

# INFO–F103 — Syllabus d'exercices

Martin Colot, Robin Petit & Cédric Simar


Correction des anciens exercices par :  
Jérôme Dossogne, François Gérard, Vincent Ho, Keno Merckx,  
Charlotte Nachtegael, Catharina Olsen, Nikita Veschikov

Remerciements à :  
Chloé Terwagne

Année académique 2023-2024



# Table des matières

<b>1</b>	<b>Rappels, notations, prérequis et digressions</b>	<b>1</b>
1.1	Définitions génériques . . . . .	1
1.2	Techniques de preuve . . . . .	1
1.2.1	Preuves universelles . . . . .	2
1.2.2	Preuves constructives . . . . .	2
1.2.3	Preuves par induction . . . . .	3
1.2.4	Preuves par énumération . . . . .	4
1.2.5	Preuves par contraposée . . . . .	5
1.2.6	Preuves par l'absurde . . . . .	5
1.3	Python et son interpréteur . . . . .	6
1.3.1	Norme vs implémentation de référence . . . . .	6
1.3.2	Strings et formatage . . . . .	7
1.3.3	Décorateurs . . . . .	8
1.3.4	Classes et types . . . . .	10
1.3.5	Encapsulation . . . . .	13
1.3.6	Fonction vs méthode, méthode liée vs non-liée . . . . .	15
1.3.7	Méthodes statiques . . . . .	17
1.3.8	Méthodes spéciales . . . . .	17
1.3.9	Propriétés . . . . .	26
1.3.10	Itérables, itérateurs et générateurs . . . . .	28
1.3.11	Adresses et cache de références interne à CPython . . . . .	32
1.3.12	Types standards et complexité . . . . .	37
1.3.13	Les types de CPython  . . . . .	40
1.4	Notions ensemblistes . . . . .	43
1.5	Notions asymptotiques de Landau . . . . .	46
1.5.1	Quelques ordres de grandeur . . . . .	49
1.6	Qualité d'un algorithme . . . . .	50
1.7	Considérations arithmétiques . . . . .	51
<b>2</b>	<b>Rappels théoriques</b>	<b>59</b>
2.1	Les ADTs . . . . .	59
2.1.1	Listes chaînées . . . . .	59
2.1.2	Piles ( <i>stacks</i> ) . . . . .	61
2.1.3	Files ( <i>queues</i> ) . . . . .	62
2.1.4	Files à double extrémité ( <i>deques</i> ) . . . . .	62
2.1.5	ADTs et complexité . . . . .	63
2.2	Récurtivité . . . . .	63
2.2.1	Récurtivité croisée . . . . .	64

2.2.2	Contexte des fonctions . . . . .	64
2.2.3	Diviser pour régner . . . . .	64
2.3	Backtracking . . . . .	65
2.3.1	Élagage ( <i>pruning</i> ) . . . . .	66
2.3.2	Recherche de solution optimale et <i>Branch and Bound</i> . . . . .	67
2.4	Arbres . . . . .	68
2.4.1	Implémentation . . . . .	70
2.4.2	Relations d'ordre ( $\leq$ ) . . . . .	71
2.5	Séquences triées . . . . .	71
2.5.1	Implémentation par vecteur . . . . .	71
2.5.2	Implémentation par liste chaînée bidirectionnelle . . . . .	73
2.5.3	Implémentation par BST . . . . .	73
2.6	Files à priorité (Heaps) . . . . .	77
2.6.1	File à priorité . . . . .	77
2.6.2	Implémentation via un heap . . . . .	77
2.6.3	Complexité . . . . .	79
2.6.4	Heapsort . . . . .	79
2.7	Les ensembles dynamiques et les dictionnaires . . . . .	81
2.7.1	Implémentation via un BST . . . . .	81
2.7.2	Implémentation via adressage direct . . . . .	81
2.7.3	Implémentation via une table de hachage . . . . .	82
2.7.4	Dictionnaires comme cas particuliers . . . . .	89
2.8	Graphes . . . . .	89
2.8.1	Représentation des graphes . . . . .	91
2.8.2	Matrice d'accessibilité . . . . .	91
2.8.3	Parcours de graphes . . . . .	94
2.9	Dérécursification . . . . .	95
2.9.1	Récursivité unaire . . . . .	96
2.9.2	Récursivité binaire . . . . .	97
2.9.3	Récursivité $n$ -aire . . . . .	97
2.10	Tris . . . . .	98
2.10.1	Tris naïfs . . . . .	98
2.10.2	Tri fusion . . . . .	101
2.10.3	Tri rapide . . . . .	103
2.10.4	Tri Shell . . . . .	106
2.11	Programmation dynamique . . . . .	108
<b>3</b>	<b>Séances de TP</b>	<b>111</b>
1	Analyse de complexité . . . . .	112
2	Les ADT (partie 1) . . . . .	116
3	Les ADT (partie 2) . . . . .	117
4	La récursivité (partie 1) . . . . .	120
5	La récursivité (partie 2) . . . . .	121
6	Le backtracking (partie 1) . . . . .	124
7	Le backtracking (partie 2) . . . . .	125
8	Arbres (partie 1) . . . . .	127
9	Arbres (partie 2) . . . . .	128
10	Séquences triées . . . . .	130
11	Files à priorité . . . . .	131

12	Hachage (partie 1) . . . . .	133
13	Hachage (partie 2) . . . . .	135
	13.1 Remarques sur le fonctionnement du hachage en Python . . . . .	137
14	Graphes (partie 1) . . . . .	140
15	Graphes (partie 2) . . . . .	141
16	Graphes (partie 3) . . . . .	142
17	Dérécursification . . . . .	146
18	Tris . . . . .	148
19	Programmation dynamique (partie 1) . . . . .	150
20	Programmation dynamique (partie 2) . . . . .	151
<b>4</b>	<b>Exercices cotés</b>	<b>153</b>
<b>5</b>	<b>Anciens examens</b>	<b>159</b>
1	Juin 2020 . . . . .	159
2	Août 2021 . . . . .	161
3	Printemps 2022 . . . . .	163
4	Juin 2022 . . . . .	164
5	Août 2022 . . . . .	166
6	Printemps 2023 . . . . .	168
7	Juin 2023 . . . . .	169
8	Août 2023 . . . . .	171

## Introduction

Ce document que vous avez devant les yeux, et que nous appellerons par la suite *correctif* ou *syllabus d'exercices* a été écrit dans sa première version pour la rentrée académique 2020-2021 dans le contexte des cours à distance à cause du Covid-19. Il a depuis été augmenté et complété afin d'y introduire des exercices supplémentaires (corrigés), des corrections d'anciens examens et bien sûr quelques digressions, prenant leur origine dans des discussions intéressantes entre les assistant(e)s et les étudiant(e)s.

Le **chapitre 1** de ce syllabus d'exercices contient des rappels (et extensions) de vos connaissances actuelles, que ce soit concernant le langage Python (c.f. la **section 1.3**), la notation  $\mathcal{O}$  de Landau et ses variations (c.f. la **section 1.5**), ou encore sur les méthodes de lecture et d'écriture de preuves en mathématique (c.f. la **section 1.2**). La **section 1.7** est quant à elle un condensé de résultats qui seront utilisés sporadiquement afin de déterminer précisément la complexité des solutions dans les exercices de TP et les exercices supplémentaires. Il ne vous est, bien entendu, pas demandé de lire cette section d'une traite, et encore moins d'en apprendre par cœur les démonstrations. Les résultats énoncés peuvent toutefois vous être utiles et les garder dans un coin de votre tête est probablement une bonne idée.


Le **chapitre 2** contient un rappel théorique pour chaque chapitre du cours lié à une ou plusieurs séance(s) de TPs. Nous vous invitons *très fortement* à lire ce rappel, afin de vous préparer, avant de venir en séance d'exercices. Le formalisme utilisé peut par moments légèrement différer de celui utilisé au cours théorique, mais les notions y sont bien sûr équivalentes.

Le **chapitre 3** contient tous les exercices qui seront vus et corrigés lors des séances d'exercices. Bien qu'absents de la table des matières, certains chapitres proposent également des exercices supplémentaires après les exercices faits en séance. Ces exercices sont, pour la plupart, à voir comme des exercices de formation pour les interrogations/examens car leur niveau de difficulté se situe quelque part entre les exercices faits en séances et des questions d'examen.

Le **chapitre 4** contient des exercices additionnels qui ont servi d'exercices cotés pendant l'année 2020-2021 (et qui avaient remplacé l'évaluation de printemps). Bien que ce fonctionnement n'ait été effectif que pendant cette année-là, ces exercices ayant été écrits et corrigés, nous les mettons à disposition dans ce correctif. Nous vous recommandons également de faire ces exercices en préparation à vos évaluations.

Le dernier chapitre (**chapitre 5**) contient une sélection de questions d'examen depuis la session de juin 2020 ainsi qu'une correction.

Si vous avez des questions sur le contenu de ce correctif, n'hésitez jamais à venir poser vos questions durant les séances d'exercices. Si vous pensez avoir trouvé une erreur (que ce soit une typo, une erreur dans un code, une erreur de raisonnement, ou toute autre erreur), n'hésitez pas à la signaler par mail, via Teams, ou encore lors des séances de TP.

En guise de conclusion à cette introduction, reprenez que ce document est là pour vous aider à suivre ce cours d'algorithmique I, pour vous montrer vers quoi l'algorithmique peut vous mener, piquer votre curiosité sur certains sujets mathématiques, mais n'est pas exhaustif sur la matière du cours. Cependant, à plusieurs reprises, des remarques et questions bonus sont proposées malgré le fait qu'elles dépassent le cadre strict du cours (le symbole  est

utilisé pour expliciter les digressions qui dépassent allègrement le cadre du cours, mais nous invitons bien entendu à lire toutes ces informations qui vous aideront à avoir une meilleure compréhension de l’algorithmique en tant que sujet d’étude). Dès lors, soyez curieux, soyez curieuses, et aimez l’algorithmique.





# Chapitre 1

## Rappels, notations, prérequis et digressions

### 1.1 Définitions génériques

**Définition 1.** Dans une structure de donnée, on appelle *donnée satellite* toute information qui n'est pas utilisée par les opérations proposées par la structure.

Par exemple dans le nœud de liste chaînée ci-dessous, la variable `next_node` sert à parcourir la structure, alors que la variable `data` n'est pas utilisée pour le parcours, mais est nécessaire (sinon la structure ne sert à rien) :

---

```
1 class Node:
2     def __init__(self, next_node, data):
3         self.next_node = next_node
4         self.data = data
```

---

**Définition 2.** Pour  $n \in \mathbb{N}$ , on définit la *factorielle* de  $n$  (que l'on note  $n!$ ) comme suit :

$$n! := \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{sinon.} \end{cases}$$

### 1.2 Techniques de preuve

En mathématique, nous pouvons regrouper les preuves en 6 grandes familles (différentes personnes feront des classifications différentes) :

1. les preuves universelles ;
2. les preuves constructives ;
3. les preuves par induction ;
4. les preuves par énumération ;
5. les preuves par contraposée ;
6. les preuves par l'absurde.

Bien sûr un résultat peut être prouvé en utilisant plusieurs de ces techniques. De plus, une preuve contient une séquence d'arguments permettant de se rapprocher de la conclusion voulue, et chacun de ces arguments peut être démontré individuellement avec des techniques différentes.

### 1.2.1 Preuves universelles

Le mot *universel* ici vient du quantificateur universel  $\forall$ . Nous utiliserons donc ici cette approche pour un résultat de la forme :

$$\forall x \in X : P(x)$$

où  $X$  est un ensemble quelconque et  $P(x)$  désigne le fait que l'élément  $x$  satisfait la propriété  $P$ .

L'approche universelle consiste en le fait de fixer un élément quelconque de  $X$  et, en utilisant uniquement les propriétés qui définissent  $X$ , de montrer que  $x$  satisfait bien la propriété  $P$ . Prenons un exemple :

| **Lemme 1.1.** *Si  $n$  est un nombre naturel pair, alors  $n + 1$  est impair.*

Démonstration. Soit  $n$  un tel nombre. Nous savons qu'il existe un nombre naturel  $k$  tel que  $n = 2k$ . Dès lors  $n + 1 = 2k + 1$ , i.e.  $n + 1$  est impair.  $\square$

### 1.2.2 Preuves constructives

Le mot *constructif* fait ici référence à la création d'un objet désiré dans le cadre du quantificateur d'existence  $\exists$ . Cette approche est donc orientée pour un résultat de la forme :

$$\exists x \in X \text{ s.t. } P(x).$$

Comme son nom l'indique, l'approche consiste donc en la création explicite d'un tel  $x$  et en le fait de montrer qu'il satisfait bien la propriété  $P$ . Prenons un exemple :

| **Lemme 1.2.** *Si  $(x_n)_n$  et  $(y_n)_n$  sont des suites réelles convergentes, alors la suite  $(z_n)_n$  définie par  $z_n = x_n + y_n$  est également convergente.*

Démonstration. Notons  $x$  et  $y$  les limites de  $x_n$  et  $y_n$  respectivement et montrons que  $z_n \xrightarrow[n \rightarrow +\infty]{} x + y$ .

Fixons  $\varepsilon > 0$ . Par définition de convergence, il existe  $N_1$  et  $N_2$  tels que si  $n > N_1$ , alors  $|x_n - x| \leq \varepsilon/2$  et si  $n > N_2$ , alors  $|y_n - y| \leq \varepsilon/2$ . Si nous notons  $N = \max\{N_1, N_2\}$ , pour tout  $n > N$ , nous savons que :

$$|z_n - (x + y)| = |x_n + y_n - x - y| = |x_n - x + y_n - y| \leq |x_n - x| + |y_n - y| \leq \varepsilon.$$

$\square$

| **Remarque.** *Le constructivisme est une école de pensée de la logique mathématique qui est fondée sur l'idée qu'une preuve n'est valide (ou du moins convaincante pour les moins orthodoxes) que si les objets manipulés sont constructibles explicitement. En particulier les constructivistes s'opposent (parfois farouchement) aux raisonnements par l'absurde, c.f. ci-dessous. Il est parfois également question d'intuitionnisme pour désigner cette école de pensée. En particulier, cette approche ne se base pas sur la loi d'exclusion de la double*

*négarion, i.e. en logique intuitionniste, il n'est pas vrai de dire que  $\neg\neg P \Rightarrow P$ . La loi du tiers exclus n'est pas non plus vérifiée en logique intuitionniste, i.e. il n'est pas vrai de dire que  $P$  est soit vrai soit faux (i.e. schématiquement  $P \vee \neg P$ <sup>a</sup>).*

a. Plus précisément la loi du tiers exclus dit  $\models P \vee \neg P$ , mais est-on à ça près ?

### 1.2.3 Preuves par induction

Les preuves par induction (également appelées preuves par récurrence) sont utilisées dans le cadre d'un résultat universel dont le paramètre est un nombre naturel (parfois entier), i.e. sous la forme :

$$\forall k \in \mathbb{N} : P(k).$$

Une telle preuve commence par montrer un *cas de base* et se continue en montrant que si le résultat est vrai pour une valeur inférieure à  $k$ , alors elle doit être vraie pour  $k$ . Notons qu'il y a deux formes différentes d'induction (mais qui sont en réalité équivalentes) : l'induction *forte* et l'induction *faible*.

Dans les deux cas, nous commençons par montrer un cas de base ( $k = k_0$ ), mais l'hypothèse d'induction diffère : dans l'induction faible, nous supposons que le résultat est vrai pour  $k$  et nous montrons qu'il l'est aussi pour  $k + 1$  alors que dans l'induction forte, nous supposons que le résultat est vrai pour toute valeur  $j$  telle que  $k_0 \leq j \leq k$  et nous montrons qu'il l'est aussi pour  $k + 1$ .

