rapide de polynômes

Nous avons appliqué « diviser pour régner » au tri (section 6.2). Dans cette section, nous l'appliquons à un problème de toute autre nature, purement numérique : la multiplication de deux polynômes, pour laquelle l'algorithme de Karatsuba (1962) bat la méthode naïve.

6.6.1 Le problème et la méthode naïve

Un polynôme $A = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}$ est donné par la liste de ses coefficients $[a_0, a_1, \ldots, a_{n-1}]$, l'indice dans la liste étant le degré du monôme. Étant donné deux polynômes A et B, on veut calculer leur produit

$$AB = \sum_k \Big(\sum_{i+j=k} a_i\, b_j \Big) X^k,$$

lui aussi sous forme de liste de coefficients.

La méthode directe applique cette formule : pour chaque paire (i, j), on ajoute $a_i b_j$ au coefficient de degré $i + j$.

```
def produit_naif(A, B):
    """Renvoie le produit des polynômes ``A`` et ``B`` (coefficient par degré).

    Un polynôme est représenté par la liste de ses coefficients, l'indice
    donnant le degré : ``[a0, a1, ..., a_{n-1}]`` représente
    ``a0 + a1 X + ... + a_{n-1} X**(n-1)``. On applique la formule du
    produit : le coefficient de degré ``k`` est la somme des ``A[i] * B[j]``
    pour ``i + j == k``.

    >>> produit_naif([1, 2, 3], [4, 5])
    [4, 13, 22, 15]
    >>> produit_naif([1, 1], [1, 1])
    [1, 2, 1]
    >>> produit_naif([], [1, 2])
    []
    """
    n = len(A)
    m = len(B)
    if n == 0 or m == 0:
        return []
    C = [0] * (n + m - 1)
    for i in range(n):
        for j in range(m):
            C[i + j] = C[i + j] + A[i] * B[j]
    return C
```

Avec deux polynômes de n coefficients chacun, la double boucle effectue n^2 multiplications de coefficients : la méthode naïve est en $O(n^2)$. Peut-on faire mieux ?

6.6.2 Couper les polynômes en deux

Supposons, pour simplifier, que A et B ont chacun $n = 2^k$ coefficients, et posons $m = n/2$. On coupe chaque polynôme en une moitié « basse » et une moitié « haute » :

$$A = A_0 + X^m A_1, \qquad B = B_0 + X^m B_1,$$

où A_0, A_1, B_0, B_1 ont chacun m coefficients. Le produit se développe alors en

$$AB = A_0 B_0 + X^m (A_0 B_1 + A_1 B_0) + X^{2m} A_1 B_1. \tag{6.6.1}$$

On a ainsi ramené le produit de deux polynômes de taille n au calcul des trois polynômes $A_0 B_0$, $A_0 B_1 + A_1 B_0$ et $A_1 B_1$, qu'il suffit de replacer aux bons degrés.

Tels quels, ces trois polynômes réclament *quatre* produits de demi-taille : A_0B_0, A_0B_1, A_1B_0 et A_1B_1. Si l'on s'arrêtait là, on obtiendrait la récurrence $t(n) = 4\,t(n/2) + O(n)$, qui — on le vérifiera en exercice — se résout en $O(n^2)$: aucun gain sur la méthode naïve. Toute l'astuce de Karatsuba tient dans l'observation qu'*il suffit de trois produits.*

Proposition 6.6.2 (L'identité de Karatsuba)**.** *Le terme central de* (6.6.1) *s'obtient à partir des deux autres produits et d'un seul produit supplémentaire,* $(A_0 + A_1)(B_0 + B_1)$ *:*

$$A_0B_1 + A_1B_0 = (A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1.$$

Démonstration. Développons le produit de droite :

$$(A_0 + A_1)(B_0 + B_1) = A_0B_0 + A_0B_1 + A_1B_0 + A_1B_1.$$

En lui retranchant A_0B_0 et A_1B_1, il reste exactement $A_0B_1 + A_1B_0$, le terme central voulu. □

Concrètement, on calcule les trois produits A_0B_0, A_1B_1 et $(A_0 + A_1)(B_0 + B_1)$ — et le terme central s'en déduit par deux soustractions, bien moins coûteuses qu'un produit. Pour que le procédé soit réellement plus rapide, ces trois produits sont eux-mêmes calculés *récursivement* par la même méthode.

```
from produit_naif import produit_naif


def ajouter(P, Q):
    """Somme de deux polynômes (listes de coefficients)."""
    R = [0] * max(len(P), len(Q))
    for i in range(len(P)):
        R[i] = R[i] + P[i]
    for j in range(len(Q)):
        R[j] = R[j] + Q[j]
    return R


def soustraire(P, Q):
    """Différence ``P - Q`` de deux polynômes."""
    R = [0] * max(len(P), len(Q))
    for i in range(len(P)):
        R[i] = R[i] + P[i]
    for j in range(len(Q)):
        R[j] = R[j] - Q[j]
    return R


def karatsuba(A, B):
```

```
    """Renvoie le produit ``A B`` par la méthode de Karatsuba.

    On coupe chaque polynôme en deux moitiés, ``A = A0 + X^m A1`` et
    ``B = B0 + X^m B1``, puis on calcule ``A B`` à l'aide de seulement
    trois produits récursifs ``A0 B0``, ``A1 B1`` et
    ``(A0 + A1)(B0 + B1)``, au lieu des quatre de la formule directe.

    >>> karatsuba([1, 2, 3], [4, 5])
    [4, 13, 22, 15]
    >>> karatsuba([1, 1], [1, 1])
    [1, 2, 1]
    >>> import random
    >>> rnd = lambda t: [random.randint(-5, 5) for _ in range(t)]
    >>> all(karatsuba(A, B) == produit_naif(A, B)
    ...     for n in range(1, 12) for m in range(1, 12)
    ...     for A, B in [(rnd(n), rnd(m))])
    True
    """
    if len(A) == 0 or len(B) == 0:
        return []
    n = max(len(A), len(B))
    if n <= 1:
        return [A[0] * B[0]]
    m = n // 2
    A0, A1 = A[:m], A[m:]
    B0, B1 = B[:m], B[m:]
    P0 = karatsuba(A0, B0)
    P2 = karatsuba(A1, B1)
    P1 = karatsuba(ajouter(A0, A1), ajouter(B0, B1))
    milieu = soustraire(soustraire(P1, P0), P2)
    C = [0] * (len(A) + len(B) - 1)
    for i in range(len(P0)):
        C[i] = C[i] + P0[i]
    for i in range(len(milieu)):
        C[i + m] = C[i + m] + milieu[i]
    for i in range(len(P2)):
        C[i + 2 * m] = C[i + 2 * m] + P2[i]
    return C
```

Les fonctions `ajouter` et `soustraire` réalisent la somme et la différence de deux polynômes coefficient par coefficient ; le cas de base traite les polynômes à un seul coefficient (une simple multiplication de nombres). Le code accepte des listes de longueurs quelconques, pas seulement des puissances de 2 : la coupure se fait alors à l'indice $m = \lfloor n/2 \rfloor$.

6.6.3 Complexité

Théorème 6.6.3 (Karatsuba multiplie en $O(n^{\log_2 3})$)**.** *Le produit de deux polynômes de n coefficients par l'algorithme de Karatsuba demande $O\left(n^{\log_2 3}\right)$ opérations, où $\log_2 3 \approx 1{,}585$.*

Comme $1{,}585 < 2$, c'est un gain réel sur la méthode naïve en $O(n^2)$, d'autant plus net que n est grand.

Démonstration. Notons $t(n)$ le coût pour deux polynômes de $n = 2^k$ coefficients. L'algorithme fait trois appels récursifs sur des polynômes de taille $n/2$, plus un travail de découpe, d'additions et de recombinaison qui est linéaire ; il existe donc une constante $a > 0$ telle que

$$t(n) \leqslant 3\,t(n/2) + a\,n.$$

Déroulons cette récurrence pour $n = 2^k$. En l'appliquant k fois,

$$t(2^k) \leqslant 3^k\,t(1) + a\sum_{i=0}^{k-1} 3^i\,2^{\,k-i}.$$

Dans la somme, mettons 3^k en facteur : $3^i\,2^{k-i} = 3^k\,(2/3)^{k-i}$, donc

$$\sum_{i=0}^{k-1} 3^i\,2^{k-i} = 3^k\sum_{i=0}^{k-1}\left(\tfrac{2}{3}\right)^{k-i} = 3^k\sum_{j=1}^{k}\left(\tfrac{2}{3}\right)^j \leqslant 3^k\sum_{j=1}^{\infty}\left(\tfrac{2}{3}\right)^j = 2\cdot 3^k,$$

car la série géométrique de raison $2/3$ a pour somme $\frac{2/3}{1-2/3} = 2$. On obtient ainsi

$$t(2^k) \leqslant 3^k\,\left(t(1) + 2a\right) = O\left(3^k\right).$$

Il ne reste qu'à réexprimer 3^k en fonction de $n = 2^k$. En prenant le logarithme en base 2, $k = \log_2 n$, donc

$$3^k = 3^{\log_2 n} = \left(2^{\log_2 3}\right)^{\log_2 n} = \left(2^{\log_2 n}\right)^{\log_2 3} = n^{\log_2 3}.$$

D'où $t(n) = O\left(n^{\log_2 3}\right)$, ce qu'il fallait démontrer. □

Remarque 6.6.4. *Tout s'est joué sur le passage de 4 à 3 produits récursifs. La récurrence $t(n) = c\,t(n/2) + O(n)$ donne $O\left(n^{\log_2 c}\right)$ — le* nombre *d'appels récursifs gouverne l'exposant. Avec $c = 4$ on retombe sur n^2, avec $c = 3$ on gagne. C'est un phénomène général du paradigme « diviser pour régner » : économiser ne serait-ce qu'un sous-problème peut faire chuter l'ordre de grandeur. Pour des degrés très grands, on sait d'ailleurs descendre jusqu'à $O(n\log n)$ par des méthodes fondées sur la transformée de Fourier rapide, hors de notre portée ici.*

6.6.4 Exercices

Exercice 6.6.5. *Vérifier en détail que la fonction* `produit_naif` *effectue exactement* $n \times m$ *multiplications de coefficients pour des polynômes de* n *et* m *coefficients, et conclure que sa complexité est en* $O(nm)$*, soit* $O(n^2)$ *lorsque les deux polynômes ont la même taille.*

Exercice 6.6.6. *Résoudre la récurrence* $t(n) = 4\,t(n/2) + a\,n$ *pour* $n = 2^k$*, sur le modèle de la preuve du théorème 6.6.3, et retrouver* $t(n) = O(n^2)$*. C'est la complexité qu'aurait un produit « diviser pour régner » qui n'économiserait pas le quatrième produit : confirmation que toute l'astuce de Karatsuba est dans l'identité de la proposition 6.6.2.*

Exercice 6.6.7. *Instrumenter l'algorithme de Karatsuba pour qu'il compte le nombre de multiplications de coefficients (le travail réellement coûteux) qu'il effectue. Vérifier que, pour deux polynômes de longueur* $n = 2^k$*, ce nombre vaut exactement* $3^k = n^{\log_2 3}$*, en accord avec le théorème 6.6.3.*

Exercice 6.6.8 (★). *La même idée s'applique au produit de deux matrices* 2×2*. Le produit naïf demande* 8 *multiplications de coefficients ; montrer qu'on peut n'en faire que* 7 *(multiplications de Strassen) à l'aide des sept produits*

$$(a_{11} + a_{22})(b_{11} + b_{22}), \quad (a_{21} + a_{22})\,b_{11}, \quad a_{11}(b_{12} - b_{22}), \quad a_{22}(b_{21} - b_{11}),$$

$$(a_{11} + a_{12})\,b_{22}, \quad (a_{21} - a_{11})(b_{11} + b_{12}), \quad (a_{12} - a_{22})(b_{21} + b_{22}),$$

en exprimant les quatre coefficients du produit comme combinaisons linéaires de ces sept quantités. En déduire, pour des matrices $n \times n$ *avec* $n = 2^k$*, un algorithme récursif en* $O\left(n^{\log_2 7}\right)$*, à comparer au produit naïf en* $O(n^3)$*.*

6.7 Sélection en temps linéaire ★

Les sections marquées par une ★ ne sont pas au programme ; elles s'adressent au lecteur curieux d'aller plus loin.

Voici un dernier emploi du paradigme « diviser pour régner », plus subtil que les précédents. On se donne une liste L de n nombres et un entier i avec $0 \leqslant i < n$, et l'on cherche son *élément de rang* i : le $(i + 1)$-ième plus petit élément, c'est-à-dire celui qui occuperait l'indice i si l'on triait L. Le rang 0 est le minimum, le rang $n - 1$ le maximum, et le rang $\lfloor n/2 \rfloor$ est la *médiane.*[3]

Une solution évidente consiste à trier L puis à lire la case d'indice i : cela coûte $O(n \log n)$. Mais trier, c'est répondre à la question pour *tous* les rangs à la fois, alors qu'on n'en veut qu'un seul : on peut espérer mieux. Nous allons construire un algorithme en temps $O(n)$ — *linéaire* — ce qui est optimal, puisqu'il faut bien lire les n éléments au moins une fois.

3. L'ancien énoncé de TD d'où cette section est tirée mélangeait des conventions 0- et 1-indexées (« l'élément de rang i est $L[\pi(i + 1)]$ ») ; nous fixons ici une convention 0-indexée unique, cohérente avec le reste du cours.

6.7.1 Sélectionner autour d'un pivot

Reprenons l'idée de partition du tri rapide, mais à *trois* voies. Choisissons un *pivot*, noté v, parmi les éléments de L, et répartissons les éléments en trois listes : A ceux qui sont strictement inférieurs à v, X ceux qui sont égaux à v, et B ceux qui sont strictement supérieurs. (À la différence de la partition à deux voies du tri rapide — section 6.3 — on isole ici les éléments égaux au pivot dans une liste X à part, ce qui permet de conclure immédiatement lorsque le rang cherché tombe dessus.) L'élément de rang i se trouve alors dans une seule de ces listes, et l'on sait laquelle rien qu'en comparant i aux tailles :

— si $i < |A|$, l'élément de rang i de L est l'élément de rang i de A ;

— si $|A| \leqslant i < |A| + |X|$, alors l'élément cherché est le pivot v lui-même ;

— sinon, $i \geqslant |A| + |X|$, et l'élément de rang i de L est l'élément de rang $i - |A| - |X|$ de B.

Contrairement au tri rapide, on ne recourt *que dans une seule* des deux sous-listes : c'est ce qui laisse espérer un coût linéaire. Encore faut-il que le pivot découpe bien la liste — sans quoi, comme pour le tri rapide, le pire cas dégénère en $O(n^2)$.

6.7.2 Un bon pivot : la médiane des médianes

Tout l'enjeu est de choisir un pivot garantissant que A et B ne sont pas trop grandes. L'idée — astucieuse — est de calculer ce pivot récursivement :

1. on découpe L en paquets de cinq éléments (le dernier pouvant en compter moins) ;
2. on calcule la médiane de chaque paquet (par un tri, c'est $O(1)$ par paquet puisqu'ils ont au plus cinq éléments) ;
3. on prend pour pivot v la *médiane de ces médianes*, calculée par un appel récursif à l'algorithme de sélection lui-même.

```
def mediane_brute(L):
    """Médiane d'une petite liste : son élément de rang ``len(L) // 2``.

    >>> mediane_brute([7, 1, 5])
    5
    >>> mediane_brute([4, 2])
    4
    """
    return sorted(L)[len(L) // 2]


def selection(L, i):
    """Renvoie l'élément de rang ``i`` de ``L`` (avec ``0 <= i < len(L)``).

    L'« élément de rang ``i`` » est le ``(i+1)``-ième plus petit élément,
    celui qui occuperait l'indice ``i`` une fois ``L`` triée (rang ``0`` =
```

```
    minimum, rang ``n-1`` = maximum). Le pivot est la médiane des médianes
    de paquets de cinq éléments, ce qui garantit une partition assez
    équilibrée pour un coût total linéaire.

    >>> selection([5, 3, 8, 1, 9, 2, 7], 0)
    1
    >>> selection([5, 3, 8, 1, 9, 2, 7], 3)
    5
    >>> selection([5, 3, 8, 1, 9, 2, 7], 6)
    9
    >>> import random
    >>> all(selection(L, i) == sorted(L)[i]
    ...     for n in range(1, 50)
    ...     for L in [[random.randint(0, 30) for _ in range(n)]]
    ...     for i in range(n))
    True
    """
    n = len(L)
    if n <= 5:
        return sorted(L)[i]
    paquets = [L[j:j + 5] for j in range(0, n, 5)]
    medianes = [mediane_brute(p) for p in paquets]
    v = selection(medianes, len(medianes) // 2)
    A = [x for x in L if x < v]
    X = [x for x in L if x == v]
    B = [x for x in L if x > v]
    if i < len(A):
        return selection(A, i)
    elif i < len(A) + len(X):
        return v
    else:
        return selection(B, i - len(A) - len(X))
```

Le cas de base trie directement les listes d'au plus cinq éléments. Pour les autres, on voit les trois appels récursifs annoncés : un sur les médianes (pour fabriquer le pivot), un sur A *ou* sur B (jamais les deux).

6.7.3 Correction

Démonstration. Montrons par récurrence forte sur $n = |L|$ que `selection` renvoie bien l'élément de rang i, pour tout i avec $0 \leqslant i < n$.

Cas de base. Si $n \leqslant 5$, la liste est triée et l'on renvoie sa case d'indice i : c'est par définition l'élément de rang i.

Hérédité. Soit $n > 5$. Les listes A, X, B forment une partition de L, avec tous les éléments de A strictement inférieurs à v, ceux de X égaux à v, ceux de B strictement

supérieurs. Dans la liste triée de L, les $|A|$ premières cases sont donc occupées par les éléments de A (dans l'ordre), les $|X|$ suivantes par les copies de v, et le reste par les éléments de B. Selon la position de i par rapport à $|A|$ et $|A|+|X|$, l'élément de rang i tombe dans A, vaut v, ou tombe dans B ; et dans le troisième cas, il y occupe le rang $i-|A|-|X|$, puisque les $|A|+|X|$ premières cases de la liste triée sont déjà passées. Les listes A, B et la liste des médianes ont toutes une longueur strictement inférieure à n (elles excluent au moins le pivot, ou comptent $\lceil n/5\rceil < n$ éléments) : l'hypothèse de récurrence s'applique à chaque appel, et la valeur renvoyée est correcte. □

6.7.4 Pourquoi des paquets de cinq, et pourquoi c'est linéaire

La clef de la complexité est que la médiane des médianes est un pivot *garanti* assez central.

Lemme 6.7.1 (Le pivot écarte un dixième des éléments de chaque côté)**.** *Avec le pivot v ci-dessus, on a $|A| \leqslant \frac{7n}{10}+3$ et, de même, $|B| \leqslant \frac{7n}{10}+3$.*

Démonstration. Traitons $|A|$, le cas de $|B|$ étant symétrique (il s'obtient en échangeant les rôles de $<$ et $>$). Notons $g=\lceil n/5\rceil$ le nombre de paquets. Le pivot v étant la médiane des g médianes, au moins $\lceil g/2\rceil$ d'entre elles lui sont supérieures ou égales. Chacune de ces médianes provient d'un paquet ; écartons l'éventuel dernier paquet incomplet, il en reste au moins $\lceil g/2\rceil-1$ qui sont des médianes de paquets *pleins* (cinq éléments). Or, dans un paquet plein, la médiane a trois éléments qui lui sont supérieurs ou égaux (elle-même et les deux plus grands) ; ces trois éléments sont donc $\geqslant v$. En comptant ces éléments paquet par paquet (ils sont tous distincts), on trouve au moins

$$3\left(\lceil \tfrac{g}{2}\rceil-1\right) \;\geqslant\; 3\left(\tfrac{g}{2}-1\right) \;\geqslant\; 3\left(\tfrac{n}{10}-1\right) \;=\; \tfrac{3n}{10}-3$$

éléments supérieurs ou égaux à v (on a utilisé $g\geqslant n/5$). Ces éléments ne sont pas dans A, qui ne contient que des éléments strictement inférieurs à v ; donc

$$|A| \;\leqslant\; n-\left(\tfrac{3n}{10}-3\right) \;=\; \tfrac{7n}{10}+3. \qquad \square$$

On comprend maintenant le choix du chiffre cinq. Notons $t(n)$ le coût de la sélection sur une liste de n éléments. Le découpage, les médianes de paquets et la partition coûtent $O(n)$; l'appel sur les médianes porte sur $\lceil n/5\rceil$ éléments, et l'appel sur A ou B sur au plus $\frac{7n}{10}+3$ éléments d'après le lemme 6.7.1. D'où

$$t(n) \;\leqslant\; t(\lceil \tfrac{n}{5}\rceil)+t\left(\tfrac{7n}{10}+3\right)+a\,n$$

pour une constante $a>0$. Le point décisif est que $\frac{1}{5}+\frac{7}{10}=\frac{9}{10}<1$: les deux sous-problèmes réunis sont *strictement* plus petits que le problème de départ, ce qui rend la récurrence linéaire.

Proposition 6.7.2 (La sélection est en temps linéaire)**.** *L'algorithme `selection` s'exécute en temps $O(n)$.*

Démonstration. Montrons qu'il existe une constante $c > 0$ telle que $t(n) \leqslant c\,n$ pour tout n. Comme la somme des deux fractions vaut $9/10 < 1$, le terme linéaire laisse une marge. Précisément, supposons $t(m) \leqslant c\,m$ pour tout $m < n$; alors

$$t(n) \leqslant c\left(\tfrac{n}{5} + 1\right) + c\left(\tfrac{7n}{10} + 3\right) + a\,n = \tfrac{9}{10}\,c\,n + 4c + a\,n.$$

Cette quantité est $\leqslant c\,n$ dès que $a\,n + 4c \leqslant \frac{1}{10}c\,n$, c'est-à-dire dès que $a + \frac{4c}{n} \leqslant \frac{c}{10}$. En prenant $c = 20\,a$, le membre de droite vaut $2a$, et la condition devient $\frac{80a}{n} \leqslant a$, soit $n \geqslant 80$. Pour $n \geqslant 80$, l'hérédité est donc acquise avec $c = 20\,a$; et quitte à agrandir c pour couvrir les (en nombre fini) valeurs $n < 80$, on obtient $t(n) \leqslant c\,n$ pour tout n. C'est bien $t(n) = O(n)$. □

6.7.5 Exercices

Exercice 6.7.3. *Dans le troisième cas de l'algorithme (la recherche se poursuit dans B), justifier soigneusement, sur un petit exemple numérique de votre choix, que le rang cherché dans B est bien $i - |A| - |X|$ et non i.*

Exercice 6.7.4 (★). *Écrire une variante* `quickselect` *qui, au lieu de la médiane des médianes, prend pour pivot un élément* tiré au hasard. *Le code est bien plus court. Vérifier expérimentalement qu'il renvoie le bon résultat, et expliquer pourquoi son coût dans le pire des cas remonte à $O(n^2)$ alors que son coût moyen reste $O(n)$ (on pourra reprendre l'avertissement sur le pivot du tri rapide, section 6.3).*

Exercice 6.7.5 (★). *Pourquoi des paquets de cinq ? Reprendre l'analyse du lemme 6.7.1 avec des paquets de* trois *éléments : montrer qu'on obtient alors $|A|, |B| \leqslant \frac{2n}{3} + O(1)$, d'où la récurrence $t(n) \leqslant t(\lceil n/3 \rceil) + t(\frac{2n}{3} + O(1)) + a\,n$. Pourquoi cette récurrence ne donne-t-elle* pas *un coût linéaire ?* (Indication : comparer $\frac{1}{3} + \frac{2}{3}$ à 1.)

7 Programmation dynamique

Résumé

Ce chapitre présente un second grand paradigme algorithmique, la *programmation dynamique*. Comme le « diviser pour régner » du chapitre précédent, elle résout un problème en combinant les solutions de sous-problèmes plus petits ; mais elle s'applique précisément là où ce découpage échoue, quand les sous-problèmes *se chevauchent* et seraient recalculés un nombre exponentiel de fois. L'idée est alors de calculer chaque sous-problème une seule fois et de *mémoriser* son résultat dans une table. Nous dégageons le principe sur une suite définie par une récurrence à deux indices, puis nous l'appliquons à deux problèmes classiques : la plus longue sous-suite commune à deux mots et le problème du sac à dos.

Prérequis

La récursivité et le raisonnement par récurrence, simple et forte (chapitre 5) ; les notations de complexité, en particulier $O(\cdot)$ (chapitre 3) ; le paradigme « diviser pour régner », à quoi la programmation dynamique s'oppose point par point (chapitre 6).

Objectifs

À l'issue de ce chapitre, vous saurez reconnaître un problème qui se prête à la programmation dynamique, identifier ses sous-problèmes et la relation de récurrence qui les relie, remplir la table correspondante « dans le bon ordre », en déduire la valeur optimale cherchée, et enfin remonter dans la table pour reconstruire une solution — pas seulement sa valeur.

Vous connaissez sans doute la suite de Fibonacci, définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$. Sa définition est récursive ; il est donc tentant de la programmer par une fonction qui s'appelle elle-même. Mais une telle fonction est catastrophiquement lente : pour calculer F_n, elle appelle F_{n-1} et F_{n-2}, qui rappellent chacun leurs deux prédécesseurs, et ainsi de suite. La valeur F_{n-2}, par exemple, est recalculée deux fois ; F_{n-3}, trois fois ; et de proche en proche le nombre d'appels explose exponentiellement. On recalcule sans fin les mêmes valeurs. *Comment éviter ce gâchis ?*

La réponse tient en un mot : *mémoriser*. Plutôt que de relancer un calcul chaque fois qu'on en a besoin, on le mène une seule fois et l'on range son résultat dans une table ; les fois suivantes, on se contente de le relire. C'est toute l'idée de la *programmation*

dynamique[1].

À première vue, cela ressemble au « diviser pour régner » du chapitre précédent : dans les deux cas on résout un problème à partir de sous-problèmes plus petits. La différence est décisive. Dans un algorithme « diviser pour régner », les sous-problèmes sont *disjoints* — le tri fusion trie deux moitiés sans aucun élément commun — et chacun n'est résolu qu'une fois. En programmation dynamique, au contraire, les sous-problèmes *se chevauchent* : un même sous-problème resurgit dans plusieurs décompositions, et c'est exactement ce chevauchement qui rend la récursion naïve exponentielle et la mémorisation payante.

7.1 Une suite définie par une récurrence sur deux indices

Dégageons le principe sur un exemple aussi dépouillé que possible, qui contient déjà toute la mécanique : une suite *à deux indices* définie par une récurrence.

On se donne une fonction $f\colon \mathbb{N}^3 \to \mathbb{N}$, vue comme une *boîte noire* : on sait l'appeler sur trois entiers et récupérer sa valeur, sans rien supposer de la façon dont elle est calculée. On se donne aussi deux suites « de bord » : une suite $(a_{i,0})_{i\in\mathbb{N}}$ (la première colonne) et une suite $(a_{0,j})_{j\in\mathbb{N}}$ (la première ligne). À partir de ces données, on *définit* une suite à deux indices $(a_{i,j})_{i,j\in\mathbb{N}}$ par la relation de récurrence

$$a_{i+1,\,j+1} = f\big(a_{i,\,j+1},\ a_{i+1,\,j},\ a_{i,\,j}\big). \tag{7.1.1}$$

Concrètement, on dispose les valeurs dans un tableau dont les lignes sont indexées par i et les colonnes par j ; la relation (7.1.1) exprime la case $a_{i+1,j+1}$ en fonction de ses trois voisines déjà placées : celle du *nord* ($a_{i,j+1}$), celle de l'*ouest* ($a_{i+1,j}$) et celle du *nord-ouest* ($a_{i,j}$).

Exemple 7.1.2. *Prenons $f(\text{nord}, \text{ouest}, \text{nord-ouest}) = \text{nord} + \text{ouest}$, avec des bords tous égaux à 1 ($a_{i,0} = a_{0,j} = 1$). La case $a_{i,j}$ compte alors le nombre de chemins croissants menant du coin $(0,0)$ à la case (i,j) en n'avançant que vers le bas ou vers la droite : c'est le coefficient binomial $\binom{i+j}{i}$. Les premières valeurs forment le triangle de Pascal couché :*

$i\backslash j$	0	1	2	3
0	1	1	1	1
1	1	2	3	4
2	1	3	6	10

Ici la case du nord-ouest n'intervient pas ; elle interviendra dans les exemples des sections suivantes.

7.1.1 La récurrence définit bien une unique suite

Avant de calculer cette suite, assurons-nous qu'elle est bien définie : que les bords et la relation (7.1.1) déterminent *une* suite *et une seule*. Ce n'est pas une formalité creuse ;

[1]. Le mot « programmation » est ici un synonyme de « planification », hérité du vocabulaire de l'optimisation des années 1950 ; il n'a rien à voir avec la programmation au sens de l'écriture de code.

c'est l'occasion de voir comment on raisonne par récurrence quand l'objet dépend de deux indices.

Proposition 7.1.3 (Unicité de la suite double)**.** *Soient deux suites* $(a_{i,j})$ *et* $(b_{i,j})$ *vérifiant toutes deux la relation* (7.1.1)*. Si elles ont les mêmes bords, c'est-à-dire si* $a_{i,0} = b_{i,0}$ *et* $a_{0,j} = b_{0,j}$ *pour tout* $i, j \in \mathbb{N}$*, alors elles sont égales :* $a_{i,j} = b_{i,j}$ *pour tous* $i, j \in \mathbb{N}$*.*

En d'autres termes, deux suites qui obéissent à la même récurrence et partent des mêmes bords ne peuvent que coïncider partout : la relation (7.1.1) ne laisse aucune liberté.

Démonstration. La difficulté n'est pas de trouver l'argument — c'est une récurrence — mais de décider *sur quel indice* elle porte. Comme la suite a deux indices, nous *imbriquons* deux récurrences : une récurrence *externe* sur i, et, à chaque étape de celle-ci, une récurrence *interne* sur j.

Récurrence externe. Pour $i \in \mathbb{N}$, notons P_i la propriété « $a_{i,j} = b_{i,j}$ pour tout $j \in \mathbb{N}$ ». Nous montrons P_i pour tout i par récurrence sur i.

Initialisation. P_0 affirme que $a_{0,j} = b_{0,j}$ pour tout j : c'est exactement l'égalité des bords du haut, donc P_0 est vraie.

Hérédité. Supposons P_i vraie pour un certain i, et montrons P_{i+1}, c'est-à-dire $a_{i+1,j} = b_{i+1,j}$ pour tout j. C'est ici qu'intervient la récurrence interne, sur j.

— *Initialisation (interne).* Pour $j = 0$, l'égalité $a_{i+1,0} = b_{i+1,0}$ est celle des bords de gauche : vraie.

— *Hérédité (interne).* Supposons $a_{i+1,j} = b_{i+1,j}$, et montrons $a_{i+1,j+1} = b_{i+1,j+1}$. La relation (7.1.1) donne

$$a_{i+1,j+1} = f(a_{i,j+1},\, a_{i+1,j},\, a_{i,j}) \qquad \text{et} \qquad b_{i+1,j+1} = f(b_{i,j+1},\, b_{i+1,j},\, b_{i,j}).$$

Or les trois arguments coïncident deux à deux : $a_{i,j+1} = b_{i,j+1}$ et $a_{i,j} = b_{i,j}$ par l'hypothèse de récurrence *externe* P_i, et $a_{i+1,j} = b_{i+1,j}$ par l'hypothèse de récurrence *interne*. Les deux appels de f portent donc sur les mêmes arguments et renvoient la même valeur : $a_{i+1,j+1} = b_{i+1,j+1}$.

La récurrence interne établit $a_{i+1,j} = b_{i+1,j}$ pour tout j, c'est-à-dire P_{i+1}. La récurrence externe est ainsi complète, et $a_{i,j} = b_{i,j}$ pour tous i, j. □

Remarque 7.1.4. *On peut aussi démontrer cette unicité « par l'absurde », en considérant un couple* (i, j) minimal *où les deux suites diffèrent et en exhibant une contradiction. C'est l'objet de l'exercice 7.1.7; l'argument y est plus court à énoncer, mais il repose sur le fait — qui mérite réflexion — qu'un ensemble non vide de couples d'entiers admet toujours un élément minimal.*

7.1.2 Un calcul récursif, mais exponentiel

Supposons qu'on sache calculer les bords : une fonction `bord_gauche` qui à i associe $a_{i,0}$, et une fonction `bord_haut` qui à j associe $a_{0,j}$. La relation (7.1.1) se traduit alors mot pour mot en une fonction récursive.

```
def suite_double_recursif(i, j, f, bord_gauche, bord_haut):
    """Calcule a[i][j] pour la suite définie par
    a[i+1][j+1] = f(a[i][j+1], a[i+1][j], a[i][j]),
    de bords a[i][0] = bord_gauche(i) et a[0][j] = bord_haut(j).

    Version récursive directe : naïve, car elle recalcule un nombre
    exponentiel de fois les mêmes valeurs intermédiaires.

    Comptage de chemins croissants dans une grille (coefficients
    binomiaux), où f(nord, ouest, nord_ouest) = nord + ouest :

    >>> nombre = lambda nord, ouest, nord_ouest: nord + ouest
    >>> uns = lambda k: 1
    >>> suite_double_recursif(2, 2, nombre, uns, uns)
    6
    >>> suite_double_recursif(4, 2, nombre, uns, uns)
    15
    """
    if i == 0:
        return bord_haut(j)
    if j == 0:
        return bord_gauche(i)
    return f(suite_double_recursif(i - 1, j, f, bord_gauche, bord_haut),
             suite_double_recursif(i, j - 1, f, bord_gauche, bord_haut),
             suite_double_recursif(i - 1, j - 1, f, bord_gauche, bord_haut))
```

Cette fonction est correcte, mais elle retombe exactement dans le piège de Fibonacci décrit en ouverture du chapitre. Pour calculer $a_{i,j}$, elle lance *trois* appels récursifs, sur $(i-1,j)$, $(i,j-1)$ et $(i-1,j-1)$; or ces trois sous-problèmes en partagent de nombreux autres. La case $(i-1,j-1)$, par exemple, est demandée à la fois directement et par l'intermédiaire de $(i-1,j)$ et de $(i,j-1)$. Les sous-problèmes *se chevauchent*, et leur nombre d'évaluations croît exponentiellement avec $i+j$. C'est précisément la situation que la programmation dynamique est faite pour corriger.

Il y a deux façons classiques d'y remédier. La première garde l'écriture récursive mais lui adjoint un cache : on range chaque valeur calculée dans une table et, avant tout appel, on regarde si la réponse n'y figure pas déjà — c'est la *mémoïsation* (« descendante », du problème vers ses sous-problèmes). La seconde renonce à la récursion et remplit directement la table « dans le bon ordre », des sous-problèmes vers le problème (« montante ») : c'est celle que nous développons maintenant, et à laquelle on réserve d'ordinaire le nom de programmation dynamique.

7.1.3 Calcul par remplissage de table

L'observation clef est qu'il n'y a, en tout, que $(n+1)(m+1)$ valeurs distinctes à calculer pour obtenir $a_{n,m}$: les cases du tableau d'indices $0 \leqslant i \leqslant n$ et $0 \leqslant j \leqslant m$. Plutôt

que de les redécouvrir indéfiniment par la récursion, calculons-les *une seule fois chacune*, et rangeons-les dans une table T où $T[i][j]$ contiendra $a_{i,j}$.

Tout est dans l'*ordre* de remplissage. La case $T[i][j]$ se calcule à partir de ses voisines du nord, de l'ouest et du nord-ouest, c'est-à-dire des cases d'indices plus petits. Il suffit donc de remplir d'abord les bords, puis de parcourir le tableau *ligne par ligne, par colonnes croissantes* : quand on arrive à la case (i, j), ses trois voisines sont déjà calculées.

```
def suite_double(n, m, f, bord_gauche, bord_haut):
    """Calcule la table des a[i][j] pour 0 <= i <= n et 0 <= j <= m,
    où a[i+1][j+1] = f(a[i][j+1], a[i+1][j], a[i][j]) et les bords
    valent a[i][0] = bord_gauche(i), a[0][j] = bord_haut(j).

    On remplit la table ligne par ligne, par colonnes croissantes :
    quand on calcule a[i][j], les trois cases dont il dépend
    (nord, ouest, nord-ouest) sont déjà connues.

    Comptage de chemins croissants dans une grille (coefficients
    binomiaux), où f(nord, ouest, nord_ouest) = nord + ouest :

    >>> nombre = lambda nord, ouest, nord_ouest: nord + ouest
    >>> uns = lambda k: 1
    >>> T = suite_double(4, 2, nombre, uns, uns)
    >>> T[2][2]
    6
    >>> T[4][2]
    15
    >>> [ligne[2] for ligne in T]
    [1, 3, 6, 10, 15]
    """
    T = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        T[i][0] = bord_gauche(i)
    for j in range(m + 1):
        T[0][j] = bord_haut(j)
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            T[i][j] = f(T[i - 1][j], T[i][j - 1], T[i - 1][j - 1])
    return T
```

Conformément à notre convention d'indexation, la table T a $n+1$ lignes et $m+1$ colonnes, indexées à partir de 0 ; `T[i][j]` désigne la case de la ligne i et de la colonne j. Les deux premières boucles installent les bords ; les deux boucles imbriquées remplissent le reste dans l'ordre voulu.

Remarque 7.1.5. *On prendra garde, dans la boucle interne, à parcourir un indice* distinct *de celui de la boucle externe (ici* `j`*, et non* `i`*) : réutiliser par mégarde la variable de la boucle externe est une erreur classique, qui fausse silencieusement tout le calcul.*

7.1.4 Coût

Le remplissage des bords coûte $O(n+m)$ opérations. Le double parcours exécute son corps — une évaluation de f et une affectation, soit un nombre constant d'opérations, puisque f est une boîte noire supposée de coût unitaire — exactement $n \times m$ fois. Le calcul de $a_{n,m}$ par remplissage de table demande donc

$$O(n \cdot m)$$

opérations, là où la version récursive en demandait un nombre exponentiel. On a échangé du *temps* contre de l'*espace* : on stocke les $(n+1)(m+1)$ cases de la table, mais on ne calcule chacune qu'une fois. C'est le compromis caractéristique de la programmation dynamique.

7.1.5 Exercices

Exercice 7.1.6. *On prend pour f la fonction $f(nord, ouest, nord\text{-}ouest) = \max(nord, ouest) + nord\text{-}ouest$, avec les bords $a_{i,0} = a_{0,j} = 1$. Calculer à la main la table des $a_{i,j}$ pour $0 \leqslant i \leqslant 3$ et $0 \leqslant j \leqslant 3$, en respectant l'ordre de remplissage « ligne par ligne, colonnes croissantes ».*

Exercice 7.1.7 (★). *Reprendre la proposition 7.1.3 et la démontrer* par l'absurde*, selon le schéma annoncé à la remarque 7.1.4 : supposer qu'il existe un couple (i, j) avec $a_{i,j} \neq b_{i,j}$, en choisir un* minimal *(i_0, j_0), montrer que $i_0 > 0$ et $j_0 > 0$, puis aboutir à une contradiction en appliquant la relation (7.1.1) en (i_0, j_0). Où, exactement, utilise-t-on la minimalité ?*

Exercice 7.1.8. *La lenteur de la version récursive ne vient pas de la récursion elle-même, mais du fait qu'elle recalcule les mêmes cases. Modifier* `suite_double_recursif` *pour qu'elle conserve dans un dictionnaire les valeurs déjà calculées et ne calcule jamais deux fois la même case (technique dite de* mémoïsation*). Vérifier que l'on retrouve les mêmes valeurs que par remplissage de table, et expliquer pourquoi le coût retombe à $O(n \cdot m)$.*

Exercice 7.1.9. *Pour mesurer le gâchis de la version récursive, modifier* `suite_double_recursif` *afin qu'elle compte le nombre total d'appels qu'elle effectue pour calculer $a_{n,m}$. Comparer ce nombre à $n \times m$ pour quelques petites valeurs (par exemple $n = m = 10$) et constater l'écart.*

7.2 Plus longue sous-suite commune : formulation

Nous avons dégagé à la section précédente le principe de la programmation dynamique sur une suite-jouet. Mettons-le à l'épreuve sur un vrai problème, emprunté au traitement des mots : la recherche d'une plus longue sous-suite commune à deux mots. Cette section

en pose le cadre et établit la *propriété fondamentale* sur laquelle reposera l'algorithme de la section suivante.

7.2.1 Le problème

On se donne un *alphabet* fini Σ — par exemple les quatre lettres $\{\mathtt{a}, \mathtt{t}, \mathtt{g}, \mathtt{c}\}$ de l'ADN, ou les caractères d'un fichier texte. Un *mot* sur Σ est une suite finie de lettres de Σ ; on note Σ^* l'ensemble des mots, et $|w|$ la longueur d'un mot w.

Conformément à notre convention d'indexation, les lettres d'un mot u de longueur n sont numérotées à partir de 0 : $u = u_0 u_1 \cdots u_{n-1}$. Pour $0 \leqslant i \leqslant n$, on note $u[:i] = u_0 u_1 \cdots u_{i-1}$ le *préfixe* de longueur i de u, c'est-à-dire ses i premières lettres ; en particulier $u[:0]$ est le *mot vide*, noté ε, et $u[:n] = u$.[2]

Définition 7.2.1 (Sous-suite, sous-suite commune)**.** *Une* sous-suite *d'un mot* $u = u_0 \cdots u_{n-1}$ *est un mot obtenu en* effaçant *certaines lettres de* u *sans changer l'ordre des lettres restantes : c'est un mot* $w = w_0 \cdots w_{k-1}$ *tel qu'il existe des indices*

$$0 \leqslant i_0 < i_1 < \cdots < i_{k-1} \leqslant n-1 \qquad \text{avec} \qquad w_\ell = u_{i_\ell} \ \text{ pour tout } 0 \leqslant \ell \leqslant k-1.$$

Un mot w *est une* sous-suite commune *à deux mots* u *et* v *s'il est à la fois une sous-suite de* u *et une sous-suite de* v.

En d'autres termes, w est une sous-suite de u si l'on retrouve les lettres de w, dans l'ordre, disséminées dans u — pas nécessairement côte à côte. Ainsi `ace` est une sous-suite de `abcde`, mais `aec` n'en est pas une (l'ordre n'y est pas).

Avertissement 7.2.2. *Il ne faut pas confondre* sous-suite *et* facteur *(ou « sous-mot contigu »). Un facteur de* u *est fait de lettres* consécutives *:* `bcd` *est un facteur de* `abcde`*. Une sous-suite, elle, autorise les trous :* `ace` *est une sous-suite mais pas un facteur. C'est de sous-suites, et non de facteurs, qu'il est question ici.*

Le *problème de la plus longue sous-suite commune* (en anglais *longest common subsequence*, d'où le sigle LCS) consiste, étant donné u et v, à déterminer une sous-suite commune à u et v de longueur maximale.

Exemple 7.2.3. *Prenons les deux brins d'ADN*

$$u = \mathtt{ttatatgcgt} \qquad \text{et} \qquad v = \mathtt{tatcccctta}.$$

Une plus longue sous-suite commune est `tattt`*, de longueur* 5 *:*

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
u	**t**	t	**a**	**t**	a	**t**	g	c	g	**t**
v	**t**	**a**	**t**	c	c	c	c	**t**	**t**	a

2. C'est exactement la tranche `u[:i]` de Python, déjà rencontrée pour les listes. L'ancien cours indexait au contraire les mots à partir de 1, « contrairement à Python » ; nous nous en tenons à la convention unique du cours, à partir de 0.

Elle correspond aux indices $0, 2, 3, 5, 9$ *dans* u *et* $0, 1, 2, 7, 8$ *dans* v*. Cette plus longue sous-suite n'est pas unique :* `tatct` *en est une autre, de même longueur.* Il peut donc exister plusieurs *plus longues sous-suites communes — c'est pourquoi on parle d'« une » et non de « la ».*

Remarque 7.2.4 (Une approche naïve, exponentielle)**.** *On pourrait songer à énumérer toutes les sous-suites de* u*, de la plus longue à la plus courte, et à tester pour chacune si c'est aussi une sous-suite de* v*. Mais un mot de longueur* n *possède* 2^n *sous-suites (une par sous-ensemble d'indices à conserver) : cette approche est de coût exponentiel, inutilisable dès que les mots dépassent quelques dizaines de lettres. C'est ce mur que la programmation dynamique va nous faire contourner.*

Remarque 7.2.5 (Où ce problème se pose)**.** *La plus longue sous-suite commune n'est pas un problème « pour le plaisir » : c'est exactement ce que calcule un système de gestion de versions tel que* git *lorsqu'il affiche les différences entre deux versions d'un fichier (la commande* `diff`*). En voyant chaque ligne comme une « lettre », la plus longue sous-suite commune aux deux versions donne les lignes inchangées ; le reste constitue les ajouts et les suppressions affichés. On retrouve le même besoin en bio-informatique, pour aligner deux séquences d'ADN, et dans certains algorithmes de compression.*

Remarque 7.2.6 (D'autres problèmes sur les mots)**.** *La plus longue sous-suite commune n'est qu'un représentant d'une vaste famille de problèmes algorithmiques sur les mots. Un autre exemple classique est la* transformation de Burrows–Wheeler*, au cœur de compresseurs comme* `bzip2`*, qui réordonne les lettres d'un texte pour le rendre plus compressible ; nous ne l'étudierons pas ici, mais elle illustre la richesse du domaine.*

7.2.2 La propriété fondamentale

Tout l'art de la programmation dynamique consiste à relier la solution d'un problème à celles de sous-problèmes plus petits. Ici, les sous-problèmes naturels sont les plus longues sous-suites communes aux *préfixes* de u et de v. Le lemme suivant — la *propriété d'optimalité des sous-structures* — est la clef de voûte de l'algorithme : il décrit comment une plus longue sous-suite commune se ramène à celle de préfixes plus courts, selon que les dernières lettres coïncident ou non.

Lemme 7.2.7 (Optimalité des sous-structures)**.** *Soient* $u = u_0 \cdots u_{n-1}$ *et* $v = v_0 \cdots v_{m-1}$ *deux mots non vides, et soit* $w = w_0 \cdots w_{k-1}$ *une plus longue sous-suite commune à* u *et* v*.*

1. *Si* $u_{n-1} = v_{m-1}$ *(les dernières lettres coïncident), alors* $w_{k-1} = u_{n-1} = v_{m-1}$*, et* $w[:k-1]$ *est une plus longue sous-suite commune à* $u[:n-1]$ *et* $v[:m-1]$.
2. *Si* $u_{n-1} \neq v_{m-1}$*, alors* $w_{k-1} \neq u_{n-1}$ *ou* $w_{k-1} \neq v_{m-1}$*, et :*
   - *si* $w_{k-1} \neq u_{n-1}$*, alors* w *est une plus longue sous-suite commune à* $u[:n-1]$ *et* v *;*
   - *si* $w_{k-1} \neq v_{m-1}$*, alors* w *est une plus longue sous-suite commune à* u *et* $v[:m-1]$.

Concrètement : quand les deux mots finissent par la même lettre, on peut toujours faire en sorte que la plus longue sous-suite commune la contienne, et l'on est ramené aux deux mots privés de leur dernière lettre ; sinon, l'une au moins des deux dernières lettres ne sert à rien, et on peut la jeter.

Démonstration. Démontrons successivement les deux points.

Point 1 : cas $u_{n-1} = v_{m-1}$. Montrons d'abord que $w_{k-1} = u_{n-1}$. Supposons par l'absurde que $w_{k-1} \neq u_{n-1}$. Comme $u_{n-1} = v_{m-1}$, on a aussi $w_{k-1} \neq v_{m-1}$. On pourrait alors *prolonger* w en lui ajoutant la lettre $u_{n-1} = v_{m-1}$ à la fin : cette lettre est la dernière de u et de v, donc d'indice strictement plus grand que tous les indices utilisés par w, et le mot $w\,u_{n-1}$ serait une sous-suite commune à u et v de longueur $k+1 > k$. Cela contredit la maximalité de w. Donc $w_{k-1} = u_{n-1} = v_{m-1}$.

Il reste à voir que $w[:k-1]$ est une plus longue sous-suite commune à $u[:n-1]$ et $v[:m-1]$. C'est bien une sous-suite commune à ces deux préfixes : la dernière lettre de w étant placée sur u_{n-1} et v_{m-1}, les $k-1$ premières le sont sur des indices $\leqslant n-2$ dans u et $\leqslant m-2$ dans v. Si elle n'était pas de longueur maximale, il existerait une sous-suite commune w' à $u[:n-1]$ et $v[:m-1]$ de longueur $\geqslant k$; en lui ajoutant la lettre $u_{n-1} = v_{m-1}$, on obtiendrait une sous-suite commune à u et v de longueur $\geqslant k+1$, de nouveau absurde.

Point 2 : cas $u_{n-1} \neq v_{m-1}$. La dernière lettre de w ne peut pas être à la fois égale à u_{n-1} et à v_{m-1}, puisque ces deux lettres diffèrent : donc $w_{k-1} \neq u_{n-1}$ ou $w_{k-1} \neq v_{m-1}$. Traitons le cas $w_{k-1} \neq u_{n-1}$; le cas $w_{k-1} \neq v_{m-1}$ est symétrique (échanger les rôles de u et v). Puisque $w_{k-1} \neq u_{n-1}$, aucune lettre de w n'est placée sur la dernière lettre de u : w est donc déjà une sous-suite de $u[:n-1]$, et reste bien sûr une sous-suite de v. C'est une sous-suite commune à $u[:n-1]$ et v. Elle y est de longueur maximale : toute sous-suite commune à $u[:n-1]$ et v est en particulier une sous-suite commune à u et v, donc de longueur $\leqslant k = |w|$. □

Traduisons ce lemme en une relation de récurrence sur les *longueurs*. Pour $0 \leqslant i \leqslant n$ et $0 \leqslant j \leqslant m$, posons

$$A[i][j] = \text{longueur d'une plus longue sous-suite commune à } u[:i] \text{ et } v[:j].$$

C'est un tableau à deux dimensions, exactement la « suite double » de la section précédente — nous l'écrivons désormais avec des crochets, $A[i][j]$, plutôt qu'avec des indices, $a_{i,j}$, pour coller à la notation des tableaux Python (`A[i][j]`) dont nous nous servirons dans le code. Ses valeurs croissent (au sens large) quand les indices croissent : $A[i][j] \leqslant A[i'][j']$ dès que $i \leqslant i'$ et $j \leqslant j'$, puisqu'une sous-suite commune à deux préfixes en est encore une de préfixes plus longs.

Proposition 7.2.8 (Récurrence sur le tableau des longueurs)**.** *On a* $A[0][j] = A[i][0] = 0$ *pour tous* i, j *(le mot vide n'a pas de sous-suite commune non vide). De plus, pour* $0 \leqslant i < n$ *et* $0 \leqslant j < m$ *:*

$$A[i+1][j+1] = \begin{cases} A[i][j] + 1 & \text{si } u_i = v_j, \\ \max(A[i+1][j],\, A[i][j+1]) & \text{sinon.} \end{cases}$$

Démonstration. Pour $i = 0$, le préfixe $u[:0]$ est le mot vide : sa seule sous-suite est le mot vide, donc $A[0][j] = 0$ pour tout j. De même $A[i][0] = 0$ pour tout i.

Soient maintenant $0 \leqslant i < n$ et $0 \leqslant j < m$. On applique le lemme 7.2.7 aux préfixes $u[:i+1]$ et $v[:j+1]$, dont les dernières lettres sont u_i et v_j.

— Si $u_i = v_j$, le point 1 donne qu'une plus longue sous-suite commune à $u[:i+1]$ et $v[:j+1]$ se termine par u_i et que, privée de cette dernière lettre, c'est une plus longue sous-suite commune à $u[:i]$ et $v[:j]$. D'où $A[i+1][j+1] = A[i][j] + 1$.

— Si $u_i \neq v_j$, le point 2 donne qu'une plus longue sous-suite commune à $u[:i+1]$ et $v[:j+1]$ est, soit une plus longue sous-suite commune à $u[:i]$ et $v[:j+1]$ (de longueur $A[i][j+1]$), soit une plus longue sous-suite commune à $u[:i+1]$ et $v[:j]$ (de longueur $A[i+1][j]$). Sa longueur est donc le plus grand des deux, $\max(A[i+1][j], A[i][j+1])$.

□

Cette proposition contient déjà tout l'algorithme. La première ligne et la première colonne du tableau A ne demandent aucun calcul ; on remplit ensuite les cases restantes *ligne par ligne, par colonnes croissantes*, exactement comme à la section 7.1, et la valeur $A[n][m]$ sera la longueur cherchée. C'est l'objet de la section suivante, où nous verrons aussi comment *reconstruire* une plus longue sous-suite, et pas seulement calculer sa longueur.

7.2.3 Exercices

Exercice 7.2.9. *Soient $u =$ `atcga` et $v =$ `tgcat`. En appliquant la proposition 7.2.8, remplir à la main le tableau A des longueurs (de taille 6×6). Quelle est la longueur d'une plus longue sous-suite commune ? En exhiber une.*

Exercice 7.2.10. *Sur l'exemple 7.2.3, vérifier que `tatct` est bien une sous-suite commune à u et v, en exhibant les indices correspondants dans chacun des deux mots. Trouver une troisième plus longue sous-suite commune.*

Exercice 7.2.11. *Écrire une fonction `est_sous_suite(w, u)` qui teste si le mot `w` est une sous-suite du mot `u`, en temps $O(|u|)$.* Indication : *balayer `u` une seule fois en avançant dans `w` chaque fois qu'on retrouve sa lettre courante.*

Exercice 7.2.12. *Justifier l'affirmation de la remarque 7.2.4 selon laquelle un mot de longueur n possède exactement 2^n sous-suites (en comptant le mot vide, et en comptant une fois chaque* façon *d'effacer des lettres). En déduire que l'approche naïve qui les énumère toutes est de coût au moins exponentiel.*

7.3 Plus longue sous-suite commune : algorithme et reconstruction

Nous avons établi à la section précédente que les longueurs des plus longues sous-suites communes aux préfixes de u et v obéissent à une récurrence (proposition 7.2.8). Il ne reste qu'à la programmer, par remplissage de table, puis à apprendre à *reconstruire* une plus longue sous-suite — car la table ne donne pour l'instant que sa *longueur*.

7.3.1 Remplir la table

La récurrence exprime $A[i+1][j+1]$ à partir de cases d'indices plus petits : celles du nord, de l'ouest et du nord-ouest. Il suffit donc, après avoir mis à zéro la première ligne et la première colonne, de parcourir la table *ligne par ligne, par colonnes croissantes* : quand on calcule une case, les cases dont elle dépend sont déjà connues.

Plutôt que de ne garder que les longueurs, retenons au passage, pour chaque case, *d'où vient* sa valeur — du nord-ouest (une lettre commune vient d'être retenue), du nord ou de l'ouest. Cette information, rangée dans une seconde table `provenance`, permettra de remonter le calcul pour reconstruire une plus longue sous-suite. C'est une technique générale en programmation dynamique : *garder la trace* des choix faits pour pouvoir reconstruire une solution, et pas seulement sa valeur.

```python
def plsc_table(u, v):
    """Calcule par programmation dynamique le tableau A des longueurs et le
    tableau des provenances pour la plus longue sous-suite commune à u et v.

    A[i][j] est la longueur d'une plus longue sous-suite commune aux préfixes
    u[:i] et v[:j] ; provenance[i][j] indique de quelle case voisine vient
    cette valeur : "diag" (nord-ouest, une lettre commune retenue), "haut"
    (nord) ou "gauche" (ouest). On remplit ligne par ligne, par colonnes
    croissantes. Coût O(len(u) * len(v)).

    >>> A, _ = plsc_table("atgcg", "agcccc")
    >>> A[-1][-1]
    3
    >>> A, _ = plsc_table("ttatatgcgt", "tatccccta")
    >>> A[-1][-1]
    5
    """
    n, m = len(u), len(v)
    A = [[0] * (m + 1) for _ in range(n + 1)]
    provenance = [[None] * (m + 1) for _ in range(n + 1)]
    for i in range(n):
        for j in range(m):
            if u[i] == v[j]:
                A[i + 1][j + 1] = A[i][j] + 1
                provenance[i + 1][j + 1] = "diag"
            elif A[i][j + 1] >= A[i + 1][j]:
                A[i + 1][j + 1] = A[i][j + 1]
                provenance[i + 1][j + 1] = "haut"
            else:
                A[i + 1][j + 1] = A[i + 1][j]
                provenance[i + 1][j + 1] = "gauche"
    return A, provenance
```

La fonction renvoie les deux tables. Conformément à notre convention, elles ont $n+1$ lignes et $m+1$ colonnes indexées à partir de 0 ; les lettres comparées dans le corps de boucle sont `u[i]` et `v[j]` (d'indices 0 à $n-1$, resp. 0 à $m-1$), qui sont les dernières lettres des préfixes `u[:i+1]` et `v[:j+1]`. On notera que les deux boucles imbriquées parcourent des indices *distincts*, `i` puis `j` : réutiliser par mégarde la variable externe dans la boucle interne fausserait tout le calcul.

Exemple 7.3.1. *Déroulons l'algorithme sur* $u =$ `atgcg` *et* $v =$ `agcccc`*. Le tableau A obtenu est :*

$A[i][j]$	ε	a	g	c	c	c	c
ε	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
t	0	1	1	1	1	1	1
g	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
g	0	1	2	3	3	3	3

Les lignes sont indexées par les lettres de u (préfixe vide ε en tête), les colonnes par celles de v. La case en bas à droite, $A[5][6] = 3$*, donne la longueur d'une plus longue sous-suite commune : ici 3. On lit « en diagonale » les lettres retenues :* `a`*,* `g`*,* `c`*, c'est-à-dire la sous-suite* `agc`*.*

7.3.2 Reconstruire une plus longue sous-suite

La longueur ne suffit pas : on veut le *mot*. C'est là que sert la table `provenance`. On part de la case (n, m) et on remonte vers $(0, 0)$ en suivant les provenances : un pas « diagonal » signale une lettre commune retenue, qu'on ajoute en tête du résultat ; les pas « nord » et « ouest » ne font qu'avancer dans la table sans rien retenir. On s'arrête en atteignant un bord (première ligne ou première colonne), où il n'y a plus de lettre à retenir.

```python
from plsc_table import plsc_table


def plsc(u, v):
    """Renvoie une plus longue sous-suite commune à u et v (le mot lui-même,
    pas seulement sa longueur).

    On calcule d'abord le tableau des provenances, puis on remonte depuis la
    case (len(u), len(v)) : à chaque pas "diag" on a retenu une lettre commune,
    qu'on ajoute en tête du résultat ; les pas "haut" et "gauche" ne font
    qu'avancer dans la table. Coût O(len(u) * len(v)).

    >>> plsc("atgcg", "agcccc")
    'agc'
```

```
    >>> plsc("", "abc")
    ''
    >>> w = plsc("ttatatgcgt", "tatccccta")
    >>> len(w)
    5
    """
    _, provenance = plsc_table(u, v)
    i, j = len(u), len(v)
    w = ""
    while i > 0 and j > 0:
        if provenance[i][j] == "diag":
            w = u[i - 1] + w
            i, j = i - 1, j - 1
        elif provenance[i][j] == "haut":
            i = i - 1
        else:
            j = j - 1
    return w
```

On ajoute chaque lettre *en tête* (`w = u[i-1] + w`) parce qu'on parcourt le mot de la fin vers le début : les lettres sont découvertes dans l'ordre inverse de leur apparition. Sur l'exemple 7.3.1, la remontée produit successivement `c`, puis `gc`, puis `agc`.

Remarque 7.3.2. *On pourrait se passer de la table* `provenance` *et remonter directement dans la table A des longueurs, en redéduisant à chaque case d'où vient sa valeur (en comparant u_{i-1} et v_{j-1}, puis les cases voisines). C'est l'objet de l'exercice 7.3.5 ; on échange alors un peu d'espace contre une comparaison de plus par pas.*

7.3.3 Coût

Proposition 7.3.3. *Le calcul d'une plus longue sous-suite commune à deux mots de longueurs n et m par les fonctions ci-dessus demande $O(n \cdot m)$ opérations.*

Démonstration. Additionnons les coûts des deux phases : le remplissage de la table, puis la reconstruction. Le remplissage des bords coûte $O(n + m)$. Les deux boucles imbriquées de `plsc_table` exécutent leur corps — un nombre constant d'opérations (une comparaison de lettres, une affectation dans chaque table) — exactement $n \times m$ fois, d'où $O(n \cdot m)$. La reconstruction par `plsc` remonte de la case (n, m) vers un bord : à chaque tour de la boucle `while`, la somme $i + j$ diminue strictement d'au moins 1 (un pas diagonal la diminue de 2, les autres de 1), en partant de $n + m$; il y a donc au plus $n + m$ tours, chacun à coût constant, soit $O(n + m)$. Le total est dominé par le remplissage de la table : $O(n \cdot m)$. □

La programmation dynamique a donc ramené un problème à première vue exponentiel (énumérer les 2^n sous-suites, remarque 7.2.4) à un coût quadratique — un gain spectaculaire, et le même schéma servira au problème du sac à dos à la section suivante.

7.3.4 Exercices

Exercice 7.3.4. *Sur* $u =$ `gattaca` *et* $v =$ `tagada`*, remplir le tableau* A *et le tableau des provenances, puis dérouler la reconstruction. Quelle plus longue sous-suite commune obtient-on, et de quelle longueur ?*

Exercice 7.3.5. *Écrire une fonction qui reconstruit une plus longue sous-suite commune en n'utilisant que le tableau* A *des longueurs,* sans *table de provenance : à chaque case* (i, j) *rencontrée lors de la remontée, redéduire le pas à faire en comparant* u_{i-1} *et* v_{j-1} *et, au besoin, les longueurs* $A[i-1][j]$ *et* $A[i][j-1]$*. Vérifier qu'on retrouve bien le même résultat que* `plsc` *sur l'exemple 7.3.1.*

Exercice 7.3.6. *La* distance d'édition *(ou de* Levenshtein*) entre deux mots est le nombre minimal d'insertions, de suppressions et de substitutions de lettres pour transformer l'un en l'autre. Elle se calcule par une programmation dynamique très proche de celle de la plus longue sous-suite commune. En notant* $D[i][j]$ *la distance entre* $u[{:}i]$ *et* $v[{:}j]$*, justifier les conditions de bord* $D[i][0] = i$ *et* $D[0][j] = j$*, puis la récurrence*

$$D[i+1][j+1] = \begin{cases} D[i][j] & \textit{si } u_i = v_j, \\ 1 + \min\big(D[i][j],\ D[i][j+1],\ D[i+1][j]\big) & \textit{sinon,} \end{cases}$$

et la programmer. (On vérifiera que la distance entre `kitten` et `sitting` vaut 3.)

Exercice 7.3.7 (★). *Pour calculer seulement la* longueur *d'une plus longue sous-suite commune (sans la reconstruire), montrer qu'il suffit de garder en mémoire deux lignes consécutives du tableau* A *à la fois, et non le tableau entier. En déduire un algorithme en espace* $O(\min(n, m))$ *(et toujours en temps* $O(n{\cdot}m)$*). Pourquoi cette astuce empêche-t-elle, en revanche, la reconstruction par remontée ?*

7.4 Le problème du sac à dos

Nous avons appliqué la programmation dynamique à un problème sur les mots. Pour en montrer la portée, traitons maintenant un problème d'*optimisation* classique — le sac à dos — qui modélise une foule de situations concrètes : choisir, sous une contrainte de budget ou de place, un ensemble d'objets de valeur totale maximale. Nous y retrouverons exactement la démarche des sections précédentes : une relation de récurrence sur des sous-problèmes, un remplissage de table, et une reconstruction par remontée.

7.4.1 Le problème

On dispose de n objets numérotés de 0 à $n-1$. L'objet i a un *poids* p_i et une *valeur* v_i, tous deux entiers strictement positifs. On dispose d'un sac de *capacité* C (un entier) : c'est le poids total qu'il peut supporter. On cherche à remplir le sac d'un sous-ensemble d'objets dont la somme des poids n'excède pas C et dont la somme des valeurs est maximale.

Formellement, on cherche un sous-ensemble $L \subseteq \{0, 1, \dots, n-1\}$ qui *maximise* la valeur totale

$$\sum_{i \in L} v_i \qquad \text{sous la contrainte de poids} \qquad \sum_{i \in L} p_i \leqslant C.$$

Avertissement 7.4.1. *La contrainte est une* inégalité large $\leqslant C$ *: un sac chargé exactement à sa capacité est autorisé. (Certains énoncés utilisent une inégalité stricte ; nous fixons une fois pour toutes l'inégalité large dans ce cours.)*

Exemple 7.4.2. *Trois objets, de poids* $p = (1, 2, 3)$ *et de valeurs* $v = (6, 10, 12)$*, et un sac de capacité* $C = 5$*. On peut prendre les objets* 0 *et* 2 *(poids* $1 + 3 = 4 \leqslant 5$*, valeur* $6 + 12 = 18$*), ou bien les objets* 1 *et* 2 *(poids* $2 + 3 = 5 \leqslant 5$*, valeur* $10 + 12 = 22$*). Prendre les trois objets pèserait* $6 > 5$ *: interdit. La meilleure valeur réalisable est* 22*, atteinte par* $\{1, 2\}$*.*

Une approche naïve énumérerait les 2^n sous-ensembles d'objets : exponentielle, donc inutilisable. La programmation dynamique va, là encore, ramener le coût à un produit.

7.4.2 Sous-problèmes et récurrence

Comme pour la plus longue sous-suite commune, la clef est de choisir les bons sous-problèmes. On les indexe par *deux* paramètres : le nombre d'objets qu'on s'autorise à utiliser, et la capacité disponible. Pour $0 \leqslant i \leqslant n$ et $0 \leqslant k \leqslant C$, posons

$$T[i][k] = \text{valeur maximale d'un sac de capacité } k \text{ ne contenant que des objets d'indices } < i.$$

Autrement dit, $T[i][k]$ est la solution du sous-problème « premiers i objets, capacité k ». La valeur cherchée est $T[n][C]$. On a clairement $T[0][k] = 0$ (aucun objet disponible) et $T[i][0] = 0$ (aucune capacité).

Proposition 7.4.3 (Récurrence du sac à dos)**.** *Pour* $0 \leqslant i < n$ *et* $0 \leqslant k \leqslant C$ *:*

$$T[i+1][k] = \begin{cases} T[i][k] & \text{si } k < p_i, \\ \max(T[i][k],\ v_i + T[i][k - p_i]) & \text{si } k \geqslant p_i. \end{cases}$$

L'idée est de raisonner sur le *dernier* objet autorisé, l'objet i : soit on le laisse de côté, soit on le prend. S'il est trop lourd ($k < p_i$), on ne peut que le laisser. Sinon, on compare les deux options et on garde la meilleure.

Démonstration. Fixons i et k. Toute solution du sous-problème « premiers $i + 1$ objets, capacité k » — c'est-à-dire tout $L \subseteq \{0, \dots, i\}$ de poids $\leqslant k$ — contient l'objet i, ou ne le contient pas. Nous évaluons séparément la meilleure valeur dans chacun des deux cas ; $T[i + 1][k]$ est le plus grand des deux.

Solutions ne contenant pas l'objet i. Une telle solution est un sous-ensemble de $\{0, \dots, i-1\}$ de poids $\leqslant k$: sa valeur est au plus $T[i][k]$, et cette valeur $T[i][k]$ est atteinte (par une solution optimale du sous-problème à i objets). La meilleure valeur sans l'objet i est donc exactement $T[i][k]$.

Solutions contenant l'objet i. Ce cas n'est possible que si $k \geqslant p_i$ (sinon l'objet seul dépasse déjà la capacité). Une telle solution L s'écrit $L = L' \cup \{i\}$ avec $L' \subseteq \{0, \ldots, i-1\}$ de poids $\leqslant k - p_i$; sa valeur est $v_i + \sum_{j \in L'} v_j$. La valeur de L' est au plus $T[i][k-p_i]$, et cette borne est atteinte ; donc la meilleure valeur contenant l'objet i est $v_i + T[i][k-p_i]$.[3]

Si $k < p_i$, seul le premier cas est possible : $T[i+1][k] = T[i][k]$. Si $k \geqslant p_i$, $T[i+1][k]$ est le maximum des deux valeurs, $\max(T[i][k],\, v_i + T[i][k-p_i])$. □

7.4.3 Algorithme et coût

La récurrence se programme par remplissage de table, ligne par ligne (indice d'objet croissant), par capacités croissantes : le calcul de $T[i+1][k]$ n'utilise que des cases de la ligne i, déjà calculée.

```python
def sac_a_dos(p, v, C):
    """Table de programmation dynamique du problème du sac à dos.

    Les objets sont numérotés de 0 à n-1 ; l'objet i a pour poids p[i] et
    pour valeur v[i] ; la capacité du sac est l'entier C. T[i][k] est la
    valeur maximale atteignable en n'utilisant que les objets d'indices < i
    sans dépasser le poids total k. La valeur optimale cherchée est T[n][C].
    Coût O(n*C).

    >>> T = sac_a_dos([1, 2, 3], [6, 10, 12], 5)
    >>> T[3][5]
    22
    >>> T[0][5]   # aucun objet disponible
    0
    >>> T[3][0]   # capacité nulle
    0
    """
    n = len(p)
    T = [[0] * (C + 1) for _ in range(n + 1)]
    for i in range(n):
        for k in range(C + 1):
            if k < p[i]:
                T[i + 1][k] = T[i][k]
            else:
                T[i + 1][k] = max(T[i][k], v[i] + T[i][k - p[i]])
    return T
```

Les deux boucles imbriquées exécutent leur corps — un test, une addition, un maximum, une affectation : un nombre constant d'opérations — exactement $n \times (C+1)$ fois. Le

3. Le détail à vérifier : si L' n'était pas optimal pour le sous-problème $(i,\, k-p_i)$, on pourrait le remplacer par un meilleur L'' de poids $\leqslant k - p_i$, et $L'' \cup \{i\}$ aurait une valeur supérieure à celle de L tout en respectant la capacité k — contredisant l'optimalité de L.

calcul de la valeur optimale $T[n][C]$ demande donc

$$O(n \cdot C)$$

opérations.

7.4.4 Reconstruire une solution

La table donne la valeur optimale, mais pas *quels objets* prendre. On la reconstruit par la même technique qu'à la section précédente : une remontée dans la table. Partons de la case (n, C). À l'étape i, l'objet $i-1$ a-t-il été utilisé ? La réponse se lit dans la table : si $T[i][k] = T[i-1][k]$, c'est qu'on pouvait atteindre la même valeur sans l'objet $i-1$, qui est donc inutile ; si $T[i][k] \neq T[i-1][k]$, c'est que l'objet $i-1$ a strictement amélioré la valeur, donc qu'il fait partie de la solution. Dans ce dernier cas, on le retient et on libère son poids ($k \leftarrow k - p_{i-1}$).

```python
from sac_a_dos import sac_a_dos


def objets_choisis(p, v, C):
    """Reconstruit une collection d'objets réalisant la valeur optimale du
    sac à dos, et non plus seulement cette valeur.

    On calcule la table T, puis on la remonte depuis la case (n, C) : à
    l'étape i, l'objet i-1 a été utilisé si et seulement si le retirer
    changerait la valeur, c'est-à-dire si T[i][k] != T[i-1][k] ; dans ce
    cas on le retient et on libère son poids. Renvoie la liste triée des
    indices choisis.

    >>> objets_choisis([1, 2, 3], [6, 10, 12], 5)
    [1, 2]
    >>> objets_choisis([2, 3, 4, 5], [3, 4, 5, 6], 5)
    [0, 1]
    >>> objets_choisis([4, 5], [1, 1], 3)   # rien ne rentre
    []
    """
    T = sac_a_dos(p, v, C)
    n = len(p)
    choisis = []
    k = C
    for i in range(n, 0, -1):
        if T[i][k] != T[i - 1][k]:
            choisis.append(i - 1)
            k -= p[i - 1]
    choisis.reverse()
    return choisis
```

Le critère est donc « $T[i][k] \neq T[i-1][k]$ » : il compare la case à celle de la *ligne du dessus*, $i-1$. Chaque pas décrémente i de 1 ; la remontée fait n tours, soit un coût $O(n)$, négligeable devant le remplissage de la table.