Une telle preuve a un intérêt car si le résultat est vrai pour  $k = 0$  et que  $P(k)$  implique  $P(k + 1)$ , alors nous savons  $P(0)$  (par le cas de base) mais également  $P(1)$  (puisque  $P(0)$ ) et  $P(2)$  (puisque  $P(1)$ ), etc.

Prenons un exemple d'induction faible :

**Lemme 1.3.** *Si  $x$  et  $y$  sont deux nombres réels et  $n$  est un nombre naturel, alors :*

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}.$$

Démonstration. Procédons par induction sur  $n$ . Prenons le cas de base  $n = 0$  : nous savons que  $(x + y)^0 = 1$  et que :

$$\sum_{k=0}^n \binom{n}{k} x^k y^{n-k} = \binom{0}{0} x^0 y^0 = 1.$$

Procédons maintenant au pas de récurrence : supposons que l'égalité soit vérifiée pour un certain  $n$  et montrons qu'elle est également vérifiée par  $n + 1$ . Pour cela rappelons-nous des égalités suivantes :

$$1 = \binom{n}{0} = \binom{n}{n} = \binom{n+1}{0} = \binom{n+1}{n+1} \quad \text{et} \quad \binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}.$$

Ensuite regardons :

$$(x + y)^{n+1} = (x + y)(x + y)^n = (x + y) \sum_{k=0}^n \binom{n}{k} x^k y^{n-k},$$

par hypothèse de récurrence. En distribuant, nous obtenons :

$$\begin{aligned}
 (x + y)^{n+1} &= \sum_{k=0}^n \binom{n}{k} x^{k+1} y^{n-k} + \sum_{k=0}^n \binom{n}{k} x^k y^{n-k+1} \\
 &= \sum_{k=1}^{n+1} \binom{n}{k-1} x^k y^{n+1-k} + \sum_{k=0}^n \binom{n}{k} x^k y^{n+1-k} \\
 &= x^{n+1} + y^{n+1} + \sum_{k=1}^n \left( \binom{n}{k-1} + \binom{n}{k} \right) x^k y^{n+1-k} \\
 &= \sum_{k=0}^{n+1} \binom{n+1}{k} x^k y^{n+1-k}.
 \end{aligned}$$

□

Prenons maintenant un exemple d'induction forte :

| **Lemme 1.4.** *Tout nombre naturel admet une représentation en base 2.*

*Démonstration.* Procédons à nouveau par induction et prenons  $n = 0$  comme cas de base. Il est en effet clair que 0 est une écriture valide en binaire.

Maintenant fixons  $n$  naturel et supposons que tout nombre  $k$  tel que  $0 \leq k \leq n$  admet une représentation binaire. Notons  $p$  la plus grande valeur telle que  $2^p \leq n$ , considérons la valeur  $m = n - 2^p \geq 0$ . Par hypothèse de récurrence, puisque  $m < n$ , nous savons que  $m$  peut s'écrire sous la forme :

$$m = \sum_{j=0}^{p-1} b_j 2^j,$$

pour  $b_0, \dots, b_{p-1} \in \{0, 1\}$  (ce qui est donc sa représentation en base 2). Sur base de cette expression, définissons la représentation de  $n$  par  $c_0, \dots, c_p \in \{0, 1\}$  définie par  $c_p = 1$  et  $c_j = b_j$  pour  $j < p$ . Cette représentation est bien celle de  $n$  puisque :

$$\sum_{j=0}^p c_j 2^j = \sum_{j=0}^{p-1} c_j 2^j + c_p 2^p = \sum_{j=0}^{p-1} b_j 2^j + 1 \cdot 2^p = m + 2^p = n.$$

□

**Remarque.** Bien que l'existence d'une représentation en base 2 (et même en n'importe quelle base  $b$  entière  $> 1$ ) se démontre aisément comme montré ci-dessus à l'aide de l'induction forte, nous pouvons également uniquement utiliser l'induction faible mais la preuve en devient plus longue car à  $n$  fixé, la seule représentation binaire de laquelle il est possible de repartir est celle de  $n - 1$  (il faut donc expliciter les reports, ce qui allonge légèrement la preuve).

#### 1.2.4 Preuves par énumération

Également appelées *preuves par analyse de cas*, les preuves par énumérations consistent, comme leur nom l'indique, à séparer le résultat à montrer en plusieurs sous-résultats et à les démontrer individuellement. Si les cas gérés couvrent bien tous les cas possibles, la preuve est valide puisque tous les cas sont démontrés. Prenons un exemple :

| **Lemme 1.5.** *Le carré de tout nombre naturel a la même parité que le nombre en question.*

Démonstration. Fixons  $n$  un nombre naturel. Analysons séparément le cas  $n$  pair et le cas  $n$  impair. Si  $n$  est pair, alors par définition il existe un entier  $k$  tel que  $n = 2k$ . Dès lors nous savons que  $n^2 = (2k)(2k) = 2(2k^2)$ , i.e.  $n^2$  est pair. Si maintenant  $n$  est impair, alors nous savons qu'il existe un entier  $k$  tel que  $n = 2k + 1$ . Dès lors :  $n^2 = (2k + 1)(2k + 1) = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$ , i.e.  $n^2$  est impair.  $\square$

| **Remarque.** *Il peut bien entendu y avoir plus de deux catégories à énumérer. Un résultat classique avec plus de catégories est le suivant : si  $p$  est un nombre premier tel que  $p > 3$ , alors soit  $p + 1$  est un multiple de 6, soit  $p - 1$  est un multiple de 6. Savez-vous décrire pourquoi ?*

### 1.2.5 Preuves par contraposée

La contraposée d'une implication est une forme d'*inverse* (mais pas de négation!) qui permet, dans une certaine mesure, de *retourner* l'implication. En effet, si nous considérons l'implication suivante  $P \Rightarrow Q$ , sa *contraposée* est l'implication  $\neg Q \Rightarrow \neg P$  (où  $\neg$  désigne la négation logique). Cette nouvelle implication a comme propriété très importante d'être en réalité strictement équivalente à la première (en effet la première est vraie si et seulement si la seconde l'est, ce qui peut s'observer, par exemple, via leur table de vérité respective). Il ne faut cependant surtout pas confondre la contraposée avec la *réciproque* (qui serait donc ici  $Q \Rightarrow P$ ) qui n'est en rien équivalente à  $P \Rightarrow Q$  !

Dans certains cas, trouver un argument pour montrer un résultat  $P \Rightarrow Q$  peut s'avérer assez compliqué, mais un argument pour  $\neg Q \Rightarrow \neg P$  peut se trouver plus facilement. Prenons un exemple :

| **Lemme 1.6.** *Si  $P : x \mapsto ax^2 + bx + c$  est un polynôme réel de degré 2 tel que, alors  $P$  admet au plus 2 racines.*

Démonstration. Procédons par contraposée et supposons que nous avons un polynôme  $P$  avec au moins 3 racines (notons-les arbitrairement  $x_0, x_1, x_2$ ). Nous savons donc que  $(x - x_i)$  divise  $P$  pour  $1 \leq i \leq 3$ . Dès lors il existe un unique polynôme  $Q$  de degré  $\geq 0$  (puisque  $P$  n'est pas identiquement nul) tel que :

$$\forall x \in \mathbb{R} : P(x) = \bar{P}(x)Q(x),$$

pour  $\bar{P}(x) = (x - x_1)(x - x_2)(x - x_3)$ . Or  $\deg \bar{P} = 3$  et donc  $\deg P = \deg \bar{P} + \deg Q \geq \deg \bar{P} = 3$ , i.e.  $P$  n'est pas de degré 2.  $\square$

### 1.2.6 Preuves par l'absurde

Un raisonnement par l'absurde peut s'utiliser lors de la démonstration d'un résultat sous la forme  $P \Rightarrow Q$  et consiste en le fait de supposer simultanément  $P$  (donc l'hypothèse du résultat) et  $\neg Q$  (donc la négation de la conclusion à laquelle on veut arriver), et d'arriver à une contradiction. Ainsi, il est impossible d'avoir à la fois  $P$  et  $\neg Q$ , ce qui est précisément la définition de l'implication  $P \Rightarrow Q$ .

Prenons un exemple :

| **Lemme 1.7.** *Le nombre  $\sqrt{2}$  est irrationnel.*

*Démonstration.* Supposons par l'absurde qu'il existe  $a, b \in \mathbb{Z}$  tels que  $\sqrt{2} = \frac{a}{b}$ . Sans perte de généralité, nous pouvons supposer que  $a$  et  $b$  sont premiers entre eux (i.e. si  $d$  divise  $a$  et  $b$ , alors  $d = 1$ ). Alors par définition de  $\sqrt{\cdot}$ , nous savons que :

$$2 = (\sqrt{2})^2 = \left(\frac{a}{b}\right)^2 = \frac{a^2}{b^2}.$$

En particulier cela implique que  $a^2 = 2b^2$ , i.e. 2 divise  $a^2$  et doit donc diviser  $a$ . Dès lors il existe un certain  $k$  tel que  $a = 2k$ . Nous pouvons alors écrire :

$$4k^2 = a^2 = 2b^2,$$

ou encore  $2k^2 = b^2$ , ce qui implique que  $b^2$  est pair, i.e.  $b$  est pair. Nous savons donc que 2 divise à la fois  $a$  et  $b$ , ce qui contredit le fait que  $a$  et  $b$  soient premiers entre eux. Nous en déduisons donc que de tels entiers  $a$  et  $b$  ne peuvent exister, et donc  $\sqrt{2}$  est irrationnel.  $\square$

| **Remarque.** *Pouvez-vous adapter cet argument pour montrer que  $\sqrt{p}$  est irrationnel pour tout  $p$  premier ?*

## 1.3 Python et son interpréteur

Le cours INFO-F103 (Algorithmique I) utilise exclusivement Python 3 comme langage d'implémentation et d'exemples, tant pour les cours théoriques que pour les séances d'exercices. Nous revenons, dans cette section, sur certains aspects importants de ce langage et nous en profitons pour rappeler (ou introduire) des notions qui seront utilisées tout au long de ce correctif. Il est important de préciser que les codes fournis ici (qui seront également disponibles sur l'UV) ont été écrits pour des versions de Python 3 à partir de Python 3.9, mais sont très certainement compatibles avec Python  $\geq 3.5$  pour la plupart. Nous vous invitons bien entendu à toujours utiliser la dernière version stable disponible (à savoir la version 3.10 lors de l'écriture de ces lignes).

### 1.3.1 Norme vs implémentation de référence

Contrairement à plein de langages classiques (tels que C, C++, Fortran, ASM x86, etc.) qui sont définis par une *norme*, Python (tout comme Pearl, R, et – certes, de manière discutable – Java), et ce depuis sa première version, est défini par son implémentation de référence : CPython. Ce nom désigne l'interpréteur *officiel* de Python, initié par Guido van Rossum (attention à garder le 'v' en minuscule, il y tient...) Son nom provient tout simplement de la contraction entre le nom du langage interprété : Python et la lettre C qui désigne le fait que cet interpréteur soit implémenté en C.

Ce n'est cependant pas la seule implémentation du langage disponible. Nous pouvons en effet mentionner Pypy et IronPython qui sont les deux alternatives les plus utilisées et encore maintenues, mais il faut tout de même retenir que même si ces implémentations ont des avantages indéniables (e.g. Pypy est clairement plus rapide que CPython), ces interpréteurs contiennent des choix d'implémentation qui peuvent poser des problèmes de compatibilité. Nous vous conseillons donc de n'utiliser un interpréteur différent de CPython que si vous êtes déjà très à l'aise avec les spécificités du langage, les détails d'implémentation et les différences potentielles de comportement lors de l'exécution de code provenant de *packages* externes. Du coup, tous les codes donnés dans ce syllabus d'exercices

ainsi que toutes les explications relatives au langage ou à l'interpréteur feront toujours (implicitement ou explicitement) référence à CPython et aucun autre interpréteur.

L'implémentation (en C donc) de CPython peut se trouver [sur GitHub](#), si ça vous amuse, ne manquez pas l'occasion d'aller y jeter un coup d'œil de temps en temps : le code est assez bien documenté et les décisions prises sur le long terme ont forcé une certaine cohérence qui rend le tout compréhensible, même pour une personne extérieure au développement du projet (pour peu que vous soyez à l'aise avec le langage C).

Python dispose tout de même d'une [documentation](#) qui contient la description de la syntaxe du langage, des fonctionnalités proposées, de la [lib standard](#), etc. La description des différents types, de leurs méthodes, des packages, etc. correspondent – dans l'écrasante majorité – à la documentation que vous pouvez trouver à l'aide de la fonction [help](#) en Python. Cette documentation devrait être le premier endroit auquel vous pensez lorsque vous cherchez une information sur un aspect du langage (e.g. à quoi correspondent les paramètres d'une fonction, que renvoie une certaine fonction, quels sont les valeurs par défaut des paramètres, quelles exceptions peuvent être lancées, etc.) La documentation propose également un [glossaire](#) que vous pouvez visiter si vous voulez avoir la définition précise d'un mot jargonnel.

### 1.3.2 Strings et formatage

À ce jour, Python propose deux méthodes pour gérer les chaînes de caractères : la classe [str](#) et la classe `bytes`. cette distinction entre [str](#) (en UTF-8) et `bytes` vient initialement de Python 2. En Python 3, et dans le cadre de ce cours, toutes les chaînes de caractères seront des instances de [str](#). La conversion de [str](#) vers `bytes` se fait par l'intermédiaire de la méthode [str.encode](#) alors que la conversion dans l'autre sens se fait via la méthode `bytes.decode`. Évidemment dans les deux cas, l'encodage de la chaîne de caractères (e.g. ASCII ou UTF-8) doit être fourni en paramètre.

Nous utiliserons donc les termes *chaîne de caractère*, *string* et *str* de manière interchangeable pour désigner un objet de type [str](#).

Le *formatage* d'un string est l'opération qui consiste à encoder la valeur de certaines variables (ou le résultat de certaines expressions) au sein d'un string. Python 3 permet de faire cela de 4 manières différentes : l'interpolation (via l'opérateur `%`), le formatage explicite (via la méthode [str.format](#)), les strings template (via donc la classe `string.Template`) et les f-strings (via le préfixe `f`).

La première approche est aujourd'hui peu utilisée (bien qu'elle ne soit pas déclarée comme obsolète ou dépréciée) et vient du formatage de chaînes de caractères en C (c.f. [la doc](#)) et ne sera pas utilisée ici. Nous nous attendons également à ce que vous ne l'utilisiez pas. Remarquons que son utilisation aujourd'hui se résume globalement à la gestion des logs, i.e. les informations que le programme décide de mentionner au cours de son exécution (ce qui contient donc les warnings et les erreurs).

La seconde approche est encore beaucoup utilisée aujourd'hui mais est en train d'être remplacée par les f-strings. Le fonctionnement est le suivant : en utilisant des accolades dans un string et en appelant la méthode [format](#), les paramètres donnés vont remplacer les accolades (pour plus d'informations, voir [la doc](#)). Voici quelques exemples :

---

```
>>> '{} - {}'.format(10, 11)
'10 - 11'
>>> '{0} - {1} - {0}'.format(10, 11)
'10 - 11 - 10'
>>> '{1} - {1} - {1}'.format(10, 11)
'11 - 11 - 11'
>>> '{zero} - {one} - {var}'.format(zero=14, one='abc', var=('a', 'b'))
'14 - abc - ('a', 'b')"
```

---

Les templates ne seront pas abordés ici car, comme l'interpolation, son utilisation est très limitée. La dernière approche, quant à elle, est de plus en plus utilisée aujourd'hui bien qu'introduite dans Python 3.6 via la [PEP-498](#) (voir la [PEP-502](#) pour plus d'informations sur le formatage de strings et sur les différentes approches, mais attention cette PEP a été rejetée). Son fonctionnement est similaire à celui de la méthode `str.format` mais sans faire d'appel explicite à une méthode. Plus précisément, les noms de variables ou les expressions à inclure dans le string est placé directement entre les accolades. Afin de spécifier que les accolades représentent bien une volonté de formatage (et pas uniquement le symbole d'accolade), ces chaînes de caractères sont notées avec le préfixe `f`. Voici quelques exemples :

---

```
>>> a = 31
>>> f'{a}'
'31'
>>> s = 'Hello'
>>> f'{a} - {s}'
'31 - Hello'
>>> f'{a:x}'
'1f'
>>> f'{a:x} - {min([10, 21, a])}'
'1f - 10'
```

---

C'est cette notation qui sera utilisée tout du long de ce syllabus d'exercices.