7.4.5 Un coût trompeur : la pseudo-polynomialité

L'algorithme semble à la fois simple et efficace, et son coût $O(n \cdot C)$ a l'air polynomial. C'est une illusion qu'il faut dissiper, car elle touche à la notion même de taille d'une entrée (chapitre 3).

Remarque 7.4.4 (Pourquoi $O(n \cdot C)$ n'est pas polynomial)**.** *La* taille *de l'entrée, c'est le nombre de symboles nécessaires pour l'écrire. Or un entier C s'écrit avec environ* $\log_2 C$ *chiffres binaires — pas C symboles ! Écrire la capacité $C = 10^9$, par exemple, ne demande qu'une trentaine de bits. Le coût $O(n \cdot C)$ est donc* exponentiel *en la taille de C : doubler le nombre de bits de C, c'est élever C au carré, et donc faire exploser le temps de calcul. On dit qu'un tel algorithme est* pseudo-polynomial *: polynomial en la* valeur *des nombres en entrée, mais pas en leur* taille*.*

Ce n'est pas un défaut de notre algorithme en particulier : on ne connaît, à ce jour, aucun algorithme résolvant le sac à dos en temps polynomial en la taille de l'entrée. Le sac à dos fait partie des problèmes dits *NP-complets*, au cœur de la fameuse question ouverte « P = NP ? », l'un des grands problèmes des mathématiques contemporaines. Pour les entrées où C reste modéré, la programmation dynamique fournit néanmoins une excellente solution pratique.

7.4.6 Exercices

Exercice 7.4.5. *Reprendre l'exemple 7.4.2 ($p = (1, 2, 3)$, $v = (6, 10, 12)$, $C = 5$). Remplir entièrement la table T (de taille 4×6), vérifier que $T[3][5] = 22$, puis dérouler la reconstruction et retrouver l'ensemble $\{1, 2\}$.*

Exercice 7.4.6. *On veut décider si une liste d'entiers strictement positifs admet un sous-ensemble de somme* exactement *égale à une cible S donnée (problème de la* somme de sous-ensemble*). Montrer que c'est le cas particulier du sac à dos où chaque objet a une valeur égale à son poids et où l'on cherche à atteindre exactement S. Écrire la fonction correspondante, qui renvoie un booléen, en temps $O(n \cdot S)$.*

Exercice 7.4.7 (★)**.** *Dans la variante du* sac à dos non borné*, chaque objet est disponible en quantité illimitée : on peut en mettre plusieurs exemplaires. Adapter la récurrence de la proposition 7.4.3 (indication : dans la branche « on prend l'objet i », on se réautorise à le reprendre, donc on lit la ligne $i+1$ et non la ligne i), puis programmer la fonction correspondante.*

Exercice 7.4.8. *Expliquer, en s'appuyant sur la remarque 7.4.4, pourquoi un sac à dos de capacité $C = 2^{50}$ avec seulement $n = 10$ objets met en échec l'algorithme, alors qu'un sac de capacité $C = 1000$ avec $n = 10^4$ objets se traite instantanément. Lequel des deux a la plus grande* taille *d'entrée ?*

8 Algorithmes sur les graphes

Résumé

Ce chapitre est consacré aux *graphes*, l'une des structures les plus universelles de l'informatique : tout réseau — routier, social, de dépendances entre tâches — s'y modélise. Après en avoir posé le vocabulaire et les deux représentations machine usuelles (matrice et listes d'adjacence), nous étudions les deux grands *parcours* qui permettent de les explorer systématiquement — en profondeur et en largeur — puis nous les mettons au travail : tester la connexité, détecter un circuit, et ordonner les tâches d'un projet par tri topologique.

Prérequis

Les notations de complexité O, Θ (chapitre 3) ; les invariants de boucle (chapitre 2) ; le raisonnement par récurrence (chapitre 5) ; les listes et leur manipulation en Python.

Objectifs

À l'issue de ce chapitre, vous saurez modéliser une situation par un graphe, choisir entre matrice et listes d'adjacence selon la densité, écrire et prouver correct un parcours en profondeur ou en largeur, et vous en servir comme brique de base pour la connexité, la détection de circuit et le tri topologique.

Une carte de métro, un réseau d'amis, les prérequis entre les chapitres d'un cours : dans chacun de ces exemples, des *objets* sont reliés deux à deux par une *relation*. Le métro relie des stations par des lignes ; le réseau social relie des personnes par des amitiés ; le cursus relie des chapitres par des dépendances. Effacez tout le reste — la géographie, les visages, le contenu — et il ne reste qu'un ensemble de points et de liens entre ces points. Cet objet épuré, c'est un *graphe*.

Comment représenter un tel réseau dans une machine, et comment l'explorer ? Voilà les deux questions de ce chapitre. La première est affaire de structure de données ; la seconde, d'algorithmes de *parcours* qui, à partir d'un sommet, visitent de proche en proche tous ceux qu'on peut atteindre. Une fois ces parcours en main, quantité de problèmes concrets — peut-on aller de toute station à toute autre ? un projet contient-il une dépendance circulaire impossible à satisfaire ? — se résolvent presque sans effort.

8.1 Définitions et représentations

8.1.1 Graphes non orientés

Définition 8.1.1 (Graphe non orienté)**.** *Un* graphe simple non orienté *est un couple* $G = (V, E)$ *où* V *est un ensemble fini, l'ensemble des* sommets*, et* E *un ensemble de* paires $\{u, v\}$ *de sommets distincts, l'ensemble des* arêtes*. L'arête* $\{u, v\}$ relie *les sommets* u *et* v*.*

Concrètement, un sommet est un point, une arête est un trait reliant deux points. « Simple » signifie qu'il n'y a ni boucle (arête d'un sommet à lui-même) ni arête multiple entre deux mêmes sommets — c'est la seule notion de graphe que nous utiliserons.

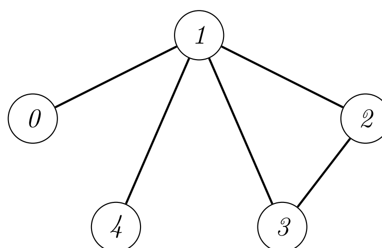
Quand $\{u, v\} \in E$, on dit que les sommets u et v sont *adjacents*, ou *voisins*, et que chacun d'eux est *incident* à l'arête $\{u, v\}$.[1] L'ensemble des voisins de v est son *voisinage*,

$$\Gamma_G(v) = \{\, u \in V : \{u, v\} \in E \,\},$$

et le nombre de ses voisins est son *degré*, noté $d_G(v) = |\Gamma_G(v)|$ (on omet l'indice G quand le graphe est clair).

Exemple 8.1.2. *Voici le graphe* $G = (V, E)$ *avec* $V = \{0, 1, 2, 3, 4\}$ *et*

$$E = \{\{0, 1\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}\}.$$



Le sommet 1 *a pour voisinage* $\Gamma(1) = \{0, 2, 3, 4\}$*, donc* $d(1) = 4$ *; le sommet* 0 *est de degré* 1*. Les degrés sont* $(1, 4, 2, 2, 1)$*.*

Remarque 8.1.3 (La somme des degrés)**.** *Dans l'exemple, la somme des degrés vaut* $1 + 4 + 2 + 2 + 1 = 10$*, soit exactement le double du nombre d'arêtes (*5*). Ce n'est pas un hasard : en sommant les degrés, chaque arête* $\{u, v\}$ *est comptée* deux fois*, une fois pour* u *et une fois pour* v*. Donc, dans tout graphe non orienté,* $\sum_{v \in V} d(v) = 2\,|E|$*. C'est le* lemme des poignées de main *: à une fête, le nombre total de mains serrées est le double du nombre de poignées de main.*

1. On prendra garde à ne pas confondre les deux relations : l'*adjacence* relie deux *sommets*, l'*incidence* relie un sommet et une *arête*. Un sommet n'est jamais « adjacent » à une arête.

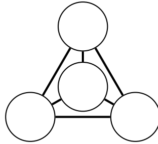
8.1.2 Quelques graphes usuels

Certaines familles de graphes reviennent si souvent qu'elles ont reçu un nom.

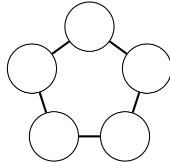
— Le *graphe complet* K_n ($n \geqslant 1$) : n sommets, et *toutes* les arêtes possibles, soit $\binom{n}{2}$ arêtes.

— Le *cycle* C_n ($n \geqslant 3$) : n sommets $0, 1, \ldots, n-1$ reliés en anneau, $\{0, 1\}, \{1, 2\}, \ldots, \{n-1, 0\}$.

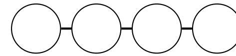
— Le *chemin* P_n ($n \geqslant 1$) : $n+1$ sommets alignés $0, 1, \ldots, n$ reliés par $\{0, 1\}, \ldots, \{n-1, n\}$.



K_4

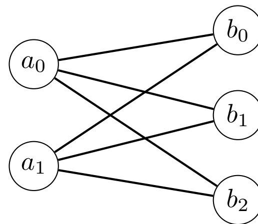


C_5



P_3

Exemple 8.1.4 (Graphe biparti complet)**.** *Pour* $n, p \geqslant 1$*, le* graphe biparti complet $K_{n,p}$ *a deux groupes de sommets,* $\{a_0, \ldots, a_{n-1}\}$ *et* $\{b_0, \ldots, b_{p-1}\}$*, et une arête entre* chaque a_i *et* chaque b_j *— mais aucune à l'intérieur d'un groupe. Voici* $K_{2,3}$ *:*



Un tel graphe modélise par exemple des compatibilités *entre deux types d'objets (candidats et postes, émetteurs et récepteurs).*

8.1.3 Matrice et listes d'adjacence

Pour manipuler un graphe dans une machine, on numérote ses sommets de 0 à $n-1$ (quitte à les renommer), conformément à notre convention d'indexation. Il reste à coder l'ensemble des arêtes. Deux représentations s'imposent.

La matrice d'adjacence. C'est la matrice M de taille $n \times n$ définie par $M[u][v] = 1$ si $\{u, v\} \in E$, et 0 sinon. En Python, c'est une liste de listes. Pour un graphe non orienté, M est *symétrique* ($M[u][v] = M[v][u]$) et nulle sur la diagonale (pas de boucle). Sur le graphe de l'exemple 8.1.2 :

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Les listes d'adjacence. On donne, pour chaque sommet u, la liste de ses voisins. C'est une liste de n listes ; pour le même graphe :

$$G = [\,[1],\ [0,2,3,4],\ [1,3],\ [1,2],\ [1]\,].$$

La case $G[u]$ est la liste des voisins de u : $G[1] = [0,2,3,4]$ dit que le sommet 1 est voisin de $0, 2, 3$ et 4.

Remarque 8.1.5 (Laquelle choisir ?)**.** *La matrice d'adjacence occupe toujours une place $\Theta(n^2)$, quel que soit le nombre d'arêtes. Les listes d'adjacence occupent une place $\Theta(n+m)$, où $m = |E|$: bien plus économique pour un graphe* creux *(peu d'arêtes), ce qui est le cas le plus fréquent en pratique. En revanche, tester si $\{u, v\}$ est une arête est immédiat avec la matrice ($O(1)$), tandis qu'il faut parcourir $G[u]$ avec les listes. Dans tout ce chapitre, sauf mention contraire, les graphes seront donnés par* listes d'adjacence *: c'est la représentation qui rend les parcours efficaces. La conversion d'une représentation à l'autre est l'objet d'un exercice ci-dessous.*

Les fonctions ci-dessous travaillent sur la représentation par listes d'adjacence : `degres` calcule la liste des degrés d'un graphe non orienté, et `listes_vers_matrice`, `matrice_vers_listes` convertissent une représentation en l'autre.

```
def degres(G):
    """Renvoie la liste des degrés des sommets d'un graphe non orienté
    donné par listes d'adjacence : G[u] est la liste des voisins du sommet u,
    et les sommets sont numérotés de 0 à len(G)-1.

    >>> G = [[1], [0, 2, 3, 4], [1, 3], [1, 2], [1]]
    >>> degres(G)
    [1, 4, 2, 2, 1]
    >>> sum(degres(G)) == 2 * 5   # somme des degrés = 2 fois le nombre d'arêtes
    True
    """
    return [len(voisins) for voisins in G]


def listes_vers_matrice(G):
    """Convertit une représentation par listes d'adjacence en matrice
    d'adjacence (liste de listes de 0 et 1). Vaut aussi bien pour un graphe
    orienté que non orienté.

    >>> listes_vers_matrice([[1], [0, 2], [1]])
    [[0, 1, 0], [1, 0, 1], [0, 1, 0]]
    """
    n = len(G)
    M = [[0] * n for _ in range(n)]
    for u in range(n):
```

```
        for v in G[u]:
            M[u][v] = 1
    return M


def matrice_vers_listes(M):
    """Conversion inverse, de la matrice d'adjacence vers les listes
    d'adjacence.

    >>> matrice_vers_listes([[0, 1, 0], [1, 0, 1], [0, 1, 0]])
    [[1], [0, 2], [1]]
    """
    n = len(M)
    return [[v for v in range(n) if M[u][v]] for u in range(n)]
```

8.1.4 Graphes orientés

Dans un réseau routier à sens uniques, ou dans un graphe de dépendances (« la tâche u doit précéder la tâche v »), les liens ont un *sens*. On parle alors de graphe *orienté*.

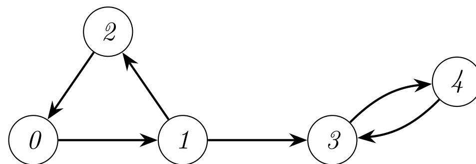
Définition 8.1.6 (Graphe orienté)**.** *Un* graphe orienté *est un couple $G = (V, A)$ où $A \subseteq V \times V$. Un élément $(u, v) \in A$ est un* arc*, allant de u* vers *v ; on le représente par une flèche de u à v.*

Cette fois l'ordre compte : (u, v) et (v, u) sont deux arcs distincts. Quand $(u, v) \in A$, on dit que v est un *successeur* de u. L'ensemble des successeurs de u est son *voisinage sortant*, noté

$$\Gamma^+_G(u) = \{\, v \in V : (u, v) \in A \,\},$$

et le nombre de ses successeurs est son *degré sortant*, noté $d^+_G(u) = |\Gamma^+_G(u)|$.

Exemple 8.1.7. *Le graphe orienté sur $V = \{0, 1, 2, 3, 4\}$ d'arcs $(0, 1), (1, 2), (2, 0), (1, 3), (3, 4), (4, 3)$:*



Ici $\Gamma^+(1) = \{2, 3\}$, donc $d^+(1) = 2$. Les arcs $(3, 4)$ et $(4, 3)$ forment un aller-retour.

On représente un graphe orienté comme un graphe non orienté : par sa matrice d'adjacence (où $M[u][v] = 1$ ssi $(u, v) \in A$, *non* nécessairement symétrique), ou par ses listes d'adjacence (où $G[u]$ est la liste des *successeurs* de u).

8.1.5 Chemins et circuits

Définition 8.1.8 (Chemin, circuit)**.** *Un* chemin *de* u_0 *à* u_p *dans un graphe orienté* $G = (V, A)$ *est une suite de sommets* $(u_0, u_1, \ldots, u_p)$ *telle que* $(u_i, u_{i+1}) \in A$ *pour tout* $0 \leqslant i \leqslant p - 1$ *(chaque pas suit un arc). Le chemin est* élémentaire *si ses sommets sont tous distincts. Un* circuit *est un chemin fermé avec* $p > 0$ *et* $u_0 = u_p$ *; il est* élémentaire *si* $u_0, \ldots, u_{p-1}$ *sont distincts.*

La même définition vaut pour un graphe *non orienté*, en remplaçant l'arc (u_i, u_{i+1}) par l'arête $\{u_i, u_{i+1}\}$; on parle alors de *chaîne* et de *cycle*. Dans l'exemple 8.1.7, $(0, 1, 2, 0)$ est un circuit élémentaire, et $(0, 1, 3, 4, 3)$ est un chemin (mais pas élémentaire, car 3 s'y répète).

8.1.6 Exercices

Exercice 8.1.9. *Écrire la matrice d'adjacence et les listes d'adjacence du cycle* C_4 *(sommets* $0, 1, 2, 3$*). Combien chacune des deux représentations utilise-t-elle de cases ?*

Exercice 8.1.10. *Vérifier que les fonctions* `listes_vers_matrice` *et* `matrice_vers_listes` *ci-dessus sont bien inverses l'une de l'autre sur le graphe de l'exemple 8.1.2. Quelle est la complexité de chacune ?*

Exercice 8.1.11. *Pour un graphe* orienté *donné par listes d'adjacence, écrire une fonction qui calcule la liste des degrés* sortants*, puis une fonction qui calcule la liste des degrés* entrants *(le degré entrant de* v *est le nombre d'arcs* (u, v)*).* Indication : *pour les degrés entrants, parcourir toutes les listes de successeurs.*

Exercice 8.1.12. *Déduire du lemme des poignées de main (remarque 8.1.3) que, dans tout graphe non orienté, le nombre de sommets de degré* impair *est pair.*

8.2 Parcours en profondeur

Nous savons représenter un graphe ; apprenons maintenant à l'*explorer*. Partons d'un sommet v et posons la question la plus naturelle qui soit : quels sommets peut-on atteindre depuis v en suivant les arcs ? Le *parcours en profondeur* y répond. Son principe : on s'enfonce dans le graphe aussi loin que possible le long d'un chemin, et quand on est bloqué, on revient sur ses pas pour explorer une branche encore inexplorée.

8.2.1 L'algorithme

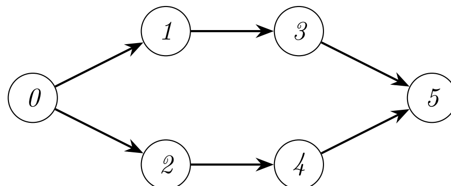
Pour ne pas tourner en rond, on tient à jour un tableau de booléens `visite` indiquant les sommets déjà rencontrés. On gère aussi une *pile* `a_explorer` des sommets découverts dont il reste à examiner les successeurs. Au départ, seul v est visité et empilé. Puis, tant que la pile n'est pas vide, on en retire le dernier sommet u et l'on parcourt ses successeurs : tout successeur encore non visité est marqué et empilé à son tour.

Le choix de retirer le *dernier* sommet empilé (une pile, « dernier entré, premier sorti ») est ce qui donne au parcours son allure « en profondeur » : on poursuit le plus récemment découvert avant de revenir aux autres. On suppose, comme en Python, que `a_explorer.pop()` retire et renvoie le dernier élément, et que `a_explorer.append(w)` ajoute w en fin de liste.

```
def parcours_profondeur(G, depart):
    """Parcours en profondeur d'un graphe orienté G donné par listes
    d'adjacence (G[u] = liste des successeurs de u, sommets 0..len(G)-1).

    Renvoie la liste de booléens visite telle que visite[w] est vrai si et
    seulement s'il existe un chemin de depart à w. On gère explicitement une
    pile a_explorer des sommets découverts mais dont les successeurs restent
    à examiner ; un sommet est marqué visité au moment où il est empilé, ce
    qui garantit qu'il n'est empilé qu'une fois. Coût O(n + m).

    >>> parcours_profondeur([[1, 2], [3], [3], []], 0)
    [True, True, True, True]
    >>> parcours_profondeur([[1, 2], [3], [3], []], 3)
    [False, False, False, True]
    >>> parcours_profondeur([[1], [2], [0], [0]], 1)
    [True, True, True, False]
    """
    n = len(G)
    visite = [False] * n
    visite[depart] = True
    a_explorer = [depart]
    while a_explorer:
        u = a_explorer.pop()
        for w in G[u]:
            if not visite[w]:
                visite[w] = True
                a_explorer.append(w)
    return visite
```

Un sommet est marqué visité *au moment même* où il est empilé : c'est essentiel, car cela garantit qu'un sommet n'est jamais empilé deux fois.

Exemple 8.2.1. *Sur le graphe orienté ci-dessous, lançons le parcours depuis le sommet* 0.



On empile 0*. On le dépile, on visite et empile ses successeurs* 1 *et* 2*. On dépile* 2 *(le dernier empilé), on visite et empile* 4*. On dépile* 4*, on visite et empile* 5*. On dépile* 5 *(aucun successeur), puis* 1*, qui fait visiter* 3 *; on dépile* 3 *(son successeur* 5 *est déjà visité), puis la pile se vide. Tous les sommets ont été atteints :* `visite` *vaut* `[True]*6`*.*

8.2.2 Correction

Nous montrons que l'algorithme *termine* et qu'à la sortie, `visite[w]` est vrai *si et seulement si* il existe un chemin de v à w.

Terminaison. Chaque sommet est marqué visité au plus une fois (le test `non visite[w]` l'empêche de l'être deux fois), et il n'est empilé qu'à cet instant : il y a donc au plus n empilements en tout. Or chaque tour de la boucle `while` dépile un sommet, et un sommet dépilé ne sera jamais réempilé. La boucle effectue donc au plus n tours, puis la pile se vide : l'algorithme termine.

Les sommets visités sont accessibles. C'est le sens facile, capturé par un invariant de boucle.

Invariant 8.2.2 (du parcours en profondeur)**.** *À la fin de chaque tour de la boucle* `while` *:*

(a) tout sommet visité est accessible depuis v (il existe un chemin de v à ce sommet) ;

(b) la pile `a_explorer` *ne contient que des sommets visités.*

Démonstration. Récurrence sur le numéro du tour. *Avant* le premier tour, seul v est visité, et v est accessible depuis lui-même par le chemin (v) ; la pile contient le seul sommet v, qui est visité. Les deux propriétés sont donc vraies initialement.

Hérédité. Supposons (a) et (b) vraies au début d'un tour, qui dépile un sommet u et traite ses successeurs. Le sommet u était dans la pile, donc visité par (b), donc accessible depuis v par (a) : fixons un chemin de v à u. Pour chaque successeur w de u qui était non visité, l'algorithme le marque visité et l'empile. Or w *est* accessible : en prolongeant le chemin de v à u par l'arc (u, w), on obtient un chemin de v à w. Ainsi tout sommet nouvellement visité est accessible, et (a) reste vraie ; et tout sommet nouvellement empilé vient d'être marqué visité, donc (b) reste vraie aussi. □

En particulier, à la sortie de la boucle, tout sommet visité est accessible depuis v.

Les sommets accessibles sont visités. C'est la réciproque, un peu plus délicate. Établissons d'abord une propriété de *clôture* valable à la terminaison.

Lemme 8.2.3. *À la fin de l'algorithme, tout successeur d'un sommet visité est visité.*

Démonstration. Soit u un sommet visité à la fin. Comme u a été marqué visité, il a été empilé au même instant. Or la pile est *vide* à la terminaison : u en a donc été retiré, lors d'un certain tour de boucle qui a examiné tous ses successeurs. À ce moment, chaque

successeur de u a été marqué visité s'il ne l'était pas déjà. Donc tout successeur de u est visité à la fin. □

On en déduit la réciproque. Soit w un sommet accessible depuis v : il existe un chemin $(v = u_0, u_1, \ldots, u_p = w)$. Montrons par récurrence sur i que u_i est visité. Le sommet $u_0 = v$ est visité (dès l'initialisation). Si u_i est visité, alors u_{i+1}, qui est un successeur de u_i, est visité d'après le lemme 8.2.3. Donc $u_p = w$ est visité.

Les sommets visités sont donc *exactement* les sommets accessibles depuis v : l'algorithme est correct.

8.2.3 Complexité

Proposition 8.2.4. *Sur un graphe à n sommets et m arcs donné par listes d'adjacence, le parcours en profondeur s'exécute en temps $O(n+m)$.*

Démonstration. L'initialisation du tableau `visite` coûte $O(n)$. Ensuite, chaque sommet est dépilé au plus une fois ; lorsqu'on dépile u, on parcourt sa liste de successeurs $\Gamma^+(u)$, soit $d^+(u)$ opérations. Le coût total des parcours de listes est donc $\sum_{u \in V} d^+(u) = m$ (chaque arc est examiné une fois, depuis son origine). Au total $O(n+m)$. □

Avec une matrice d'adjacence, en revanche, retrouver les successeurs d'un sommet demande de parcourir toute une ligne, soit $O(n)$ par sommet et $O(n^2)$ en tout : c'est l'une des raisons pour lesquelles on privilégie les listes d'adjacence pour les parcours.

8.2.4 Exercices

Exercice 8.2.5. *Sur le graphe de l'exemple 8.2.1, dérouler le parcours depuis le sommet* 1*, en indiquant à chaque tour le contenu de la pile et le sommet dépilé. Quels sommets sont atteints ?*

Exercice 8.2.6. *Écrire une version* récursive *du parcours en profondeur : une fonction qui marque le sommet courant, puis s'appelle récursivement sur chacun de ses successeurs non encore visités. Vérifier qu'elle calcule le même tableau* `visite` *que la version itérative.*

Exercice 8.2.7. *Adapter l'algorithme pour qu'il renvoie la* liste triée *des sommets accessibles depuis v, plutôt que le tableau de booléens. En déduire une fonction qui teste s'il existe un chemin d'un sommet s à un sommet t donnés.*

Exercice 8.2.8. *Le parcours s'applique tel quel à un graphe* non orienté*, à condition de voir chaque arête $\{u, v\}$ comme la donnée des deux successeurs (de u vers v et de v vers u). Sur le graphe non orienté de l'exemple 8.1.2 (section précédente), quels sommets sont atteints depuis le sommet* 0 *? Que signifie, pour un graphe non orienté, le fait que tous les sommets soient atteints ?*

8.3 Parcours en largeur et distances

Le parcours en profondeur de la section précédente détermine *quels* sommets sont accessibles, mais pas *à quelle distance*. Dans cette section, nous décrivons le *parcours en largeur*, qui calcule pour chaque sommet la longueur du plus court chemin qui y mène depuis la source.

Définition 8.3.1 (Distance dans un graphe)**.** *Dans un graphe orienté G, la* distance *$\delta(s, u)$ d'un sommet s à un sommet u est la longueur (le nombre d'arcs) d'un plus court chemin de s à u. On pose $\delta(s, s) = 0$, et $\delta(s, u) = +\infty$ si u n'est pas accessible depuis s.*

Concrètement, $\delta(s, u)$ est le plus petit nombre d'arcs qu'il faut emprunter pour aller de s à u ; s'il n'existe aucun chemin, cette distance est infinie.

8.3.1 L'algorithme

Le principe d'exploration par couches se réalise avec une *file*, structure « premier entré, premier sorti » (en anglais *FIFO*) : les sommets en sont retirés dans l'ordre où ils y ont été ajoutés. C'est ce qui distingue le parcours en largeur du parcours en profondeur, qui utilisait une *pile* (« dernier entré, premier sorti »). En traitant les sommets dans l'ordre de leur découverte, on épuise une couche entière avant d'attaquer la suivante.

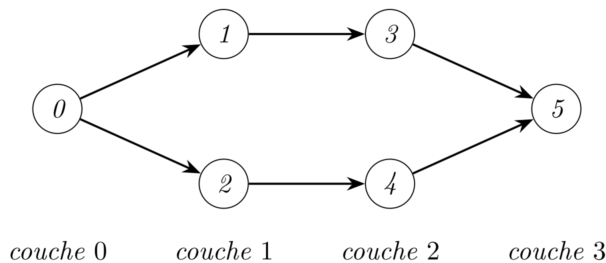
On maintient une liste `distance` où `distance[v]` sera $\delta(s, v)$. La case vaut `None` tant que v n'a pas été découvert : elle joue donc aussi le rôle de marque « déjà visité », ce qui dispense d'un tableau de couleurs séparé. Au départ, seule la source a une distance (0) et entre dans la file. Puis, tant que la file n'est pas vide, on défile un sommet u et l'on examine ses successeurs : tout successeur encore non découvert reçoit la distance `distance`$[u] + 1$ et entre à son tour dans la file.

```
def parcours_largeur(G, source):
    """Parcours en largeur d'un graphe orienté G (listes d'adjacence) depuis
    le sommet source. Renvoie la liste distance telle que distance[v] est la
    longueur du plus court chemin de source à v (nombre d'arcs), ou None si v
    n'est pas accessible.

    On utilise une file (premier entré, premier sorti) réalisée par une liste
    et un indice de tête : file[tete] est le prochain sommet à défiler. La
    case distance[v] joue aussi le rôle de marque « déjà découvert » : v est
    découvert exactement quand sa distance cesse de valoir None. Coût O(n + m).

    >>> parcours_largeur([[1, 2], [3], [3], []], 0)
    [0, 1, 1, 2]
    >>> parcours_largeur([[1, 2], [3], [3], []], 3)
    [None, None, None, 0]
    >>> parcours_largeur([[1], [2], [0], [0]], 1)
    [2, 0, 1, None]
    """
```

```
    n = len(G)
    distance = [None] * n
    distance[source] = 0
    file = [source]
    tete = 0
    while tete < len(file):
        u = file[tete]
        tete += 1
        for v in G[u]:
            if distance[v] is None:
                distance[v] = distance[u] + 1
                file.append(v)
    return distance
```

La file est réalisée par une liste `file` et un indice de tête `tete` : défiler, c'est lire `file[tete]` puis avancer `tete` d'un cran ; enfiler, c'est ajouter en fin de liste. Ces deux opérations sont en $O(1)$.

Exemple 8.3.2. *Parcours en largeur depuis* 0 *sur le graphe ci-dessous. Les sommets se rangent en couches selon leur distance à* 0 *:*



On enfile 0 *(distance* 0*). On le défile, on découvre* 1 *et* 2 *(distance* 1*). On défile* 1*, on découvre* 3 *(distance* 2*) ; on défile* 2*, on découvre* 4 *(distance* 2*). On défile* 3*, on découvre* 5 *(distance* 3*) ; on défile* 4*, dont le successeur* 5 *est déjà découvert ; on défile* 5*. La liste finale est* `distance` $= [0, 1, 1, 2, 2, 3]$.

8.3.2 Correction

Montrons que le parcours en largeur calcule bien les distances : à la fin, $\texttt{distance}[v] = \delta(s, v)$ pour tout v (avec la convention $\texttt{None} = +\infty$). La clef est un invariant *par couches.* Pour $d \in \mathbb{N}$, notons

$$V_d = \{\, u \in V : \delta(s, u) = d \,\}$$

la d-ième couche, et D la plus grande distance finie atteinte. (On admettra que $V_d \neq \emptyset$ pour tout $d \leqslant D$: un plus court chemin vers un sommet de distance D traverse un sommet de chaque distance $0, 1, \ldots, D$; c'est l'objet de l'exercice 8.3.6.)

Invariant 8.3.3 (par couches)**.** *Pour tout d avec $0 \leqslant d \leqslant D$: après avoir défilé tous les sommets des couches $V_0, \ldots, V_{d-1}$ (soit $|V_0| + \cdots + |V_{d-1}|$ tours de boucle), la file contient* exactement *les sommets de la couche V_d, chacun une fois, et* `distance`$[v] = \delta(s, v)$ *pour tout sommet v des couches $V_0, \ldots, V_d$.*

Démonstration. Récurrence sur d.

Initialisation ($d = 0$). Avant tout tour de boucle, la file contient le seul sommet s, et $V_0 = \{s\}$ car $\delta(s, s) = 0$ et aucun autre sommet n'est à distance 0. De plus `distance`$[s] = 0 = \delta(s, s)$. La propriété est vraie.

Hérédité. Supposons-la vraie au rang $d < D$: la file contient alors exactement V_d, et les distances des couches $\leqslant d$ sont correctes. Les $|V_d|$ tours suivants défilent précisément les sommets de V_d (la file les contient tous, et rien d'autre). Examinons ce qu'ils découvrent.

Tout sommet découvert pendant ces tours appartient à V_{d+1}. Soit v un successeur, non encore découvert, d'un sommet $u \in V_d$ qu'on défile. On lui affecte `distance`$[v] =$ `distance`$[u] + 1 = d + 1$. Comme (u, v) est un arc et $\delta(s, u) = d$, on a $\delta(s, v) \leqslant d + 1$. Par ailleurs v n'a pas été découvert lors du traitement des couches $\leqslant d$; par hypothèse de récurrence, ce traitement a fixé la distance de *tous* les sommets des couches $\leqslant d$, donc v n'est dans aucune de ces couches, soit $\delta(s, v) > d$. Ainsi $\delta(s, v) = d + 1$, c'est-à-dire $v \in V_{d+1}$; et sa distance est correctement fixée.

Tout sommet de V_{d+1} est découvert pendant ces tours. Soit $w \in V_{d+1}$. Un plus court chemin de s à w, de longueur $d+1$, s'écrit $(s, \ldots, u, w)$ où u est à distance d, donc $u \in V_d$. Quand on défile u (ce qui arrive pendant ces tours), on examine son successeur w : ou bien w a déjà été découvert par un autre sommet de V_d, ou bien il l'est à cet instant. Dans les deux cas, w a été découvert pendant ces tours, et une seule fois (un sommet n'est enfilé qu'au moment où il est découvert).

Ces $|V_d|$ tours ont donc défilé exactement V_d et enfilé exactement V_{d+1}. Comme la file fonctionne en « premier entré, premier sorti » et que les sommets de V_{d+1} ne sont enfilés qu'*après* ceux de V_d, les $|V_d|$ défilements ne retirent que des sommets de V_d : à la fin, la file contient exactement V_{d+1}, et les distances des couches $\leqslant d + 1$ sont correctes. C'est la propriété au rang $d + 1$. □

Appliqué à $d = D$, l'invariant donne `distance`$[v] = \delta(s, v)$ pour tout sommet accessible. Quant aux sommets inaccessibles, leur distance n'est jamais modifiée et reste `None` $= +\infty = \delta(s, v)$. L'algorithme est donc correct.

8.3.3 Complexité

Proposition 8.3.4 (Coût du parcours en largeur)**.** *Sur un graphe à n sommets et m arcs donné par listes d'adjacence, le parcours en largeur s'exécute en temps $O(n + m)$.*

Démonstration. L'initialisation coûte $O(n)$. Chaque sommet est enfilé au plus une fois (au moment de sa découverte), donc défilé au plus une fois ; lorsqu'on défile u, on parcourt sa liste de successeurs, soit $d^+(u)$ opérations. Le coût total de ces parcours est $\sum_u d^+(u) = m$. Les opérations sur la file étant en $O(1)$, le total est $O(n + m)$. □

Comme pour le parcours en profondeur, une représentation par matrice d'adjacence porterait ce coût à $O(n^2)$.

8.3.4 Exercices

Exercice 8.3.5. *Dérouler le parcours en largeur depuis le sommet* 0 *sur le graphe non orienté de l'exemple 8.1.2 (chaque arête* $\{u, v\}$ *se lit comme deux arcs* $u \to v$ *et* $v \to u$*), en indiquant le contenu de la file à chaque tour et la distance de chaque sommet.*

Exercice 8.3.6. *Montrer que* $V_d \neq \emptyset$ *pour tout* $d \leqslant D$ *(propriété admise dans la preuve de l'invariant 8.3.3).* Indication : *considérer un plus court chemin vers un sommet de distance* D *et regarder ses sommets successifs.*

Exercice 8.3.7. *Modifier l'algorithme pour qu'il mémorise, en plus des distances, le* père *de chaque sommet (le sommet depuis lequel on l'a découvert). En déduire une fonction qui reconstruit un plus court chemin de la source à un sommet cible donné, ou renvoie qu'il n'en existe pas.*

Exercice 8.3.8. *Exhiber un petit graphe et une source pour lesquels le parcours en* profondeur*, en numérotant les sommets dans l'ordre où il les visite, n'attribue* pas *à chaque sommet sa distance à la source. Cela confirme que seul le parcours en largeur calcule les distances.*

8.4 Connexité et composantes connexes

Nous avons construit deux parcours (sections 8.2 et 8.3) ; mettons-les au travail. Dans cette section, nous définissons la *connexité* d'un graphe non orienté et calculons ses *composantes connexes*. Les graphes y sont *non orientés* : une arête $\{u, v\}$ se parcourt dans les deux sens, donc les listes d'adjacence sont symétriques ($v \in G[u]$ équivaut à $u \in G[v]$).

8.4.1 La relation d'accessibilité

Rappelons (section 8.1) qu'une *chaîne* de x à y est une suite de sommets ($x = u_0, u_1, \ldots, u_p = y$) dont chaque paire consécutive $\{u_i, u_{i+1}\}$ est une arête. Posons $x \sim y$ s'il existe une chaîne de x à y.

Proposition 8.4.1. *La relation* $\sim$ *est une relation d'équivalence sur l'ensemble des sommets.*

Autrement dit, $\sim$ est réflexive, symétrique et transitive — les trois propriétés qui permettent de regrouper les sommets en classes.

Démonstration. Vérifions les trois propriétés.

- *Réflexivité.* La suite réduite au seul sommet (x) est une chaîne de x à x (de longueur 0) ; donc $x \sim x$.

— *Symétrie.* Si $x \sim y$ par une chaîne $(x, u_1, \dots, u_{p-1}, y)$, alors la chaîne renversée $(y, u_{p-1}, \dots, u_1, x)$ relie y à x : c'est bien une chaîne, car les arêtes ne sont pas orientées. Donc $y \sim x$.

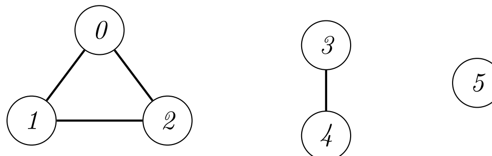
— *Transitivité.* Si $x \sim y$ par une chaîne $(x, \dots, y)$ et $y \sim z$ par une chaîne $(y, \dots, z)$, leur concaténation $(x, \dots, y, \dots, z)$ est une chaîne de x à z. Donc $x \sim z$.

□

Définition 8.4.2 (Connexité, composantes connexes)**.** *Les classes d'équivalence de la relation $\sim$ sont les* composantes connexes *de G. Le graphe est* connexe *s'il n'a qu'une seule composante, c'est-à-dire si deux sommets quelconques sont toujours reliés par une chaîne.*

Concrètement, une composante connexe est un « morceau » du graphe d'un seul tenant : on peut s'y déplacer d'arête en arête entre deux quelconques de ses sommets, mais on n'en sort jamais.

Exemple 8.4.3. *Le graphe non orienté ci-dessous a trois composantes connexes : le triangle $\{0, 1, 2\}$, l'arête $\{3, 4\}$, et le sommet isolé $\{5\}$.*



Le graphe n'est pas connexe ; il le deviendrait si l'on ajoutait, par exemple, les arêtes $\{2, 3\}$ et $\{4, 5\}$.

8.4.2 Calculer les composantes

Comment, en machine, découper le graphe en composantes ? L'observation clef relie les composantes aux parcours : un parcours (en profondeur ou en largeur) lancé depuis un sommet s visite *exactement* les sommets accessibles depuis s, c'est-à-dire — le graphe étant non orienté — exactement la composante de s. Il suffit donc de balayer les sommets : dès qu'on en rencontre un qui n'est encore attribué à aucune composante, on lance un parcours depuis lui pour marquer toute sa composante, et on passe au numéro suivant.

```
def composantes_connexes(G):
    """Calcule les composantes connexes d'un graphe NON orienté G donné par
    listes d'adjacence (les listes sont symétriques : si v est dans G[u],
    alors u est dans G[v]).

    Renvoie la liste composante telle que composante[v] est le numéro de la
    composante de v ; les composantes sont numérotées 0, 1, 2, ... dans
    l'ordre où le balayage des sommets les découvre. On lance un parcours
    depuis chaque sommet pas encore attribué, ce qui marque toute sa
```

```
    composante. Coût O(n + m).

    >>> composantes_connexes([[1], [0], [3], [2], []])
    [0, 0, 1, 1, 2]
    >>> composantes_connexes([[1, 2], [0, 2], [0, 1]])
    [0, 0, 0]
    >>> composantes_connexes([[], [], []])
    [0, 1, 2]
    """
    n = len(G)
    composante = [None] * n
    nb = 0
    for depart in range(n):
        if composante[depart] is None:
            composante[depart] = nb
            file = [depart]
            tete = 0
            while tete < len(file):
                u = file[tete]
                tete += 1
                for v in G[u]:
                    if composante[v] is None:
                        composante[v] = nb
                        file.append(v)
            nb += 1
    return composante


def nombre_composantes(G):
    """Renvoie le nombre de composantes connexes de G. Le graphe est connexe
    si et seulement si ce nombre vaut 1.

    >>> nombre_composantes([[1], [0], [3], [2], []])
    3
    >>> nombre_composantes([[1, 2], [0, 2], [0, 1]])
    1
    """
    composante = composantes_connexes(G)
    if not composante:
        return 0
    return max(composante) + 1
```

La fonction `composantes_connexes` renvoie, pour chaque sommet, le numéro de sa composante ; `nombre_composantes` en déduit leur nombre, et donc un test de connexité (le graphe est connexe ssi ce nombre vaut 1). On a réutilisé ici le parcours en largeur (file

et indice de tête), mais un parcours en profondeur conviendrait tout aussi bien : pour la connexité, seul importe *l'ensemble* des sommets atteints, pas l'ordre dans lequel on les visite.

Proposition 8.4.4. *La fonction* `composantes_connexes` *attribue à deux sommets le même numéro si et seulement s'ils sont dans la même composante connexe ; son coût est* $O(n+m)$.

Démonstration. Correction. Lorsqu'on lance le parcours depuis un sommet `depart` non encore attribué, il marque du numéro courant tous les sommets accessibles depuis `depart`, et eux seuls (correction du parcours, section 8.3). Comme le graphe est non orienté, ces sommets forment exactement la composante de `depart` (proposition 8.4.1). Chaque composante reçoit ainsi un unique numéro, attribué au moment où son premier sommet (dans l'ordre du balayage) est rencontré.

Complexité. La boucle de balayage examine chaque sommet une fois. Les parcours lancés, pris ensemble, visitent chaque sommet une seule fois (un sommet attribué n'est jamais réexploré) et examinent chaque arête un nombre borné de fois : le coût cumulé est $O(n+m)$, comme pour un unique parcours. □

8.4.3 Parcours en largeur ou en profondeur ?

Pour la connexité, les deux parcours sont interchangeables : on l'a dit, seul compte l'ensemble des sommets atteints. Le choix se fait sur d'autres critères. Le parcours en *largeur* fournit en prime les *distances* à la source (section précédente), précieuses si l'on cherche de plus courts chemins. Le parcours en *profondeur*, lui, s'écrit très naturellement de façon récursive, et c'est sur lui que reposeront la détection de circuit et le tri topologique des sections suivantes. On gardera donc les deux dans sa boîte à outils.

8.4.4 Exercices

Exercice 8.4.5. *Sur le graphe de l'exemple 8.4.3, dérouler* `composantes_connexes` *et retrouver le tableau des numéros de composante. Combien de composantes trouve-t-on ?*

Exercice 8.4.6. *Modifier l'algorithme pour qu'il renvoie la* liste des composantes*, chacune donnée comme la liste de ses sommets, plutôt que le tableau des numéros. En déduire une fonction* `est_connexe(G)` *renvoyant un booléen.*

Exercice 8.4.7. *Montrer qu'un graphe non orienté à n sommets et c composantes connexes a au moins $n-c$ arêtes.* Indication : *chaque parcours qui découvre une composante à k sommets emprunte au moins $k-1$ arêtes pour les atteindre.*

Exercice 8.4.8 (★). *Un graphe est* biparti *si l'on peut colorier ses sommets en deux couleurs de sorte qu'aucune arête ne relie deux sommets de même couleur (les graphes $K_{n,p}$ de la section 8.1 en sont l'exemple type). Adapter le parcours en largeur pour tester si un graphe est biparti : colorier la source en* 0*, ses voisins en* 1*, et alterner ; le graphe n'est pas biparti dès qu'une arête joint deux sommets de même couleur. Pourquoi ce critère est-il correct ?*

8.5 Détection d'un circuit

Nous avons étudié les graphes non orientés (section 8.4) ; passons aux graphes *orientés*. Dans cette section, nous donnons un algorithme qui détecte si un graphe orienté contient un *circuit* — la question décisive, par exemple, pour savoir si un système de dépendances entre tâches est réalisable.

8.5.1 Circuits et circuits élémentaires

Commençons par une simplification utile : pour chercher un circuit, on peut se contenter de chercher un circuit *élémentaire* (sans sommet répété).

Proposition 8.5.1 (Un circuit en cache toujours un élémentaire)**.** *Un graphe orienté possède un circuit si et seulement s'il possède un circuit élémentaire.*

Démonstration. Un circuit élémentaire *est* un circuit, ce qui donne le sens de droite à gauche. Réciproquement, supposons que G possède un circuit, et choisissons-en un de longueur *minimale*, disons $(u_0, u_1, \ldots, u_p = u_0)$. S'il n'était pas élémentaire, deux de ses sommets coïncideraient, $u_i = u_j$ avec $i < j < p$; mais alors $(u_i, u_{i+1}, \ldots, u_j = u_i)$ serait un circuit strictement plus court, ce qui contredit la minimalité. Le circuit choisi est donc élémentaire. □

8.5.2 L'algorithme : retirer les puits

Appelons *puits* un sommet de degré sortant nul (aucun arc n'en part). L'idée de l'algorithme est qu'*un puits ne peut appartenir à aucun circuit* : un circuit qui le traverserait devrait en ressortir par un arc, qu'il n'a pas. On peut donc le retirer sans rien changer à la présence de circuits, puis recommencer. L'algorithme s'écrit ainsi :

— tant qu'il existe un puits (sommet de degré sortant nul), le retirer du graphe (avec les arcs qui y arrivent) ;

— à la fin, si le graphe est vide, renvoyer « pas de circuit » ; sinon, renvoyer « circuit ».

Tout repose sur deux lemmes.

Lemme 8.5.2 (Tout degré sortant non nul force un circuit)**.** *Si tout sommet d'un graphe orienté G a un degré sortant strictement positif ($d^+(u) > 0$ pour tout u), alors G possède un circuit.*

Démonstration. Partons d'un sommet quelconque v_0. Comme $d^+(v_0) > 0$, il a un successeur v_1 ; comme $d^+(v_1) > 0$, v_1 a un successeur v_2 ; et ainsi de suite, on construit un chemin $v_0, v_1, v_2, \ldots$ qui ne s'arrête jamais. Or le graphe n'a qu'un nombre fini de sommets : il existe donc un premier indice q tel que v_q a déjà été rencontré, soit $v_q = v_p$ pour un certain $p < q$. La portion $(v_p, v_{p+1}, \ldots, v_q = v_p)$ est alors un circuit de G. □

Lemme 8.5.3 (Retirer un puits préserve l'acyclicité)**.** *Soit v un puits de G (donc $d^+(v) = 0$). Alors G est sans circuit si et seulement si $G \setminus \{v\}$ est sans circuit.*

Démonstration. Si $G \setminus \{v\}$ possède un circuit, ce même circuit est dans G. Pour la réciproque, supposons que G possède un circuit C. Ce circuit ne peut pas passer par v : s'il le traversait, il en sortirait par un arc (v, v'), ce qui donnerait $d^+(v) > 0$, contradiction. Donc C évite v : c'est aussi un circuit de $G \setminus \{v\}$. □

Proposition 8.5.4 (Correction de l'algorithme)**.** *L'algorithme de retrait des puits répond correctement à la question « G possède-t-il un circuit ? ».*

Démonstration. Démontrons la correction par récurrence sur le nombre n de sommets de G. Si $n = 0$, le graphe vide est sans circuit, et l'algorithme renvoie correctement « pas de circuit ». Supposons $n > 0$ et le résultat acquis pour tout graphe à $n - 1$ sommets. Deux cas selon l'existence d'un puits.

— *Aucun puits :* tout sommet vérifie $d^+(u) > 0$. Alors G a un circuit d'après le lemme 8.5.2, et l'algorithme s'arrête immédiatement en renvoyant « circuit » : c'est correct.

— *Il existe un puits v :* l'algorithme le retire et poursuit sur $G \setminus \{v\}$, qui a $n - 1$ sommets. Par hypothèse de récurrence, l'algorithme répond correctement pour $G \setminus \{v\}$. Or, d'après le lemme 8.5.3, G est sans circuit si et seulement si $G \setminus \{v\}$ l'est : la réponse obtenue vaut donc aussi pour G.

□

8.5.3 Mise en œuvre

Plutôt que de retirer effectivement des sommets, il est plus simple de les *marquer* : un sommet marqué est considéré comme retiré. Un sommet peut être marqué dès que tous ses successeurs le sont déjà — c'est la condition « être un puits du graphe résiduel ». On répète tant qu'un tel sommet existe.

Avertissement 8.5.5. *Le sommet que l'on marque à chaque tour doit être choisi* parmi les sommets non encore marqués*. Sans cette précaution, on pourrait re-sélectionner indéfiniment un sommet déjà marqué (dont les successeurs restent marqués) et la boucle ne terminerait jamais.*

```
def a_un_circuit(G):
    """Teste si le graphe orienté G (listes d'adjacence, G[u] = successeurs
    de u) possède un circuit.

    On retire répétitivement un sommet dont tous les successeurs sont déjà
    retirés - un puits du graphe résiduel, qui ne peut appartenir à aucun
    circuit restant. S'il reste des sommets non retirés quand plus aucun
    retrait n'est possible, c'est que le graphe a un circuit. Coût O(n(n+m)).

    >>> a_un_circuit([[1, 2], [2], []])      # graphe sans circuit
    False
    >>> a_un_circuit([[1], [2], [0]])        # circuit 0 -> 1 -> 2 -> 0
```

```
    True
    >>> a_un_circuit([[1], [2], [0], [0]])  # circuit + un sommet pendant
    True
    >>> a_un_circuit([[], []])              # deux sommets isolés
    False
    """
    n = len(G)
    retire = [False] * n
    nb_retires = 0
    progres = True
    while progres:
        progres = False
        for u in range(n):
            if not retire[u] and all(retire[v] for v in G[u]):
                retire[u] = True
                nb_retires += 1
                progres = True
    return nb_retires < n
```

À chaque tour de la boucle externe, on tente de marquer un sommet supplémentaire. Si un tour complet ne marque rien (`progres` reste faux), c'est qu'aucun puits résiduel n'existe : les sommets restants ont tous un successeur non marqué, donc forment un sous-graphe où tout degré sortant est > 0 — un circuit, par le lemme 8.5.2. La fonction renvoie alors `True` car il reste des sommets non retirés.

Exemple 8.5.6. *Sur le graphe de gauche (sans circuit), on marque successivement* 2 *(puits), puis* 1*, puis* 0 *: tous les sommets sont marqués, donc pas de circuit. Sur le graphe de droite, les sommets* $0, 1, 2$ *forment un circuit : aucun n'est jamais un puits résiduel, et l'algorithme conclut à la présence d'un circuit.*

8.5.4 Complexité

À chaque tour de la boucle externe qui progresse, on marque au moins un sommet ; il y a donc au plus n tours utiles (plus un dernier tour sans progrès). Chaque tour parcourt tous les sommets et, pour chacun, ses successeurs, soit $O(n+m)$ opérations. La complexité totale est donc $O\big(n(n+m)\big)$.

Remarque 8.5.7. *On peut faire mieux. L'exercice 8.5.10 propose une détection de circuit en* un seul *parcours en profondeur, en temps* $O(n+m)$ *: on colorie les sommets en blanc / gris / noir, et l'on détecte un circuit dès qu'on rencontre un arc vers un sommet « gris » (en cours d'exploration).*

8.5.5 Exercices

Exercice 8.5.8. *Sur les deux graphes de l'exemple 8.5.6, dérouler l'algorithme en indiquant à chaque tour quel sommet est marqué (ou qu'aucun ne l'est).*

Exercice 8.5.9. *Sur le graphe orienté d'arcs* $(0, 1), (1, 2), (2, 0), (2, 3), (3, 1)$*, exhiber un circuit* non *élémentaire, puis un circuit élémentaire. Combien ce graphe a-t-il de circuits élémentaires ?*

Exercice 8.5.10. *Écrire une détection de circuit par un* unique *parcours en profondeur récursif, en coloriant chaque sommet en blanc (pas encore vu), gris (en cours d'exploration) ou noir (terminé) : il y a un circuit si et seulement si l'exploration rencontre un arc vers un sommet gris. Vérifier le résultat contre la fonction* `a_un_circuit` *sur quelques graphes, et justifier le critère « arc vers un sommet gris ».*

Exercice 8.5.11. *Pourquoi la notion de « puits » et l'algorithme ci-dessus ne s'appliquent-ils pas tels quels à la recherche d'un* cycle *dans un graphe* non orienté *?* (On pourra examiner ce que devient un sommet de degré 1.)

8.6 Tri topologique

Terminons ce chapitre par une application directe de la section précédente. Reprenons un graphe de dépendances entre tâches : un arc (u, v) signifie « la tâche u doit être faite avant la tâche v ». On souhaite ordonner toutes les tâches en une seule liste compatible avec ces contraintes — un calendrier où chaque tâche figure après toutes celles dont elle dépend. C'est ce qu'on appelle un *tri topologique*.

Définition 8.6.1 (Tri topologique)**.** *Soit* $G = (V, A)$ *un graphe orienté à* n *sommets. Un* tri topologique *de* G *est une numérotation des sommets par une bijection* $\pi: V \to \{0, 1, \dots, n-1\}$ *telle que*

$$(u, v) \in A \implies \pi(u) < \pi(v).$$

Concrètement, si l'on dispose les sommets sur une ligne par ordre de π croissant, *tous les arcs vont de la gauche vers la droite.* Aucun arc ne « revient en arrière ».

Proposition 8.6.2 (Un graphe avec circuit n'a pas de tri topologique)**.** *Si* G *possède un circuit, il n'admet aucun tri topologique.*

Démonstration. Supposons que G admette un tri topologique π et un circuit $(u_0, u_1, \dots, u_p = u_0)$. Chaque arc (u_i, u_{i+1}) impose $\pi(u_i) < \pi(u_{i+1})$; en enchaînant le long du circuit, on obtient

$$\pi(u_0) < \pi(u_1) < \cdots < \pi(u_p) = \pi(u_0),$$

soit $\pi(u_0) < \pi(u_0)$, ce qui est absurde. Un graphe muni d'un tri topologique est donc nécessairement sans circuit. □

Nous allons voir que, réciproquement, *tout* graphe sans circuit admet un tri topologique — et que l'algorithme de la section précédente, à peine modifié, en construit un.

8.6.1 L'algorithme

L'idée reprend celle du retrait des puits. Rappelons qu'un graphe sans circuit possède toujours un puits (un sommet de degré sortant nul) : c'est la contraposée du lemme 8.5.2, qui affirmait que sans puits il y aurait un circuit. Or un puits ne dépend d'aucun autre sommet pour le suivre : on peut le placer *en dernier*. On lui attribue donc la plus grande position, on le retire, et l'on recommence sur le graphe restant, qui est encore sans circuit (lemme 8.5.3).

```python
def tri_topologique(G):
    """Renvoie un tri topologique d'un graphe orienté sans circuit G (listes
    d'adjacence) : une liste ordre des sommets telle que tout arc (u, v) va
    d'un sommet à un sommet situé plus loin dans la liste, autrement dit
    ordre.index(u) < ordre.index(v).

    On remplit les positions de la fin vers le début : à chaque étape on
    choisit un puits du graphe résiduel (un sommet dont tous les successeurs
    sont déjà placés), on lui donne la plus grande position libre, et on le
    retire. Lève une ValueError si G possède un circuit (aucun puits à un
    moment où il reste des sommets). Coût O(n(n+m)).

    >>> tri_topologique([[1, 2], [2], []])
    [0, 1, 2]
    >>> tri_topologique([[], [0], [0, 1]])
    [2, 1, 0]
    >>> tri_topologique([[1], [0]])
    Traceback (most recent call last):
        ...
    ValueError: le graphe possède un circuit
    """
    n = len(G)
    retire = [False] * n
    ordre = [None] * n
    for position in range(n - 1, -1, -1):
        choisi = None
        for u in range(n):
            if not retire[u] and all(retire[w] for w in G[u]):
                choisi = u
                break
        if choisi is None:
            raise ValueError("le graphe possède un circuit")
        ordre[position] = choisi
        retire[choisi] = True
    return ordre
```

La fonction renvoie la liste `ordre`, où `ordre[k]` est le sommet placé en position k ; on

remplit les positions de la dernière vers la première. À chaque étape, on cherche un puits du graphe résiduel — un sommet non retiré dont tous les successeurs sont déjà retirés. S'il n'en existe pas alors qu'il reste des sommets, c'est que le graphe a un circuit : la fonction lève alors une `ValueError`.

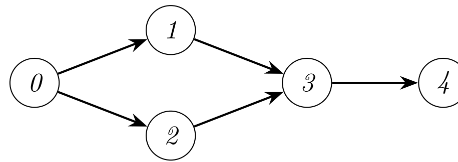
Proposition 8.6.3 (Correction du tri topologique)**.** *Sur un graphe sans circuit, la fonction* `tri_topologique` *renvoie une liste* `ordre` *qui est un tri topologique : pour tout arc* (u, v)*, le sommet* u *apparaît avant* v *dans* `ordre`*.*

Démonstration. Comme le graphe est sans circuit, à chaque étape le graphe résiduel — encore sans circuit par le lemme 8.5.3 — possède un puits (lemme 8.5.2) : l'algorithme trouve donc toujours un sommet à placer, et remplit bien les n positions.

Soit maintenant un arc (u, v). Le sommet u a v pour successeur ; tant que v n'est pas retiré, u n'est pas un puits du graphe résiduel et ne peut donc pas être choisi. Ainsi v est retiré (donc placé) *avant* u. Or l'algorithme remplit les positions de la plus grande vers la plus petite : être retiré plus tôt, c'est recevoir une position plus grande. Donc la position de v est supérieure à celle de u, c'est-à-dire que u apparaît avant v dans `ordre`. C'est bien la propriété d'un tri topologique. □

En posant $\pi(u) =$ la position de u dans `ordre`, on obtient une bijection vérifiant la définition 8.6.1 : tout graphe sans circuit admet donc un tri topologique, et l'algorithme en exhibe un.

Exemple 8.6.4. *Sur le graphe ci-dessous (sans circuit), le seul puits est d'abord* 4*, puis* 3*. Restent alors* 0*,* 1 *et* 2 *: les puits sont maintenant* 1 et 2*, et le code retient le plus petit indice, donc* 1*, puis* 2*, et enfin* 0*. On place ces sommets de la fin vers le début, ce qui donne l'ordre* $(0, 2, 1, 3, 4)$*. Tous les arcs y vont bien de la gauche vers la droite.*



Ce tri n'est pas unique : $(0, 1, 2, 3, 4)$ *en est un autre, car* 1 *et* 2 *ne sont liés par aucun arc et peuvent être placés dans n'importe quel ordre relatif.*

8.6.2 Complexité

L'analyse est identique à celle de la détection de circuit : à chaque étape on parcourt au plus tous les sommets et leurs successeurs pour trouver un puits, soit $O(n + m)$, et il y a n étapes. Le coût est donc $O\big(n(n + m)\big)$.

Remarque 8.6.5. *Comme pour la détection de circuit, on peut descendre à* $O(n + m)$*. Une méthode élégante, l'*algorithme de Kahn*, part au contraire des* sources *(sommets de degré* entrant *nul) et les place du début vers la fin, en maintenant à jour les degrés entrants ; elle détecte du même coup un éventuel circuit. C'est l'objet de l'exercice 8.6.8.*

8.6.3 Exercices

Exercice 8.6.6. *Donner un tri topologique du graphe orienté d'arcs* $(0,1),(0,3),(1,2),(3,2),(2,4),(3,4)$*. Y en a-t-il plusieurs ? En proposer deux différents.*

Exercice 8.6.7. *Montrer qu'un graphe sans circuit admet un* unique *tri topologique si et seulement si, à chaque étape de l'algorithme, le graphe résiduel possède un* unique *puits.* (On pourra relier cette unicité à l'existence d'un chemin passant par tous les sommets.)

Exercice 8.6.8. *Écrire l'algorithme de Kahn, qui calcule un tri topologique en temps* $O(n+m)$ *: calculer le degré entrant de chaque sommet, placer dans une file les sommets de degré entrant nul, puis défiler un sommet, l'ajouter à l'ordre, et décrémenter le degré entrant de ses successeurs (en enfilant ceux qui tombent à* 0*). Comment l'algorithme détecte-t-il un circuit ?*

Exercice 8.6.9. *Que renvoie (ou que se passe-t-il pour) la fonction* `tri_topologique` *si on l'exécute sur un graphe* possédant *un circuit ? Justifier à partir du code et de la proposition 8.6.2.*

9 Algorithmes de plus court chemin

Résumé

Ce chapitre traite d'un problème omniprésent : trouver le chemin le plus court d'un point à un autre dans un réseau dont les liens portent un *poids* — une distance, une durée, un coût. Nous formalisons d'abord la notion de *chemin pondéré* et de *distance*, puis nous dégageons une opération élémentaire, le *relâchement* d'un arc, dont tous les algorithmes du chapitre ne sont que des ordonnancements astucieux. Nous en déduisons l'algorithme de *Bellman–Ford* (et sa variante accélérée sur les graphes sans circuit), puis l'algorithme de *Dijkstra*, plus rapide mais réservé aux poids positifs. Une dernière section montre comment ces algorithmes résolvent un problème en apparence sans rapport : la faisabilité d'un système de contraintes.

Prérequis

Le vocabulaire des graphes orientés et leurs représentations machine (chapitre 8) ; la notion de distance dans un graphe non pondéré et le parcours en largeur (section 8.3) ; le tri topologique (section 8.6) ; le raisonnement par récurrence (chapitre 5) ; les notations de complexité O (chapitre 3).

Objectifs

À l'issue de ce chapitre, vous saurez calculer la distance et un plus court chemin d'une source à tous les sommets d'un graphe pondéré, choisir l'algorithme adapté selon le signe des poids et la structure du graphe, prouver sa correction à l'aide du lemme de relâchement ou d'un invariant de boucle, et reconnaître un problème de contraintes de différences comme un problème de plus court chemin déguisé.

Lorsqu'un GPS calcule un itinéraire, il ne cherche pas le trajet comportant le moins de routes, mais celui dont la *somme des durées* est minimale : chaque tronçon porte un poids — sa longueur, le temps qu'on met à le parcourir, le péage qu'on y paie — et c'est ce poids cumulé qu'on veut minimiser. Le parcours en largeur du chapitre précédent ne suffit plus : il compte les arcs, pas leur poids, et le chemin le moins peuplé en arcs n'est presque jamais le plus court en distance.

Comment, dans un réseau dont chaque lien porte un poids, trouver le chemin de poids total minimal entre deux points ? C'est la question de ce chapitre. Nous allons voir qu'une seule idée — améliorer petit à petit une estimation de la distance, arc par arc — suffit à tout résoudre, et que la difficulté n'est pas *quoi* faire mais *dans quel ordre* le faire.

9.1 Chemins pondérés et lemme de relâchement

9.1.1 Graphes pondérés, chemins et distances

Définition 9.1.1 (Graphe orienté pondéré)**.** *Un* graphe orienté pondéré *est un graphe orienté $G = (V, A)$ muni d'une fonction de* poids w *qui à chaque arc $(u, v) \in A$ associe un nombre réel $w(u, v)$, appelé le* poids *de l'arc.*

Concrètement, on dessine le poids à côté de chaque flèche. Selon le contexte, ce poids modélise une longueur, une durée, un coût, voire une valeur négative (un gain, une remise) : nous autoriserons les poids de signe quelconque, quitte à voir au fil du chapitre quelles précautions cela impose.

Définition 9.1.2 (Longueur d'un chemin, distance)**.** *La* longueur *(ou* poids*) d'un chemin $(u_0, u_1, \ldots, u_p)$ est la somme des poids des arcs qui le composent :*

$$\sum_{i=0}^{p-1} w(u_i, u_{i+1}).$$

La distance *d'un sommet s à un sommet v, notée $\delta(s, v)$, est la plus petite longueur d'un chemin de s à v. On pose $\delta(s, v) = +\infty$ s'il n'existe aucun chemin de s à v.*

En d'autres termes, parmi tous les chemins menant de s à v, la distance retient celui dont les poids cumulés sont les plus faibles — pas celui qui emprunte le moins d'arcs. Lorsque tous les poids valent 1, la longueur d'un chemin redevient son nombre d'arcs et l'on retrouve exactement la distance du graphe non pondéré (définition 8.3.1, section 8.3) : le présent chapitre généralise donc le parcours en largeur.

Avertissement 9.1.3 (Un circuit de poids négatif rend la distance indéfinie)**.** *La distance n'est pas toujours bien définie ! Si un* circuit de poids strictement négatif *est accessible depuis s et mène à v, on peut le parcourir autant de fois qu'on veut pour faire baisser la longueur sans limite : l'infimum vaut alors $-\infty$ et aucun chemin n'est le plus court. Nous préciserons ce cas à la section 9.3 ; jusque-là, on supposera qu'aucun tel circuit n'existe, de sorte que $\delta(s, v)$ est un vrai nombre (ou $+\infty$).*

9.1.2 Le problème, et les deux tableaux d et p

On se fixe un sommet s, la *source*, et l'on cherche à calculer $\delta(s, v)$ pour *tous* les sommets v du graphe — c'est le problème des *plus courts chemins d'une source unique*. On veut de plus pouvoir reconstruire un chemin de longueur minimale vers chaque sommet, pas seulement connaître sa longueur.

Pour cela, tous les algorithmes du chapitre maintiennent deux tableaux indexés par les sommets :

- un tableau d, où $d[v]$ est la *meilleure longueur connue* d'un chemin de s à v trouvé jusqu'ici ; à la fin, on aura $d[v] = \delta(s, v)$;
- un tableau p, où $p[v]$ est un *prédécesseur* de v sur ce chemin.

Le tableau p encode les chemins « à l'envers » : en posant $u_0 = v$ et $u_{i+1} = p[u_i]$, on remonte de v vers s ; si $u_m = s$, alors $(u_m, u_{m-1}, \dots, u_1, u_0)$ est un plus court chemin de s à v. On *initialise* ces tableaux en posant

$$d[s] = 0, \qquad d[v] = +\infty \text{ pour } v \neq s, \qquad p[v] = \texttt{None} \text{ pour tout } v,$$

ce qui traduit qu'au départ on ne connaît qu'un seul chemin : celui, vide, de s à lui-même.

Convention 9.1.4. *En machine, un graphe pondéré est donné par* listes d'adjacence pondérées *:* `G[u]` *est la liste des couples* $(v, w(u, v))$ *pour les arcs* (u, v) *sortant de* u*. La valeur* $+\infty$ *est représentée par* `float('inf')`*, et le prédécesseur indéfini par* `None`*. Les sommets sont numérotés* $0, \dots, n-1$ *comme au chapitre 8.*

9.1.3 Le relâchement d'un arc

Voici l'unique brique de calcul du chapitre. *Relâcher*[1] l'arc (u, v), c'est se demander : le chemin que je connais vers u, prolongé par l'arc (u, v), est-il plus court que le meilleur chemin connu vers v ? Si oui, on met à jour.

```
def relacher(d, p, u, v, poids):
    """Relâche l'arc (u, v) de poids `poids`.

    Les deux tableaux d et p décrivent les chemins trouvés jusqu'ici depuis la
    source : d[x] est la meilleure longueur connue d'un chemin de la source à x
    (float('inf') si aucun n'a encore été trouvé), et p[x] le prédécesseur de x
    sur ce chemin (None si aucun). Relâcher (u, v) teste si l'on raccourcit le
    chemin vers v en passant par u puis l'arc (u, v) ; si oui, d et p sont mis à
    jour. Renvoie True si d[v] a été amélioré, False sinon.

    >>> d = [0, float('inf'), float('inf')]
    >>> p = [None, None, None]
    >>> relacher(d, p, 0, 1, 5)
    True
    >>> d
    [0, 5, inf]
    >>> p
    [None, 0, None]
    >>> relacher(d, p, 0, 1, 5)    # 5 n'améliore pas la valeur 5 déjà atteinte
    False
    >>> relacher(d, p, 2, 1, 1)    # d[2] vaut +infini : aucun chemin par 2
    False
    """
    if d[u] + poids < d[v]:
```

1. Le terme est consacré ; il évoque une corde tendue entre s et v que chaque amélioration vient un peu détendre. On dit aussi parfois « traiter » l'arc.

```
        d[v] = d[u] + poids
        p[v] = u
        return True
    return False
```

En d'autres termes, relâcher (u, v) remplace $d[v]$ par $d[u] + w(u, v)$ *lorsque cette valeur est plus petite*, et note alors u comme nouveau prédécesseur de v. L'opération ne fait jamais *augmenter* une case de d : elle ne peut que la diminuer ou la laisser inchangée.

Remarque 9.1.5. *Tout au long d'un algorithme qui ne modifie d que par des relâchements, on a en permanence $d[v] \geqslant \delta(s, v)$ pour tout sommet v. En effet, chaque fois qu'on affecte $d[v] = d[u] + w(u, v)$, cette valeur est la longueur d'un* vrai *chemin de s à v (le chemin connu vers u, prolongé par l'arc (u, v)) ; or $d[u]$ est lui-même, par récurrence, la longueur d'un chemin de s à u. Une case de d contient donc toujours la longueur d'un chemin existant, qui ne peut être inférieure à la longueur* minimale *$\delta(s, v)$.*

9.1.4 Le lemme de relâchement

La force du relâchement tient à la propriété suivante, qui sera la clef de correction de *tous* les algorithmes du chapitre. Elle dit qu'il suffit de relâcher les arcs d'un plus court chemin *dans l'ordre* pour que sa longueur soit correctement calculée — peu importe ce qu'on relâche entre-temps.

Lemme 9.1.6 (de relâchement)**.** *Soit $(s = u_0, u_1, \ldots, u_p)$ un plus court chemin de s à u_p. Supposons qu'après l'initialisation, on effectue une suite quelconque de relâchements qui contient,* comme sous-suite *(c'est-à-dire dans cet ordre, mais pas forcément consécutivement), les relâchements des arcs*

$$(u_0, u_1),\ (u_1, u_2),\ \ldots,\ (u_{p-1}, u_p).$$

Alors $d[u_p] = \delta(s, u_p)$ à la fin.

Démonstration. On raisonne par récurrence sur p, le nombre d'arcs du chemin.

Initialisation ($p = 0$). Le chemin est réduit au sommet $s = u_0$, et $d[s] = 0 = \delta(s, s)$ dès l'initialisation. Comme un relâchement ne fait que diminuer une case de d sans jamais passer sous $\delta(s, s)$ (remarque 9.1.5), on a $d[s] = 0$ jusqu'à la fin.

Hérédité. Supposons la propriété acquise pour les chemins à p arcs, et soit $(u_0, \ldots, u_p, u_{p+1})$ un plus court chemin à $p + 1$ arcs. Un sous-chemin d'un plus court chemin est lui-même un plus court chemin[2] ; donc $(u_0, \ldots, u_p)$ est un plus court chemin de s à u_p, et la suite de relâchements contient ceux de ses arcs dans l'ordre. Par hypothèse de récurrence, $d[u_p] = \delta(s, u_p)$ à la fin — et même dès que le relâchement de (u_{p-1}, u_p) a eu lieu, puisque $d[u_p]$ ne peut plus descendre sous $\delta(s, u_p)$.