### 1.3.3 Décorateurs

En Python, un *décorateur* est une fonction qui renvoie une autre fonction. Ce concept est intéressant car il permet en un sens de *factoriser* certains traitements applicables à plusieurs fonctions (ou méthodes comme on le verra plus loin) mais est également applicable à des classes.

Voici un exemple trivial de décorateur :

---

```
1 def decorator():
2     return max
```

---

Cet exemple n'est cependant pas très intéressant. L'intérêt principal d'un décorateur est de s'ajouter au fonctionnement d'une fonction quelconque. Supposons par exemple que nous cherchons à avoir un message de debug affichant quelle fonction a été appelée à quel moment. Nous pouvons écrire un décorateur pour ça et appliquer ce décorateur à



plusieurs de nos fonctions :

---

```

1 from datetime import datetime
2
3 start = datetime.now()
4
5 def log_to_stdout(function):
6     def wrapper(*args, **kwargs):
7         offset = datetime.now() - start
8         print(f'[{offset}] called function "{function.__name__}"')
9         return function(*args, **kwargs)
10    return wrapper

```

---

La syntaxe pour appliquer un décorateur à une fonction est la suivante :

---

```

1 @decorator_function
2 def wrapped_function(...):
3     ...

```

---

qui en réalité correspond à :

---

```

1 def wrapped_function(...):
2     ...
3 wrapped_function = decorator_function(wrapped_function)

```

---

Ces deux syntaxes sont en effet équivalentes, la première est juste plus lisible (on appelle cela du *sucre syntaxique*). Nous pouvons donc écrire un court programme qui récupère 5 nombres entiers sur l'input standard et qui en affiche la somme mais en décorant les fonctions impliquées à l'aide de notre fonction `log_to_stdout` :

---

```

1 @log_to_stdout
2 def print_sum(array):
3     print(sum(array))
4
5 @log_to_stdout
6 def main():
7     array = []
8     for _ in range(5):
9         array.append(int(input('Entier: ')))
10    print_sum(array)

```

---

Une exécution de ce programme pourrait par exemple donner ceci (ou on voit donc les inputs et les outputs comme sur un terminal) :

```

[0:00:00.000006] called function "main"
Entier: 1
Entier: 2
Entier: 3
Entier: 4

```

```
Entier: 5
[0:00:01.599762] called function "print_sum"
15
```

Un décorateur peut également être paramétrisé sous la forme suivante :

---

```
1 @decorator(param1, param2, ...)
2 def function(...):
3     ...
```

---

Pour faire cela, il faut que la fonction décorateur ne prenne pas la fonction *wrapped* (enveloppée) mais bien les paramètres, et la fonction renvoyée doit, elle, être le *wrapper* :

---

```
1 def decorator(param1, param2, ...):
2     def decorator(func):
3         def wrapper(*args, **kwargs):
4             ... # utilise les paramètres de decorator
5             ret = func(*args, **kwargs)
6             ... # utilise les paramètres de decorator
7             return ret
8         return wrapper
9     return decorator
```

---

Remarquons qu'afin de conserver le nom de la fonction, le docstring, etc. il est possible d'utiliser le décorateur `functools.wraps` :

---

```
1 import functools
2 def decorator(func):
3     @functools.wraps(func)
4     def wrapper(*args, **kwargs):
5         ...
6     return wrapper
```

---

### 1.3.4 Classes et types

Python est un langage qui permet de faire de la *programmation orienté objet* (POO). Les détails de ce paradigme seront vus en bloc 2, mais les classes seront tout de même utilisées à moult reprises dans ce document, sans rentrer dans les détails du fonctionnement de la POO.

En python, *tout est objet* : tout ce que vous pouvez manipuler a un *type* bien défini et c'est ce dernier qui définit les opérations que l'on peut faire sur cet objet (quelles méthodes peuvent être appelées sur l'objet en question, quels opérateurs sont disponibles, à quelles fonctions cet objet peut être passé en paramètre, etc.) Cette notion de *type* est directement liée à la notion de *classe* puisque les classes permettent de définir des types, en plus des types déjà existant. En effet les types déjà disponibles (e.g. `int`, `float`, `str`, `tuple`, `list`, `set`, `dict`, etc.) sont codés directement en C dans l'interpréteur (c.f. les fichiers `*object.c` dans le code de CPython).

Une classe *définit* un type dans le sens suivant : lors de l'écriture d'une classe C, il faut

définir les *méthodes* qui peuvent être appelées sur cet objet, et il faut également définir quels sont les *attributs* d'un objet de ce type C.

Contrairement à d'autres langages (e.g. C++ ou Java), Python est un langage fondamentalement *dynamique*. Il est donc possible de rajouter des méthodes et des attributs à un objet bien après son initialisation. Cependant, il est fortement recommandé de ne pas faire cela (hors contextes très spécifiques) car la lisibilité du code en est fortement impactée. De manière générale, tous les attributs sont créés et initialisés (potentiellement à *None*) dans le constructeur, i.e. la méthode `__init__`. Toutes les méthodes d'une classe (à part les méthodes statiques dont nous parlerons plus loin) prennent en premier paramètre une variable *spéciale* (appelée *self* par convention; bien que ce nom n'est pas strictement obligatoire, il serait déraisonnable de l'appeler autrement) dans le sens où cette variable est une référence vers l'objet de type C sur lequel la méthode est appelée. Cet objet est fondamental afin de pouvoir en manipuler les attributs.

Par exemple définissons le type trivial T contenant une unique variable *v* initialisée à 0 et une unique méthode *increment* qui a pour seul objectif d'incrémenter (donc augmenter de 1) l'attribut *v* :

---

```
1 class T:
2     def __init__(self):
3         self.v = 0
4     def increment(self):
5         self.v += 1
```

---

Nous voyons bien que le constructeur définit l'attribut *v* : si nous retirons la ligne 3 de ce code, un objet de type T n'aura pas son attribut *v*. De plus, la méthode *increment* prend bien le paramètre *self*, ce qui permet d'accéder à l'attribut *v* de l'*instance* sur laquelle la méthode est appelée.

En effet si nous créons deux objets (respectivement *t1* et *t2*) de type T, ces deux objets auront chacun *leur propre* attribut *v* avec sa valeur. Dès lors si nous appelons *increment* sur *t1* mais pas sur *t2*, alors l'attribut *v* de *t1* vaudra 1 alors que l'attribut *v* de *t2* vaudra toujours 0, comme juste après son initialisation :

---

```
1 t1 = T()
2 t2 = T()
3 t1.increment()
4 assert t1.v == 1
5 assert t2.v == 0
```

---

En réalité, l'interpréteur comprend l'instruction `t1.increment()` comme étant `type(t1).increment(t1)` (i.e. `T.increment(t1)`), et *t1* est donc passé en paramètre à la *fonction* `T.increment`, d'où la présence de *self*.

Toutefois, comme mentionné plus haut : *tout est objet*. En particulier nos variables *t1* et *t2* sont des objets (et leur *type* est T) mais le type T en lui-même est un objet et a un type ! Son type est `type`... En effet, en Python il existe un type `type` et tout type est de type `type`, même le type `type`...

---

```
>>> class T:
...     pass
...
>>> t = T()
>>> type(t)
<class '__main__.T'>
>>> type(T)
<class 'type'>
>>> type(type)
<class 'type'>
```

---

Dès lors le `type` `T` est un objet (en un sens, c'est une variable), et donc comme tout objet, il peut avoir des attributs. Ce qu'on appelle les *attributs de classe* sont des attributs de l'objet relatif à une classe. Ces derniers se distinguent des attributs *classiques* dans le sens où deux objets de même type ont chacun leurs propres attributs, mais partagent les mêmes attributs de classe. Nous pouvons, par exemple, créer un type `T` comme celui présenté ci-dessus mais qui aurait comme attribut de classe un compteur d'instances : le type `T` aura donc sa propre variable qui sera augmentée à chaque fois qu'un objet de type `T` est créé :

---

```
1 class T:
2     counter = 0
3     def __init__(self):
4         self.v = 0
5         T.counter += 1
6     def increment(self):
7         self.v += 1
```

---

Nous voyons bien que dans ce contexte, l'attribut `v` des instances et l'attribut `counter` du type `T` sont fondamentalement différents. D'ailleurs même si l'attribut `T.counter` n'est modifié que dans le constructeur ici, les attributs de classe peuvent être utilisés partout, même en dehors de la classe :

---

```
>>> t1 = T()
>>> t2 = T()
>>> T.counter
2
>>> t1.increment()
>>> T.counter
2
>>> t3 = T()
>>> T.counter
3
```

---

Notons que l'utilisation la plus classique des attributs de classe est la présence de constantes : cela permet de définir des constantes qui n'apparaissent pas dans le *scope* global des variables mais bien uniquement dans le *scope* qui a un sens.

### 1.3.5 Encapsulation

Un principe très important en POO et qui est également très important dans le cadre des types de données abstraits (ADT ou *abstract data types*) est la notion d'*encapsulation*. Le principe est le suivant : si un type représente un objet par les attributs qu'il contient et par les actions que l'on peut faire sur cet objet (via des méthodes), depuis l'*extérieur* de la classe (donc en dehors de son code de définition), la représentation interne des objets (en particulier les attributs et leur type) ne doit pas être accessible. Cela implique que les attributs d'un objet ne doivent *jamais* être manipulés (ne fut-ce qu'en lecture) par d'autres objets. Certains langages (tels que Java ou C++ par exemple) permettent d'explicitement cette notion et de préciser quels attributs sont *publics* (donc visible depuis l'extérieur) et lesquels sont *privés* (donc invisibles depuis l'extérieur). Remarquons également que ceci est valable pour les attributs des instances des classes mais peut également s'appliquer aux attributs des classes en elles-mêmes : il arrive que les attributs de classe ne soient pas accessibles depuis l'extérieur car ce ne serait pas pertinent. Notons également qu'il existe une visibilité intermédiaire appelée *protégée* mais dont nous ne parlerons pas ici car elle n'a de sens que dans le cadre de l'héritage dont nous ne parlerons (presque) pas ici (on aimerait bien, mais comme vous le constaterez, il y a déjà beaucoup à faire, ne brûlons pas d'étape...)

À première vue, nous pourrions penser que Python ne permet pas cela puisque un attribut, nommé de manière somme toute classique, est tout à fait accessible en lecture *et* écriture depuis l'extérieur :

---

```
>>> class T:
...     def __init__(self):
...         self.x = 0
...
>>> t = T()
>>> t.x # lecture de l'attribut x
0
```

---

est un code tout à fait *valide* et ne provoquera aucune erreur.

Il est cependant possible de forcer un attribut à être privé en Python (ce principe s'applique d'ailleurs également aux méthodes car de toute façon, en Python, les méthodes sont, sous une certaine forme, des attributs, oui ça peut paraître confus). Il faut pour cela faire précéder le nom de l'attribut par un double underscore. Attention, nous ne mettons pas de double underscore à la fin (c.f. la [sous-section 1.3.8](#)) ! Nous pouvons dès lors adapter le code précédent comme ceci :

---

```
>>> class T:
...     def __init__(self):
...         self.__x = 0
...
>>> t = T()
>>> t.__x # lecture de l'attribut x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'T' object has no attribute '__x'
```

---

L'attribut `__x` n'est accessible que depuis l'intérieur de la classe `T`.

Si les attributs sont privés mais que l'on veut tout de même permettre un accès à ces derniers, il faut passer des *getters* et des *setters*, i.e. des méthodes dont le but est d'interfacer les attributs avec le monde extérieur. En particulier, il ne faut en faire que pour les attributs pour lesquels une interface a un sens, et les *setters* se chargent bien souvent de faire une vérification pour s'assurer de la validité de l'objet modifié.

En pratique il est rare, en Python, d'utiliser cette notation avec le double underscore puisqu'elle alourdit fortement le code et le rend moins lisible. La plupart du temps, les attributs sont nommés normalement, mais il existe certaines conventions utilisées par différentes personnes pour désigner un attribut comme étant privé et donc ne devant pas être manipulé depuis l'extérieur, sans explicitement interdire une telle manipulation. Le principe est donc de *faire confiance* (drôle d'idée, je vous l'accorde) aux autres et de supposer que votre classe sera bien utilisée. Parmi ces conventions, une revient plutôt fréquemment et consiste à ajouter le suffixe `_` (donc un underscore) aux attributs que l'on veut désigner comme privés. Si vous voyez une telle nomenclature dans ce correctif, c'est très certainement ce qu'elle désigne.

Un *getter*, donc une méthode dont le but est de récupérer la valeur d'un attribut, *sans le modifier*, ne prend aucun paramètre (si ce n'est `self` bien entendu) et se nomme, par convention `get_*` alors qu'un *setter*, dont le but est de modifier un attribut sans chercher à récupérer sa nouvelle valeur, prend en paramètre la nouvelle valeur à donner et se nomme, par convention `set_*`.

Si nous prenons comme exemple une classe représentant l'heure de la journée (donc heure, minute, seconde), nous aurons trois attributs (que nous appellerons ici `hour_`, `minute_`, `second_`). Si nous voulons en plus donner la possibilité d'augmenter le temps par une seconde, nous implémentons une méthode `tick()` qui mettra les valeurs à jour (en passant à la minute supérieure ou à l'heure supérieure si nécessaire) :

---

```
1 class Time:
2     def __init__(self, h=0, m=0, s=0):
3         self.hour_ = h
4         self.minute_ = m
5         self.second_ = s
6
7     def tick(self):
8         self.second_ += 1
9         if self.second_ == 60:
10            self.second_ = 0
11            self.minute_ += 1
12        if self.minute_ == 60:
13            self.minute_ = 0
14            self.hour_ += 1
15        if self.hour_ == 24:
16            self.hour_ = 0
```

---

Maintenant, bien qu'il soit *techniquement* possible de directement accéder aux attributs, si nous voulons permettre de faire de tels accès *proprement*, nous ajouterons un *getter* et un *setter* pour chacun de ces attributs :

---

```

1     def get_hour(self):
2         return self.hour_
3     def get_minute(self):
4         return self.minute_
5     def get_second(self):
6         return self.second_
7
8     def set_hour(self, new_h):
9         if 0 <= new_h < 24:
10            self.hour_ = new_h
11     def set_minute(self, new_m):
12         if 0 <= new_m < 60:
13            self.minute_ = new_m
14     def set_second(self, new_s):
15         if 0 <= new_s < 60:
16            self.second_ = new_s

```

---

Nous avons ici ajouté une vérification aux setters afin de s'assurer de ne pas permettre de tout casser en un simple appel de fonction (nous pouvons également bien sûr ajouter un lancement d'exception, e.g. `ValueError` si la condition n'est pas vérifiée).

Il existe une version plus *pythonic* de gérer l'encapsulation, mais nous la verrons dans la sous-section 1.3.9. Faisons tout de même une remarque sur le fonctionnement des attributs *privés* en Python. Puisque cette notion n'existe pas, au sens formel, dans le langage, il a fallu ruser pour le simuler. En particulier, si un type `T` définit un attribut `x` mais veut le rendre privé via le double underscore, il sera accessible depuis `self.__x` mais en réalité cet attribut s'appellera `_T__x` où le premier underscore permet de ne pas le lister parmi les attributs *classiques*, le `T` est le nom du type, le double underscore derrière est le séparateur et le `x` est le nom de l'attribut. En particulier nous voyons que nous pouvons *tricher* en procédant de la sorte :

---

```

>>> class Type:
...     def __init__(self):
...         self.__private_attr = 0
...
>>> var = Type()
>>> var._Type__private_attr
0

```

---

Évidemment, ceci sert uniquement à vous montrer le principe de *name mangling* (i.e. le *mâchage de nom*, concept qui existe dans moult autres langages, c.f. le *name mangling* de labels en ASM pour résoudre les surcharges en C++) mais vous ne devez bien sûr *absolument jamais* accéder à des attributs privés de cette manière !

### 1.3.6 Fonction vs méthode, méthode liée vs non-liée

Considérons le type `T` suivant :

---

```
>>> class T:
...     ''' docstring de T '''
...     class_var = 0
...     def f(): pass
```

---

Puisque tout est objet en Python (même les types, même les fonctions et les méthodes!), l'objet `T` (de type `type`) a un *attribut* pour chacune de ses méthodes en plus de ses attributs de classe. Nous pouvons voir cela avec son attribut `__dict__` :

---

```
>>> dict(T.__dict__)
{'__module__': '__main__',
 'class_var': 0,
 'f': <function T.f at 0x7f0e6fc64820>,
 '__doc__': ' docstring de T ', ...}
```

---

où ... a été mis ici car le reste ne nous intéresse pas dans le cas présent. Nous pouvons donc voir certains objets ont un attribut `__module__` qui contient un string contenant le module dans lequel cet objet a été défini. En particulier tous les types ont cet attribut (mais tous les objets ne sont pas obligés de le définir). Nous voyons également l'attribut `__doc__` qui contient le docstring de la classe.

Si nous disposons d'un objet `t`, instance de `T`, observons que `T.f` et `t.f` ne désignent pas la même chose !

---

```
>>> T
<class '__main__.T'>
>>> t = T()
>>> t
<__main__.T object at 0x7f0e70065cd0>
>>> type(T.f)
<class 'function'>
>>> type(t.f)
<class 'method'>
```

---

En effet, `T.f` est une *fonction* (dans le sens classique d'une fonction) mais `t.f` est une *méthode*. En effet, c'est une fonction qui est *liée* à un objet en particulier et qui agit sur cet objet. De plus, lors de l'appel `t.f()`, le paramètre `self` est ajouté *automatiquement*, ce qui implique bien que `t.f` et `T.f` ont une différence fondamentale de comportement. On parle également de *fonction liée* pour parler de méthode :

---

```
>>> T.f
<function T.f at 0x7f0e6fc64820>
>>> t.f
<bound method T.f of <__main__.T object at 0x7f0e70065cd0>>
>>> t.f is T.f
False
>>> t.f.__func__ is T.f
True
```

---



De plus, même si ces deux fonctions ne sont, factuellement, pas les mêmes, `t.f` a besoin de `T.f` pour s'exécuter. Une méthode a donc un attribut `__func__` qui n'est autre que la fonction à appeler. Il faut également à `t.f` savoir quel est l'objet sur lequel la méthode doit être appelée, i.e. ce que va être le paramètre `self`. Il y a, pour cela, un attribut `__self__` à toute méthode liée :

---

```
>>> t.f.__self__
<__main__.T object at 0x7f0e70065cd0>
>>> t.f.__self__ is t
True
```

---

### 1.3.7 Méthodes statiques

Certaines fonctions n'ont de sens que dans le cadre d'un certain type (i.e. dans le cadre d'une classe) mais ne travaillent pas sur un objet de ce type (i.e. le paramètre `self` n'est pas utilisé). Une méthode (donc dans une classe) qui n'a pas besoin de ce paramètre `self` est appelée *méthode statique* et peut se déclarer à l'aide du décorateur `staticmethod`. Toute fonction déclarée avec ce décorateur ne doit évidemment pas prendre le paramètre `self` !

---

```
>>> class T:
...     def __init__(self):
...         self.x = 0
...     @staticmethod
...     def f():
...         return 1
...
>>> t = T()
>>> t.f()
1
>>> T.f()
1
```

---

Bien que le langage le permette, il est rare d'appeler une méthode statique sur une instance de la classe. Les appels se font la plupart du temps directement sur la classe (donc `T.f` au lieu de `t.f`). D'ailleurs, beaucoup de langages orientés objets ne permettent tout simplement pas l'appel de méthode statique sur une instance.

Le fonctionnement interne du décorateur `staticmethod` n'est pas simplement une fonction comme vu dans la [sous-section 1.3.3](#) mais est en réalité un *descripteur*. Nous n'en parlerons pas ici, mais vous pouvez bien sûr aller lire la [documentation](#) si le cœur vous en dit.

### 1.3.8 Méthodes spéciales

Précédemment, nous avons déjà vu des *attributs spéciaux*, i.e. ceux qui commencent et terminent par un double underscore. De votre côté, vous ne devez *jamais* utiliser cette notation pour nommer vos propres variables/fonctions. Cette convention est réservée pour le langage lui-même et permet de spécifier certains comportements de vos objets sous certaines fonctions *built-in* (donc définies directement dans l'interpréteur).

Parmi ces attributs, nous avons vu `__doc__`, `__module__`, `__dict__`, mais vous connaissez aussi `__name__` (e.g. dans `if __name__ == '__main__':`) et il y en a (plein) d'autres. Vous avez [ici](#) la liste des attributs spéciaux pour les objets appelables (*callable*) et [ici](#) la liste pour les modules.

En plus de ces attributs, il existe des *méthodes spéciales*. Ces méthodes sont donc des fonctions définies dans une classe et dont le nom commence et finit par un double underscore. À nouveau, hormis les méthodes spéciales existantes, vous ne devez *jamais* définir vos propres méthodes selon cette convention. Nous en distinguerons ici deux types : les méthodes spéciales qui correspondent à une fonction *built-in* et celles qui correspondent à un opérateur.

Python propose certaines fonctions déjà existantes et dont le comportement sur vos propres classes peut être spécifié. Par exemple nous pouvons citer les fonctions `str` et `repr` qui permettent respectivement de récupérer un string pour afficher l'élément et un string pour représenter l'objet (`repr` est la fonction utilisée pour représenter un objet en mode interactif). Ces fonctions peuvent être paramétrisées sur vos propres classes afin de (par exemple) permettre de les afficher. Pour cela, il va falloir implémenter les méthodes spéciales `__str__` et `__repr__`. Bien souvent, le nom de la méthode spéciale à implémenter est le nom de la fonction *built-in* entouré de double underscores. La fonction `repr` doit donc être vue comme ceci (schématiquement, bien sûr, des précisions suivront) :

---

```
def repr(x):
    return x.__repr__()
```

---

Ce principe est à garder en tête pour toutes les autres méthodes spéciales. Prenons un exemple et définissons une classe `IntegerInterval` qui, comme son nom l'indique, représente un intervalle entier (c.f. la [section 1.4](#)) :

---

```
1 class IntegerInterval:
2     def __init__(self, a, b):
3         self.a = a
4         self.b = max(a, b)
5
6     def __str__(self):
7         return f'[{self.a}, {self.b}]'
8
9     def __repr__(self):
10        return f'<IntegerInterval object at 0x{id(self):x}: {str(self)}>'
```

---

Considérons maintenant la fonction `len` qui doit renvoyer la taille du conteneur donné en paramètre (comme vous l'avez très certainement déjà vu sur des listes, tuples, dictionnaires, etc.) Si vous voulez que cette fonction soit généralisable à vos propres types, il vous faut implémenter la méthode `__len__` dans votre classe. Sur notre classe `IntegerInterval`, nous pouvons l'implémenter, par exemple, comme ceci :

---

```
1     def __len__(self):
2         return self.b - self.a + 1
```

---

Nous pouvons dès lors regarder le comportement des fonctions `repr`, `str` et `len` avec le

code suivant (en gardant en tête que `print(x)` va appeler `str(x)` si `x` n'est pas déjà un `str`) :

---

```

1 for b in (8, 1, 0, -2):
2     I = IntegerInterval(0, b)
3     print(repr(I))
4     print(I)
5     print(len(I))

```

---

qui donne comme output :

```

<IntegerInterval object at 0x7fc11c5dfe50: [0, 8]>
[0, 8]
9
<IntegerInterval object at 0x7fc11c5dfc40: [0, 1]>
[0, 1]
2
<IntegerInterval object at 0x7fc11c5dfe50: [0, 0]>
[0, 0]
1
<IntegerInterval object at 0x7fc11c5dfc40: [0, 0]>
[0, 0]
1

```

Nous pouvons observer que les adresses reviennent malgré que les variables soient des instances différentes. Nous en discuterons dans la [sous-section 1.3.11](#).

La deuxième famille de fonctions spéciales est celle des fonctions qui correspondent à un opérateur. Plus précisément, Python (à nouveau comme beaucoup d'autres langages) <sup>1</sup> permet de spécifier le comportement des opérateurs sur un type *user-defined*. En effet, les opérateurs sont tous liés à une méthode spéciale qui sera appelée (si elle existe). Prenons l'exemple d'une classe représentant un nombre rationnel (donc une fraction de nombre entiers). Il est clair que la somme est définie sur  $\mathbb{Q}$  :

$$\frac{a}{b} + \frac{p}{q} = \frac{aq}{bq} + \frac{bp}{bq} = \frac{aq + bp}{bq}.$$

Si nous implémentons la classe `Rational`, mettons une méthode `reduce` qui nous permet de toujours avoir la fonction sous forme réduite (ou irréductible). Ajoutons également une méthode `__str__` afin de permettre d'afficher aisément nos objets. Finalement, ajoutons la méthode `__add__` qui, en plus de `self` doit prendre l'élément que l'on souhaite ajouter à `self` pour déterminer le résultat ; et qui doit renvoyer cette instance de `Rational` :

---

1. C++ permet de *surcharger* les opérateurs par exemple mais comme Python ne supporte pas la notion-même de surcharge, il a fallu suivre une autre route.

---

```

1 def gcd(a, b):
2     while b > 0:
3         a, b = b, a%b
4     return a
5
6 class Rational:
7     def __init__(self, a=0, b=1):
8         self.num_ = a
9         self.den_ = b
10        self.reduce()
11
12    def reduce(self):
13        d = gcd(self.num_, self.den_)
14        self.num_ //= d
15        self.den_ //= d
16
17    def __str__(self):
18        return f'{self.num_} / {self.den_}'
19
20    def __add__(self, other):
21        num = self.num_ * other.den_ + other.num_ * self.den_
22        den = self.den_ * other.den_
23        return Rational(num, den)
24
25 if __name__ == '__main__':
26     x = Rational(3, 5)
27     y = Rational(1, 10)
28     z = x+y
29     print(type(z))
30     print(f'({x}) + ({y}) == ({z})')
```

---

Le code ci-dessus affichera ceci :

```
<class '__main__.Rational'>
(3 / 5) + (1 / 10) == (7 / 10)
```

En effet, lorsque l'interpréteur voit une expression sous la forme  $a + b$ , il va l'interpréter de la manière suivante : `a.__add__(b)` (ou encore `type(a).__add__(a, b)`).

Il existe également la méthode spéciale `__radd__` qui permet de spécifier le comportement de l'opérateur `+`. En effet, dans le cas où l'objet `a` ne définit pas la méthode `__add__`, il est tout de même possible d'effectuer cette addition si l'objet `b` définit `__radd__`. Dans ce cas, l'expression `a + b` va être comprise comme `b.__radd__(a)` (ou encore à nouveau `type(b).__radd__(b, a)`).

Le `r` qui préfixe le nom de la méthode vient de *right* en anglais qui signifie *droite* puisque si `__radd__` est appelée, c'est que `self` est l'opérande de droite de la somme. Si la somme que vous définissez est commutative, cela ne change rien et vos méthodes `__add__` et `__radd__` peuvent être strictement équivalentes, mais si la somme ne l'est pas, alors le traitement dans ces deux cas doit être différent.

Notons également qu'il est possible de définir une méthode `__add__` mais de ne vouloir la spécifier que sur certains types mais pas sur d'autres. Pour cela, il faut user de `isinstance` afin de déterminer le type des objets. En particulier si nous voulons que notre classe `Rational` puisse supporter l'addition entre deux rationnels ou entre un rationnel et un entier, mais pas les autres types, nous pouvons traiter individuellement les cas où `other` est de type `int`, de type `Rational` ou aucun des deux. Lorsqu'une méthode spéciale qui précise le fonctionnement d'un opérateur ne veut pas fonctionner sur un certain type, elle doit renvoyer `NotImplemented` :

---

```

1 class Rational:
2     def __init__(self, a=0, b=1):
3         self.num_ = a
4         self.den_ = b
5         self.reduce()
6
7     def reduce(self):
8         d = gcd(self.num_, self.den_)
9         self.num_ //= d
10        self.den_ //= d
11
12    def __str__(self):
13        return f'{self.num_} / {self.den_}'
14
15    def __add__(self, other):
16        if isinstance(other, int):
17            return Rational(self.num_ + other * self.den_, self.den_)
18        elif isinstance(other, Rational):
19            num = self.num_ * other.den_ + other.num_ * self.den_
20            den = self.den_ * other.den_
21            return Rational(num, den)
22        else:
23            return NotImplemented
24
25    def __radd__(self, other):
26        return self + other # notre somme est commutative
27
28 if __name__ == '__main__':
29     x = Rational(3, 5)
30     y = Rational(1, 10)
31     print(f'({x}) + ({y}) == ({x + y})')
32     print(f'({x}) + 1 == ({x + 1})')
33     print(f'({x}) + ({y}) == ({x + 0.1})')
```

---

qui affichera :

$(3 / 5) + (1 / 10) == (7 / 10)$

$(3 / 5) + (1 / 10) == (8 / 5)$

Traceback (most recent call last):

File "<stdin>", line 62, in <module>

print(f'({x}) + ({y}) == ({x + 0.5})')

`TypeError: unsupported operand type(s) for +: 'Rational' and 'float'`

En effet, c'est l'interpréteur qui remarque, en ayant récupéré `NotImplemented`, que cette implémentation ne gère pas les types donnés. Sur base de tout cela, nous pouvons écrire le fonctionnement de l'opérateur `+` comme ceci :

---

```

1 def addition(a, b):
2     ret = NotImplemented
3     if hasattr(a, '__add__'):
4         ret = a.__add__(b)
5     if ret is NotImplemented:
6         if hasattr(b, '__radd__'):
7             ret = b.__radd__(a)
8     if ret is NotImplemented:
9         raise TypeError(
10             f"unsupported operand type(s) for +: '{type(a)}' and
              ↪ '{type(b)}'"
11         )
12     return ret

```

---

Il nous faut encore parler de l'opérateur `+=` qui est différent de l'opérateur `+`. En effet même si on s'attend à ce que `x` ait la même valeur après `x += y` et après `x = x + y`, il y a une différence fondamentale entre les deux. En effet, lors que l'on écrit `x = x + y`, une variable temporaire (appelons-la ici `temp` même si elle n'obtient jamais de *nom* explicitement) est créée et contient le résultat de `x + y`, ensuite la référence de `x` est modifiée pour pointer vers cette variable temporaire. Schématiquement, nous pouvons comprendre l'instruction `x = x + y` comme ceci (où `addition` est la fonction ci-dessus) :

---

```

1 temp = addition(x, y)
2 del x
3 x = temp
4 del temp # supprime la référence mais ne détruit pas l'objet

```

---

Dès lors, `x = x + y` crée un nouvel objet et la référence de `x` n'est pas la même avant et après l'assignation.