2. Sinon, en remplaçant le morceau de u_0 à u_p par un chemin plus court, on obtiendrait un chemin de u_0 à u_{p+1} plus court que le minimum : contradiction.

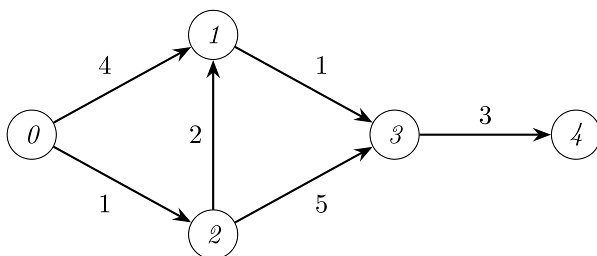
Or, dans la sous-suite, le relâchement de l'arc (u_p, u_{p+1}) vient *après* celui de (u_{p-1}, u_p). À ce moment, on a déjà $d[u_p] = \delta(s, u_p)$. Le relâchement de (u_p, u_{p+1}) donne alors

$$\delta(s, u_{p+1}) \leqslant d[u_{p+1}] \leqslant d[u_p] + w(u_p, u_{p+1}) = \delta(s, u_p) + w(u_p, u_{p+1}) = \delta(s, u_{p+1}),$$

où l'inégalité de gauche est la remarque 9.1.5, celle du milieu vient de l'effet du relâchement, et l'égalité de droite du fait que $(u_0, \ldots, u_{p+1})$ est un plus court chemin. Toutes les inégalités sont donc des égalités : $d[u_{p+1}] = \delta(s, u_{p+1})$. Les relâchements ultérieurs ne modifieront plus cette valeur, déjà minimale. □

Ce lemme déplace toute la difficulté sur une seule question : comment garantir que, pour *chaque* sommet, les arcs d'un de ses plus courts chemins sont relâchés dans l'ordre ? Les sections suivantes y répondent chacune à leur manière — par force brute (Bellman–Ford), en exploitant l'absence de circuit (variante DAG), ou en exploitant la positivité des poids (Dijkstra).

Exemple 9.1.7. *Considérons le graphe pondéré suivant, de source* 0 *:*



Le chemin direct $(0, 1)$ *a pour longueur* 4*, mais le détour* $(0, 2, 1)$ *ne coûte que* $1 + 2 = 3$ *: la distance* $\delta(0, 1)$ *vaut* 3*, pas* 4*. Les distances depuis* 0 *sont* $\delta = [0, 3, 1, 4, 7]$*. Relâcher l'arc* $(0, 2)$ *d'abord, puis* $(2, 1)$*, puis* $(1, 3)$*, puis* $(3, 4)$ *— c'est-à-dire les arcs du plus court chemin vers* 4 *dans l'ordre — amène bien* $d[4]$ *à* 7*, conformément au lemme 9.1.6.*

9.1.5 Exercices

Exercice 9.1.8. *Sur le graphe de l'exemple 9.1.7, on relâche les arcs dans l'ordre malheureux* $(1, 3), (3, 4), (0, 2), (2, 1), (0, 1)$*. Donner le contenu des tableaux* d *et* p *après cette séquence. Quelle case n'a pas encore atteint sa distance, et pourquoi ? Combien de passes sur tous les arcs faut-il pour que tout converge ?*

Exercice 9.1.9. *Montrer qu'à tout instant d'un algorithme procédant par relâchements, en suivant les prédécesseurs* $u_0 = v$*,* $u_{i+1} = p[u_i]$ *depuis un sommet* v *tel que* $d[v] < +\infty$*, on finit par atteindre la source* s*, et le chemin obtenu a pour longueur* $d[v]$*.* Indication : *montrer que chaque relâchement préserve cette propriété.*

Exercice 9.1.10. *On considère le graphe à deux sommets* 0 *et* 1 *avec les arcs* $(0, 1)$ *de poids* 1 *et* $(1, 0)$ *de poids* -3*. Décrire le circuit de poids négatif. Montrer qu'en relâchant indéfiniment les deux arcs en alternance, les valeurs de* $d[0]$ *et* $d[1]$ *tendent vers* $-\infty$*. Que vaut* $\delta(0, 1)$ *?*

Exercice 9.1.11. *Un plus court chemin emprunte-t-il toujours le moins d'arcs possible parmi les chemins de même longueur ? Et réciproquement, le chemin comportant le moins d'arcs est-il toujours le plus court ? Justifier chaque réponse, par une preuve ou un contre-exemple tiré de l'exemple 9.1.7.*

9.2 L'algorithme de Bellman–Ford

Le lemme de relâchement (section 9.1) ramène le calcul des plus courts chemins à une seule exigence : pour chaque sommet, relâcher dans l'ordre les arcs d'un de ses plus courts chemins. Dans cette section, nous y parvenons par la méthode la plus simple qui soit — relâcher *tous* les arcs, et recommencer assez de fois pour être sûr de n'avoir rien oublié.

9.2.1 L'idée : une suite universelle de relâchements

Combien de fois faut-il tout relâcher ? La réponse tient dans une remarque sur la *longueur* des plus courts chemins.

Lemme 9.2.1 (Un plus court chemin est sans répétition)**.** *Si les poids des arcs sont positifs ou nuls, alors entre deux sommets reliés il existe un plus court chemin* élémentaire *(sans sommet répété), donc comportant au plus $n-1$ arcs, où n est le nombre de sommets.*

Démonstration. Partons d'un plus court chemin de s à v. S'il repasse par un sommet x, le morceau compris entre les deux visites de x est un circuit, de longueur $\geqslant 0$ puisque les poids sont positifs ou nuls. En le supprimant, on obtient un chemin de s à v de longueur *au plus* égale, donc encore minimale, et comportant strictement moins d'arcs. En répétant, on aboutit à un plus court chemin sans répétition. Un tel chemin visite chaque sommet au plus une fois : il a donc au plus n sommets, soit au plus $n-1$ arcs. □

Voilà la clef. Si l'on relâche *tous* les arcs du graphe, puis de nouveau tous les arcs, et ainsi de suite $n-1$ fois, alors la suite des relâchements ainsi produite contient, comme sous-suite et dans le bon ordre, les arcs de n'importe quel plus court chemin élémentaire : son premier arc est relâché lors de la première passe (ou avant), son deuxième lors de la deuxième passe (ou avant), etc. Comme un tel chemin a au plus $n-1$ arcs, $n-1$ passes suffisent. Le lemme de relâchement fait le reste.

9.2.2 L'algorithme et sa correction

```python
from relachement import relacher


def bellman_ford_positif(G, s):
    """Plus courts chemins de la source s à tous les sommets, par l'algorithme
    de Bellman-Ford. Le graphe pondéré G est donné par listes d'adjacence
    pondérées : G[u] est la liste des couples (v, poids) des arcs sortant de u.
```

```
    Renvoie le couple (d, p) où d[v] est la distance de s à v (float('inf') si
    v est inaccessible) et p[v] un prédécesseur de v sur un plus court chemin
    (None pour la source et les sommets inaccessibles). On effectue n-1 passes
    de relâchement de tous les arcs ; cette suite contient comme sous-suite les
    arcs de n'importe quel plus court chemin sans répétition de sommet (au plus
    n-1 arcs), pris dans l'ordre. Coût O(n*m).

    >>> G = [[(1, 4), (2, 1)], [(3, 1)], [(1, 2), (3, 5)], [(4, 3)], []]
    >>> d, p = bellman_ford_positif(G, 0)
    >>> d
    [0, 3, 1, 4, 7]
    >>> p
    [None, 2, 0, 1, 3]
    >>> bellman_ford_positif(G, 3)[0]   # depuis 3, seuls 3 et 4 sont atteints
    [inf, inf, inf, 0, 3]
    """
    n = len(G)
    d = [float('inf')] * n
    p = [None] * n
    d[s] = 0
    for _ in range(n - 1):
        for u in range(n):
            for (v, poids) in G[u]:
                relacher(d, p, u, v, poids)
    return d, p
```

Proposition 9.2.2 (Correction de Bellman–Ford, poids positifs)**.** *Si tous les poids sont positifs ou nuls, l'algorithme ci-dessus calcule* $d[v] = \delta(s, v)$ *pour tout sommet* v*.*

Démonstration. Soit v un sommet accessible depuis s (si v ne l'est pas, $d[v]$ n'est jamais relâché et reste $+\infty = \delta(s, v)$). D'après le lemme 9.2.1, il existe un plus court chemin $(s = u_0, u_1, \ldots, u_p = v)$ avec $p \leqslant n - 1$. La boucle principale relâche tous les arcs du graphe $n - 1$ fois de suite ; la suite de relâchements obtenue contient donc, dans l'ordre, le relâchement de (u_0, u_1) lors de la première passe, celui de (u_1, u_2) lors de la deuxième, …, celui de (u_{p-1}, u_p) lors de la p-ième (au plus la $(n - 1)$-ième). Ces relâchements apparaissent dans cet ordre : par le lemme de relâchement 9.1.6, $d[v] = \delta(s, v)$ à la fin. □

Exemple 9.2.3. *Sur le graphe de l'exemple 9.1.7 (source* 0*), suivons d passe par passe. Après l'initialisation,* $d = [0, \infty, \infty, \infty, \infty]$*. La première passe relâche* $(0, 1)$ *et* $(0, 2)$ *— d'où* $d[1] = 4$*,* $d[2] = 1$ *— puis, l'arc* $(2, 1)$ *étant relâché* dans la même passe*,* $d[1]$ *tombe à* $1 + 2 = 3$ *; elle relâche aussi* $(1, 3)$ *et* $(3, 4)$*, donnant* $d[3] = 4$ *et* $d[4] = 7$*. On obtient* $d = [0, 3, 1, 4, 7]$*, déjà correct. Les passes suivantes ne changent plus rien — c'est le signe que tout a convergé.*

9.2.3 Complexité

Proposition 9.2.4 (Coût de Bellman–Ford)**.** *Sur un graphe à n sommets et m arcs donné par listes d'adjacence, l'algorithme s'exécute en temps $O(n(n+m))$.*

Démonstration. L'initialisation coûte $O(n)$. Chacune des $n-1$ passes parcourt les n listes d'adjacence — soit $O(n)$ pour visiter les sommets et $O(m)$ pour parcourir tous les arcs, au total $O(n+m)$ — et n'effectue par arc qu'un relâchement, en $O(1)$. Le coût total est donc $O(n) + (n-1) \cdot O(n+m) = O(n(n+m))$. □

Si le graphe est dense (m de l'ordre de n^2), ce coût est $O(n^3)$; c'est aussi ce qu'on obtient avec une représentation par *matrice* d'adjacence, où chaque passe coûte $O(n^2)$ pour retrouver les arcs. Nous verrons à la section 9.4 que, lorsque les poids sont positifs, on peut faire bien mieux.

9.2.4 Variante : les graphes sans circuit

Lorsque le graphe est un *DAG* (de l'anglais *directed acyclic graph*, graphe orienté sans circuit), on peut se passer des $n-1$ passes : une seule suffit, à condition de relâcher les arcs *dans le bon ordre*. Cet ordre, nous le connaissons déjà : c'est le tri topologique du chapitre 8[3].

```
from relachement import relacher
from tri_topologique import tri_topologique


def plus_courts_chemins_dag(G, s):
    """Plus courts chemins de la source s dans un graphe pondéré SANS CIRCUIT
    (DAG), donné par listes d'adjacence pondérées G[u] = [(v, poids), ...].

    On trie d'abord les sommets dans l'ordre topologique, puis on relâche les
    arcs sortants de chaque sommet en suivant cet ordre, en une seule passe.
    Comme tout arc d'un plus court chemin va d'un sommet à un sommet situé plus
    loin dans l'ordre topologique, les arcs de ce chemin sont relâchés dans le
    bon ordre : le lemme de relâchement garantit la correction. Coût O(n + m),
    meilleur que les O(n*m) passes de Bellman-Ford. Lève ValueError si G a un
    circuit. Renvoie (d, p) comme bellman_ford_positif.

    >>> G = [[(1, 2), (2, 4)], [(2, 1), (3, 7)], [(3, 3)], []]
    >>> d, p = plus_courts_chemins_dag(G, 0)
    >>> d
    [0, 2, 3, 6]
    >>> p
```

3. Rappel : un tri topologique (section 8.6) range les sommets de sorte que tout arc aille d'un sommet à un sommet situé *plus loin* dans l'ordre. Il en existe un si et seulement si le graphe est sans circuit.

```
    [None, 0, 1, 2]
    >>> plus_courts_chemins_dag(G, 1)[0]    # depuis 1 : 0 est inaccessible
    [inf, 0, 1, 4]
    """
    n = len(G)
    successeurs = [[v for (v, _) in G[u]] for u in range(n)]
    ordre = tri_topologique(successeurs)
    d = [float('inf')] * n
    p = [None] * n
    d[s] = 0
    for u in ordre:
        for (v, poids) in G[u]:
            relacher(d, p, u, v, poids)
    return d, p
```

Proposition 9.2.5 (Correction et coût de la variante DAG). *Sur un graphe pondéré sans circuit à n sommets et m arcs, l'algorithme ci-dessus calcule $d[v] = \delta(s, v)$ pour tout v, en temps $O(n + m)$.*

Démonstration. Correction. Soit $(s = u_0, \ldots, u_p = v)$ un plus court chemin. Chacun de ses arcs (u_i, u_{i+1}) va d'un sommet à un sommet plus loin dans l'ordre topologique ; les sommets $u_0, u_1, \ldots, u_p$ apparaissent donc dans cet ordre relatif au fil de la boucle. Quand on traite u_i, on relâche tous ses arcs sortants, en particulier (u_i, u_{i+1}). Les relâchements de $(u_0, u_1), \ldots, (u_{p-1}, u_p)$ ont donc lieu dans l'ordre voulu, et le lemme de relâchement 9.1.6 donne $d[v] = \delta(s, v)$. (Cette fois on n'a pas eu besoin que les poids soient positifs : l'absence de circuit suffit, et le lemme s'applique tel quel.)

Coût. Le tri topologique coûte $O(n + m)$ (ou moins selon l'implémentation, voir section 8.6) ; la boucle de relâchement parcourt une seule fois chaque liste d'adjacence, soit $O(n + m)$. Le total est $O(n + m)$. □

Remarque 9.2.6. *La variante DAG accepte les poids* négatifs *sans aucune modification : un graphe sans circuit n'a, en particulier, pas de circuit de poids négatif, donc les distances restent bien définies (avertissement 9.1.3). C'est tout l'inverse de l'algorithme de Dijkstra de la section 9.4, qui exigera des poids positifs mais tolérera les circuits.*

9.2.5 Exercices

Exercice 9.2.7. *Sur le graphe de l'exemple 9.1.7, on a vu qu'une seule passe suffit à tout converger. Construire un graphe à n sommets et une source pour lesquels Bellman–Ford a réellement besoin des $n - 1$ passes : après k passes, exactement $k + 1$ sommets ont leur distance correcte.* Indication : *un chemin où les arcs sont numérotés à rebours de l'ordre de parcours.*

Exercice 9.2.8. *Améliorer l'algorithme pour qu'il s'arrête dès qu'une passe complète n'a effectué* aucun *relâchement (aucune case de d n'a changé). Justifier que la réponse*

reste correcte, et que sur un graphe où tout converge en k passes, le coût tombe à $O((k+1)(n+m))$.

Exercice 9.2.9. *Adapter la variante DAG pour calculer, dans un graphe sans circuit, le plus* long *chemin d'une source à chaque sommet (utile pour ordonnancer des tâches et trouver le « chemin critique » d'un projet).* Indication : *que devient le relâchement ?*

9.3 Cas général et détection d'un circuit négatif

Jusqu'ici, la correction de Bellman–Ford reposait sur des poids positifs ou nuls (section 9.2) — sauf sur un graphe sans circuit, où la variante DAG tolérait déjà n'importe quel signe (remarque 9.2.6). Dans cette section, nous traitons le cas général : des poids de signe quelconque, sur un graphe quelconque. Deux questions se posent alors. L'algorithme reste-t-il correct ? Et que faire du cas, signalé dès l'avertissement 9.1.3, où un circuit de poids strictement négatif rend la distance indéfinie ?

9.3.1 Les poids négatifs ne gênent pas — à une condition

Reprenons l'argument de correction de Bellman–Ford. Tout reposait sur le fait qu'un plus court chemin est *élémentaire*, donc comporte au plus $n-1$ arcs (lemme 9.2.1) ; mais ce lemme supposait les poids positifs ou nuls. Heureusement, l'hypothèse exacte dont on a besoin est plus faible.

Lemme 9.3.1 (Un plus court chemin reste élémentaire sans circuit négatif)**.** *Supposons qu'aucun circuit de poids strictement négatif ne soit accessible depuis s. Alors, pour tout sommet v accessible depuis s, il existe un plus court chemin de s à v qui est* élémentaire *(sans sommet répété), donc comportant au plus $n-1$ arcs.*

Démonstration. Partons d'un plus court chemin de s à v. S'il repasse par un sommet x, le morceau compris entre les deux visites de x est un circuit, et ce circuit est accessible depuis s ; par hypothèse, son poids est $\geqslant 0$. En le supprimant, on obtient un chemin de s à v de longueur *au plus* égale, donc encore minimale, et comportant strictement moins d'arcs. En répétant, on aboutit à un plus court chemin sans répétition, qui visite chaque sommet au plus une fois : au plus n sommets, soit au plus $n-1$ arcs. □

C'est exactement le lemme 9.2.1, où l'hypothèse « poids $\geqslant 0$ » a été remplacée par la plus faible « aucun circuit négatif accessible » — la seule chose dont la preuve se servait était que le circuit supprimé ne raccourcissait pas le chemin. Du coup, *toute* la preuve de correction de Bellman–Ford (proposition 9.2.2) s'applique telle quelle : les $n-1$ passes de relâchement contiennent les arcs de n'importe quel plus court chemin élémentaire comme sous-suite ordonnée, et le lemme de relâchement 9.1.6 conclut. Autrement dit :

> *Tant qu'aucun circuit de poids strictement négatif n'est accessible depuis la source, les $n-1$ passes de Bellman–Ford calculent les distances, que les poids soient positifs ou non.*

Reste donc un seul danger : le circuit de poids négatif. S'il y en a un d'accessible, la notion même de distance s'effondre (avertissement 9.1.3), et les $n-1$ passes produisent des valeurs de d qui ne veulent rien dire. Le bon réflexe n'est pas de calculer au hasard, mais de *détecter* cette situation et de prévenir l'appelant.

9.3.2 Détecter un circuit négatif

Nous venons de voir que les $n-1$ passes calculent les distances en l'absence de circuit négatif accessible. Reste à *détecter* ce cas, et l'idée est simple : après les $n-1$ passes, on en fait *une de plus*, non pas pour mettre à jour d, mais pour *vérifier*. Si un arc (u, v) peut encore être relâché — c'est-à-dire si $d[u] + w(u, v) < d[v]$ —, c'est qu'aucune valeur stable n'a été atteinte, et l'on verra que cela trahit la présence d'un circuit négatif accessible. Sinon, tout est stabilisé et les distances sont correctes.

```
from relachement import relacher


def bellman_ford(G, s):
    """Bellman-Ford dans le cas général, poids de signe quelconque, avec
    détection d'un circuit de poids négatif accessible depuis la source.

    Le graphe pondéré G est donné par listes d'adjacence pondérées
    G[u] = [(v, poids), ...]. On effectue n-1 passes de relâchement de tous les
    arcs, puis une passe supplémentaire de vérification : si un arc peut encore
    être relâché, c'est qu'un circuit de poids négatif est accessible depuis s.

    Renvoie le triplet (d, p, ok). Si ok vaut True, aucun circuit négatif n'est
    accessible et d, p contiennent les distances et prédécesseurs comme dans le
    cas positif. Si ok vaut False, un circuit négatif est accessible et les
    distances ne sont pas toutes définies (d, p ne sont pas exploitables).
    Coût O(n*(n + m)).

    >>> # poids négatifs sans circuit négatif : distances bien définies
    >>> bellman_ford([[(1, 4), (2, 5)], [], [(1, -3)]], 0)
    ([0, 2, 5], [None, 2, 0], True)
    >>> # circuit 1 -> 2 -> 1 de poids -2, accessible depuis 0
    >>> bellman_ford([[(1, 1)], [(2, -1)], [(1, -1)]], 0)[2]
    False
    >>> # le même circuit existe mais n'est pas accessible depuis la source 0
    >>> bellman_ford([[], [(2, -1)], [(1, -1)]], 0)[2]
    True
    """
    n = len(G)
    d = [float('inf')] * n
    p = [None] * n
```

```
    d[s] = 0
    for _ in range(n - 1):
        for u in range(n):
            for (v, poids) in G[u]:
                relacher(d, p, u, v, poids)
    for u in range(n):
        for (v, poids) in G[u]:
            if d[u] + poids < d[v]:
                return d, p, False
    return d, p, True
```

Proposition 9.3.2 (Correction de Bellman–Ford, cas général)**.** *Soit G un graphe pondéré et s une source. À l'issue de l'algorithme ci-dessus :*

1. *la valeur* `ok` *renvoyée vaut* `False` *si et seulement si un circuit de poids strictement négatif est accessible depuis s ;*
2. *lorsque* `ok` *vaut* `True`*, on a $d[v] = \delta(s, v)$ pour tout sommet v.*

Démonstration. Traitons d'abord le cas où *aucun* circuit de poids strictement négatif n'est accessible depuis s. Par le lemme 9.3.1, l'argument de la proposition 9.2.2 s'applique : après les $n-1$ passes, $d[v] = \delta(s, v)$ pour tout v. Vérifions qu'alors la passe finale ne détecte rien, c'est-à-dire que pour tout arc (u, v) on a $d[u]+w(u, v) \geqslant d[v]$. C'est l'*inégalité triangulaire* : un plus court chemin de s à u, prolongé par l'arc (u, v), est un chemin de s à v, donc

$$\delta(s, v) \leqslant \delta(s, u) + w(u, v), \qquad \text{soit} \qquad d[v] \leqslant d[u] + w(u, v).$$

Aucun arc ne vérifie donc $d[u] + w(u, v) < d[v]$, et l'algorithme renvoie `True`.

Réciproquement, supposons que l'algorithme renvoie `True`, et montrons qu'aucun circuit négatif n'est accessible. Renvoyer `True` signifie qu'à la passe finale,

$$d[v] \leqslant d[u] + w(u, v) \qquad \text{pour tout arc } (u, v). \tag{$*$}$$

Soit $(c_0, c_1, \ldots, c_p = c_0)$ un circuit accessible depuis s quelconque. Tous ses sommets sont accessibles, donc $d[c_i] < +\infty$ pour tout i (chaque $d[c_i]$ a été abaissé à la longueur d'un vrai chemin). En appliquant $(*)$ à chacun des arcs (c_i, c_{i+1}) du circuit et en sommant les p inégalités, les termes $d[c_i]$ se télescopent le long du circuit $(c_p = c_0)$:

$$\sum_{i=0}^{p-1} d[c_{i+1}] \leqslant \sum_{i=0}^{p-1} d[c_i] + \sum_{i=0}^{p-1} w(c_i, c_{i+1}), \qquad \text{d'où} \qquad 0 \leqslant \sum_{i=0}^{p-1} w(c_i, c_{i+1}).$$

Le poids de tout circuit accessible est donc $\geqslant 0$: aucun n'est strictement négatif. Cela établit le point 1 par contraposée, et le point 2 est le calcul de distances obtenu au premier paragraphe. □

Remarque 9.3.3. *La détection ne coûte presque rien : la passe de vérification parcourt une fois de plus les listes d'adjacence, en $O(n + m)$, négligeable devant les $O(n(n + m))$ des $n - 1$ passes principales (proposition 9.2.4). Le coût total reste $O(n(n + m))$.*

Avertissement 9.3.4 (Quand `ok` vaut `False`, d n'a aucun sens)**.** *Il ne faut pas confondre les deux issues. Lorsque `ok` vaut `True`, d et p contiennent les distances et les prédécesseurs, exactement comme dans le cas positif. Mais lorsque `ok` vaut `False`, les valeurs de d sont le résultat d'un calcul interrompu sur un problème mal posé : certaines distances valent $-\infty$ et aucune case de d n'est exploitable. La seule information utile est alors le booléen lui-même : « un circuit négatif est accessible ».*

9.3.3 Exemples

Exemple 9.3.5. *Considérons le graphe à trois sommets de source 0 avec les arcs $(0, 1)$ de poids 4, $(0, 2)$ de poids 5 et $(2, 1)$ de poids -3. Le chemin direct vers 1 coûte 4, mais le détour $(0, 2, 1)$ ne coûte que $5 - 3 = 2$: le poids* négatif *de l'arc $(2, 1)$ rend ce détour avantageux. Il n'y a pourtant aucun circuit, donc aucun circuit négatif : les distances sont bien définies, et l'algorithme renvoie $d = [0, 2, 5]$, $p = [\texttt{None}, 2, 0]$ et $\texttt{ok} = \texttt{True}$.*

Exemple 9.3.6. *Prenons à présent trois sommets $0, 1, 2$ avec les arcs $(0, 1)$ de poids 1, $(1, 2)$ de poids -1 et $(2, 1)$ de poids -1. Le circuit $1 \to 2 \to 1$ pèse $(-1) + (-1) = -2 < 0$, et il est accessible depuis la source 0 : on peut le parcourir indéfiniment pour faire baisser $d[1]$ et $d[2]$ sans limite. La passe de vérification trouve donc toujours un arc à relâcher, et l'algorithme renvoie $\texttt{ok} = \texttt{False}$. Mais si l'on coupe l'arc $(0, 1)$ — laissant la source 0 sans arc sortant —, le même circuit existe toujours mais n'est plus* accessible *: les cases $d[1]$ et $d[2]$ restent $+\infty$, aucun arc n'est relâchable, et l'algorithme renvoie `True`. C'est bien l'*accessibilité *du circuit, et non sa seule existence, qui compte — les trois doctests de `bellman_ford` illustrent ces situations.*

9.3.4 Exercices

Exercice 9.3.7. *L'algorithme se contente de signaler la présence d'un circuit négatif. Modifier la passe de vérification pour qu'elle en* exhibe *un : lorsqu'un arc (u, v) vérifie encore $d[u] + w(u, v) < d[v]$, montrer qu'en remontant les prédécesseurs p à partir de v on tombe nécessairement sur un sommet déjà rencontré, et que le morceau ainsi bouclé est un circuit de poids strictement négatif.*

Exercice 9.3.8. *On a utilisé l'inégalité triangulaire $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$ pour tout arc (u, v), en supposant les distances bien définies. Démontrer soigneusement cette inégalité à partir de la définition 9.1.2, et expliquer pourquoi elle peut être* fausse *lorsqu'un circuit négatif est accessible.*

Exercice 9.3.9. *Pourquoi exactement $n - 1$ passes avant la vérification, et pas n ? Montrer que sur un graphe* sans *circuit négatif, une n-ième passe de relâchement ne modifierait jamais d — c'est une autre façon de voir que la passe de vérification ne détecte rien.* Indication : *les distances sont déjà atteintes après $n - 1$ passes.*

9.4 L'algorithme de Dijkstra

Bellman–Ford traite tous les cas, mais à un prix : $O(n(n+m))$, car il relâche *tous* les arcs $n-1$ fois sans jamais savoir lesquels sont déjà définitifs. Lorsque les poids sont *positifs ou nuls*, on peut être bien plus malin. L'algorithme de Dijkstra [4] fixe les distances *une par une*, dans l'ordre croissant, et ne revient jamais sur une distance acquise — exactement comme le parcours en largeur (section 8.3) explorait les sommets par couches de distance croissante, mais cette fois la « distance » compte les poids, pas les arcs.

9.4.1 L'idée : fixer les sommets par distance croissante

Le parcours en largeur traitait les sommets dans l'ordre du nombre d'arcs qui les sépare de la source. Transposons l'idée aux poids. Colorions chaque sommet :

— *noir* si sa distance est *définitivement* connue ;

— *gris* s'il a déjà été atteint (sa case d est finie) mais pas encore fixé ;

— *blanc* s'il n'a pas encore été atteint ($d = +\infty$).

Au départ, seule la source est grise. À chaque tour, on choisit *le sommet gris de plus petite distance connue*, on le déclare noir — sa distance ne bougera plus — et l'on relâche ses arcs sortants, ce qui peut rendre gris de nouveaux sommets blancs ou améliorer la distance de sommets déjà gris.

Tout repose sur une affirmation : *au moment où on le noircit, le sommet gris de plus petite distance a déjà sa distance finale.* C'est ici, et seulement ici, que la positivité des poids intervient — nous le démontrerons. En machine, on n'a pas besoin d'un tableau de couleurs séparé : « noir » se lit dans un tableau `fixe`, et « gris »/« blanc » se distinguent selon que $d[v]$ est fini ou non.

```
from relachement import relacher


def dijkstra(G, s):
    """Algorithme de Dijkstra : plus courts chemins d'une source s à tous les
    sommets, pour des poids positifs ou nuls.

    Le graphe pondéré G est donné par listes d'adjacence pondérées
    G[u] = [(v, poids), ...]. On maintient un tableau fixe : fixe[v] vaut True
    quand d[v] a atteint sa valeur définitive (sommet « noir »). À chaque tour,
    on fixe le sommet non encore fixé de plus petite distance connue, puis on
    relâche ses arcs sortants. Coût O(n^2).

    Renvoie le couple (d, p) des distances et des prédécesseurs. Un sommet
    inaccessible garde d[v] = float('inf') et p[v] = None.
```

4. Du nom d'Edsger Dijkstra, qui le publia en 1959. C'est l'un des algorithmes les plus utilisés au monde : tout calcul d'itinéraire routier ou de routage réseau en est une variante.

```
    >>> dijkstra([[(1, 4), (2, 1)], [(3, 1)], [(1, 2), (3, 5)], [(4, 3)], []], 0)
    ([0, 3, 1, 4, 7], [None, 2, 0, 1, 3])
    >>> dijkstra([[(1, 2)], [], [(0, 1)]], 0)
    ([0, 2, inf], [None, 0, None])
    """
    n = len(G)
    d = [float('inf')] * n
    p = [None] * n
    d[s] = 0
    fixe = [False] * n
    for _ in range(n):
        u = None
        for v in range(n):
            if not fixe[v] and d[v] != float('inf'):
                if u is None or d[v] < d[u]:
                    u = v
        if u is None:
            break
        fixe[u] = True
        for (v, poids) in G[u]:
            if not fixe[v]:
                relacher(d, p, u, v, poids)
    return d, p
```

La boucle `for v in range(n)` qui cherche le minimum joue le rôle de la file du parcours en largeur : elle extrait, parmi les sommets non encore fixés, celui de plus petite distance. Si ce minimum est $+\infty$ (tous les sommets restants sont blancs), c'est qu'ils sont inaccessibles : on s'arrête.

9.4.2 Correction

La correction se lit sur un invariant de boucle. Notons N l'ensemble des sommets noirs (fixés).

Invariant 9.4.1 (de Dijkstra)**.** *Avant chaque tour de la boucle principale, si N compte k sommets, alors :*

1. *N est constitué des k sommets les plus proches de s (ceux de plus petite distance) ;*
2. *pour tout sommet noir $v \in N$, on a $d[v] = \delta(s, v)$;*
3. *pour tout sommet non noir $v \notin N$, $d[v]$ est la plus petite longueur d'un chemin de s à v dont tous les sommets* intermédiaires *sont noirs ($+\infty$ s'il n'en existe aucun).*

Concrètement : les noirs sont définitivement réglés (point 2) et forment un noyau des plus proches voisins (point 1) ; la case d d'un sommet non encore fixé ne reflète, elle, que les chemins passant par ce noyau (point 3) — elle peut donc encore diminuer quand le

noyau s'agrandit. Un sommet non noir est gris exactement quand le chemin du point 3 existe, c'est-à-dire quand un sommet noir pointe vers lui.

Démonstration. Récurrence sur le nombre de tours déjà effectués. On démarre la récurrence *après* le premier tour : c'est l'initialisation propre, car avant ce tour les arcs de s n'ont pas encore été relâchés et le point 3 ne tient pas.

Initialisation. Au premier tour, le seul sommet de distance finie est s ($d[s] = 0$) : l'algorithme noircit donc s, puis relâche ses arcs sortants. Après ce tour, $N = \{s\}$ ($k = 1$). Le point 1 est vrai (s, à distance 0, est bien le plus proche de lui-même) ; le point 2 aussi, car $d[s] = 0 = \delta(s, s)$ et un relâchement ne peut faire descendre $d[s]$ sous 0 (remarque 9.1.5). Pour le point 3, soit $v \neq s$: un chemin de s à v dont tous les sommets intermédiaires sont dans $N = \{s\}$ ne peut avoir *aucun* sommet intermédiaire (s est une extrémité, pas un intermédiaire), c'est donc l'arc (s, v) s'il existe. Or le relâchement des arcs de s a justement porté $d[v]$ à $w(s, v)$ pour chaque successeur v de s, et l'a laissé à $+\infty$ sinon : c'est exactement le point 3.

Hérédité. Supposons l'invariant vrai avec N de taille $k < n$, et soit u le sommet non noir de plus petite distance $d[u]$ choisi à ce tour (s'il n'y en a pas de fini, tous les sommets restants sont inaccessibles : aucun sommet n'est noirci, N et d ne changent plus, et l'invariant est préservé tel quel). Montrons d'abord le point crucial :

$$d[u] = \delta(s, u).$$

On sait déjà $d[u] \geqslant \delta(s, u)$ (remarque 9.1.5). Pour l'autre sens, prenons un plus court chemin P de s à u et appelons w son *premier* sommet hors de N (il en existe un, car $u \notin N$; on peut avoir $w = u$). Le morceau de P allant de s à w n'a que des sommets intermédiaires noirs ; par le point 3 de l'invariant, $d[w]$ est $\leqslant$ sa longueur. Comme w est non noir et u est le non-noir de plus petite distance, $d[u] \leqslant d[w]$. Enfin, le morceau de P allant de w à u a une longueur $\geqslant 0$, *puisque les poids sont positifs ou nuls.* Comme $\delta(s, u)$ est la longueur de P, en recollant ses deux morceaux :

$$\delta(s, u) = \underbrace{(\text{longueur de } s \text{ à } w)}_{\geqslant d[w] \geqslant d[u]} + \underbrace{(\text{longueur de } w \text{ à } u)}_{\geqslant 0} \geqslant d[u].$$

Donc $d[u] = \delta(s, u)$: le point 2 reste vrai pour le nouveau noir u. Comme $d[u]$ est le plus petit parmi les sommets non noirs, et que tout sommet encore *blanc* a une distance $\delta \geqslant d[u]$ (le même découpage de chemin s'applique, son premier sommet hors de N étant gris donc de distance $\geqslant d[u]$), u est bien le $(k + 1)$-ième plus proche de s : le point 1 est préservé.