L'opérateur `+=` quant à lui *modifie* l'objet `x` et cette variable intermédiaire `temp` n'est jamais créée et donc (*a priori* mais nous détaillerons dans la [sous-section 1.3.11](#)) la référence de `x` reste inchangée :

---

```

>>> x, y = [0, 1, 2], [3, 4, 5, 6]
>>> hex(id(x))
'0x7f119d4163c0'
>>> hex(id(y))
'0x7f119d418c00'
>>> x += y
>>> hex(id(x))
'0x7f119d4163c0'
>>> x = x + y
>>> hex(id(x))

```

'0x7f119d416a80'

---

Afin de permettre à vos classes de supporter une augmentation via l'opérateur `+=`, il vous faut définir la méthode `__iadd__`. Cette méthode doit prendre (bien évidemment) `self` en premier paramètre et prend en second paramètre le membre de droite (donc le second opérande). Notons aussi que cette méthode doit renvoyer `self` ! La raison sera explicitée dans la sous-section 1.3.11. Voici un exemple d'implémentation de cette méthode sur la classe `Rational` :

---

```

1 class Rational:
2     def __init__(self, a=0, b=1):
3         self.num_ = a
4         self.den_ = b
5         self.reduce()
6
7     def reduce(self):
8         d = gcd(self.num_, self.den_)
9         self.num_ //= d
10        self.den_ //= d
11
12    def __str__(self):
13        return f'{self.num_} / {self.den_}'
14
15    def __add__(self, other):
16        if isinstance(other, int):
17            return Rational(self.num_ + other * self.den_, self.den_)
18        elif isinstance(other, Rational):
19            num = self.num_ * other.den_ + other.num_ * self.den_
20            den = self.den_ * other.den_
21            return Rational(num, den)
22        else:
23            return NotImplemented
24
25    def __iadd__(self, other):
26        if isinstance(other, int):
27            self.num_ += other * self.den_
28        elif isinstance(other, Rational):
29            self.num_ = self.num_ * other.den_ + other.num_ * self.den_
30            self.den_ *= other.den_
31            self.reduce()
32        else:
33            return NotImplemented
34        return self # très important
35
36    def __radd__(self, other):
37        return self + other # notre somme est commutative
38
39 if __name__ == '__main__':
40     x = Rational(3, 5)

```

```

41     y = Rational(1, 10)
42     print(x)
43     x += 1
44     print(x)
45     x += y
46     print(x)

```

---

qui affichera :

```

3 / 5
8 / 5
17 / 10

```

Notons que contrairement à l'opérateur `+`, l'opérateur `+=` ne peut pas exister sous une forme *droite*, i.e. une méthode `__riadd__` puisque c'est bien le premier opérande qui doit être modifié et par le principe d'encapsulation, il n'aurait aucun sens que la méthode `__riadd__(self, other)` laisse `self` intact et modifie `other`. Dès lors l'instruction `x += y` est obligatoirement interprétée comme `x.__iadd__(y)` si cette méthode existe.

Tout ce que nous avons dit ici concernait uniquement les opérateurs `+` et `+=`. Bien entendu, il existe ces mêmes méthodes spéciales pour les opérateurs `-` (dont le nom est `sub`); `*` (dont le nom est `mul`); `/` (dont le nom est `truediv`); etc. La liste exhaustive est accessible dans [la documentation](#).

Notons qu'au delà des opérateurs que l'on pourrait qualifier d'*arithmétiques* et des opérateurs booléens, Python (comme d'autres langages bien sûr) considère `()` (tel que dans un appel de fonction) comme un opérateur et considère `[]` (tel que lors de l'accès à un élément dans une liste) comme un opérateur également. Le premier correspond à la méthode `__call__` et le second est séparé en deux en fonction de si l'*indixage* est en lecture ou en écriture. Dans le premier cas, la méthode associée est `__getitem__` et dans le second cas, la méthode associée est `__setitem__`. Notons également `__delitem__` qui correspond à l'instruction `del obj[i]` qui correspond donc à `obj.__delitem__(i)`. Regardons cela sur un exemple. Écrivons une classe `Callback` qui permet de stocker une fonction à appeler pour plus tard à laquelle on peut donner autant de paramètres que l'on veut :

---

```

1  class Callback:
2      def __init__(self, f, *args):
3          self.fct_ = f
4          self.params_ = list(args)
5
6      def add_param(self, param):
7          self.params_.append(param)
8
9      def __call__(self):
10         return self.fct_(*self.params_)
11
12     def __getitem__(self, i):
13         return self.params_[i]
14
15     def __setitem__(self, i, newval):
16         assert i < len(self.params_)

```



```

17         self.params_[i] = newval
18
19     def __delitem__(self, i):
20         del self.params_[i]
21
22 if __name__ == '__main__':
23     callback = Callback(print)
24     callback.add_param(0)
25     # ah non je ne veux pas afficher 0
26     del callback[0] # appelle callback.__delitem__(0)
27     callback.add_param(1)
28     # oups, je voulais dire -1
29     callback[0] = -1 # appelle callback.__setitem__(0, -1)
30     callback.add_param(2)
31     # c'était quoi le premier paramètre encore ?
32     print(callback[0]) # appelle callback.__getitem__(0)
33     callback() # et on affiche le tout

```

---

Nous pouvons également mentionner la méthode spéciale `__contains__` qui permet d'utiliser l'opérateur `in`. En effet `x in container` est équivalent à `container.__contains__(x)`. Cette méthode doit renvoyer un booléen. Les comparaisons se font également via des opérateurs. Que ce soit l'opérateur `==` qui correspond à `__eq__`, l'opérateur `!=` qui correspond à `__neq__`, l'opérateur `<` qui correspond à `__lt__` ou encore `<=` qui correspond à `__le__` (idem pour `>` et `>=`, c.f. la [documentation](#)). Notons que par défaut, Python propose une implémentation de `__eq__` qui est `__eq__(self, other):`  
`↪ return self is other`, i.e. deux objets sont égaux si et seulement ils sont en réalité le même objet. En particulier cela implique que `x == deepcopy(x)` sera évalué à `False` pour (presque) tout objet `x`. De plus Python propose également une implémentation de `__neq__` qui renvoie simplement la négation de `__eq__`.

D'ailleurs, bien que le fait d'avoir une méthode `__eq__` et une des quatre méthodes de comparaison est suffisant pour déduire les autres, Python ne le fait pas automatiquement. En particulier, si seules les méthodes `__eq__` et `__lt__` sont implémentées, le fait d'avoir `x < y` et `x == y` évalués à `True` n'est pas suffisant pour que `x <= y` soit évalué à `True` : Python ira tout de même chercher la méthode `__le__` et se rendra compte qu'elle n'existe pas. Il existe cependant un décorateur (c.f. la [sous-section 1.3.3](#)) qui s'en charge : `total_ordering` demande qu'une seule des méthodes `__lt__`, `__le__`, `__gt__`, `__ge__` soit implémentée et déduit automatiquement les autres. Voici un exemple d'utilisation sur notre classe `Rational` qui nous permet d'utiliser tous les opérateurs de comparaison :

---

```

1 import functools
2 @functools.total_ordering
3 class Rational:
4     #...
5     def __eq__(self, other):
6         return self.num_ * other.den_ == self.den_ * other.num_
7
8     def __lt__(self, other):
9         return self.num_ * other.den_ < self.den_ * other.num_

```

Notons tout de même qu'à l'instar des méthodes `__r*__` pour les opérateurs arithmétiques, Python gère en un sens une certaine symétrie concernant les opérateurs de comparaison. En effet, lors de l'évaluation de l'expression `x < y`, si `x.__lt__(y)` renvoie `NotImplemented` (soit parce que la méthode n'existe pas, soit parce qu'elle a renvoyé `NotImplemented`), alors Python tentera d'appeler `y.__gt__(x)`, et si cet appel de fonction renvoie un booléen, alors cette valeur correspondra à l'expression `x < y`, ce qui est logique puisque les opérateurs `<` et `>` sont symétriques. Il en va, bien entendu, de même pour `__ge__` et `__le__`.

### 1.3.9 Propriétés

Les *propriétés* sont une forme d'*attributs virtuels* : ils permettent de créer des fonctions qui se manipuleront comme des attributs mais dont le code sera exécuté dès qu'on cherchera à les lire/écraser. Elles se déclarent avec le décorateur `property` et s'utilisent de la manière suivante :

---

```

1 class C:
2     def __init__(self):
3         self.__x = 1
4
5     @property
6     def x(self):
7         return self.__x
8
9     @property
10    def twice_x(self):
11        return self.__x*2
12
13 c = C()
14 assert c.x == 1 # appelle la propriété x
15 assert c.twice_x == 2 # appelle la propriété twice_x

```

---

Une propriété est donc une fonction qui se comporte comme un attribut. Attention à ne pas mettre de parenthèses pour faire explicitement l'appel de fonction car le décorateur s'en occupe déjà. Écrire `c.x()` sera interprété comme `(c.x)()` et lancera donc l'exception suivante :

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable

```

Cela ne permet par contre pas de modifier les attributs. En effet sur base du code ci-dessus, l'instruction `c.x = 2` lancera une exception :

```
AttributeError: can't set attribute.
```

Cependant, il est possible d'ajouter un *setter* à une propriété :

---

```
1 class C:
2     def __init__(self):
3         self.__x = 1
4
5     @property
6     def x(self):
7         return self.__x
8
9     @property
10    def twice_x(self):
11        return self.__x*2
12
13    @twice_x.setter
14    def twice_x(self, two_x):
15        self.__x = two_x // 2
16
17 c = C()
18 assert c.x == 1
19 c.twice_x = 4
20 assert c.x == 2
```

---

Écrire `c.twice_x = XXX` va donc être compris comme devant appeler le *setter* de `c.twice_x`. Notez également que la propriété (qui est donc un *getter*) et le *setter* doivent obligatoirement avoir le même nom.

Comme nous pouvons l'observer sur l'exemple ci-dessus, une propriété ne doit pas du tout obligatoirement correspondre à un attribut *réel*. De plus, depuis l'extérieur, nous n'avons aucun moyen de distinguer un *réel* attribut d'une propriété en lecture/écriture (donc avec un *setter*) puisque les deux se comportent strictement de la même manière.

Nous utiliserons beaucoup les propriétés dans ce correctif : c'est en Python la *bonne* (dans le sens la plus pythonic) manière de gérer des attributs privés avec *getter/setter* car tout se passe de manière totalement transparente.

Finissons ici sur une remarque assez importante : la transparence du traitement des propriétés est la raison pour laquelle elles sont si populaires, mais peut avoir un net désavantage. En effet, il est assez bien admis qu'accéder à une variable/un attribut est une opération *triviale* d'un point de vue *calculatoire* (disons qu'accéder à un attribut se fait assez logiquement en temps constant, ou encore en  $\mathcal{O}(1)$  pour plus de précision, c.f. la [section 1.5](#) pour les définitions précises). Dès lors, il faut réserver les propriétés à des traitements qui ne sont pas trop gourmands et réserver les traitements plus lourds dans des méthodes. Une utilisation commune (mais qui doit être bien documentée !) est la suivante : si une classe permet de récupérer la solution à un problème, il faut s'assurer que le problème ait été résolu avant de demander la solution (pas aberrant jusque là). Mais proposer une propriété `solution` qui appellerait systématiquement la méthode `solve` serait déraisonnable car pourrait amener à recalculer maintes fois une solution déjà trouvée. Une manière de contourner cela est la suivante :

---

```

1 class Solver:
2     def __init__(self):
3         ...
4         self.solution_ = None
5
6     def solve(self):
7         ...
8         self.solution_ = ...
9
10    @property
11    def solution(self):
12        if self.solution_ is None:
13            self.solve()
14        return self.solution_

```

---

En effet, si la méthode `solve` n'a pas encore été appelée, l'attribut `solution_` sera toujours à `None`, ce qui peut se détecter quand l'attribut virtuel (i.e. la propriété) `solution` va vouloir être lue, et la méthode `solve` peut donc être appelée à ce moment là. Comme la solution trouvée est gardée dans l'attribut `solution_`, la prochaine fois que l'attribut virtuel `solution` sera accédé, comme l'attribut `solution_` ne sera plus à `None`, il pourra directement être renvoyé (ce qui se fait donc en temps constant).

Notons pour finir (oui on était censé finir au paragraphe précédent, mais c'est juste une phrase) que tout comme le décorateur `staticmethod`, le décorateur `property` est un descripteur et n'est pas simplement une fonction. À nouveau, bien que ça ne soit pas l'envie qui manque, nous n'explicitons pas le sujet ici.

### 1.3.10 Itérables, itérateurs et générateurs

Un *itérable* est un objet sur lequel on peut *itérer*. Plus précisément, c'est un objet que l'on peut donner en paramètre à la fonction *built-in* `iter`. Le comportement de la fonction `iter` se paramétrise via la méthode spéciale `__iter__` (c.f. la sous-section 1.3.8). Cette fonction doit renvoyer un *itérateur*, i.e. un objet que l'on peut donner à la fonction `next`. Tout comme `iter`, cette fonction se paramétrise par la méthode spéciale `__next__`.

Un itérateur est donc un objet qui a pour but de renvoyer tous les éléments d'un objet à tour de rôle à chaque fois que la fonction `next` lui est appliquée. Lorsque `next` est appelée sur un itérateur qui a déjà renvoyé le dernier élément possible, il faut lancer une exception `StopIteration`. Prenons un exemple sur une liste (qui est donc un itérable) :

---

```

>>> l = [1, 2, 3]
>>> iterator = iter(l)
>>> iterator
<list_iterator object at 0x7fea6aa087f0>
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3

```

```
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

---

Nous observons bien ici que la variable `iterator` est de type `list_iterator` et qu'en appelant successivement la fonction `next` (qui va donc appeler la méthode `iterator.__next__`), nous récupérons tous les éléments de la liste jusqu'à arriver à la fin, en quel cas l'exception voulue est lancée. Nous pouvons donc écrire nous-mêmes un itérateur pour la classe `list` :

```
>>> class ListIterator:
...     def __init__(self, l):
...         self.l_ = l
...         self.idx_ = 0
...     def __next__(self):
...         if self.idx_ >= len(self.l_):
...             raise StopIteration()
...         self.idx_ += 1
...         return self.l_[self.idx_-1]
...     def __iter__(self):
...         return self
...
>>> iterator = ListIterator(l)
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in __next__
StopIteration
```

---

Remarquons qu'afin de pouvoir utiliser la syntaxe `for element in obj`, il faut que `obj` soit un itérable puisqu'en réalité cette ligne est équivalente à `for element in iter(obj)`. C'est également pour cette raison qu'un itérateur doit aussi être un itérable (i.e. définir une méthode `__iter__`) qui doit renvoyer `self` (i.e. l'itérateur en question). Cela permet d'appeler `iter` sur un itérateur sans que cela ne pose de problème.