Il reste le point 3 pour les sommets restés non noirs après l'ajout de u à N. Soit $v \notin N \cup \{u\}$, et considérons les chemins de s à v dont les sommets intermédiaires sont dans $N \cup \{u\}$. Un tel chemin qui *évite* u a ses intermédiaires dans N : sa longueur était déjà prise en compte dans l'ancien $d[v]$ (point 3 avant ce tour). Un tel chemin qui *passe* par u mais ne se termine pas par l'arc (u, v) atteint v depuis un dernier intermédiaire $x \in N$: sa longueur est $\geqslant \delta(s, x) + w(x, v) = d[x] + w(x, v)$, une valeur déjà comparée à $d[v]$ lorsqu'on a relâché l'arc (x, v) en noircissant x, donc $\geqslant$ l'ancien $d[v]$. Aucun de

ces chemins ne fait mieux que l'ancien $d[v]$. La seule amélioration possible vient donc d'un chemin se terminant par (u, v), de longueur $\delta(s, u) + w(u, v) = d[u] + w(u, v)$ — exactement la valeur que le relâchement de (u, v) vient de comparer à $d[v]$. Après avoir relâché tous les arcs sortants de u, $d[v]$ est donc le minimum sur toutes ces possibilités : le point 3 est rétabli. □

Proposition 9.4.2 (Correction de Dijkstra)**.** *Si tous les poids sont positifs ou nuls, l'algorithme de Dijkstra calcule $d[v] = \delta(s, v)$ pour tout sommet v.*

Démonstration. La boucle tourne jusqu'à ce que tout sommet accessible soit noir (un sommet accessible finit par devenir gris, donc par être choisi). À l'arrêt, le point 2 de l'invariant 9.4.1 donne $d[v] = \delta(s, v)$ pour tout sommet noir, donc pour tout sommet accessible ; les sommets restés blancs sont inaccessibles et gardent $d[v] = +\infty = \delta(s, v)$. □

9.4.3 Complexité

Proposition 9.4.3 (Coût de Dijkstra)**.** *Avec la recherche du minimum par balayage présentée ci-dessus, l'algorithme de Dijkstra s'exécute en temps $O(n^2)$, quelle que soit la représentation du graphe.*

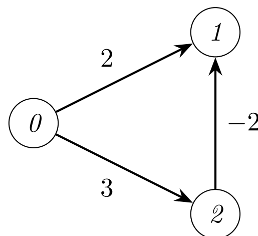
Démonstration. La boucle principale tourne au plus n fois. À chaque tour, la recherche du sommet gris de distance minimale balaie les n sommets, en $O(n)$; le relâchement des arcs sortants du sommet fixé coûte, lui, $O(d^+(u))$. Le balayage domine : le coût d'un tour est $O(n)$, et le coût total $O(n^2)$. (Les relâchements, cumulés sur toute l'exécution, coûtent $\sum_u d^+(u) = O(m)$, absorbés par le $O(n^2)$ puisque $m \leqslant n^2$.) □

Comparons à Bellman–Ford, en $O(n(n + m)) = O(n^2 + nm)$. Sur un graphe *dense* (m de l'ordre de n^2), Dijkstra est en $O(n^2)$ contre $O(n^3)$: un facteur n gagné. Sur un graphe *creux* (m de l'ordre de n), les deux sont en $O(n^2)$; mais on peut alors accélérer la recherche du minimum à l'aide d'une structure de données appelée *tas*, qui abaisse le coût à $O((n + m) \log n)$. Cette amélioration, hors de notre portée pour l'instant, sera possible une fois les tas étudiés (chapitre 11) ; nous nous en tenons ici à la version par balayage, déjà parfaitement utilisable.

9.4.4 Pourquoi les poids doivent être positifs

La positivité n'est pas un détail technique : c'est l'hypothèse sans laquelle l'algorithme *rend de mauvaises réponses.* Voici pourquoi.

Exemple 9.4.4 (Dijkstra échoue sur un poids négatif)**.** *Considérons le graphe à trois sommets de source* 0*, avec les arcs* $(0, 1)$ *de poids* 2*,* $(0, 2)$ *de poids* 3 *et* $(2, 1)$ *de poids* -2.



La vraie distance de 0 *à* 1 *est* 1*, par le détour* $(0,2,1)$ *de poids* $3+(-2)=1$*. Mais Dijkstra noircit* 0*, relâche ses arcs (*$d[1]=2$*,* $d[2]=3$*), puis choisit le gris de plus petite distance : c'est* 1*, avec* $d[1]=2$*. Il* fixe *alors* $d[1]=2$ *et ne reviendra jamais dessus — alors même qu'on n'a pas encore exploré le détour par* 2 *qui aurait donné* 1*. L'algorithme renvoie* $d[1]=2\neq 1=\delta(0,1)$ *: faux ! La faille est exactement à l'endroit pointé par la preuve : le morceau de chemin allant de* w *à* u*, supposé de longueur* $\geqslant 0$*, ne l'est plus, et fixer le minimum gris devient illégitime.*

Pour des poids de signe quelconque, il faut donc revenir à Bellman–Ford (section 9.2). Chaque algorithme a son domaine : Dijkstra exige des poids positifs mais tolère les circuits ; la variante DAG (remarque 9.2.6) tolère les poids négatifs mais exige l'absence de circuit ; Bellman–Ford ne demande ni l'un ni l'autre, et le paie en temps.

9.4.5 Exercices

Exercice 9.4.5. *Dérouler l'algorithme de Dijkstra sur le graphe de l'exemple 9.1.7 (source* 0*) : indiquer, tour après tour, le sommet noirci et l'état du tableau* d*. Vérifier qu'on retrouve* $\delta=[0,3,1,4,7]$*.*

Exercice 9.4.6. *On ne veut souvent que la distance à un seul sommet* cible t*, non à tous. Montrer qu'on peut arrêter l'algorithme dès que* t *devient noir, sans changer la réponse pour* t*.* Indication : *que dit le point 2 de l'invariant au moment où* t *est noirci ?*

Exercice 9.4.7. *Comme pour le parcours en largeur (solution* `chemin_plus_court` *de la section 8.3), le tableau* p *permet de reconstruire un plus court chemin et pas seulement sa longueur. Écrire une fonction qui, à partir de la source* s *et d'une cible, renvoie un plus court chemin de* s *à la cible (liste de sommets), ou* `None` *si la cible est inaccessible.*

9.5 Une application : les contraintes de différences

Terminons le chapitre par une application qui peut surprendre. Bellman–Ford ne sert pas qu'à calculer des plus courts chemins : il résout aussi, sans modification, un problème qui n'a en apparence rien à voir — décider si un système d'inégalités admet une solution. La détection de circuit négatif de la section 9.3 y joue le rôle central.

9.5.1 Le problème

On se donne n variables réelles $x_0, \dots, x_{n-1}$ et un ensemble d'inégalités, chacune de la forme

$$x_j - x_i \leqslant c,$$

où c est une constante réelle (de signe quelconque). De telles inégalités s'appellent des *contraintes de différences*[5] : elles ne portent que sur la *différence* de deux variables. La question est celle de la *faisabilité* : existe-t-il des valeurs des x_i satisfaisant *toutes* les contraintes à la fois ? Quand de telles valeurs existent, on dit que le système est *faisable* (on dit aussi qu'il *admet une solution*) ; sinon, il est *infaisable*.

Exemple 9.5.1. *De telles contraintes apparaissent dès qu'on planifie des événements dans le temps. Si x_i est l'instant où débute la tâche i, la contrainte « la tâche j commence au plus 5 minutes après la tâche i » s'écrit $x_j - x_i \leqslant 5$, et « au moins 3 minutes après » s'écrit $x_i - x_j \leqslant -3$. Décider si un emploi du temps est réalisable, c'est décider la faisabilité d'un système de contraintes de différences.*

9.5.2 La traduction en graphe

À un système S de contraintes de différences sur $x_0, \dots, x_{n-1}$, on associe un graphe pondéré G comme suit :

- ses sommets sont $0, 1, \dots, n-1$ — un par variable — plus un sommet supplémentaire n, qui servira de *source* ;
- chaque contrainte $x_j - x_i \leqslant c$ donne un arc (i, j) de poids c ;
- on ajoute enfin, de la source vers chaque variable, un arc (n, i) de poids 0 (pour $0 \leqslant i \leqslant n-1$).

Les arcs depuis la source garantissent que *tous* les sommets-variables sont accessibles depuis n. Le choix des poids n'est pas un hasard : un arc (i, j) de poids c impose, après calcul des distances, $d[j] \leqslant d[i] + c$, soit exactement $d[j] - d[i] \leqslant c$ — la contrainte de départ, lue sur les distances. C'est tout l'enjeu de la traduction.

```python
from bellman_ford import bellman_ford


def contraintes_differences(n, contraintes):
    """Faisabilité d'un système de contraintes de différences x_j - x_i <= c.

    Les n variables sont x_0, ..., x_{n-1}. La liste contraintes contient des
    triplets (i, j, c) signifiant la contrainte x_j - x_i <= c. On construit un
    graphe pondéré à n+1 sommets : un sommet par variable (0, ..., n-1), plus un
    sommet source n relié à chaque variable par un arc de poids 0. Chaque
```

5. C'est un cas particulier de la *programmation linéaire*, qui cherche à optimiser une quantité linéaire sous des contraintes affines ; ici, on ne cherche pas à optimiser, seulement à savoir si les contraintes sont compatibles.

```
    contrainte (i, j, c) devient un arc (i, j) de poids c. Bellman-Ford depuis la
    source n renvoie ok = True si et seulement si le système est faisable, et
    les distances fournissent alors une solution.

    Renvoie une solution [x_0, ..., x_{n-1}] si le système est satisfiable, ou
    None sinon.

    >>> contraintes_differences(3, [(0, 1, 5), (1, 2, 3), (2, 0, -2)])
    [-2, 0, 0]
    >>> contraintes_differences(2, [(0, 1, 1), (1, 0, -2)]) is None
    True
    """
    source = n
    G = [[] for _ in range(n + 1)]
    for i in range(n):
        G[source].append((i, 0))
    for (i, j, c) in contraintes:
        G[i].append((j, c))
    d, p, ok = bellman_ford(G, source)
    if not ok:
        return None
    return d[:n]
```

9.5.3 Correction

Proposition 9.5.2 (Bellman–Ford décide la faisabilité)**.** *Soit S un système de contraintes de différences sur $x_0, \dots, x_{n-1}$ et G le graphe associé. On applique Bellman–Ford (section 9.3) à G depuis la source n.*

— *Si l'algorithme renvoie `ok` $=$ `True`, le système est faisable, et $(d[0], \dots, d[n-1])$ en est une solution.*

— *S'il renvoie `ok` $=$ `False`, le système n'a aucune solution.*

Démonstration. Cas `True`. Tous les sommets-variables sont accessibles depuis la source n (par les arcs de poids 0), donc $d[i] < +\infty$ pour tout $i \in \{0, \dots, n-1\}$, et par la proposition 9.3.2 ces $d[i]$ sont des distances. Posons $z_i = d[i]$ et vérifions que z satisfait S. Soit une contrainte $x_j - x_i \leqslant c$: par construction, G contient l'arc (i, j) de poids c. Comme l'algorithme a renvoyé `True`, sa passe de vérification a constaté $d[j] \leqslant d[i] + c$ (inégalité $(*)$), c'est-à-dire $z_j - z_i \leqslant c$. Toutes les contraintes sont donc satisfaites : z est une solution.

Cas `False`. Par la proposition 9.3.2, un circuit de poids strictement négatif est accessible depuis n. Ce circuit n'emprunte pas la source : le sommet n n'a aucun arc *entrant*, il ne peut donc figurer sur aucun circuit. Le circuit ne fait intervenir que des sommets-variables, disons $(c_0, c_1, \dots, c_p = c_0)$ avec $c_k \in \{0, \dots, n-1\}$. Chacun de ses arcs (c_k, c_{k+1}), de poids γ_k, provient d'une contrainte $x_{c_{k+1}} - x_{c_k} \leqslant \gamma_k$ de S, et l'on sait que $\sum_{k=0}^{p-1} \gamma_k < 0$.

Or toute solution de S devrait satisfaire la somme de ces p inégalités ; mais leur membre de gauche se télescope le long du circuit ($c_p = c_0$) :

$$0 = \sum_{k=0}^{p-1} (x_{c_{k+1}} - x_{c_k}) \leqslant \sum_{k=0}^{p-1} \gamma_k < 0,$$

ce qui est impossible. Le système S n'a donc aucune solution. □

Remarquons la symétrie avec la section 9.3 : le circuit de poids négatif, qui y signalait l'*absence* de distance bien définie, signale ici l'*absence* de solution. C'est la même obstruction, vue sous deux éclairages.

Exemple 9.5.3. *Prenons trois variables et les contraintes* $x_1 - x_0 \leqslant 5$*,* $x_2 - x_1 \leqslant 3$ *et* $x_0 - x_2 \leqslant -2$*. Le graphe a quatre sommets (*$0, 1, 2$ *et la source* 3*), et le circuit* $0 \to 1 \to 2 \to 0$ *a pour poids* $5 + 3 + (-2) = 6 \geqslant 0$ *: aucun circuit négatif, le système est faisable. L'algorithme renvoie la solution* $(d[0], d[1], d[2]) = (-2, 0, 0)$*, qu'on vérifie sans peine :* $0 - (-2) = 2 \leqslant 5$*,* $0 - 0 = 0 \leqslant 3$*,* $-2 - 0 = -2 \leqslant -2$*. Si l'on remplaçait la dernière contrainte par* $x_0 - x_2 \leqslant -7$*, le circuit pèserait* $5 + 3 - 7 = 1$ *encore positif ; mais avec* $x_0 - x_2 \leqslant -9$*, il pèserait* $-1 < 0$ *et le système deviendrait infaisable.*

9.5.4 Exercices

Exercice 9.5.4. *Trois tâches* $0, 1, 2$ *doivent être planifiées : la tâche* 1 *commence au moins* 2 *unités après la tâche* 0*, la tâche* 2 *au moins* 3 *unités après la tâche* 1*, et la tâche* 2 *doit s'achever — on la suppose instantanée — au plus* 4 *unités après le début de la tâche* 0*. Écrire les contraintes de différences correspondantes, construire le graphe, et déterminer si l'emploi du temps est réalisable.*

Exercice 9.5.5. *Montrer que si* $(x_0, \ldots, x_{n-1})$ *est une solution d'un système de contraintes de différences, alors* $(x_0 + t, \ldots, x_{n-1} + t)$ *en est une aussi, pour toute constante* $t \in \mathbb{R}$*. En déduire qu'un tel système, dès qu'il a une solution, en a une infinité — et expliquer pourquoi cela justifie qu'on puisse arbitrairement « ancrer » les distances à la source.*

Exercice 9.5.6. *Écrire une fonction qui, étant donné une assignation des variables et la liste des contraintes, renvoie* `True` *si l'assignation satisfait toutes les contraintes,* `False` *sinon. L'utiliser pour contrôler la solution renvoyée par* `contraintes_differences` *sur l'exemple 9.5.3.*

10 Algorithmes de recherche de motif

Résumé

Ce chapitre s'attaque à un problème que vous rencontrez chaque fois que vous tapez `Ctrl+F` dans un document : trouver toutes les occurrences d'un petit mot, le *motif*, à l'intérieur d'un grand texte. Nous décrivons d'abord la solution évidente — comparer le motif à chaque position du texte — puis nous l'accélérons de deux façons indépendantes : par *hachage glissant* (l'algorithme de Karp–Rabin), qui rend les comparaisons rapides en moyenne, et par *automate*, qui ne relit jamais deux fois la même lettre du texte. Le fil conducteur est l'analyse du coût : combien de comparaisons de lettres faut-il, et peut-on faire mieux que l'évidence ?

Prérequis

Le vocabulaire des mots sur un alphabet — alphabet Σ, mot, longueur, préfixe, *facteur* (lettres consécutives) opposé à *sous-suite* (avec trous) — introduit au chapitre 7 (section 7.2) ; la convention d'indexation à partir de 0 (convention 1.2.1) ; les notations de complexité O (chapitre 3) ; l'arithmétique modulaire élémentaire (pour Karp–Rabin).

Objectifs

À l'issue de ce chapitre, vous saurez poser proprement le problème de la recherche d'un motif dans un texte, écrire et analyser l'algorithme naïf, et comprendre comment le hachage glissant puis le précalcul d'un automate font chuter le coût des cas favorables d'un facteur égal à la longueur du motif.

Ouvrez n'importe quel traitement de texte, tapez quelques lettres dans la barre de recherche, et l'éditeur surligne instantanément toutes leurs apparitions dans un document de plusieurs centaines de pages. Un moteur de recherche fait de même à l'échelle du web ; un logiciel de bio-informatique cherche un gène — une courte suite de lettres `a`, `t`, `g`, `c` — dans un génome de milliards de bases. Dans tous ces cas, le problème est le même : on dispose d'un long *texte* et d'un court *motif*, et l'on veut savoir où, et combien de fois, le motif apparaît dans le texte.

Comment trouver toutes les occurrences d'un mot dans un texte, le plus vite possible ? La réponse évidente — glisser le motif le long du texte et comparer à chaque position — fonctionne, mais relit beaucoup de lettres inutilement. Nous verrons que deux idées très différentes, le hachage et les automates, permettent de s'en approcher autrement, voire de faire strictement mieux.

10.1 Algorithme naïf de recherche de motif

10.1.1 Le problème

On se place sur un *alphabet* fini Σ — l'ensemble des caractères autorisés — et l'on manipule des *mots* sur Σ, c'est-à-dire des suites finies de lettres. Nous avons déjà rencontré ce vocabulaire au chapitre 7 (section 7.2) ; rappelons-en l'essentiel. La longueur d'un mot w est notée $|w|$, et ses lettres sont numérotées *à partir de* 0, conformément à la convention du cours : $w = w_0 w_1 \cdots w_{k-1}$ pour un mot de longueur k. On note $w[i]$ la lettre d'indice i et $w[i{:}j]$ le *facteur* formé des lettres $w_i, w_{i+1}, \ldots, w_{j-1}$ — exactement la tranche `w[i:j]` de Python.

Remarque 10.1.1. *La recherche de motif porte sur des* facteurs*, pas sur des sous-suites. Un facteur est fait de lettres* consécutives *du texte (`bra` est un facteur de `abracadabra`), alors qu'une sous-suite autorise les trous (`aaa` est une sous-suite de `abracadabra`, mais pas un facteur). C'est la distinction posée à la section 7.2 ; ici, tout est question de facteurs.*

On se donne donc un *texte* $C \in \Sigma^*$ de longueur $n = |C|$ et un *motif* $M \in \Sigma^*$ de longueur $m = |M|$, en général bien plus court ($m \leqslant n$). On cherche tous les endroits où M apparaît dans C.

Définition 10.1.2 (Décalage, occurrence)**.** *On dit que le motif M* apparaît dans *le texte C* avec le décalage *i lorsque*

$$M = C[i{:}i+m], \qquad \textit{c'est-à-dire} \qquad M[j] = C[i+j] \ \textit{ pour tout } 0 \leqslant j \leqslant m-1.$$

On dit aussi que M a une occurrence *à la position i dans C.*

En d'autres termes, le décalage i indique de combien de crans on a fait glisser le motif vers la droite pour le superposer au texte. Pour que le facteur $C[i{:}i+m]$ ait bien m lettres, il faut $i + m \leqslant n$: les décalages possibles sont donc les entiers

$$i \in \{0, 1, \ldots, n-m\},$$

ce qui en fait exactement $n - m + 1$. *Le décalage* 0 *en fait partie* : il correspond au motif placé tout au début du texte, et il serait fâcheux de l'oublier. [1]

Exemple 10.1.3. *Le motif $M =$ `abr` apparaît dans $C =$ `abracadabra` (*de longueur *11) avec les décalages* 0 *et* 7 *:*

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
C	***a***	***b***	***r***	*a*	*c*	*a*	*d*	***a***	***b***	***r***	*a*

Le problème de la recherche de motif consiste à renvoyer la liste de tous ces décalages, ici $[0, 7]$.

1. C'est un piège classique. Si l'on fait commencer les décalages à 1, on rate l'occurrence située au tout début du texte — par exemple le motif `abr` dans `abracadabra`.

10.1.2 L'algorithme

L'idée la plus directe découle de la définition : pour chaque décalage i possible, on compare le motif M au facteur $C[i{:}i+m]$, et l'on retient i si les deux coïncident.

```python
def recherche_naive(C, M):
    """Renvoie la liste des décalages où le motif M apparaît dans le texte C.

    On dit que M apparaît dans C avec le décalage i lorsque M coïncide avec le
    facteur C[i:i+m], c'est-à-dire les m lettres de C à partir de l'indice i. On
    teste successivement chacun des décalages possibles i = 0, 1, ..., n - m en
    comparant M au facteur correspondant. La liste renvoyée est croissante.

    Le motif M doit être non vide (m >= 1) : c'est une précondition, sans quoi
    un motif vide « coïnciderait » avec tous les décalages, y compris au-delà de
    la fin du texte. La fonction écarte ce cas et lève une ValueError.

    Coût O((n - m + 1) * m) comparaisons de lettres dans le pire cas, atteint
    par exemple sur C = 'aaaa...' et M = 'aa...a'.

    >>> recherche_naive("abracadabra", "abr")
    [0, 7]
    >>> recherche_naive("aaaaa", "aa")
    [0, 1, 2, 3]
    >>> recherche_naive("abc", "x")
    []
    >>> recherche_naive("abc", "abcd")   # motif plus long que le texte
    []
    >>> recherche_naive("abc", "")       # motif vide : precondition violee
    Traceback (most recent call last):
        ...
    ValueError: motif vide
    """
    n = len(C)
    m = len(M)
    if m == 0:
        raise ValueError("motif vide")
    decalages = []
    for i in range(n - m + 1):
        if M == C[i:i + m]:
            decalages.append(i)
    return decalages
```

La fonction parcourt les décalages $i = 0, 1, \ldots, n-m$ (la borne `range(n - m + 1)` couvre bien le décalage $n - m$, le dernier possible) et ajoute i à la liste résultat chaque fois que

l'égalité $M = C[i{:}i+m]$ est vérifiée. La correction est immédiate : par la définition 10.1.2, i est ajouté à la liste si et seulement si M apparaît avec le décalage i, et tous les décalages possibles sont examinés une fois et une seule.

Avertissement 10.1.4. *La recherche suppose le motif* non vide *($m \geqslant 1$). C'est une* précondition*, au même titre que « la liste est non vide » pour la recherche du minimum (section 2.1) : un motif vide n'a pas de décalage bien défini — il « coïnciderait » avec* tous *les décalages, y compris au-delà de la fin du texte. La fonction commence donc par écarter ce cas et lève une erreur.*

10.1.3 Coût de l'algorithme

On compte le nombre de *comparaisons de lettres*, l'opération élémentaire naturelle ici. Tester l'égalité $M = C[i{:}i+m]$ revient à comparer les deux mots lettre à lettre ; cela coûte *au plus* m comparaisons, et parfois moins, car la comparaison s'arrête dès la première lettre qui diffère. Comme il y a $n-m+1$ décalages à tester, le nombre total de comparaisons est au plus

$$(n-m+1)\,m.$$

Dans le cas où m est petit devant n, cette quantité est de l'ordre de $n\,m$.

Proposition 10.1.5 (Le pire cas est atteint)**.** *La borne $(n-m+1)\,m$ est* atteinte *: il existe des entrées de taille n, m pour lesquelles l'algorithme effectue exactement ce nombre de comparaisons.*

Démonstration. Prenons un alphabet contenant la lettre $\mathtt{a}$, le texte $C = \mathtt{aa}\cdots\mathtt{a}$ de n lettres $\mathtt{a}$ et le motif $M = \mathtt{aa}\cdots\mathtt{a}$ de m lettres $\mathtt{a}$, avec $m \leqslant n$. Pour chacun des $n-m+1$ décalages, le facteur $C[i{:}i+m]$ est lui aussi égal à $\mathtt{a}^m$: la comparaison va jusqu'au bout, soit m comparaisons à chaque fois. Le total est donc exactement $(n-m+1)\,m$ comparaisons. □

Ce pire cas — un texte et un motif très répétitifs — montre que l'algorithme naïf peut vraiment être quadratique : pour $m \approx n/2$, on atteint de l'ordre de $n^2/4$ comparaisons. Les deux sections suivantes cherchent à échapper à cette répétition de comparaisons inutiles, chacune par une idée différente.

10.1.4 Exercices

Exercice 10.1.6. *Décrire un* meilleur cas *pour l'algorithme naïf, c'est-à-dire des entrées de taille n, m sur lesquelles le nombre de comparaisons est de l'ordre de $n-m+1$ seulement (une comparaison par décalage).* Indication : *comment faire en sorte que chaque test d'égalité échoue dès la première lettre ?*

Exercice 10.1.7. *Écrire une fonction* `premiere_occurrence_motif(C, M)` *qui renvoie le plus petit décalage où M apparaît dans C, ou -1 si M est absent, et qui s'arrête dès la première occurrence trouvée. Sur quelles entrées cette variante est-elle bien plus rapide que la recherche complète ? Sur quelles entrées ne gagne-t-elle rien ?*

Exercice 10.1.8. *Les occurrences d'un motif peuvent se* chevaucher *: le motif `aa` apparaît trois fois dans `aaaa`, aux décalages* 0*,* 1 *et* 2*. Écrire une fonction `compte_occurrences(C, M)` qui renvoie le nombre de décalages où M apparaît dans C, et vérifier qu'elle compte bien* 3 *pour `aa` dans `aaaa`. Combien de fois le motif a^m apparaît-il dans le texte a^n ?*

Exercice 10.1.9. *Montrer qu'un motif de longueur m a au plus $n - m + 1$ occurrences dans un texte de longueur n, et que cette borne est atteinte. En déduire que la* taille de la sortie *de l'algorithme de recherche peut être de l'ordre de n : même un algorithme idéal ne saurait coûter moins que $\Omega(n)$ dans ce cas.*

10.2 L'algorithme de Karp–Rabin

Nous avons vu à la section précédente que l'algorithme naïf passe l'essentiel de son temps à comparer des mots lettre à lettre, et que ces comparaisons sont souvent gâchées. L'algorithme de Karp–Rabin [2] garde le principe du balayage, mais remplace chaque comparaison de mots par une comparaison de *nombres*, en général bien plus rapide. L'idée — appelée *hachage* — est de résumer chaque mot par un entier, son *empreinte*, et de ne comparer les mots eux-mêmes que lorsque leurs empreintes coïncident.

10.2.1 Des mots à leurs empreintes

Pour calculer avec des mots, on les voit comme des nombres. On note d la taille de l'alphabet et, pour simplifier, on suppose que les lettres *sont* les entiers $\{0, 1, \ldots, d-1\}$ — ce qui évite toute conversion (la lettre `a` vaut 0, `b` vaut 1, etc.). Un mot $M = M_0 M_1 \cdots M_{m-1}$ est alors lu comme l'écriture en base d de l'entier

$$M_0\, d^{m-1} + M_1\, d^{m-2} + \cdots + M_{m-2}\, d + M_{m-1}.$$

Ainsi, pour $d = 10$, le mot `13452` est tout simplement l'entier 13452.

Définition 10.2.1 (Empreinte d'un mot)**.** *On se fixe un entier q, le* module*. L'*empreinte *d'un mot w est la valeur de w lue en base d, prise* modulo q*. On la note $\bar{w}$.*

Réduire modulo q garantit que toutes les empreintes restent des entiers inférieurs à q : en choisissant q assez petit pour qu'une empreinte tienne dans un registre du processeur, on rend les comparaisons d'empreintes immédiates. [3] L'empreinte se calcule en $O(m)$ opérations par le *schéma de Horner* [4], sans jamais former de grande puissance de d.

La propriété fondamentale de l'empreinte est une *implication*, à sens unique :

$$\bar{M} \neq \bar{w} \implies M \neq w.$$

2. Du nom de Richard Karp et Michael Rabin, qui le publièrent en 1987.
3. On s'arrange en pratique pour qu'aucun nombre intermédiaire ne dépasse $d \times q$, et l'on choisit q tel que $d \times q$ tienne dans un mot machine (typiquement 64 bits) ; les empreintes se manipulent alors en arithmétique simple précision.
4. Que vous verrez ou avez vu en travaux pratiques : on évalue $M_0 d^{m-1} + \cdots + M_{m-1}$ sans calculer aucune puissance, en partant de 0 et en répétant « multiplier par d, ajouter la lettre, réduire modulo q ».

Autrement dit, si deux mots ont des empreintes différentes, ils sont à coup sûr différents. La réciproque est fausse : deux mots distincts peuvent partager la même empreinte (puisqu'on a réduit modulo q). On dit alors qu'il y a *collision*, ou *faux positif*. C'est pourquoi, lorsque les empreintes coïncident, on ne peut pas conclure tout de suite : il faut confirmer en comparant réellement les deux mots.

10.2.2 Faire glisser l'empreinte

L'astuce qui rend l'algorithme efficace est la suivante : une fois connue l'empreinte d'une fenêtre $C[i{:}i+m]$ du texte, on obtient celle de la fenêtre suivante $C[i+1{:}i+1+m]$ en *temps constant*, sans la recalculer depuis le début. Il suffit de retirer la contribution de la lettre $C[i]$ qui sort par la gauche, de décaler tout le reste d'un cran (multiplication par d), puis d'ajouter la lettre $C[i+m]$ qui entre par la droite :

$$\overline{C[i+1{:}i+1+m]} = \big(d\times(\,\overline{C[i{:}i+m]}-C[i]\cdot h\,)+C[i+m]\big) \bmod q, \qquad \text{où } h = d^{m-1} \bmod q.$$

Exemple 10.2.2. *Avec $d = 10$, $m = 4$ et le texte C = 23451, la première fenêtre $C[0{:}4]$ = 2345 a pour valeur 2345. La fenêtre suivante $C[1{:}5]$ = 3451 s'en déduit (ici sans réduction, q étant pris très grand) :*

$$10 \times (2345 - 10^3 \times 2) + 1 = 10 \times 345 + 1 = 3451.$$

On a bien retiré le 2 de tête, décalé, puis ajouté le 1 de queue.

La puissance $h = d^{m-1} \bmod q$ se calcule une fois pour toutes (en $O(m)$ multiplications, ou $O(\log m)$ par exponentiation rapide). Chaque glissement coûte ensuite un nombre constant d'opérations.

10.2.3 L'algorithme

On calcule l'empreinte $\bar{M}$ du motif et l'empreinte de la première fenêtre, puis on balaye le texte : à chaque décalage, on compare les empreintes, et l'on ne déclenche la comparaison lettre à lettre que lorsqu'elles coïncident — pour écarter un éventuel faux positif. Entre deux décalages, on fait glisser l'empreinte de la fenêtre.

```
def karp_rabin(C, M, d, q):
    """Recherche du motif M dans le texte C par hachage glissant (Karp-Rabin).

    Le texte C et le motif M sont des listes de lettres codées par des entiers de
    {0, ..., d-1}, où d est la taille de l'alphabet ; q est un entier (idéalement
    premier) servant de module. À chaque mot on associe son empreinte : la valeur
    du mot lu comme un entier en base d, prise modulo q. On calcule l'empreinte du
    motif et celle de la première fenêtre C[0:m], puis on FAIT GLISSER la fenêtre
    d'un cran à chaque étape en mettant à jour l'empreinte en temps constant :

        empreinte(C[i+1:i+1+m]) = (d * (empreinte(C[i:i+m]) - C[i]*h) + C[i+m]) mod
```

```
    où h = d^(m-1) mod q. Lorsque les empreintes coïncident, on compare réellement
    le motif et la fenêtre lettre à lettre pour écarter un éventuel faux positif.
    Renvoie la liste croissante des décalages i tels que M = C[i:i+m].

    Le motif M doit être non vide (m >= 1) : c'est une précondition, comme pour
    l'algorithme naïf. La fonction écarte ce cas et lève une ValueError.

    >>> # motif [1, 0, 1] dans le texte binaire [1, 0, 1, 0, 1] : décalages 0 et 2
    >>> karp_rabin([1, 0, 1, 0, 1], [1, 0, 1], 2, 7)
    [0, 2]
    >>> # mêmes occurrences quel que soit le module q choisi
    >>> karp_rabin([1, 0, 1, 0, 1], [1, 0, 1], 2, 13)
    [0, 2]
    >>> # motif absent
    >>> karp_rabin([3, 1, 4, 1, 5], [2, 6], 10, 11)
    []
    >>> # motif plus long que le texte
    >>> karp_rabin([1, 2], [1, 2, 3], 10, 11)
    []
    >>> # motif vide : precondition violee
    >>> karp_rabin([1, 0, 1], [], 2, 7)
    Traceback (most recent call last):
        ...
    ValueError: motif vide
    """
    n = len(C)
    m = len(M)
    if m == 0:
        raise ValueError("motif vide")
    if m > n:
        return []
    h = pow(d, m - 1, q)
    empreinte_M = 0
    empreinte_C = 0
    for i in range(m):
        empreinte_M = (d * empreinte_M + M[i]) % q
        empreinte_C = (d * empreinte_C + C[i]) % q
    decalages = []
    for i in range(n - m + 1):
        if empreinte_M == empreinte_C and M == C[i:i + m]:
            decalages.append(i)
        if i < n - m:
            empreinte_C = (d * (empreinte_C - C[i] * h) + C[i + m]) % q
```

```
    return decalages
```

La correction se lit sur la propriété fondamentale : un décalage i n'est retenu que si l'on a vérifié $M = C[i{:}i+m]$ pour de bon, donc la liste renvoyée est exactement celle des occurrences (les mêmes que pour l'algorithme naïf, définition 10.1.2). Le test d'empreinte ne sert qu'à *éviter* des comparaisons complètes, jamais à en supprimer une qui conclurait à une occurrence : une vraie occurrence a forcément la même empreinte que le motif, donc passe le premier test.

Avertissement 10.2.3. *Comme l'algorithme naïf, Karp–Rabin suppose le motif* non vide *(*$m \geqslant 1$*) : c'est la même* précondition *(avertissement 10.1.4), et la fonction lève une erreur sur un motif vide.*

10.2.4 Coût de l'algorithme

Pire cas. Dans le pire des cas, Karp–Rabin n'est pas meilleur que l'algorithme naïf. Reprenons $C = \mathtt{a}^n$ et $M = \mathtt{a}^m$: toutes les fenêtres sont égales à $\mathtt{a}^m$, donc *toutes* ont la même empreinte que le motif. Chaque décalage déclenche alors une comparaison complète, et l'on retombe sur $(n - m + 1)\,m$ comparaisons. Le hachage ne protège pas contre un texte adversaire.

Une estimation heuristique du cas favorable. Le gain de Karp–Rabin se lit non pas dans le pire cas, mais « en moyenne ». L'analyse qui suit *sort du cadre du pire cas* adopté dans tout le cours (convention de la section 3.2.2) : elle repose sur un modèle probabiliste des entrées que nous ne formalisons pas. Nous la présentons donc comme une *heuristique*, une justification intuitive du bon comportement observé en pratique, et non comme un théorème.

Avertissement 10.2.4. *L'argument ci-dessous suppose qu'une fenêtre du texte a « une chance sur* q *» d'avoir la même empreinte que le motif sans lui être égale. Cette « probabilité » n'est pas définie rigoureusement (elle dépend du texte, du motif, du choix de* q*) ; le raisonnement est une estimation, pas une preuve.*

Décomposons le travail. Le calcul des deux empreintes initiales coûte $O(m)$. Le balayage lui-même — faire glisser l'empreinte le long du texte — coûte $O(1)$ par décalage, soit $O(n)$ au total. Restent les comparaisons complètes ; elles ne sont déclenchées que sur les décalages où les empreintes coïncident, c'est-à-dire :

— les *vraies occurrences* : notons x leur nombre, chacune coûtant $O(m)$;

— les *faux positifs* : sous l'hypothèse ci-dessus, ils sont de l'ordre de n/q, chacun coûtant aussi $O(m)$.

L'espérance du nombre d'opérations est donc de l'ordre de

$$\underbrace{O(m)}_{\text{empreintes}} + \underbrace{O(n)}_{\text{balayage}} + \underbrace{O(m\,(x + n/q))}_{\text{comparaisons complètes}} .$$

Lorsque le motif apparaît peu (disons $x = O(1)$) et que l'on choisit $q \geqslant m$, ce coût se simplifie en $O(n)$, puisque $m \leqslant n$ et $m/q \leqslant 1$: on obtient une recherche *linéaire* en la taille du texte. C'est le cas favorable typique, et le mérite de l'algorithme.