Notons que dans plein de cas (typiquement itérer simplement sur un conteneur), il peut être pénible de devoir écrire la classe d'un itérateur alors qu'en réalité on voudrait (par exemple) simplement écrire une boucle `for`. Pour cela, nous pouvons plutôt utiliser les *générateurs*. Ce sont des fonctions qui, au lieu d'utiliser un `return` pour s'arrêter en renvoyer une valeur, utilise l'instruction `yield`. Le fonctionnement est assez similaire dans le sens où lorsqu'un `yield` est exécuté, la fonction s'arrête et la valeur est retournée, mais quand elle est appelée à nouveau (enfin pas exactement mais nous détaillerons juste après), elle

reprend exactement à l'endroit où elle était restée. En particulier `yield` va souvent être utilisé au sein d'une boucle. Nous pouvons donc réécrire beaucoup plus simplement notre itérateur de liste avec un générateur :

---

```
>>> def list_iterator(l):
...     for i in range(len(l)):
...         yield l[i]
...
>>> iterator = list_iterator(l)
>>> type(iterator)
<class 'generator'>
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

---

Bien sûr, c'est le rôle de l'interpréteur de déduire qu'il doit renvoyer un générateur (donc un itérateur) parce que l'instruction `yield` a été rencontrée (mais faites toujours très attention quand vous écrivez une fonction qui mélange `return` et `yield`, ça a tendance à mal se passer, il vaut toujours mieux choisir l'un ou l'autre). L'interpréteur crée donc un générateur en définissant *lui-même* la méthode `__next__` sur base du `yield`, mais les détails d'implémentation ne nous concernent pas ici : seul le fonctionnement est important.

Notons aussi que tout comme les listes peuvent être créées de manière condensée grâce aux *list comprehensions*, les générateurs le peuvent également via les *expressions de générateurs* dont la syntaxe est identique à celle des *list comprehensions* si ce n'est qu'il ne faut pas des crochets mais des parenthèses comme délimiteurs. Nous avons donc la forme la plus condensée de notre itérateur de liste :

---

```
>>> iterator = (l[i] for i in range(len(l)))
>>> iterator
<generator object <genexpr> at 0x7fea6ab324a0>
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

---

L'intérêt des itérateurs (et en particulier des générateurs) est qu'en plus de permettre

d'écrire facilement des itérateurs, l'entièreté des valeurs générées ne doit pas être stockée simultanément. Cela veut dire qu'en particulier il est possible de faire des générateurs infinis (e.g. faire un générateur qui va produire tous les nombres naturels) :

---

```

1 def naturals(start=0):
2     n = start
3     while True:
4         yield n
5         n += 1

```

---

Voire même encore mieux générer tous les nombres premiers :

---

```

1 def not_multiple_of(iterable, n):
2     for m in iterable:
3         if m%n != 0:
4             yield m
5
6 def gen_primes(numbers=None):
7     if numbers is None:
8         numbers = naturals(2)
9     p = next(numbers)
10    yield p
11    for new_prime in gen_primes(not_multiple_of(numbers, p)):
12        yield new_prime
13
14 if __name__ == '__main__':
15     primes = gen_primes()
16     for i in range(100):
17         print(next(primes))

```

---

Notons qu'ici, les valeurs générées sont bien gardées quelque part par la nature récursive du code. De plus, un générateur peut être *étendu* via l'instruction `yield from`. En effet, au lieu de devoir explicitement itérer sur les valeurs produites par l'appel récursif (l. 11-12), nous pouvons utiliser `yield from gen_primes(not_multiple_of(numbers, p))`.

Le fait de ne pas devoir construire explicitement l'entièreté du conteneur sur lequel on veut itérer est très pratique et est bien souvent utilisé en Python. En particulier, nous pouvons mentionner les fonctions `sum`, `map`, `enumerate` ou encore `reversed` (cette liste est bien sûr non exhaustive). Ces fonctions prennent en paramètre un itérable (donc potentiellement un itérateur) et renvoient également potentiellement un itérateur. La fonction `map` prend en paramètre un itérable et une fonction et renvoie un itérateur qui va appliquer cette fonction à tous les éléments de l'itérable. Nous pouvons donc l'écrire comme ceci :

---

```

1 def map(fct, iterable):
2     for x in iterable:
3         yield fct(x)

```

---

`enumerate(iterable, start=0)` prend en paramètre un itérable et potentiellement un entier et renvoie un itérateur qui produit des paires `(i, x)` où `i` est l'indice (en commen-

cant de start) et  $x$  est le  $i$ -start ème élément de l'itérable. Nous pouvons l'écrire comme ceci :

---

```

1 def enumerate(iterable, start=0):
2     n = start
3     for x in iterable:
4         yield n, x
5         n += 1

```

---

Parlons tout de même de `reversed` qui prend un itérable en paramètre et qui renvoie un itérateur qui parcourt cet itérable de la fin vers le début. Comme cette notion de *début* et de *fin* n'a pas de sens pour tout conteneur, nous ne pouvons pas en faire une implémentation générique. Mais si on veut permettre de produire un tel itérateur sur un type *user-defined*, il faut définir la méthode spéciale (c.f. la [sous-section 1.3.8](#)) `__reversed__`. Par exemple la méthode `list.__reversed__` peut ressembler à ceci :

---

```

1 def __reversed__(self):
2     for i in range(len(self)-1, -1, -1):
3         yield self[i]

```

---

Tant que nous en sommes à parler de `range`, ce type (car oui, `range` n'est pas une fonction, ou tout du moins la fonction `range()` n'est autre que le constructeur de la classe `range`) est basé sur la notion d'itérateur, c.f. l'[Exercice 3.7](#) pour plus d'informations.

### 1.3.11 Adresses et cache de références interne à CPython

Cette section est importante pour la [sous-section 13.1](#) mais n'est pas fondamentale pour pouvoir coder en Python de manière générale.

En Python, tous les objets ont leur propre type, mais aux yeux de l'interpréteur (ou tout du moins en ce qui concerne CPython), tous les objets sont en réalité des pointeurs vers un objet de type `PyObject` (qui est donc une structure contenant les informations importantes sur le nom du type, le module duquel il vient, un conteneur des attributs, un conteneur des méthodes, une structure pour l'héritage, etc.) Cette adresse peut en réalité être récupérée du côté Python en passant par la fonction `id`. Pour un objet  $x$ , appeler `id(x)` donne un entier permettant d'identifier sans ambiguïté l'objet  $x$  (ou du moins tant que  $x$  existe !) En pratique, afin de faire cela, CPython a choisi de renvoyer l'adresse de l'objet de type `PyObject` vu par l'interpréteur. Cette adresse ne peut en effet pas être partagée par plusieurs objets et est donc unique.

Il faut toutefois se rappeler du fonctionnement de la gestion de la mémoire de Python : l'interpréteur dispose d'un module que l'on appelle le *garbage collector* qui est chargé de déterminer quelles sont les zones mémoires qui ne sont plus utilisées par le programme en cours et qui peuvent donc être libérées. Il est possible d'interagir avec le *garbage collector* via le module `gc`, mais nous vous conseillons de faire preuve de beaucoup de prudence si vous décidez de suivre cette route.

En particulier, bien que le fonctionnement précis du *garbage collector* ne soit pas documenté (afin d'éviter que des programmes ne se reposent sur les particularités du gc qui peuvent être amenées à changer d'une version à l'autre), nous pouvons en dire certaines choses. Par exemple, tant qu'il existe des références vers un certain objet, il ne pourra être



libéré (et heureusement !) Cependant, libérer la dernière référence vers un objet ne garantit absolument pas que le gc s'en occupera directement. En effet, ce dernier doit faire un choix entre efficacité en mémoire et efficacité en temps de calcul. Il est assez clair qu'appeler le gc après chaque instruction pour voir s'il peut faire de la place est très loin d'être optimal car il tournera, la majorité du temps, dans le vide. À l'inverse, il faut bien l'appeler de temps en temps sinon un code aussi simple que celui-ci serait rapidement amené à planter à cause d'une allocation impossible alors qu'à chaque instant, l'espace mémoire nécessaire est tout à fait raisonnable :

---

```
1 for i in range(1000000):
2     l = list(range(i))
```

---

Afin de supprimer une référence vers un objet, il faut soit que le flux d'exécution du programme atteigne la fin du scope de vie de cette variable (e.g. à la fin d'une fonction, toutes les variables locales arrivent en fin de vie) soit expliciter la suppression de la référence via l'instruction `del` :

---

```
>>> a = 31
>>> b = 16
>>> a, b
(31, 16)
>>> del a
>>> a, b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

---

Parlons maintenant de quelques comportements particuliers concernant les références, les *singletons* (donc les objets qui par nature ne peuvent être dupliqués), les entiers et les strings.

Certaines valeurs sont, par choix d'implémentation, uniques. Parmi celles-ci nous pouvons compter `None`, `True` et `False`. En effet, plusieurs *variables* de type `bool` peuvent valoir la même valeur, mais toutes ces variables font en réalité référence à la même instance :

---

```
>>> a = True
>>> b = True
>>> a is b
True
>>> id(a) == id(b)
True
>>> a = not a
>>> id(a) == id(False)
True
>>> l = [1, 3, 2]
>>> id(l.sort()) == id(None)
True
```

---

Pour cette raison, les comparaisons à `None`, `True` et `False` se font toujours via l'opérateur `is` et pas via l'opérateur `==`. De tels objets sont appelés des *singletons* puisqu'il ne peut exister

qu'une seule instance ayant cette valeur à tout instant de vie de l'interpréteur.

Discutons maintenant le comportement particulier des entiers et des strings en Python (tout du moins avec CPython). Nous avons déjà parlé ci-dessus de l'opérateur `+=` et du fait que l'expression `a += b` *modifiait* l'objet `a` mais ne créait pas un nouvel objet (contrairement à `a = a+b`). Cette information était en réalité inexacte (allez disons *incomplète*). En effet, si pour une liste, ceci est bien vérifié :

---

```
>>> l = [1, 2, 3]
>>> old_id = id(l)
>>> l += [4]
>>> new_id = id(l)
>>> old_id == new_id
True
```

---

Ce n'est pas le cas pour un entier :

---

```
1 >>> a = 1
2 >>> old_id = id(a)
3 >>> a += 1
4 >>> new_id = id(a)
5 >>> old_id == new_id
6 False
```

---

La raison est la suivante : les entiers sont *immuables*. Un objet immuable ne peut donc, par définition, pas être modifié. De là, il y a deux manières de gérer un tel type : soit, comme les *tuples*, aucun opérateur de modification n'est rendu disponible, soit, comme les *int*, ces opérateurs existent mais vont, de manière transparente, créer une nouvelle variable et remplacer la *référence* interne de l'objet (oui oui, c'est possible...). D'ailleurs, si vous vous demandiez pourquoi `__iadd__` devait systématiquement renvoyer `self` (parce que bien sûr, vous vous le demandiez, je n'en doute pas), c'est précisément pour cette raison : la méthode ne doit en réalité pas *obligatoirement* renvoyer `self` mais doit en réalité renvoyer la (potentiellement) nouvelle référence à mettre dans l'objet après exécution de l'opérateur `+=`. Nous pouvons donc maintenant comprendre qu'en réalité `a += b` n'est *pas* interprété comme `a.__iadd__(b)` mais bien comme `a = a.__iadd__(b)`, ce qui est équivalent dans le cas où `__iadd__` renvoie `self`, mais qui joue une différence sur les types immuables. Vous pouvez donc implémenter votre propre type immuable de la sorte :

---

```
>>> class C:
...     def __init__(self, x):
...         self.x = x
...     def __iadd__(self, other):
...         new_c = C(self.x+other.x)
...         return new_c
...
>>> c1 = C(10)
>>> c2 = C(15)
>>> old_id = id(c1)
>>> c1 += c2
>>> new_id = id(c1)
```

```
>>> c1.x
25
>>> old_id == new_id
False
```

---

Mais je vous vois venir, vous pensez probablement : *ah, donc les entiers sont des singletons ?* Ce qui est, ma foi, légitime comme pensée, d'autant plus si vous le tentez directement dans l'interpréteur :

---

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a += 1
>>> a is b
False
>>> a -= 1
>>> a is b
True
```

---

En effet il apparaît ici clairement que 1 est un singleton : si deux variables sont toutes les deux égales à 1, alors elles sont en réalité la même variable (dans le sens où elles ont le même identifiant). Cependant, il faut faire attention ici :

---

```
>>> a = 1024
>>> b = 1024
>>> a is b
False
```

---

Voilà qui n'aidera pas pour la confusion ambiante. La raison est donnée *ici* : en effet puisque les *petits entiers* sont très souvent utilisés (e.g. comme indices dans des conteneurs), il a été choisi de les garder en cache et d'en faire des singletons (cela permet de ne pas avoir à réallouer de la mémoire à chaque fois qu'un objet valant 2 est demandé). Il a également fallu déterminer ce que *petit* voulait dire dans ce contexte, et l'intervalle entier (fermé) de -5 à 256 a été considéré comme un bon compromis.

Comme quoi, c'est un peu arbitraire, mais au final ce n'est pas très compliqué : les entiers sont immuables mais CPython garde tout de même en cache les petites valeurs. Donc à partir de 257 (compris), deux variables ayant la même valeur n'auront pas le même identifiant. Sauf que... :

---

```
1 >>> a = 1024
2 >>> b = 1024
3 >>> a is b
4 False
5 >>> a, b = 1024, 1024
6 >>> a is b
7 True
8 >>> c, d = 1024, 1024
9 >>> c is d
```

```

10 True
11 >>> a is c
12 False

```

---

Ce n'est (bien évidemment) pas aussi simple que ça. En effet, lorsque CPython lit une instruction, il va en extraire les *constantes* (i.e. ici les littéraux) et ne va pas les dupliquer si ce n'est pas nécessaire. En particulier, puisque `a, b = 1024, 1024` est en réalité équivalent à `a, b = (1024, 1024)`, où le tuple `(1024, 1024)` est d'abord construit et puis est *unpacked* dans les variables `a` et `b`, lors de la construction du tuple, l'interpréteur est libre d'effectuer ses propres optimisations.

Ce principe s'applique également directement aux chaînes de caractères (donc aux strings, `str`) : les `str` sont immuables :

```

1 >>> a = 'a'
2 >>> old_id = id(a)
3 >>> a += a
4 >>> new_id = id(a)
5 >>> old_id == new_id
6 False

```

---

Et une forme de *caching* existe sur les strings de petite taille :

```

1 >>> a = 'a'
2 >>> b = (a+a)[len(a):]
3 >>> a == b
4 True
5 >>> a is b
6 True
7 >>> a = 'abcdefghijklmnopqrstuvwxy'
8 >>> b = (a+a)[len(a):]
9 >>> a == b
10 True
11 >>> a is b
12 False

```

---

Le principe d'optimisation des instructions est également applicable à la gestion des strings. En effet `a = 'a'*100` ou `a = 'abc' + 'def'` ne vont pas appeler les opérateurs `__add__` et `__mul__` respectivement, mais l'interpréteur CPython va directement appliquer la duplication ou concaténation afin de gagner du temps. Les détails de la gestion des strings est parfois un peu opaque et il ne faut en aucun cas écrire un code qui se baserait sur ce comportement puisqu'il est susceptible de changer à chaque version de l'interpréteur.

### Le module `dis` (📄)

Nous l'avons dit plus haut, Python est un langage interprété et son interpréteur est (au moins dans le cadre de ce cours) CPython. Ce dernier est donc en charge de lire toutes les lignes de code, de les comprendre et de les exécuter dans le bon ordre. Pour cela, CPython fonctionne sur un système de *bytecode* qui est donc une version *précompilée* du code Python vers un langage plus proche d'un langage d'assemblage (c.f. le cours de fonction-

nement des ordinateurs pour plus d'informations sur le bytecode en général). Ce procédé est bien sûr tout à fait opaque et nous ne nous en rendons même pas compte lorsque nous exécutons un script. Il en reste tout de même parfois un signe, même après la fin de l'exécution d'un fichier : le fameux fichier `*.pyc` (pour *Python Compiled*). Ces fichiers au format binaire (donc illisible avec un éditeur de texte standard) ont un intérêt : lorsque CPython exécute un fichier (parce que vous l'avez appelé via la commande `python3 file.py` par exemple), il doit obligatoirement passer par ce bytecode (donc cette version précompilée de votre code Python). Cependant il est souhaitable d'éviter de devoir refaire ce procédé à chaque fois que le fichier `file.py` est exécuté (tout du moins tant qu'il n'est pas modifié). C'est pour cela que l'interpréteur crée ce fichier.

Notons qu'il est possible de demander directement à CPython quel est le bytecode correspondant à une instruction Python (où à un ensemble d'instructions d'ailleurs). Pour cela, nous pouvons utiliser le module `dis`, qui est un module standard, c.f. la [doc](#). C'est en effet ce module qui me permet d'affirmer que CPython va optimiser l'addition de strings si les deux opérands sont des littéraux :

---

```
>>> import dis
>>> dis.dis("a = 'abc' + 'def'")
 1          0 LOAD_CONST          0 ('abcdef')
          2 STORE_NAME          0 (a)
          4 LOAD_CONST          1 (None)
          6 RETURN_VALUE
```

---

Mais arrêtons-nous là sur le bytecode, ce n'est (et c'est bien triste) pas le sujet de ce cours.

### 1.3.12 Types standards et complexité

Au vu de la [documentation](#), la classification des types en Python n'est pas triviale. Explicitons ici cette classification et faisons un récapitulatif de la complexité des opérations disponibles sur ces types.

#### Types numériques

Cette section fait l'hypothèse que les notions ci-dessous sont acquises (c.f. INFO-F102 — Fonctionnement des ordinateurs pour plus d'informations) :

- les nombres sont représentés en binaire ;
- le CPU dispose de registres de taille *fixe* (32 ou 64 bits probablement sur votre machine) ;
- les entiers négatifs suivent la représentation par complément à 2 ;
- les nombres *décimaux* (ou plutôt à *virgule flottante*) sont encodés selon la norme IEEE-754.

**Représentations** Les types numériques que nous verrons ici sont respectivement `int`, `float`, `bool` et `complex`.

Dans sa version 2 (qui, rappelons-le, n'est plus maintenue et ne devrait *sous aucun prétexte* être utilisée à la place de Python 3, hormis potentiellement des raisons de compatibilité obscures), Python proposait deux types entiers : `int` et `long`. Le premier était un entier (signé) sur 16 bits et le second était un type entier à *taille variable*, i.e. une séquence

de bits de longueur variable. Depuis le passage à Python 3 (et même en réalité un peu avant, c.f. [PEP-237](#)), ces deux types ont été *fusionnés* : le type `long` n'existe plus, mais le comportement de l'actuel type `int` correspond à l'ancien type `long`, i.e. les entiers en Python sont actuellement *tous* à taille variable.

Le type `float` est en réalité (modulo ce dont tout objet a besoin aux yeux de l'interpréteur) un `double` en C, et fait donc 64 bits (sauf si le compilateur utilisé pour compiler l'interpréteur CPython ne supporte pas ce type pour l'architecture de destination, ce qui est disons... peu probable). Notons que bien que les entiers peuvent avoir une taille (et donc une précision) arbitraire, ce n'est absolument pas le cas des `float`, et donc tous les problèmes d'arrondis, de troncature, etc. sont également d'application en Python. Nous ne attarderons pas sur cela ici car nous n'utiliserons presque pas ce type dans ce cours. Pour plus de détails sur la norme IEEE-754, c.f. le cours INFO-F102 — Fonctionnement des ordinateurs, et pour plus d'informations sur les problèmes de stabilité liés à l'utilisation de nombres à virgule flottante, c.f. le cours INFO-F205 — Calcul formel et numérique.

Le type `complex` est représenté via deux `double` en C, en particulier toutes les remarques sur le type `float` sont également d'application ici. À nouveau, nous n'utiliserons pas ce type dans ce cours, nous en resterons donc là.

Le type `bool` est un peu particulier dans le sens où ce dernier n'existe pas aux yeux de l'interpréteur... En effet, bien qu'il soit tout à fait possible de créer des variables booléennes dans un code Python, l'interpréteur CPython ne définit pas un type `PyBoolObject` comme il définit pourtant `PyLongObject`, `PyFloatObject` et `PyComplexObject`. L'interpréteur se contente de définir deux *variables* `Py_True` et `Py_False` et s'arrange pour systématiquement retomber dessus, ce qui est possible puisque ces derniers sont des singletons (c.f. la discussion à ce sujet dans la [sous-section 1.3.11](#)). Ces deux variables sont en réalité de type `PyLongObject` et valent respectivement 1 et 0.

Mais trêve de digressions, les détails de représentation des booléens au sein de l'interpréteur CPython, bien que très intéressants, ne sont pas nécessaires. Gardons en tête que le type *existe* en Python, mais que les booléens se comportent comme des entiers, comme nous pouvons le remarquer en utilisant `isinstance` :

---

```
>>> isinstance(True, int)
True
>>> isinstance(False, int)
True
```

---

**Complexités** Il est clair que les opérations arithmétiques sur `float` (et par extension `complex`) se font en  $\mathcal{O}(1)$ . De même, les opérateurs booléens peuvent également s'exécuter en  $\mathcal{O}(1)$ . Nous pouvons également dire que les opérations arithmétiques sur des entiers *classiques* (entendons par là de taille bornée et définie, e.g. les entiers en C/C++) puisqu'il s'agit d'opérations directement prévues dans l'ALU. Mais les entiers dont nous disposons ici ne sont pas de taille bornée. Nous pouvons donc aisément nous dire convaincre que multiplier deux nombres chacun sur  $\sim 10\,000$  chiffres demandera beaucoup plus de travail que si les entiers en question étaient sur quelques chiffres. En effet, puisque le nombre  $N$  nécessite  $1 + \lfloor \log_2 n \rfloor \sim \log_2 N$  bits pour le représenter, l'addition de deux nombres bornés par  $N$  demande  $\mathcal{O}(\log_2 N)$  opérations et la multiplication naïve de deux tels nombres nécessite  $\mathcal{O}((\log N)^2)$  opérations. En réalité, CPython est plus efficace que ça et n'utilise pas l'algorithme naïf pour la multiplication de deux nombres entiers mais utilise l'algo-

rithme de Karatsuba qui requiert  $\mathcal{O}((\log N)^{\log_2 3})$  opérations. Pour plus d'informations sur les algorithmes de multiplication de nombres entiers (ou de matrices) et sur des exposants fractionnaires dans les complexités, c.f. la discussion dans l'[Exercice 5.6](#).

### Conteneurs de type séquence

Les *séquences* sont des types définis par la présence des opérations suivantes (c.f. [ici](#)) :

- `x in s` via `__contains__`;
- `s1 + s2` via `__add__`;
- `n * s` via `__mul__`;
- `s[i]` via `__getitem__` (les *slices* doivent également être gérées par `__getitem__`);
- `len(s)` via `__len__`;
- `s.index(x)`;
- `s.count(x)`.

Notons que la documentation précise également que les fonctions `min` et `max` peuvent être appelées sur une séquence, alors qu'il n'est pas requis qu'une séquence définisse une méthode `__iter__` ! La raison est que la discussion de la [sous-section 1.3.10](#) est incomplète : tout comme l'instruction `a + b` ne correspond pas simplement à `a.__add__(b)` (c.f. discussion de la [sous-section 1.3.8](#)), la fonction `iter` ne se contente pas simplement d'appeler la méthode `__iter__` sur un objet. En effet, comme on peut le voir dans le [code de CPython](#), la fonction `iter` se comporte plutôt comme ceci :

---

```

1 def iter(obj):
2     if hasattr(obj, '__iter__'):
3         return obj.__iter__()
4     if hasattr(obj, '__getitem__'):
5         if hasattr(obj, '__len__'):
6             return (obj[i] for i in range(len(obj)))
7         else:
8             return iterseq(obj)
9     raise TypeError(f"'{type(obj)}' object is not iterable");
10
11 def iterseq(obj):
12     i = 0
13     while True:
14         try:
15             yield obj[i]
16         except IndexError:
17             raise StopIteration
18     i += 1

```

---

Les séquences que nous manipulerons ici sont les suivantes :

- `list`;
- `tuple`;
- `range`.

Plus de détails sur le fonctionnement du type `list` peuvent être trouvés dans l'[Exercice 3.5](#) et plus de détails sur le type `range` peuvent être trouvés dans l'[Exercice 3.7](#). Le comportement du type `tuple` peut être assimilé à celui du type `list` mais sans les méthodes de

modification.<sup>2</sup>

Reprenons la complexité des opérations dans le tableau suivant, où nous considérons que  $k$  est un entier ;  $s$ ,  $s_1$  et  $s_2$  sont des séquences de longueur respective  $n$ ,  $n_1$  et  $n_2$  ;  $x$  est un objet quelconque et `slice_indices` est une slice (c.f. l’Exercice 3.6) de longueur  $\ell$  :

	list	tuple	range
<code>x in s</code>	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
<code>s1 + s2</code>	$\Theta(n_1 + n_2)$	$\Theta(n_1 + n_2)$	
<code>k * s</code>	$\Theta(kn)$	$\Theta(kn)$	
<code>s[i]</code>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<code>s[slice_indices]</code>	$\Theta(\ell)$	$\mathcal{O}(\ell)$	$\mathcal{O}(1)$
<code>len(s)</code>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
<code>s.index(x)</code>	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
<code>s.count(x)</code>	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
<code>l.append(x)</code>	$\mathcal{O}(n)$		
<code>l.insert(k, x)</code>	$\mathcal{O}(n)$		
<code>l.pop(i)</code>	$\mathcal{O}(n)$		

TABLE 1.1 – Récapitulatif des complexités des opérations sur les séquences.

Quelques remarques concernant les slices :

- Si `slice_indices` est défini comme ceci : `slice_indices = slice(i, j, k)`, alors la taille de ce slice est  $\mathcal{O}((j - i)/k)$  mais n’est pas  $\Theta((j - i)/k)$ . En effet puisque ces conteneurs permettent que l’indice de fin d’un slice soit plus grand que la taille du conteneur,  $j$  peut être arbitrairement grand par rapport  $n$  et donc  $(j - i)/k$  peut être arbitrairement grand par rapport à  $(n - i)/k$ . Nous pouvons tout de même dire que  $\ell = \Theta((\min\{n, j\} - i)/k)$
- Les complexités données sont relatives à la création des sous-conteneurs (i.e. via `__getitem__`), mais dans le cas d’un objet de type `list` (qui est donc mutable), les slices peuvent être utilisées dans `__setitem__`, voire même dans `__delitem__`, et dans ces cas-ci la complexité est dominée par respectivement  $\Theta(\ell + n)$  et  $\Theta(n)$ .

Les méthodes `append`, `insert` et `pop` ne sont données que pour le type `list` puisque ces opérations ne font pas partie du protocole de séquence décrit plus haut. La complexité donnée ici est donc une borne supérieure correspondant au pire des cas. Plus d’informations peuvent être trouvées dans l’Exercice 3.5.

### 1.3.13 Les types de CPython 🚩

Comme mentionné plus haut, l’interpréteur CPython est codé en C et ne dispose donc pas des outils proposés par la programmation orienté objet. En particulier, il n’est pas fait usage de l’héritage dans le code de CPython, notion qui existe pourtant en Python. En effet, tout type en Python hérite de `object`, et d’une manière assez similaire, le type de tout objet Python au sein de l’interpréteur commence par une instance de `PyObject`. L’intérêt est le suivant : si  $p$  est un pointeur vers un objet Python au sein de l’interpréteur (e.g. un `PyTupleObject` ou un `PyListObject`), alors `(PyObject*)p` est un pointeur valide un `PyObject`.

2. Alors en réalité, l’implémentation est assez différente et je ne me gênerai pas d’en dire tout le mal que j’en pense dans la sous-section 1.3.13.



Le type `PyObject` en lui-même est défini de la manière suivante (c.f. [le code de CPython](#)) :

---

```

1 struct _object {
2     size_t refcount;
3     PyTypeObject* type;
4 };
5 typedef struct _object PyObject;

```

---

où `refcount` est le compteur de référence de l'objet, utilisé par le *garbage collector* (c.f. [sous-section 1.3.11](#)) et `type` est un pointeur vers un objet de type `PyTypeObject` contenant toutes les informations relatives au type (au sens Python) de l'objet (en particulier le nom, la structure d'héritage, les pointeurs vers fonctions correspondant aux méthodes, etc.).

Toutes (enfin disons *la plupart*) les fonctions de l'interpréteur qui prennent un objet Python en paramètre prennent donc un pointeur vers un `PyObject` et doivent effectuer le *cast* explicitement. Pour cela, il faut d'abord vérifier que le paramètre reçu corresponde bien à un objet du type demandé. Ceci est effectué via des *fonctions* de l'interpréteur (en réalité ce sont des macros qui font toutes appel à `PyType_CheckExact` ou `PyType_FastSubclass` ; le premier est une macro qui correspond à un appel à la fonction `Py_IS_TYPE` qui compare le type de l'objet reçu à l'adresse de l'*unique* instance de `PyTypeObject` qui correspond au type recherché – car oui, les types sont des singletons – et le second fait appel à `PyType_HasFeature` pour vérifier certains flags correspondant aux types recherchés) et une exception est lancée si le type ne correspond pas.

**We need to talk about les tuples. . .** Le type `list` est défini par le type C `PyListObject` défini (plus ou moins) comme suit (c.f. [le code de CPython](#)) :

---

```

1 typedef struct {
2     PyObject base;
3     PyObject** items;
4     size_t allocated;
5 } PyListObject;

```

---

où `base` est l'instance de `PyObject` comme discuté ci-dessus, `items` est un pointeur (qui contiendra l'adresse d'un tableau alloué dynamiquement) vers des `PyObject*` (i.e. des objets Python quelconques) et `allocated` donne la taille allouée du conteneur (attention, ceci ne correspond pas à `len(l)`, c.f. [l'Exercice 3.5](#)). Il est en effet assez logique qu'une liste peut donc être créée (schématiquement) comme suit :

---

```

1 PyObject* allocate_list(size_t size) {
2     PyListObject* ret = PyObject_GC_New(PyListObject, &PyList_Type);
3     ret->items = malloc(size * sizeof(*ret));
4     return (PyObject*)ret;
5 }

```

---

où `PyObject_GC_New` est une fonction qui alloue un nouvel objet, spécifie son type (via l'attribut `PyObject.type` et l'inclut dans la *vision* du *garbage collector*). Dès lors une liste de

taille `size` va donc allouer dynamiquement un tableau de `size` pointeurs vers `PyObject`.

Et `tuple` dans tout ça ? Et bien l'implémentation des tuples utilise un vieux concept du C appelé FAM (*Flexible Array Member*) dont le concept est le suivant : si un type se *termine* (au sens de sa représentation binaire d'un point de vue croissant des adresses) par un tableau (statique), l'utilisation de la syntaxe `obj.array[i]` permet de sortir de l'espace statique alloué pour cet objet. Dès lors si ledit tableau est de taille 1 mais que l'espace mémoire de l'instance de ce type est allouée dynamiquement en utilisant plus de mémoire que nécessaire, nous pouvons simuler un tableau alloué dynamiquement en économisant un niveau d'indirection ! Conceptuellement :

---

```

1 typedef struct {
2     size_t size;
3     int array[1];
4 } int_array_t;
5
6 int_array_t* allocate_array(size_t size) {
7     assert(size > 0);
8     int_array_t* ret = malloc(sizeof(size_t) + size * sizeof(int));
9     ret->size = size;
10    return ret;
11 }
```

---

Pour cela, une allocation dynamique des objets de type `int_array_t` est nécessaire, mais si `p` est un pointeur de type `int_array_t*`, alors `p->array[i]` est un accès *valide* à un tableau semi-statique.

Tout cela peut sembler ingénieux (et *en effet* permet d'obtenir un code plus efficace puisque un niveau d'indirection est économisé) mais est fondamentalement confus pour des gens lisant le code. Venons-en donc aux tuples. Voici (à nouveau schématiquement) la représentation du type `PyTupleObject` de CPython ainsi qu'une fonction d'allocation d'un tuple de la bonne taille :

---

```

1 typedef struct {
2     PyObject* base;
3     Py_ssize_t size;
4     PyObject* obj[1];
5 } PyTupleObject;
6
7 PyObject* allocate_tuple(Py_ssize_t size) {
8     assert(size > 0);
9     PyTupleObject* ret = malloc(
10        sizeof(PyTupleObject) + (size-1) * sizeof(PyObject*)
11    );
12    ret->size = size;
13    return ret;
14 }
```

---

Conclusion : c'est dégueulasse.

## 1.4 Notions ensemblistes

**Définition 3.** Nous utilisons les notations suivantes pour désigner les intervalles entiers (pour  $a, b \in \mathbb{Z}$ ) :

- $\llbracket a, b \rrbracket := \{m \in \mathbb{Z} \text{ s.t. } a \leq m \leq b\} = \{a, a+1, \dots, b-1, b\}$ ;
- $\llbracket a, b \llbracket := \{m \in \mathbb{Z} \text{ s.t. } a \leq m < b\} = \{a, a+1, \dots, b-1\}$ ;
- $\llbracket a, b \rrbracket := \{m \in \mathbb{Z} \text{ s.t. } a < m \leq b\} = \{a+1, \dots, b-1, b\}$ ;
- $\llbracket a, b \llbracket := \{m \in \mathbb{Z} \text{ s.t. } a < m < b\} = \{a+1, \dots, b-1\}$ .