10.2.5 Exercices

Exercice 10.2.5. *Écrire une fonction* `valeur_horner(M, d, q)` *qui calcule l'empreinte d'un mot M (liste de lettres de $\{0, \ldots, d-1\}$) par le schéma de Horner, en $O(m)$ opérations et sans calculer de puissance de d. Vérifier qu'elle renvoie* 13452 *pour le mot* `13452` *avec $d = 10$ et un module q assez grand.*

Exercice 10.2.6. *On prend $d = 2$ et le module $q = 2$. Montrer que l'empreinte d'un mot binaire n'est alors rien d'autre que sa* parité *(le nombre de* 1 *modulo* 2*). En déduire qu'avec ce module, l'algorithme déclenche une comparaison complète sur* toutes *les fenêtres ayant la même parité que le motif, et donc qu'un module trop petit ruine le gain attendu. Donner un texte et un motif sur lesquels il y a au moins un faux positif.*

Exercice 10.2.7. *Écrire une variante* `karp_rabin_compteur(C, M, d, q)` *qui renvoie, en plus de la liste des décalages, le nombre de comparaisons complètes effectuées. Vérifier que ce nombre, diminué du nombre d'occurrences, donne bien le nombre de faux positifs. Faire varier q et observer la décroissance du nombre de faux positifs.*

Exercice 10.2.8. *Confirmer l'analyse du pire cas : sur $C = \mathtt{a}^n$ et $M = \mathtt{a}^m$, montrer que chaque test d'empreinte conclut à l'égalité, et que l'algorithme effectue donc autant de comparaisons de lettres que l'algorithme naïf. Le hachage a-t-il un quelconque effet ici ?*

10.3 Recherche par automate

L'algorithme de Karp–Rabin accélère le cas favorable, mais reste quadratique dans le pire cas, car il peut relire plusieurs fois les mêmes lettres du texte. Nous présentons maintenant une méthode qui, après un *précalcul* ne portant que sur le motif, lit le texte *une seule fois*, lettre par lettre, sans jamais revenir en arrière — et garantit donc une phase de recherche linéaire, même dans le pire cas. L'outil sous-jacent est un *automate* ; il n'est pas nécessaire d'en connaître la théorie pour comprendre la construction.

10.3.1 Préfixes, suffixes, et la fonction σ

Rappelons (section 7.2) qu'un *préfixe* d'un mot u est un début de u : pour $0 \leqslant \ell \leqslant |u|$, le préfixe de longueur ℓ est $u[:\ell] = u_0 u_1 \cdots u_{\ell-1}$. Symétriquement, un *suffixe* est une fin de u.

Définition 10.3.1 (Suffixe)**.** *Soit u un mot de longueur k et $0 \leqslant \ell \leqslant k$. Le* suffixe de longueur *ℓ de u est le mot formé de ses ℓ* dernières *lettres :*

$$u[k-\ell:] = u_{k-\ell}\, u_{k-\ell+1} \cdots u_{k-1}.$$

Pour $\ell = 0$ c'est le mot vide ε, et pour $\ell = k$ c'est u tout entier.

En d'autres termes, un mot v est un suffixe de u lorsque u se termine par v. Par exemple, les suffixes de `bonjour` sont ε, `r`, `ur`, `our`, `jour`, `njour`, `onjour` et `bonjour` — à ne pas confondre avec ses *préfixes* ε, `b`, `bo`, `bon`, `bonj`, `bonjo`, `bonjou` et `bonjour`, qui sont ses débuts.

Toute l'idée de l'algorithme tient dans une seule quantité. On fixe le motif M, de longueur m, et l'on regarde, pour un mot x quelconque, à quel point x se termine par un début de M.

Définition 10.3.2 (La fonction σ_M)**.** *Pour un mot x, on note $\sigma_M(x)$ la longueur du plus long préfixe de M qui est aussi un suffixe de x.*

On a toujours $0 \leqslant \sigma_M(x) \leqslant m$ (le préfixe vide convient toujours, et aucun préfixe de M ne dépasse M lui-même). L'intérêt de cette fonction est l'équivalence suivante, immédiate à partir de la définition :

$$\sigma_M(x) = m \quad \Longleftrightarrow \quad M \text{ est un suffixe de } x \quad \Longleftrightarrow \quad M \text{ apparaît à la fin de } x.$$

Autrement dit, repérer les occurrences de M dans le texte C revient à repérer les préfixes $C[:i]$ pour lesquels $\sigma_M(C[:i]) = m$.

Exemple 10.3.3. *Prenons M = `ababaca` (donc $m = 7$). Alors $\sigma_M($`abab`$) = 4$, car `abab` est à la fois un suffixe de `abab` et le préfixe de longueur 4 de M. En revanche $\sigma_M($`ababab`$) = 4$ également : le plus long préfixe de M qui termine `ababab` est encore `abab` (le préfixe `ababa` de longueur 5 n'est pas un suffixe de `ababab`). Enfin $\sigma_M($`xyz`$) = 0$: aucun préfixe non vide de M ne termine `xyz`.*

10.3.2 L'algorithme de recherche, en supposant σ_M connue

Supposons un instant que l'on sache calculer σ_M. On lit alors le texte $C = C_0 C_1 \cdots C_{n-1}$ de gauche à droite ; après avoir lu les i premières lettres, on dispose de la valeur $\ell = \sigma_M(C[:i])$. Chaque fois que $\ell = m$, l'équivalence ci-dessus garantit que M se termine à la position $i-1$, c'est-à-dire qu'il apparaît avec le décalage $i - m$; on enregistre ce décalage. Tout repose donc sur la mise à jour de ℓ d'une lettre à la suivante.

Calculer $\sigma_M(C[:i])$ *à partir de rien* à chaque étape coûterait $O(m)$ par lettre, soit $O(nm)$ en tout : pas mieux que le naïf. La clef est de mettre à jour ℓ en temps constant, grâce à un précalcul effectué une fois pour toutes sur le motif. C'est la *phase de précalcul*.

10.3.3 La table de transition

Tout repose sur le lemme suivant, qui dit que la prochaine valeur de σ_M ne dépend que de la valeur courante et de la lettre lue — pas du texte déjà parcouru.

Lemme 10.3.4 (Mise à jour de σ_M)**.** *Soit x un mot et a une lettre. Si $\sigma_M(x) = \ell$, alors*

$$\sigma_M(xa) = \sigma_M(M[:\ell] \cdot a),$$

où xa désigne le mot x suivi de la lettre a, et $M[:\ell] \cdot a$ le préfixe de longueur ℓ de M suivi de a.

Démonstration. Posons $\ell = \sigma_M(x)$. Par définition, $M[:\ell]$ est un préfixe de M qui est un suffixe de x : les ℓ dernières lettres de x sont donc exactement $M[:\ell]$.

Montrons d'abord que $\sigma_M(xa) \leqslant \ell + 1$. Si l'on avait $\sigma_M(xa) = k$ avec $k > \ell + 1$, alors $M[:k]$ serait un suffixe de xa ; en lui retirant sa dernière lettre, $M[:k-1]$ serait un suffixe de x, avec $k - 1 > \ell$ — un préfixe de M plus long que ℓ terminant x, ce qui contredit $\sigma_M(x) = \ell$. Donc $\sigma_M(xa) \leqslant \ell + 1$.

Ainsi, le plus long préfixe de M qui est suffixe de xa a une longueur au plus $\ell + 1$: il ne fait intervenir que les $\ell + 1$ dernières lettres de xa, c'est-à-dire les ℓ dernières lettres de x suivies de a. Or ces ℓ dernières lettres de x sont $M[:\ell]$. Le plus long préfixe de M suffixe de xa est donc le même que celui de $M[:\ell] \cdot a$; les deux mots ont la même valeur de σ_M. □

Le membre de droite, $\sigma_M(M[:\ell] \cdot a)$, ne dépend que de l'entier $\ell \in \{0, 1, \ldots, m\}$ et de la lettre $a \in \Sigma$. On peut donc *tout précalculer* dans une table : pour chaque état ℓ et chaque lettre a,

$$\texttt{table}[\ell][a] = \sigma_M(M[:\ell] \cdot a).$$

On la calcule naïvement — pour chaque case, on cherche le plus long préfixe de M qui termine $M[:\ell] \cdot a$, en essayant les longueurs décroissantes.

```python
def calcule_table(M, alphabet):
    """Construit la table de transition de l'automate de recherche du motif M.

    table[l][a] vaut sigma(M[:l] + a) : la longueur du plus long préfixe de M qui
    est aussi un suffixe du mot M[:l] + a. Cette valeur ne dépend que de l'état
    courant l (un entier de 0 à m) et de la lettre lue a, jamais du texte. On la
    calcule naïvement : pour chaque (l, a), on cherche le plus grand k tel que
    M[:k] soit un suffixe de M[:l] + a, en essayant les longueurs k décroissantes.

    Coût O(|alphabet| * m^3) (m+1 états, |alphabet| lettres, chaque case en
    O(m^2)) ; négligeable devant le texte quand le motif est court.

    >>> calcule_table("aa", "ab")
    [{'a': 1, 'b': 0}, {'a': 2, 'b': 0}, {'a': 2, 'b': 0}]
    """
    m = len(M)
    table = []
    for l in range(m + 1):
        mot_prefixe = M[:l]
        ligne = {}
        for a in alphabet:
            mot = mot_prefixe + a
            k = min(m, len(mot))
            while M[:k] != mot[len(mot) - k:]:
                k -= 1
```

```
            ligne[a] = k
        table.append(ligne)
    return table


def recherche_automate(C, M, alphabet):
    """Renvoie la liste des décalages où le motif M apparaît dans le texte C.

    On précalcule la table de transition de M (phase de précalcul), puis on
    lit le texte une seule fois en mettant à jour un état l = longueur du plus
    long préfixe de M qui termine le texte lu jusqu'ici. Chaque fois que l atteint
    m = |M|, c'est que M vient de se terminer : on enregistre le décalage. La
    phase de recherche est en O(n), chaque lettre du texte n'étant lue qu'une fois.
    Toutes les lettres de C et de M doivent appartenir à alphabet.

    Le motif M doit être non vide (m >= 1) : c'est une précondition. Avec un motif
    vide, l'état l = 0 vaudrait déjà m = 0 et l'algorithme signalerait une fausse
    occurrence à chaque lettre ; la fonction écarte ce cas et lève une ValueError.

    >>> recherche_automate("abababacaba", "ababaca", "abc")
    [2]
    >>> recherche_automate("aaaa", "aa", "ab")
    [0, 1, 2]
    >>> recherche_automate("abracadabra", "abr", "abcdr")
    [0, 7]
    >>> recherche_automate("abcabc", "xyz", "abcxyz")
    []
    >>> recherche_automate("abc", "", "abc")   # motif vide : precondition violee
    Traceback (most recent call last):
        ...
    ValueError: motif vide
    """
    n = len(C)
    m = len(M)
    if m == 0:
        raise ValueError("motif vide")
    table = calcule_table(M, alphabet)
    decalages = []
    l = 0
    for i in range(n):
        l = table[l][C[i]]
        if l == m:
            decalages.append(i - m + 1)
    return decalages
```

La table joue le rôle de la fonction de transition d'un automate fini : les états sont les entiers $0, 1, \ldots, m$, l'état $\ell = m$ signale une occurrence, et lire une lettre fait passer de l'état ℓ à l'état $\texttt{table}[\ell][a]$.[5]

10.3.4 L'algorithme complet et son coût

L'algorithme `recherche_automate` ci-dessus enchaîne les deux phases : il construit la table, puis lit le texte en maintenant l'état ℓ, qu'il met à jour par un simple accès `table[l][C[i]]`. La correction découle du lemme 10.3.4 : une fois lue la lettre C_i d'indice i, l'état vaut $\sigma_M(C[:i+1])$[6] ; or $\sigma_M(C[:i+1]) = m$ signifie que M se termine en position i, c'est-à-dire apparaît avec le décalage $i - m + 1$ — exactement ce que l'algorithme enregistre.

Avertissement 10.3.5. *Ici encore, le motif est supposé* non vide *($m \geqslant 1$) : c'est la même* précondition *que pour les deux algorithmes précédents (avertissement 10.1.4). Avec un motif vide, l'état initial $\ell = 0$ vaudrait déjà $m = 0$ et l'algorithme signalerait une occurrence à chaque lettre ; la fonction l'écarte explicitement et lève une erreur.*

Proposition 10.3.6 (Coût de la recherche par automate)**.** *Sur un alphabet fixé (de taille $O(1)$), la phase de précalcul coûte $O(m^3)$ et la phase de recherche $O(n)$, soit un coût total de*

$$O(n + m^3).$$

Démonstration. La table a $(m+1) \times |\Sigma|$ cases. Le calcul d'une case essaie au plus $m+1$ longueurs de préfixe, et chaque essai compare deux mots de longueur au plus m, en $O(m)$: une case coûte donc $O(m^2)$, et la table $O(|\Sigma|\, m^3)$, soit $O(m^3)$ à alphabet fixé. La phase de recherche effectue n itérations, chacune réduite à un accès dans la table et une comparaison, en $O(1)$: elle coûte $O(n)$. □

Lorsque le motif est bien plus court que le texte, le terme $O(n)$ domine et l'algorithme est *linéaire en la taille du texte*. Par exemple, dès que $m \leqslant n^{1/3}$, on a $m^3 \leqslant n$ et le coût total est $O(n)$ — chaque lettre du texte n'est lue qu'une fois, sans aucun retour en arrière. C'est là le gain décisif sur l'algorithme naïf : le pire cas lui-même devient linéaire.

Remarque 10.3.7. *Le précalcul $O(m^3)$ de la table est volontairement naïf, pour rester élémentaire ; il existe des constructions bien plus rapides (en $O(m)$ pour une variante proche, celle de Knuth, Morris et Pratt, fondée sur une « fonction d'échec »). Comme le motif est en général court, ce précalcul n'est de toute façon pas le goulot d'étranglement : c'est la longueur n du texte qui gouverne le coût.*

5. Si vous connaissez les automates finis, la table `table` est exactement la table de transition de l'automate reconnaissant les mots terminant par M ; cette section en construit l'essentiel sans en supposer la théorie.

6. Récurrence sur le nombre de lettres lues. Initialisation : avant toute lecture, $\ell = 0 = \sigma_M(\varepsilon)$. Hérédité : si $\ell = \sigma_M(C[:i])$ avant de lire C_i, alors la nouvelle valeur $\texttt{table}[\ell][C_i] = \sigma_M(M[:\ell] \cdot C_i) = \sigma_M(C[:i+1])$ par le lemme.

10.3.5 Exercices

Exercice 10.3.8. *Pour le motif $M =$ `aba`, calculer à la main $\sigma_M(x)$ pour $x =$ `a`, $x =$ `ab`, $x =$ `aab`, $x =$ `ababa` et $x =$ `abc`. Vérifier dans chaque cas qu'on lit bien la longueur du plus long préfixe de M qui termine x.*

Exercice 10.3.9. *Calculer la table de transition pour le motif $M =$ `ababaca` sur l'alphabet $\Sigma = \{$`a`, `b`, `c`$\}$, puis dérouler l'algorithme de recherche sur le texte `abababacaba` : indiquer l'état ℓ après chaque lettre lue et retrouver le décalage de l'unique occurrence.*

Exercice 10.3.10. *Écrire une fonction `sigma_naive(M, x)` qui calcule $\sigma_M(x)$ directement, en essayant les longueurs de préfixe décroissantes. Combien de comparaisons de lettres effectue-t-elle au pire ? En déduire qu'utiliser cette fonction à chaque lettre du texte, sans table, redonnerait un algorithme en $O(nm)$.*

Exercice 10.3.11. *Modifier l'algorithme de recherche par automate pour qu'il renvoie le* premier *décalage où M apparaît (ou -1 s'il est absent), en s'arrêtant dès que l'état atteint m. La phase de recherche reste-t-elle linéaire dans le pire cas ?*

11 Les tas ★

Ce chapitre tout entier est *hors programme* : il est signalé par une ★ et s'adresse au lecteur curieux d'aller plus loin. On y construit une structure de données — le *tas* — qui réalise efficacement une *file de priorité*, et l'on en tire au passage un second algorithme de tri en $O(n \log n)$, le *tri par tas*.

Résumé

Une *file de priorité* est une collection d'éléments munis d'une priorité, dans laquelle on veut pouvoir insérer un élément et en retirer celui de priorité maximale. Ce chapitre décrit une structure de données simple et efficace pour la réaliser : le *tas*, un tableau que l'on lit comme un arbre binaire presque complet et dans lequel chaque nœud domine ses fils. Nous apprenons à organiser un tableau en tas en temps linéaire, à y insérer et en extraire le maximum en $O(\log n)$, et nous en déduisons le *tri par tas*, un tri en place de complexité $O(n \log n)$.

Prérequis

Le vocabulaire des *arbres* — racine, fils, feuille, hauteur — introduit au chapitre 5 (section 5.5) ; les notions de *pile* (chapitre 5, section 5.1.2) et de *file* (chapitre 8, section 8.3) ; la convention d'indexation à partir de 0 (convention 1.2.1) ; les notations de complexité O (chapitre 3) ; le tri fusion et sa borne $O(n \log n)$ (chapitre 6, section 6.2).

Objectifs

À l'issue de ce chapitre, vous saurez ce qu'est une file de priorité, reconnaître et manipuler la *propriété de tas*, lire un tableau comme un arbre binaire presque complet, construire un tas en temps linéaire, y insérer et en extraire le maximum, et enfin trier un tableau en place grâce à un tas.

Nous avons déjà rencontré deux structures où l'on range des éléments pour les ressortir un à un. Dans une *pile* (chapitre 5), c'est le *dernier* élément entré qui ressort en premier — comme une pile d'assiettes, on prend toujours celle du dessus. Dans une *file* (chapitre 8), c'est au contraire le *premier* entré qui sort en premier — comme une file d'attente à un guichet, où l'on est servi dans l'ordre d'arrivée.

Mais que faire si l'on veut ressortir les éléments par ordre d'importance, et non d'arrivée ? C'est exactement ce que demande une *file de priorité* : chaque élément y porte une *priorité* (un nombre), et lorsqu'on retire un élément, c'est toujours celui de priorité maximale. Pensez à la file d'attente d'un service d'urgences, où l'on traite d'abord les cas

les plus graves, quel que soit leur ordre d'arrivée ; ou à un ordonnanceur de tâches qui exécute en priorité la tâche la plus urgente. La difficulté est de réaliser ces deux opérations — *insérer* et *extraire le maximum* — sans avoir à reparcourir toute la collection à chaque fois. La structure de *tas* [1] y parvient en gardant les éléments « presque triés », et nous donnera de surcroît un nouvel algorithme de tri.

11.1 Files de priorité et propriété de tas

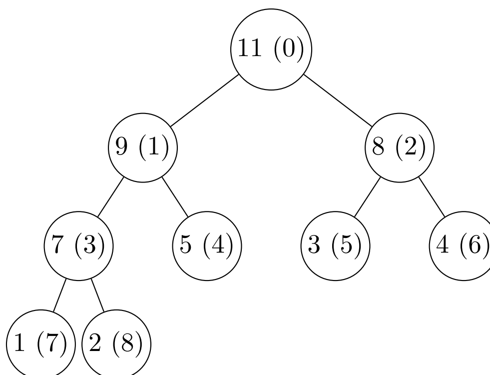
11.1.1 Un tableau lu comme un arbre

L'idée de départ est de ranger les éléments dans un simple tableau, mais de *lire* ce tableau comme un arbre binaire. Précisément, on visualise un tableau A de longueur n comme un *arbre binaire presque complet* : un arbre binaire dont tous les niveaux sont remplis, sauf peut-être le dernier, qui l'est de gauche à droite. La case d'indice 0 est la racine ; on parcourt ensuite l'arbre niveau par niveau (par profondeur croissante), de gauche à droite, en suivant les indices.

Voyons cela sur un exemple. Le tableau de longueur 9

Indice	0	1	2	3	4	5	6	7	8
Valeur	11	9	8	7	5	3	4	1	2

représente l'arbre suivant, où chaque nœud porte sa valeur et, entre parenthèses, son indice dans le tableau :



Cette lecture n'a rien d'arbitraire : la numérotation niveau par niveau relie la place d'un nœud et celle de ses fils par une formule simple.

Proposition 11.1.1 (Indices des fils et du père)**.** *Dans un arbre binaire presque complet numéroté de cette façon, le nœud d'indice p a pour fils gauche le nœud d'indice $2p+1$ et pour fils droit le nœud d'indice $2p+2$ (lorsque ces indices sont $< n$). Réciproquement, le père du nœud d'indice i (pour $i \geqslant 1$) est le nœud d'indice $\lfloor (i-1)/2 \rfloor$.*

1. De l'anglais *heap*. Le mot évoque un amas en vrac, mais ne vous y trompez pas : un tas bien fait est une structure très ordonnée.

En d'autres termes, pour descendre vers un fils on double l'indice (et on ajoute 1 ou 2) ; pour remonter vers le père on retranche 1 et l'on divise par 2.

Démonstration. Il y a 2^ℓ nœuds à la profondeur ℓ (la racine étant à la profondeur 0), donc le premier nœud de la profondeur ℓ — le plus à gauche — porte l'indice $2^0 + 2^1 + \cdots + 2^{\ell-1} = 2^\ell - 1$. Numérotons les nœuds d'une même profondeur ℓ de gauche à droite par $k = 0, 1, \ldots$: le k-ième a donc l'indice $2^\ell - 1 + k$. Ses deux fils sont à la profondeur $\ell + 1$, aux places $2k$ et $2k + 1$ de la gauche, donc d'indices $2^{\ell+1} - 1 + 2k$ et $2^{\ell+1} - 1 + 2k + 1$. Or si $p = 2^\ell - 1 + k$, alors $2p + 1 = 2^{\ell+1} - 1 + 2k$ et $2p + 2 = 2^{\ell+1} - 1 + 2k + 1$: ce sont exactement les indices des deux fils. La formule du père s'en déduit en inversant celles des fils : si $i = 2p + 1$ ou $i = 2p + 2$, alors $\lfloor (i-1)/2 \rfloor = p$ dans les deux cas. □

On retiendra ces formules sous la forme, en machine, de trois fonctions élémentaires : `parent(i)` renvoie `(i - 1) // 2`, `fils_gauche(i)` renvoie `2 * i + 1`, et `fils_droit(i)` renvoie `2 * i + 2`.

Remarque 11.1.2. *La littérature présente souvent les tas avec une indexation à partir de* 1 *: les fils de p sont alors en $2p$ et $2p + 1$, et le père de i en $\lfloor i/2 \rfloor$, formules un brin plus jolies. Fidèles à la convention du cours (convention 1.2.1), nous indexons à partir de* 0 *comme toutes les listes Python, au prix de ce « +1 » dans les formules.*

11.1.2 La propriété de tas

Un tas n'est pas n'importe quel arbre : on lui demande que chaque nœud *domine* ses fils.

Définition 11.1.3 (Propriété de tas)**.** *Un tableau A de longueur n* vérifie la propriété de tas à la position *i si la valeur $A[i]$ est supérieure ou égale à celle de chacun de ses fils présents dans le tableau.*

Concrètement, en utilisant les indices des fils (proposition 11.1.1), A vérifie la propriété de tas à la position i si et seulement si l'une des trois situations suivantes a lieu :

— $2i + 1 \geqslant n$: le nœud i n'a aucun fils, la propriété est vide ;

— $2i + 1 = n - 1$ (un seul fils, le gauche) : $A[i] \geqslant A[2i + 1]$;

— $2i + 2 \leqslant n - 1$ (deux fils) : $A[i] \geqslant \max(A[2i + 1], A[2i + 2])$.

Cette propriété, exigée *partout*, donne sa structure au tas. Pour l'énoncer proprement, rappelons qu'un nœud j est un *descendant* d'un nœud i s'il existe un chemin descendant de i à j dans l'arbre ; par convention i est descendant de lui-même, et un descendant *strict* de i est un descendant différent de i.

Définition 11.1.4 (Tableau organisé en tas)**.** *Un tableau A est* organisé en tas à la position *i si la propriété de tas est vérifiée en tout descendant de i (i compris). Il est* organisé en tas *(tout court) s'il l'est à la position* 0*, c'est-à-dire si la propriété de tas est vérifiée à toute position.*

Dit autrement, un tableau est organisé en tas lorsque, dans l'arbre associé, chaque valeur est supérieure ou égale à toutes celles qui figurent dans son sous-arbre. Le tableau de notre exemple est organisé en tas : on vérifie sur l'arbre qu'en chaque nœud la valeur domine bien ses deux fils. En particulier, *la plus grande valeur se trouve à la racine*, en position 0 — c'est toute l'utilité de la structure pour une file de priorité, et nous le démontrerons au moment de l'extraction (section 11.3).

Exemple 11.1.5. *Le tableau*

Indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*
Valeur	*5*	*1*	*3*	*6*	*4*	*7*	*2*

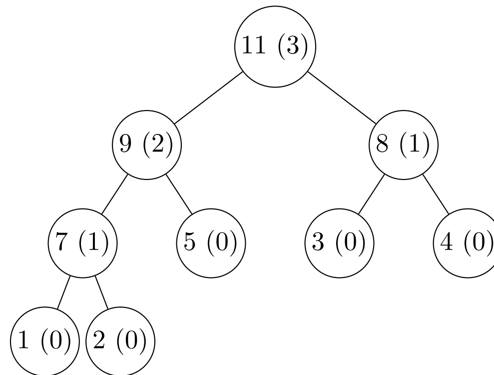
n'est pas *organisé en tas. La propriété de tas est déjà violée à la position* 1 *: le nœud de valeur* 1 *a pour fils gauche (indice* $2 \cdot 1 + 1 = 3$*) le nœud de valeur* 6*, et* $1 < 6$*. Dessinez l'arbre pour le voir : la valeur* 7*, enfouie en position* 5*, devrait « remonter » vers la racine. Nous apprendrons à la section 11.2 comment réorganiser un tel tableau en tas.*

11.1.3 Hauteur d'un nœud

La complexité des opérations sur un tas se mesurera en fonction de la hauteur des nœuds ; rappelons cette notion (vue au chapitre 5) dans le cadre qui nous occupe.

Définition 11.1.6 (Hauteur d'un nœud)**.** *La* hauteur *d'un nœud est la longueur maximale d'un chemin descendant de ce nœud jusqu'à une feuille. En particulier, une feuille est de hauteur* 0*.*

Sur l'arbre de notre exemple, les hauteurs valent (entre parenthèses) :



Le point essentiel, qui resservira constamment, est que cet arbre est *peu profond* : comme il est presque complet, sa hauteur croît seulement comme le logarithme du nombre de nœuds.

Proposition 11.1.7 (Un tas est peu profond)**.** *La hauteur de la racine d'un arbre binaire presque complet à* n *nœuds vaut* $\lfloor \log_2 n \rfloor$*. En particulier, tout nœud est de hauteur au plus* $\lfloor \log_2 n \rfloor$*.*

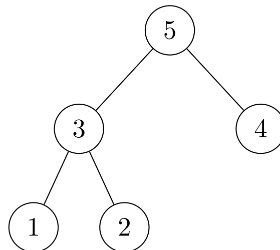
Démonstration. Un arbre binaire dont tous les niveaux $0, 1, \ldots, h$ sont entièrement remplis possède $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nœuds. Un arbre binaire presque complet de hauteur h a donc tous ses niveaux pleins jusqu'à $h - 1$ (soit $2^h - 1$ nœuds) et au moins un nœud au niveau h : son nombre de nœuds n vérifie $2^h \leqslant n \leqslant 2^{h+1} - 1 < 2^{h+1}$. En passant au logarithme en base 2, on obtient $h \leqslant \log_2 n < h + 1$, c'est-à-dire $h = \lfloor \log_2 n \rfloor$. □

11.1.4 Taille du tas et longueur du tableau

Une dernière convention, technique mais commode, nous sera utile pour le tri par tas (section 11.4). Il arrive qu'on veuille ne considérer comme *faisant partie du tas* qu'un préfixe du tableau, et réserver la fin du tableau à un autre usage. On distingue donc la *longueur* du tableau A — le nombre total de cases, noté $\texttt{len}(A)$ — de la *taille* du tas — le nombre $t \leqslant \texttt{len}(A)$ de cases qui participent effectivement au tas. Seules les positions $0, 1, \ldots, t - 1$ font alors partie du tas ; les cases d'indice $\geqslant t$ sont ignorées par toutes les opérations.

Avertissement 11.1.8. *Ne confondez pas la* longueur $\texttt{len}(A)$ *du tableau et la* taille t *du tas. Toutes les formules d'indices (fils, père) restent valables, mais un nœud d'indice* $\geqslant t$*, même présent dans le tableau, ne compte pas comme un fils : la propriété de tas ne porte que sur les positions* $0, \ldots, t - 1$*.*

Exemple 11.1.9. *Si* $A = [5, 3, 4, 1, 2, 11, 12]$ *a pour longueur* 7 *mais que la taille du tas est* $t = 5$*, alors seules les cinq premières cases comptent. Le tas correspond à l'arbre*



et il est bien organisé en tas : en chaque nœud, la valeur domine ses fils. Les valeurs 11 *et* 12*, rangées en positions* 5 *et* 6*, sont* hors du tas *et ne le contredisent pas, bien qu'elles soient plus grandes que la racine.*

En machine, plutôt que de stocker la taille dans une case réservée du tableau, nous la passerons simplement en *paramètre* aux fonctions qui en ont besoin — c'est plus clair et cela évite tout indice spécial. Les fonctions qui travaillent sur le tableau entier (taille égale à la longueur) s'en passeront.

11.1.5 Exercices

Exercice 11.1.10. *Dessinez l'arbre binaire presque complet associé au tableau* $[9, 7, 8, 3, 6, 2, 5, 1]$*. Vérifiez position par position s'il est organisé en tas, et corrigez la première violation que vous rencontrez en échangeant deux valeurs.*

Exercice 11.1.11. *Écrivez une fonction* `est_tas(A)` *qui renvoie* `True` *si la liste A est organisée en tas, et* `False` *sinon. (Indication : il suffit de vérifier la propriété de tas en chaque position, à l'aide des indices de fils.)*

Exercice 11.1.12. *Programmez les trois fonctions d'indices* `parent`*,* `fils_gauche` *et* `fils_droit`*, et vérifiez sur l'arbre de l'exemple que* `parent(fils_gauche(p))` *et* `parent(fils_droit(p))` *valent bien p pour quelques positions p.*

Exercice 11.1.13. *Montrez que, dans un arbre binaire presque complet à n nœuds, les nœuds d'indices $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots, n-1$ sont exactement les* feuilles *(les nœuds sans fils). Combien y a-t-il de feuilles ? (Indication : p est une feuille si et seulement si $2p+1 \geqslant n$.)*

11.2 Construction d'un tas en temps linéaire

Nous avons défini à la section précédente ce qu'est un tableau organisé en tas. Étant donné un tableau quelconque, nous voulons maintenant en *permuter les éléments* pour qu'il devienne un tas. Nous procédons en deux temps : d'abord une brique élémentaire, la fonction `entasser`, qui répare la propriété de tas en un nœud lorsque ses deux sous-arbres sont déjà des tas ; puis l'algorithme de construction proprement dit, qui applique cette brique du bas vers le haut.

11.2.1 La fonction `entasser`

Supposons que l'on appelle `entasser` sur une position p dans une situation très particulière : les deux sous-arbres issus de p (enracinés en $2p+1$ et $2p+2$) sont *déjà* organisés en tas, mais la valeur $A[p]$, elle, peut être trop petite et violer la propriété de tas en p. La fonction rétablit alors la propriété en faisant « descendre » cette valeur à sa place.

L'idée est la suivante. On regarde les trois valeurs en p, en son fils gauche et en son fils droit (ceux qui existent), et l'on note m la position de la plus grande. Si $m = p$, la propriété de tas est déjà vérifiée en p et il n'y a rien à faire. Sinon, on échange $A[p]$ avec $A[m]$: la racine porte désormais la plus grande des trois valeurs, donc la propriété de tas est rétablie en p. Mais l'échange a pu casser la propriété de tas en m ; on relance donc `entasser` en m, et ainsi de suite jusqu'à atteindre une feuille.

```
def entasser(A, p, taille):
    """Réorganise le sous-arbre de racine ``p`` pour qu'il soit organisé en tas,
    en supposant que ses deux sous-arbres fils le sont déjà. Seules les positions
    ``0`` à ``taille - 1`` de ``A`` font partie du tas ; ``A`` est modifié en
    place. Fonction récursive : on échange la racine avec le plus grand de ses
    fils s'il la dépasse, puis on recommence à cette position.

    >>> A = [1, 14, 10, 8, 7, 9, 3]
    >>> entasser(A, 0, len(A))
```

```
    >>> A
    [14, 8, 10, 1, 7, 9, 3]
    """
    gauche = 2 * p + 1
    droite = 2 * p + 2
    m = p
    if gauche < taille and A[gauche] > A[m]:
        m = gauche
    if droite < taille and A[droite] > A[m]:
        m = droite
    if m != p:
        A[p], A[m] = A[m], A[p]
        entasser(A, m, taille)
```

Remarquons que `entasser` ne fait que permuter des éléments du tableau : elle ne crée ni ne détruit aucune valeur. Sa propriété essentielle est la suivante.

Proposition 11.2.1 (Correction de `entasser`)**.** *Si l'on appelle* `entasser`(A, p, t) *alors que les sous-arbres enracinés en* $2p+1$ *et* $2p+2$ *sont organisés en tas (pour la taille* t*), alors, après l'exécution, le tableau est organisé en tas à la position* p*.*

Démonstration. Par récurrence sur la hauteur h du nœud p.

Initialisation ($h = 0$). Le nœud p est une feuille : il n'a pas de fils, la propriété de tas y est vérifiée (un nœud sans fils ne peut pas la violer) et l'algorithme ne modifie pas le tableau (on a $m = p$). Le tableau est donc organisé en tas à la position p.

Hérédité. Supposons la propriété acquise pour tout nœud de hauteur $< h$, et soit p de hauteur $h \geqslant 1$. L'algorithme calcule la position m du maximum des valeurs en p, $2p+1$ et $2p+2$ (parmi celles qui existent), puis distingue deux cas.

Cas $m = p$. La valeur $A[p]$ domine ses fils, donc la propriété de tas est vérifiée en p ; comme les deux sous-arbres sont déjà des tas par hypothèse, le tableau est organisé en tas à la position p et l'algorithme s'arrête sans rien modifier.

Cas $m = 2p+1$ (le fils gauche ; le cas $m = 2p+2$ du fils droit se traite de façon identique). On échange $A[p]$ et $A[2p+1]$. Après l'échange, $A[p]$ vaut l'ancien maximum des trois : la propriété de tas est désormais vérifiée en p. Par ailleurs l'échange n'a touché que les positions p et $2p+1$, donc le sous-arbre du fils droit $2p+2$ est resté un tas, et les deux sous-arbres du fils gauche $2p+1$ sont eux aussi restés des tas. Le nœud $2p+1$ est un fils de p, donc *de hauteur au plus* $h-1$; l'hypothèse de récurrence s'applique à l'appel `entasser`$(A, 2p+1, t)$ et garantit qu'après celui-ci, le sous-arbre enraciné en $2p+1$ est organisé en tas. Comme la propriété de tas est vérifiée en p et que les deux sous-arbres de p sont des tas, le tableau est organisé en tas à la position p.[2] □

Proposition 11.2.2 (Coût de `entasser`)**.** *La fonction* `entasser`(A, p, t) *effectue* $O(\textit{hauteur}(p))$ *opérations, soit* $O(\log n)$ *pour un tas de taille* n*.*

2. C'est bien la *hauteur* du fils qui intervient ici, et non sa profondeur : la récurrence décroît parce que la hauteur du fils est *majorée* par $h-1$, tandis que sa profondeur, elle, augmente d'une unité.

Démonstration. Chaque appel effectue un travail constant (comparer trois valeurs, faire au plus un échange) puis se relance, le cas échéant, sur un fils, dont la hauteur est strictement plus petite. La suite des positions visitées est donc un chemin descendant dans l'arbre, de longueur au plus hauteur(p) : le coût total est $O(\text{hauteur}(p))$. Comme tout nœud d'un arbre presque complet à n nœuds a une hauteur $\leqslant \lfloor \log_2 n \rfloor$ (proposition 11.1.7), ce coût est $O(\log n)$. □

La borne en $O(\text{hauteur}(p))$ est plus précise que le simple $O(\log n)$, et c'est elle — pas sa version affaiblie — qui nous permettra d'obtenir une bonne borne sur la construction du tas.

11.2.2 Construire le tas du bas vers le haut

Pour organiser *tout* le tableau en tas, il suffit d'appeler `entasser` en chaque position, mais dans le bon ordre : *du dernier nœud interne vers la racine*. Ainsi, lorsqu'on traite une position p, ses deux fils ont déjà été traités et sont à la racine de sous-arbres organisés en tas — la précondition de `entasser` est satisfaite.

Inutile de traiter les feuilles : un sous-arbre réduit à une feuille est déjà un tas. Or les feuilles sont exactement les nœuds d'indices $\lfloor n/2 \rfloor, \dots, n-1$ (un nœud p est une feuille si et seulement si $2p+1 \geqslant n$). On part donc du dernier nœud interne, d'indice $\lfloor n/2 \rfloor - 1$, et l'on remonte jusqu'à la position 0.

```python
from entasser import entasser


def construire_tas(A):
    """Permute les éléments de ``A`` pour l'organiser en tas, et le renvoie. On
    appelle ``entasser`` sur chaque nœud interne, du dernier vers la racine : à
    chaque appel, les deux sous-arbres fils sont déjà des tas. Les nœuds
    d'indices ``len(A) // 2`` à ``len(A) - 1`` sont des feuilles, qu'il est
    inutile de traiter.

    >>> construire_tas([5, 1, 3, 6, 4, 7, 2])
    [7, 6, 5, 1, 4, 3, 2]
    >>> construire_tas([])
    []
    """
    n = len(A)
    for p in range(n // 2 - 1, -1, -1):
        entasser(A, p, n)
    return A
```

Proposition 11.2.3 (Correction de `construire_tas`). *Après l'exécution de* `construire_tas`(A)*, le tableau A est organisé en tas.*

Démonstration. Montrons, par récurrence décroissante sur p, l'invariant suivant : *à la fin de l'itération qui traite la position p, le tableau est organisé en tas en toute position $\geqslant p$.*

Avant la première itération, toutes les positions $\geqslant \lfloor n/2 \rfloor$ sont des feuilles, donc organisées en tas. Lorsqu'on traite une position p, ses deux fils $2p+1$ et $2p+2$ sont d'indices strictement plus grands que p, donc — par l'invariant à l'étape précédente — déjà racines de sous-arbres organisés en tas. La précondition de `entasser` (proposition 11.2.1) est satisfaite : après l'appel, le tableau est organisé en tas à la position p, et il l'est resté en toute position $> p$ (que `entasser`(A, p, n) n'a pu modifier qu'à l'intérieur du sous-arbre de p, qu'elle a précisément rendu conforme). L'invariant est donc préservé.

À la fin de la boucle ($p = 0$), le tableau est organisé en tas à la position 0, c'est-à-dire organisé en tas. □

11.2.3 Un coût linéaire

On pourrait croire la construction en $O(n \log n)$: il y a $O(n)$ appels à `entasser`, chacun de coût $O(\log n)$. Cette borne est correcte, mais grossière. La plupart des nœuds sont *près des feuilles*, donc de très petite hauteur, et une analyse plus fine donne en réalité une borne *linéaire*.

Proposition 11.2.4 (Coût de `construire_tas`). *La construction d'un tas de taille n par `construire_tas` a une complexité $O(n)$.*

Démonstration. Posons $H = \lfloor \log_2 n \rfloor$, la hauteur de la racine (proposition 11.1.7). Un nœud à la profondeur d a une hauteur au plus $H - d$; et il y a au plus 2^d nœuds à la profondeur d. D'après la borne fine de `entasser` (proposition 11.2.2), le coût total est donc majoré par

$$O\left(\sum_{d=0}^{H} 2^d \, (H-d)\right).$$

Posons $i = H - d$ dans la somme :

$$\sum_{d=0}^{H} 2^d \, (H-d) = \sum_{i=0}^{H} 2^{H-i} \, i = 2^H \sum_{i=0}^{H} \frac{i}{2^i} \leqslant 2^H \sum_{i=0}^{+\infty} \frac{i}{2^i}.$$

La série $\sum_{i \geqslant 0} i \, 2^{-i}$ converge (son terme général décroît géométriquement : $\frac{i}{2^i} \leqslant (2/3)^i$ pour i assez grand) ; sa somme vaut 2, mais seule importe ici sa *finitude*. Le coût est donc $O(2^H) = O(2^{\log_2 n}) = O(n)$. □

Remarque 11.2.5. *Ce résultat peut surprendre :* organiser *n valeurs en tas coûte moins cher que les* trier *! Il n'y a pourtant pas de contradiction avec la borne $\Omega(n \log n)$ des tris par comparaison (chapitre 6) : un tas n'est pas un tableau trié. Il ne garantit que la domination locale de chaque nœud sur ses fils, ce qui est bien plus faible qu'un ordre total. Nous verrons à la section 11.4 comment passer du tas au tableau trié, et c'est cette seconde phase qui coûtera les $O(n \log n)$ attendus.*

Exemple 11.2.6. *Reprenons le tableau* $[\,5,\ 1,\ 3,\ 6,\ 4,\ 7,\ 2\,]$ *de l'exemple 11.1.5, qui n'était pas un tas. Il a* 7 *éléments, donc* $\lfloor 7/2 \rfloor - 1 = 2$ *est le dernier nœud interne ; on appelle* `entasser` *en positions* 2, 1 *puis* 0. *En position* 2, *la valeur* 3 *est dépassée par son fils* 7 *(indice* 5*) : on les échange. En position* 1, *la valeur* 1 *est dépassée par son fils* 6 *(indice* 3*) : on les échange. En position* 0 *enfin, la valeur* 5 *« remonte » le maximum* 7 *jusqu'à la racine. On obtient* $[\,7,\ 6,\ 5,\ 1,\ 4,\ 3,\ 2\,]$, *qui est bien un tas.*

11.2.4 Exercices

Exercice 11.2.7. *Déroulez à la main l'appel* `entasser`$(A, 0, 7)$ *sur le tableau* $A = [\,2,\ 8,\ 6,\ 1,\ 10,\ 9,\ 3\,]$ *(les deux sous-arbres issus de la racine sont-ils bien des tas ?). Indiquez le tableau obtenu après chaque échange.*

Exercice 11.2.8. *Réécrivez* `entasser` *sous forme* itérative *(sans récursion), à l'aide d'une boucle* `while` *qui fait descendre la valeur tant qu'un de ses fils la dépasse. Vérifiez qu'elle produit le même résultat que la version récursive. Quel est l'intérêt de cette version pour l'occupation mémoire ?*

Exercice 11.2.9. *Justifiez précisément que les nœuds d'indices* $\lfloor n/2 \rfloor, \ldots, n-1$ *sont exactement les feuilles de l'arbre. Que se passerait-il, sur le plan de la* correction, *si l'on faisait par erreur démarrer la boucle de* `construire_tas` *à l'indice* $n-1$ *plutôt qu'à* $\lfloor n/2 \rfloor - 1$ *? Et sur le plan du* coût *?*

Exercice 11.2.10. *On pourrait songer à construire le tas* du haut vers le bas *(en traitant les positions* $0, 1, \ldots, n-1$ *dans cet ordre). Montrez sur un petit exemple que cette stratégie ne fonctionne pas : la précondition de* `entasser` *(les deux sous-arbres sont déjà des tas) n'est alors plus garantie.*

11.3 Extraction du maximum et insertion

Nous savons maintenant construire un tas. Reste à réaliser les deux opérations qui font d'un tas une *file de priorité* : extraire l'élément de priorité maximale, et insérer un nouvel élément. Nous allons voir que toutes deux se font en $O(\log n)$, en « réparant » la propriété de tas le long d'un unique chemin de l'arbre.

11.3.1 Le maximum est à la racine

Commençons par justifier l'affirmation annoncée à la section 11.1 : dans un tas, le plus grand élément occupe la racine.

Lemme 11.3.1 (Le maximum est en position 0)**.** *Si* $A[0:t]$ *est organisé en tas, alors* $A[0] \geqslant A[j]$ *pour toute position* j *du tas. Autrement dit, le maximum du tas se trouve en position* 0.

Démonstration. Soit j une position du tas. Considérons le chemin qui remonte de j jusqu'à la racine : j, puis son père, puis le père de son père, et ainsi de suite jusqu'à la

position 0. À chaque pas de ce chemin, on passe d'un nœud à son père ; or la propriété de tas affirme exactement que la valeur d'un nœud est inférieure ou égale à celle de son père. Les valeurs rencontrées le long du chemin sont donc croissantes, et en particulier $A[j] \leqslant A[0]$. □

11.3.2 Extraction du maximum

Pour extraire le maximum, on le lit en position 0. Mais on ne peut pas simplement « retirer » la racine : il faut reboucher le trou et conserver un tas. L'astuce est de remplacer la racine par le *dernier* élément du tas (celui qui occupe la dernière feuille), de diminuer la taille du tas d'une unité, puis de faire « redescendre » cette valeur à sa place avec `entasser`.

```
from entasser import entasser


def extraction_max(A, taille):
    """Retire et renvoie le maximum du tas ``A[0:taille]``. On échange la racine
    (le maximum) avec la dernière case du tas, on diminue la taille d'une unité -
    ce qui sort l'ancien maximum du tas, désormais rangé en fin de tableau - puis
    on rétablit la propriété de tas à la racine avec ``entasser``. Renvoie le
    couple (maximum, nouvelle taille). Lève une erreur si le tas est vide.

    >>> A = [7, 6, 5, 1, 4, 3, 2]
    >>> extraction_max(A, 7)
    (7, 6)
    >>> A
    [6, 4, 5, 1, 2, 3, 7]
    >>> extraction_max([], 0)
    Traceback (most recent call last):
        ...
    ValueError: extraction dans un tas vide
    """
    if taille == 0:
        raise ValueError("extraction dans un tas vide")
    A[0], A[taille - 1] = A[taille - 1], A[0]
    entasser(A, 0, taille - 1)
    return A[taille - 1], taille - 1
```

Concrètement, on échange $A[0]$ et $A[t-1]$ — l'ancien maximum se retrouve ainsi rangé en position $t-1$, juste *au-delà* du tas rétréci, ce qui sera précieux pour le tri par tas (section 11.4) — puis on appelle `entasser` sur la racine pour le tas de taille $t-1$.

Avertissement 11.3.2. *On ne peut pas extraire d'un* tas vide *($t = 0$). La fonction commence donc par ce cas et lève une erreur : c'est une* précondition *de l'extraction, au même titre que « la liste est non vide » pour la recherche du minimum (section 2.1).*

Proposition 11.3.3 (Correction de l'extraction)**.** *Appelée sur un tas $A[0:t]$ non vide, la fonction `extraction_max` renvoie le maximum du tas, et après son exécution le sous-tableau $A[0:t-1]$ est organisé en tas.*

Démonstration. Par le lemme 11.3.1, la valeur initialement en position 0 est le maximum du tas ; l'échange la place en position $t-1$, d'où elle est renvoyée. Il reste à voir qu'après l'appel, $A[0:t-1]$ est un tas. Après l'échange, les deux sous-arbres enracinés aux positions 1 et 2 n'ont pas été modifiés (l'échange n'a touché que les positions 0 et $t-1$, et $t-1$ est une feuille pour le tas réduit) : ils sont donc toujours organisés en tas pour la taille $t-1$. La précondition de `entasser` (proposition 11.2.1) est ainsi satisfaite, et l'appel `entasser`$(A, 0, t-1)$ rend $A[0:t-1]$ organisé en tas. □

Proposition 11.3.4 (Coût de l'extraction)**.** *L'extraction du maximum d'un tas de taille n s'effectue en $O(\log n)$.*

Démonstration. L'échange et la mise à jour de la taille coûtent $O(1)$; le seul travail notable est l'appel à `entasser`, de coût $O(\log n)$ (proposition 11.2.2). □

11.3.3 Insertion

L'insertion d'une priorité v procède en miroir de l'extraction. On place v dans la première case libre — la nouvelle dernière feuille, en position t — ce qui agrandit le tas d'une unité ; puis on fait « remonter » cette valeur tant qu'elle dépasse son père, jusqu'à ce qu'elle trouve sa place.

Plutôt que d'échanger v avec son père à chaque étape, on procède plus économiquement : on creuse un « trou » à la position de v et on le fait monter en y faisant *descendre* les pères trop petits ; quand le père est assez grand (ou qu'on a atteint la racine), on dépose v dans le trou.

```
def inserer(A, taille, v):
    """Insère la priorité ``v`` dans le tas ``A[0:taille]`` et renvoie la
    nouvelle taille. La valeur prend place dans la première case libre, puis
    « remonte » tant qu'elle dépasse son père : on creuse un trou que l'on fait
    monter en y faisant descendre les pères trop petits, et l'on y dépose ``v``.
    Le tableau doit avoir une case libre, c'est-à-dire ``taille < len(A)``.

    >>> A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1, None]
    >>> inserer(A, 10, 15)
    11
    >>> A
    [16, 15, 10, 8, 14, 9, 3, 2, 4, 1, 7]
    >>> inserer([5, 3, 1], 3, 9)
    Traceback (most recent call last):
        ...
    ValueError: insertion dans un tas plein
    """
```

```
    if taille == len(A):
        raise ValueError("insertion dans un tas plein")
    i = taille
    while i > 0 and A[(i - 1) // 2] < v:
        A[i] = A[(i - 1) // 2]
        i = (i - 1) // 2
    A[i] = v
    return taille + 1
```

Avertissement 11.3.5. *L'insertion suppose qu'il reste une case libre, c'est-à-dire $t <$ `len`(A). Si le tableau est déjà* plein *($t =$ `len`(A)), il n'y a pas de position t où loger le nouvel élément : la fonction lève alors une erreur. (Avec une liste Python, on pourrait aussi agrandir le tableau par* `append`*; nous conservons ici un tableau de capacité fixe pour bien distinguer la* taille *du tas de la* longueur *du tableau, avertissement 11.1.8.)*

Cette correction est souvent laissée en exercice ; démontrons-la, car c'est l'occasion de voir un invariant de boucle sur une opération qui *monte* dans l'arbre, là où `entasser` descendait.

Proposition 11.3.6 (Correction de l'insertion)**.** *Si $A[0:t]$ est organisé en tas et $t <$ `len`(A), alors après* `inserer`(A, t, v) *le sous-tableau $A[0:t+1]$ est organisé en tas et contient, en plus des éléments initiaux, la valeur v.*

Démonstration. Que les valeurs présentes soient les anciennes plus v est clair : la boucle ne fait que recopier des valeurs déjà présentes vers le bas, et la seule valeur nouvellement écrite est le v final. Montrons que $A[0:t+1]$ est un tas, à l'aide de l'invariant suivant, où i désigne la position du « trou » courant.

Invariant. Au début de chaque itération, si l'on écrivait v dans le trou i, le tableau $A[0:t+1]$ serait organisé en tas, à la seule exception possible de l'arête reliant le trou i à son père : la valeur v pourrait y dépasser celle de son père. Toutes les autres arêtes sont conformes, et le sous-arbre situé sous i est un tas de valeurs toutes $\leqslant v$.

Initialisation. Avant la boucle, $i = t$: le trou est la nouvelle feuille. Le tas initial $A[0:t]$ étant conforme et la position t n'ayant pas de fils, écrire v en t ne crée qu'une seule arête nouvelle, celle entre t et son père. L'invariant est vérifié.

Hérédité. Supposons l'invariant vrai au trou i, et la boucle s'exécute : c'est que $i > 0$ et $A[\lfloor (i-1)/2 \rfloor] < v$. Notons $q = \lfloor (i-1)/2 \rfloor$ le père. Le corps recopie $A[q]$ dans le trou ($A[i] \leftarrow A[q]$) puis fait remonter le trou en q. Vérifions l'invariant au nouveau trou q.

— *Arête entre q et l'ancien trou i* : le père (futur v) doit dominer l'enfant, qui vaut désormais $A[q]$; or la condition de boucle donne précisément $A[q] < v$.

— *Arête entre q et l'autre fils* (le frère de i) : ce frère est inchangé. Comme l'invariant garantissait la conformité en q (l'arête $(q, \text{frère})$ ne touche pas le trou i), on avait $A[q] \geqslant$ frère ; et le futur père v vérifie $v > A[q] \geqslant$ frère.

- *Sous-arbre sous l'ancien trou i* : on y a écrit $A[q]$. Dans le tas initial, $A[q]$ dominait déjà tous ses descendants, donc en particulier les fils de i : le sous-arbre enraciné en i reste un tas.
- Les autres arêtes sont inchangées et restaient conformes.

La seule arête éventuellement en défaut est désormais celle entre le nouveau trou q et *son* père, et le sous-arbre sous q est un tas de valeurs $\leqslant v$: l'invariant est préservé.

Terminaison et conclusion. La boucle s'arrête soit parce que $i = 0$ (le trou est la racine, qui n'a pas de père : aucune arête en défaut), soit parce que $A[\lfloor (i-1)/2 \rfloor] \geqslant v$ (l'arête vers le père est alors conforme). Dans les deux cas, après l'écriture finale $A[i] \leftarrow v$, *toutes* les arêtes sont conformes : $A[0 : t+1]$ est organisé en tas. □

Proposition 11.3.7 (Coût de l'insertion)**.** *L'insertion dans un tas de taille n s'effectue en $O(\log n)$.*

Démonstration. À chaque itération, le trou remonte d'un niveau vers la racine ; le nombre d'itérations est donc majoré par la hauteur de l'arbre, soit $\lfloor \log_2 n \rfloor$ (proposition 11.1.7). Chaque itération coûte $O(1)$, d'où un coût total $O(\log n)$. □

Exemple 11.3.8. *Insérons la priorité* 15 *dans le tas* $A = [\,16, 14, 10, 8, 7, 9, 3, 2, 4, 1\,]$ *de taille* 10*, le tableau ayant une case libre. La valeur* 15 *part de la position* 10*. Son père (position* $\lfloor 9/2 \rfloor = 4$*) vaut* $7 < 15$ *: on fait descendre* 7*, le trou monte en* 4*. Le nouveau père (position* $\lfloor 3/2 \rfloor = 1$*) vaut* $14 < 15$ *: on fait descendre* 14*, le trou monte en* 1*. Le père (position* 0*) vaut* $16 \geqslant 15$ *: on s'arrête et l'on dépose* 15 *en position* 1*. On obtient* $[\,16, 15, 10, 8, 14, 9, 3, 2, 4, 1, 7\,]$*, qui est bien un tas.*

11.3.4 Exercices

Exercice 11.3.9. *Partez du tas* $[\,16, 14, 10, 8, 7, 9, 3, 2, 4, 1\,]$ *(taille* 10*) et effectuez deux extractions du maximum successives. Donnez le tableau et la taille après chaque extraction, et vérifiez à chaque fois que le préfixe actif est un tas.*

Exercice 11.3.10. *Écrivez une fonction* `augmenter_priorite(A, i, v)` *qui remplace la valeur en position i par une priorité v* supérieure ou égale *à l'ancienne, et rétablit la propriété de tas. (Indication : la valeur ne peut que devenir trop grande pour son père ; faites-la remonter comme à l'insertion.) Pourquoi l'opération symétrique —* diminuer *une priorité — demanderait-elle au contraire un* `entasser` *?*

Exercice 11.3.11. *Réécrivez l'insertion en* échangeant *à chaque étape la valeur avec son père (au lieu de décaler les pères vers le bas). Vérifiez qu'on obtient le même tas. Combien d'écritures dans le tableau chaque version fait-elle par niveau parcouru ?*

Exercice 11.3.12. *On peut construire un tas en partant du tas vide et en insérant les n éléments un à un. Quelle est la complexité de cette méthode dans le pire cas ? Comparez-la à celle de* `construire_tas` *(proposition 11.2.4), et concluez sur la méthode à préférer lorsque tous les éléments sont connus d'avance.*

11.4 Tri par tas

Nous avons promis, en ouvrant ce chapitre, qu'un tas fournirait un second algorithme de tri en $O(n \log n)$, à côté du tri fusion (chapitre 6). Le voici. L'idée est limpide une fois qu'on dispose d'un tas : organiser le tableau en tas place le maximum à la racine ; il suffit alors de *retirer le maximum* à répétition pour vider le tas par ordre décroissant — et donc remplir le tableau par ordre croissant, de la fin vers le début.

11.4.1 L'algorithme

C'est exactement le mécanisme de l'extraction du maximum (section 11.3), déroulé en place. On organise d'abord le tableau en tas. Puis, à chaque tour, on échange la racine $A[0]$ (le maximum du tas) avec la dernière case du tas, on réduit la taille du tas d'une unité — la valeur sortie est ainsi rangée *définitivement* à sa place, juste après le tas — et l'on rétablit la propriété de tas à la racine avec `entasser`.

```
from construire_tas import construire_tas
from entasser import entasser


def tri_par_tas(A):
    """Trie ``A`` par ordre croissant, en place, et le renvoie. On organise
    d'abord ``A`` en tas, puis on extrait le maximum à répétition : à chaque
    tour, on échange la racine (le maximum du tas) avec la dernière case du tas,
    on réduit le tas d'une unité - l'élément sorti se range ainsi définitivement
    en fin de tableau - et on rétablit la propriété de tas avec ``entasser``.

    >>> tri_par_tas([5, 1, 3, 6, 4, 7, 2])
    [1, 2, 3, 4, 5, 6, 7]
    >>> tri_par_tas([])
    []
    >>> tri_par_tas([42])
    [42]
    >>> tri_par_tas([3, 1, 2, 1, 3])
    [1, 1, 2, 3, 3]
    """
    construire_tas(A)
    for i in range(len(A) - 1, 0, -1):
        A[0], A[i] = A[i], A[0]
        entasser(A, 0, i)
    return A
```

Remarquons que le tas *rétrécit* d'un cran à chaque tour, tandis que la zone triée, à sa droite, *grandit* d'autant : le tableau se range donc en place, sans aucune liste auxiliaire.

Proposition 11.4.1 (Correction du tri par tas)**.** *Après l'exécution de* `tri_par_tas`(A)*, le tableau* A *est trié par ordre croissant.*

Démonstration. Notons n la longueur de A. Après l'appel à `construire_tas`, le tableau $A[0:n]$ est organisé en tas (proposition 11.2.3). La correction repose alors sur l'invariant de boucle suivant.

Invariant. Au début de l'itération d'indice i*, le sous-tableau* $A[i+1:n]$ *contient les* $n-1-i$ *plus grands éléments du tableau, déjà triés par ordre croissant, et* $A[0:i+1]$ *est organisé en tas et contient les* $i+1$ *autres éléments.*

Initialisation. Avant le premier tour ($i = n-1$), la zone triée $A[n:n]$ est vide (elle contient bien les 0 plus grands éléments), et $A[0:n]$ est le tas que l'on vient de construire.

Hérédité. Supposons l'invariant vrai au début de l'itération i. Comme $A[0:i+1]$ est un tas, sa racine $A[0]$ en est le maximum (lemme 11.3.1) ; et comme $A[i+1:n]$ contient déjà les $n-1-i$ plus grands éléments du tableau, ce maximum $A[0]$ est le plus grand des éléments restants, c'est-à-dire le $(i+1)$-ième plus petit du tableau. L'échange de $A[0]$ et $A[i]$ le range en position i : la zone $A[i:n]$ contient maintenant les $n-i$ plus grands éléments, et elle est triée (l'élément placé en i est inférieur ou égal à tous ceux de $A[i+1:n]$, eux-mêmes déjà triés). Il reste à voir que $A[0:i]$ redevient un tas : l'échange n'a perturbé la propriété de tas qu'à la racine (les deux sous-arbres de 0 sont restés des tas), donc l'appel `entasser`$(A, 0, i)$ rétablit le tas (proposition 11.2.1). L'invariant est donc vrai au début de l'itération $i-1$.

Conclusion. La boucle s'arrête après l'itération $i = 1$. L'invariant au début de cette dernière itération, et l'effet de celle-ci, donnent : $A[1:n]$ contient les $n-1$ plus grands éléments triés, et $A[0]$ l'unique élément restant — nécessairement le plus petit. Le tableau $A[0:n]$ est donc trié par ordre croissant. □

11.4.2 Complexité

Proposition 11.4.2 (Coût du tri par tas)**.** *Le tri par tas trie un tableau de* n *éléments en* $O(n \log n)$ *opérations.*

Démonstration. La construction du tas coûte $O(n)$ (proposition 11.2.4). La boucle effectue ensuite $n-1$ tours ; chaque tour fait un échange en $O(1)$ et un appel à `entasser` sur un tas de taille au plus n, de coût $O(\log n)$ (proposition 11.2.2). Le coût total est donc $O(n) + (n-1) \cdot O(\log n) = O(n \log n)$. □

Exemple 11.4.3. *Trions* [5, 1, 3, 6, 4, 7, 2]*. La construction du tas (exemple de la section 11.2) donne* [7, 6, 5, 1, 4, 3, 2]*. Puis, à chaque tour, le maximum courant (en gras) part se ranger en fin de zone active :*

tas construit	[**7**, 6, 5, 1, 4, 3, 2]
après le tour $i = 6$	[**6**, 4, 5, 1, 2, 3, | 7]
après le tour $i = 5$	[**5**, 4, 3, 1, 2, | 6, 7]
après le tour $i = 4$	[**4**, 2, 3, 1, | 5, 6, 7]
après le tour $i = 3$	[**3**, 2, 1, | 4, 5, 6, 7]
après le tour $i = 2$	[**2**, 1, | 3, 4, 5, 6, 7]
après le tour $i = 1$	[1, | 2, 3, 4, 5, 6, 7]

La barre verticale sépare le tas (à gauche) de la zone déjà triée (à droite). On lit le tableau trié sur la dernière ligne.

11.4.3 Un tri en place

Remarque 11.4.4. *Le tri par tas partage avec le tri fusion la complexité optimale* $O(n \log n)$ *(la borne inférieure* $\Omega(n \log n)$ *des tris par comparaison, chapitre 6, vaut pour les deux). Mais il a sur lui un avantage : il trie* en place*, sans allouer de seconde liste, là où le tri fusion recopie les éléments dans un tableau auxiliaire. À condition d'écrire* `entasser` *de façon* itérative *(exercice 11.2.8), le tri par tas n'utilise qu'un nombre* constant *de variables auxiliaires — la version récursive d'*`entasser`*, elle, occupe* $O(\log n)$ *cases de pile d'appels. En contrepartie, le tri par tas n'est pas* stable *: il peut inverser l'ordre relatif de deux éléments égaux (exercice 11.4.8).*

Ainsi s'achève notre étude des tas : une seule structure — un tableau lu comme un arbre — nous aura donné à la fois une file de priorité efficace et un tri en place optimal.

11.4.4 Exercices

Exercice 11.4.5. *Déroulez le tri par tas à la main sur* [3, 8, 2, 5, 1, 4]*. Donnez le tableau après la construction du tas, puis après chacun des cinq tours de la boucle.*

Exercice 11.4.6. *En réutilisant la version itérative d'*`entasser` *(exercice 11.2.8), écrivez un tri par tas qui n'utilise qu'un nombre constant de variables auxiliaires. Vérifiez qu'il trie correctement quelques tableaux, y compris avec des valeurs répétées.*

Exercice 11.4.7. *Écrivez une fonction* `k_plus_grands(L, k)` *qui renvoie les* k *plus grands éléments de* L*, par ordre décroissant, à l'aide d'un tas. Quelle est sa complexité en fonction de* n *et* k *? Pour quelles valeurs de* k *est-elle préférable à un tri complet suivi d'une lecture des* k *derniers éléments ?*

Exercice 11.4.8. *On trie par tas la liste de couples* $[(1, a), (1, b)]$ *en ne comparant que les premières composantes (toutes deux égales à* 1*). Suivez l'exécution et constatez que l'ordre relatif de* $(1, a)$ *et* $(1, b)$ *peut être inversé : le tri par tas n'est pas* stable*. (On rappelle qu'un tri est* stable *s'il préserve l'ordre d'arrivée des éléments de même clé.)*

Exercice 11.4.9. *Comment trier un tableau par ordre* décroissant *avec un tas ? (Indication : quelle propriété de tas faut-il imposer pour que le* minimum *se retrouve à la racine ?)*

Index

A
accumulateur, 54
adjacence, 112
algorithme, 5, 6
  adaptatif, 38
  Bellman-Ford, 141
  Dijkstra, 151
  exponentiel, 20
  polynomial, 20
  produit de matrices, 23
algorithme de Kahn, 132
algorithme de Knuth-Morris-Pratt, 169
algorithme d'Euclide
  décroissance des restes, 53
  dérécursivation, 53
  optimalité, 54
algorithme naïf
  recherche de motif, 160
alphabet, 99, 158
arbre, 57
  des appels récursifs, 44
  descendant, 173
  feuille, 57
  hauteur d'un nœud, 174
  nombre de feuilles, 59
  nœud interne, 57
arbre binaire
  hauteur, 174
  numérotation des nœuds, 172
  presque complet, 172
arbre de décision, 76
arc, 115
arête, 112
automate, 165
  complexité, 169
  précalcul, 166
    optimisation, 169
  transition, 166

B
balayage, 33
Bellman-Ford
  cas général, 146
  chemin élémentaire, 140
  circuit négatif accessible, 147
  complexité, 142
    cas général, 146
  contraintes de différences, 154
  correction, 141
  exemple, 141
  poids négatifs sans circuit, 147
  variante DAG, 143
borne inférieure, 24, 25
  tris, 77
borne optimale, 24, 25

C
cas de base, 42
cas récursif, 46
chaîne, 116
chemin, 113, 116
  élémentaire, 116, 140
    cas général, 144
circuit, 116, 127, 130
  de poids négatif, 136
    accessible, 147
    détection, 146
    impact sur les chemins, 144
  détection, 128, 129
  élémentaire, 116, 127
coefficient binomial, 94
collision, 162
comparaison, 28

comparaison de lettres, 160
complexité, 6, 21–23, 119, 122
    exponentielle, 100
    Karatsuba, 86
    minoration, 24
    pire cas, 22
    pseudo-polynomiale, 110
    tri fusion, 69
    tri par comptage, 81
composante connexe, 123, 124, 126
condition d'arrêt, 46
connexité, 123, 124, 126
`construire_tas`
    complexité, 179
    correction, 178
contraintes de différences, 153
    exemple, 155
    faisabilité, 154
    planification, 153
correction, 7
cycle, 113

D

DAG, 142
    plus courts chemins, 143
    poids négatifs, 143
débordement de pile, 46
décalage, 158
degré, 112
    somme des degrés, 112
    sortant, 115, 127
dérécursivation, 51
Dijkstra
    complexité, 151
    correction, 151
    invariant, 149
    poids négatifs
        échec, 151
distance, 120
    dans un graphe pondéré, 136
    non définie par circuit négatif, 136
domination, 18

E

élément minimal, 95
empreinte
    d'un mot, 161
`entasser`
    complexité, 177
    correction, 177
expression arithmétique, 59

F

facteur, 99
    d'un mot, 158
factorielle
    correction, 43
faisabilité, 153
    système faisable, 153
faux positif, 162
file, 120
file de priorité, 171
    extraction du maximum, 182
    insertion, 183
    maximum, 180
fonction récursive, 41
fusion, 63
    invariant, 65

G

grand O, 18, 19
    dualité, 19
    transitivité, 19
grand Ω, 18
    dualité, 19
grand Θ, 18–20, 25
graphe, 112
    acyclique, 127, 129
    biparti, 113
    biparti complet, 113
    complet, 113
    connexe, 124
    non orienté, 112
    orienté, 115
    orienté pondéré, 136
    pondéré
        représentation machine, 137

H
hachage, 161
- empreinte, 161

hachage glissant
- exemple, 162

hauteur
- d'un nœud, 174

I
implémentation, 6
incidence, 112
indexation, 7
invariant de boucle, 6, 12–16, 65, 118
- Dijkstra, 149

K
Karatsuba
- algorithme, 86
- identité, 84

Karp-Rabin
- analyse heuristique, 164
- précondition, 164

L
LCS, 99
lemme de relâchement, 138
lemme des poignées de main, 112
liste, 7, 9
liste d'adjacence, 114
logarithme binaire, 31
longueur
- d'un chemin, 136

M
matrice d'adjacence, 114
médiane, 87
médiane des médianes, 88, 90
mémoïsation, 47, 96
- coût, 49
- généralisation, 49

mot, 99, 158
motif, 158
- vide, 160

N
NP-complet, 110

O
occurrence
- d'un motif, 158

opération élémentaire, 17
optimalité des sous-structures, 100
optimisation, 106
ordre lexicographique, 28
ordre relatif, 28
ordre total, 28
ordre total strict, 28

P
parcours
- en largeur, 120–122
- en profondeur, 116–119, 129

partition, 71
permutation, 31
pile, 116
- d'appels, 43

pivot, 71
plus court chemin, 120
- source unique, 136

plus longue sous-suite commune, 99
- algorithme, 104
- applications, 100
- complexité, 105
- exemple, 99
- reconstruction, 105
- récurrence, 101

poids
- d'un arc, 136

polynôme, 20
précondition, 14
préfixe, 99
priorité
- dans une file de priorité, 171

produit de polynômes, 84
programmation dynamique, 49, 94
- optimalité des sous-structures, 100

programme, 6
propriété de tas, 173
- violation, 174

pseudocode, 8
pseudo-polynomialité, 110

puits, 127, 128

R
raisonnement par l'absurde, 95
rang, 87
recherche
  dichotomique, 30
  séquentielle, 28
recherche de motif
  algorithme naïf
    pire cas, 160
  exemple, 158
  précondition, 160
recherche par automate
  complexité, 169
  fonction sigma, 166
  précondition, 169
récurrence
  double, 95
récursion infinie, 46
récursion terminale, 50
  dérécursivation, 51
récursivité croisée, 55
récursivité mutuelle, 55, 56
  correction, 56
relâchement, 137
  exemple, 139
  lemme de, 138
  minorant de la distance, 138
relation d'équivalence, 123

S
sac à dos, 107
  contrainte, 107
  exemple, 107
  récurrence, 107
sélection
  pivot, 90
  temps linéaire, 90
σ_M (fonction), 166
  exemple, 166
  mise à jour, 166
sommet, 112
source, 136
sous-problèmes
  chevauchement, 94
sous-suite, 99, 158
sous-suite commune, 99
spécification, 9
successeur, 115
suffixe
  d'un mot, 165
suite
  Fibonacci, 49, 54
suite double
  unicité, 95

T
tableau, 7, 9
taille de l'entrée, 21
tas, 172
  construction, 178
    exemple, 180
  construction en temps linéaire, 179
  construction linéaire, 179
  extraction
    complexité, 182
    correction, 182
    précondition, 181
  hauteur logarithmique, 174
  indexation à partir de zéro, 173
  indices des fils et du père, 172
  insertion
    complexité, 184
    correction, 183
    exemple, 184
    précondition, 183
  maximum à la racine, 180
  opération entasser, 177
  tableau organisé en tas, 173
  taille du tas, 175
  taille vs longueur du tableau, 175
temps de calcul, 21
terminaison, 15, 16
  récursion, 46
texte, 158
transformation de Burrows–Wheeler, 100

transitivité, 28
tri, 31
  à bulles, 33
  en place, 32, 187
  fusion, 69
  par comparaison, 28, 76, 77
  par comptage, 81
  par insertion, 36, 37
  par sélection, 34, 35
  par tas, 186
    complexité, 186
    correction, 185
    en place, 187
    exemple, 186
  rapide, 71
  stable, 187
tri topologique, 130, 132
triangle de Pascal, 94
trichotomie, 28

V
variable de boucle, 97
voisinage, 112
  sortant, 115