**Définition 4.** Soient deux ensembles  $E$  et  $F$  (finis ou infinis). Le *produit cartésien* entre  $E$  et  $F$  est l'ensemble  $E \times F$  défini par :

$$E \times F := \{(x, y) \text{ s.t. } x \in E, y \in F\}.$$

Soit une suite finie d'ensembles  $E_1, \dots, E_n$ . Le *produit cartésien* des ensembles  $E_i$  est l'ensemble défini par :

$$\prod_{i=1}^n E_i := \{(x_1, \dots, x_n) \text{ s.t. } \forall i \in \llbracket 1, n \rrbracket : x_i \in E_i\}.$$

On note  $E^n$  le produit cartésien de  $n$  copies de l'ensemble  $E$ .

**Définition 5.** Soient deux ensembles  $E$  et  $F$ . On définit :

- l'*intersection* de  $E$  et  $F$ , notée  $E \cap F$  par l'ensemble  $E \cap F := \{x \in E \text{ s.t. } x \in E \text{ et } x \in F\}$ ;
- l'*union* de  $E$  et  $F$ , notée  $E \cup F$ , par l'ensemble  $E \cup F := \{x \text{ s.t. } x \in E \text{ ou } x \in F\}$ ;
- la *différence* entre  $E$  et  $F$ , notée  $E \setminus F$  (et parfois  $E - F$ , mais nous n'utiliserons pas cette notation ici), par l'ensemble  $E \setminus F = \{x \text{ s.t. } x \in E \text{ et } x \notin F\}$ .

On dit que  $E$  et  $F$  sont *disjoints* si  $E \cap F = \emptyset$ . Afin d'insister sur le fait que deux ensembles dont on souhaite prendre l'union sont disjoints, on peut écrire  $E \sqcup F$  afin de désigner l'union  $E \cup F$  sous l'hypothèse  $E \cap F = \emptyset$ . La notation  $\uplus$  est également parfois utilisée, mais n'apparaîtra pas dans ce document.

**Définition 6.** Soient deux ensembles (quelconques)  $E$  et  $F$ . On appelle *relation* (binaire) entre  $E$  et  $F$  tout sous-ensemble  $\mathcal{R} \subset E \times F$ . On dit que  $e \in E$  et  $f \in F$  sont en *relation*  $\mathcal{R}$ , que l'on note  $e\mathcal{R}f$  lorsque  $(e, f) \in \mathcal{R}$ .

Pour un ensemble (quelconque)  $E$ , une relation  $\sim$  entre  $E$  et lui-même est dite *d'équivalence* lorsque :

- réflexivité** si  $x \in E$ , alors  $x \sim x$ ;
- transitivité** si  $x, y, z \in E$  tels que  $x \sim y$  et  $y \sim z$ , alors  $x \sim z$ ;
- symétrie** si  $x, y \in E$  tels que  $x \sim y$ , alors  $y \sim x$ .

**Remarque.** La relation d'équivalence triviale sur un ensemble est une forme d'identité (chaque élément n'est en relation qu'avec lui-même). Par exemple  $=$  est une relation d'équivalence sur  $\mathbb{R}$ . On peut cependant construire des relations d'équivalence bien moins triviale, par exemple la congruence modulo un nombre entier est une relation d'équivalence (i.e. à  $n$  fixé,  $x \sim y$  lorsque  $x$  modulo  $n$  et  $y$  modulo  $n$  donnent la même valeur, ou plus proprement lorsque  $x - y$  est un multiple de  $n$ ).


Une relation d'ordre est également une relation binaire (usuellement notée  $\leq$ ) entre un ensemble et lui-même qui doit être réflexive et transitive, mais qui est anti-symétrique,

c'est-à-dire si  $x, y \in E$  tels que  $x \leq y$  et  $y \leq x$ , alors  $x = y$ .

**Remarque.** Les relations entre cardinalités d'ensembles se définissent normalement en terme de fonctions bijectives, injectives ou surjectives afin d'être entièrement rigoureux. Cependant, bien que l'envie ne manque pas, cette matière est gardée pour le cours INFO-F307 — mathématiques discrètes.

Dans le doute (parce qu'à choisir, amusons-nous un petit peu) : on dit, pour deux ensembles finis  $E$  et  $F$ , que  $|E| \leq |F|$  s'il existe une application injective  $\varphi : E \rightarrow F$ ;  $|E| \geq |F|$  s'il existe une application surjective  $\phi : E \rightarrow F$  et  $|E| = |F|$  s'il existe une application bijective  $\psi : E \rightarrow F$ . On peut même uniquement se réduire au cas  $|E| \leq |F|$  lorsqu'il existe une application injective  $E \rightarrow F$  puisque l'existence d'une telle application implique l'existence d'une application surjective  $\phi : F \rightarrow E$  (savez-vous le montrer?) et donc le cas  $|E| = |F|$  se déduit de l'existence d'une application injective et d'une application surjective qui impliquent l'existence d'une application bijective.

Nous pouvons même aller plus loin en parlant du théorème de Bernstein (ou Bernstein-Schröder ou encore Bernstein-Schröder-Cantor).

**Théorème 1.8** (Bernstein ). Si  $\phi : E \rightarrow F$  et  $\psi : F \rightarrow E$  sont injectives, alors il existe  $\theta : E \rightarrow F$  bijective.

Démonstration. Nous allons noter  $\phi^{-1} : \phi(E) \rightarrow E : \phi(x) \mapsto x$  et  $\psi^{-1} : \psi(F) \rightarrow E : \psi(y) \mapsto y$  les inverses partielles. Pour tout  $x \in E$ , définissons la chaîne  $\mathcal{C}_x \subseteq E$  comme suit :

$$\mathcal{C}_x := \left\{ (\psi \circ \phi)^k(x) \right\}_{k \geq 0}.$$

Notons que cette chaîne est potentiellement infinie mais peut-être finie (e.g. si  $\psi(\phi(x)) = x$ ). Maintenant définissons la relation d'équivalence suivante sur  $E : x_1 \sim x_2$  lorsque  $x_1 \in \mathcal{C}_{x_2}$  ou  $x_2 \in \mathcal{C}_{x_1}$ . Il est clair que  $x \in \mathcal{C}_x$  pour tout  $x$ , et donc  $\sim$  est réflexive. De plus  $\sim$  est symétrique par définition. Il reste à vérifier qu'elle est transitive : prenons  $x_1, x_2, x_3 \in E$  tels que  $x_1 \sim x_2$  et  $x_2 \sim x_3$ . Séparons l'analyse en 4 cas :

- (i)  $x_1 \in \mathcal{C}_{x_2}$  et  $x_2 \in \mathcal{C}_{x_3}$  ;
- (ii)  $x_1, x_3 \in \mathcal{C}_{x_2}$  ;
- (iii)  $x_2 \in \mathcal{C}_{x_1} \cap \mathcal{C}_{x_3}$  ;
- (iv)  $x_2 \in \mathcal{C}_{x_1}$  et  $x_3 \in \mathcal{C}_{x_2}$ .

Dans le cas (i), il existe  $k_1 \geq 0$  tel que  $x_1 = (\psi \circ \phi)^{k_1}(x_2)$  et  $k_2 \geq 0$  tel que  $x_2 = (\psi \circ \phi)^{k_2}(x_3)$ , et donc  $x_1 = (\psi \circ \phi)^{k_1+k_2}(x_3)$ , i.e.  $x_1 \sim x_3$ . Dans le cas (ii), il existe  $k_1, k_3 \geq 0$  tels que  $x_1 = (\psi \circ \phi)^{k_1}(x_2)$  et  $x_3 = (\psi \circ \phi)^{k_3}(x_2)$ . Sans perte de généralité supposons  $k_1 \leq k_3$  et donc :

$$x_3 = (\psi \circ \phi)^{k_3-k_1} \left( (\psi \circ \phi)^{k_1}(x_2) \right) = (\psi \circ \phi)^{k_3-k_1}(x_1),$$

i.e.  $x_1 \sim x_3$ . Dans le cas (iii), il existe  $k_1, k_3 \geq 0$  tels que  $(\psi \circ \phi)^{k_1}(x_1) = x_2 = (\psi \circ \phi)^{k_3}(x_3)$ . Sans perte de généralité supposons  $k_1 \leq k_3$  et donc :

$$x_1 = (\psi \circ \phi)^{-k_1}(x_2) = (\psi \circ \phi)^{k_3-k_1}(x_3),$$

i.e.  $x_1 \sim x_3$ . Finalement, dans le cas (iv), tout comme dans le cas (i), nous savons que  $x_3 \in \mathcal{C}_{x_1}$  et donc  $x_1 \sim x_3$ .

Dans tous les cas, nous savons que  $x_1 \sim x_3$ , et nous pouvons déduire que  $\sim$  est bien une relation d'équivalence. En particulier  $E / \sim$  est une partition de  $E$  (notons les classes d'équivalence  $[x]_{\sim}$ ).

Définissons maintenant  $\theta : E \rightarrow F : x \mapsto \theta(x)$  où nous définissons  $\theta(x)$  comme ceci :

- S'il existe  $\tilde{x} \in E$  tel que  $[x]_{\sim} = \mathcal{C}_{\tilde{x}}$ , alors nous savons que soit (a)  $x \notin \psi(F)$  soit (b)  $\psi(x) \notin \phi(E)$ . Dans le cas (a),  $\phi|_{[x]_{\sim}}$  est une bijection donc  $\theta(x) = \phi(x)$  et dans le cas (b),  $\psi^{-1}|_{[x]_{\sim}}$  est une bijection et donc  $\theta(x) = \psi^{-1}(x)$ .
- Sinon  $(\psi \circ \phi)$  est une bijection sur  $[x]_{\sim}$  et donc  $\theta(x) = \phi(x)$ .

Il est clair que  $\theta$  est bien définie sur  $E$ , et par construction  $\theta$  est bien inversible.  $\square$

**Proposition 1.9.** Soient deux ensembles finis  $E$  et  $F$ . Alors :

1.  $|\emptyset| = 0$ ;
2. si  $E \subseteq F$ , alors  $|E| \leq |F|$ ;
3.  $|E \times F| = |E| \cdot |F|$ ;
4.  $|E \cup F| \leq |E| + |F|$ ;
5. si de plus  $E$  et  $F$  sont disjoints, alors  $|E \sqcup F| = |E| + |F|$ ;

*Démonstration.* Commençons par le dernier point. Il est clair que si  $E = \{x_1, \dots, x_n\}$  et  $F = \{y_1, \dots, y_m\}$  sont eux ensembles disjoints, alors :

$$E \sqcup F = \{x_1, \dots, x_n, y_1, \dots, y_m\}$$

contient bien  $n + m = |E| + |F|$  éléments.

Il est également clair que  $|\emptyset| = 0$ . Cela peut, par exemple se montrer par le fait que tout ensemble  $E$  satisfait l'égalité  $E = E \sqcup \emptyset$ , donc  $|E| = |E| + |\emptyset|$ , ce qui implique  $|\emptyset| = 0$ .

Maintenant observons que si  $E \subseteq F$ , alors  $F = E \sqcup (F \setminus E)$ . En particulier :

$$|F| = |E| + \underbrace{|F \setminus E|}_{\geq 0} \geq |E|.$$

Dès lors, pour deux ensembles quelconques  $E$  et  $F$  :

$$E \cup F = E \sqcup (F \setminus E),$$

en particulier :

$$|E \cup F| = |E| + |F \setminus E|.$$

Or  $F \setminus E \subseteq F$ , donc en particulier  $|F \setminus E| \leq |F|$ .

Finalement, nous pouvons écrire :

$$E \times F = \bigsqcup_{x \in E} \{(x, y) \text{ s.t. } y \in F\} = \bigsqcup_{x \in E} \bigsqcup_{y \in F} \{(x, y)\}.$$

Dès lors en prenant la cardinalité :

$$|E \times F| = \sum_{x \in E} \sum_{y \in F} \underbrace{|\{(x, y)\}|}_{=1} = |E| |F|.$$

$\square$

## 1.5 Notions asymptotiques de Landau

**Définition 7.** Soient deux fonctions  $f, g : \mathbb{N} \rightarrow \mathbb{R}$ . On dit que :

—  $f = \mathcal{O}(g)$  (qui se lit *f est un grand O de g*) lorsque :

$$\exists N \in \mathbb{N}, C > 0 \text{ s.t. } \forall n > N : |f(n)| \leq C |g(n)| ;$$

—  $f = \Omega(g)$  (qui se lit *f est un grand Omega de g*) lorsque :

$$\exists N \in \mathbb{N}, C > 0 \text{ s.t. } \forall n > N : |f(n)| \geq C |g(n)| ,$$

i.e. lorsque  $g = \mathcal{O}(f)$  ;

—  $f = \Theta(g)$  (qui se lit *f est un grand Theta de g*) lorsque :

$$\exists N \in \mathbb{N}, C_1, C_2 > 0 \text{ s.t. } \forall n > N : C_1 f(n) \leq g(n) \leq C_2 f(n),$$

i.e. lorsque  $f = \mathcal{O}(g)$  et  $f = \Omega(g)$ .

Donc  $f = \mathcal{O}(g)$  désigne le fait que  $g$  est une borne supérieure sur l'ordre de grandeur de la croissance de la fonction  $f$  ; et  $f = \Theta(g)$  désigne le fait que l'ordre de grandeur de la croissance de la fonction  $f$  est le même que celui de la fonction  $g$ .

**Attention :**  $f = \Theta(g)$  n'implique pas que  $f = Cg$  pour une constance  $C > 0$ . Pouvez-vous trouver un contre-exemple ?

**Remarque.** Pour deux fonctions  $f, g : \mathbb{N} \rightarrow \mathbb{R}$ ,  $f = \mathcal{O}(g)$  si et seulement si la fonction  $n \mapsto \left| \frac{f(n)}{g(n)} \right|$  est bornée (à partir d'un certain  $n_0$ ).

Il est cependant incorrect de dire que  $f = \mathcal{O}(g)$  seulement si la limite suivante existe et est finie (savez-vous trouver pourquoi ?) :

$$\lim_{n \rightarrow +\infty} \left| \frac{f(n)}{g(n)} \right|.$$

Cette condition est tout de même une condition suffisante pour que  $f = \mathcal{O}(g)$ .

**Définition 8.** Soient deux fonctions  $f, g : \mathbb{N} \rightarrow \mathbb{R}_0^+$ . On dit que  $f$  et  $g$  sont équivalentes asymptotiquement, que l'on note  $f \sim g$ , lorsque :

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 1.$$

**Proposition 1.10.** Soient deux fonctions  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ . Si  $f \sim g$ , alors  $f = \Theta(g)$  ;

*Démonstration.* Par hypothèse, nous savons que  $\frac{f(n)}{g(n)} \xrightarrow{n \rightarrow +\infty} 1$ , et donc  $\frac{g(n)}{f(n)} \xrightarrow{n \rightarrow +\infty} 1$  également. Par la remarque ci-dessus, nous pouvons déduire  $f = \mathcal{O}(g)$  et  $g = \mathcal{O}(f)$ , i.e.  $f = \Theta(g)$ .  $\square$

**Proposition 1.11.** Soient  $f_1, f_2, g_1, g_2 : \mathbb{N} \rightarrow \mathbb{R}^+$  telles que  $f_1 = \mathcal{O}(g_1)$  et  $f_2 = \mathcal{O}(g_2)$ . Alors  $f_1 \cdot f_2 = \mathcal{O}(g_1 \cdot g_2)$ .

**Remarque.** Par abus de notation, nous notons également cela comme suit :

$$\mathcal{O}(g_1) \cdot \mathcal{O}(g_2) = \mathcal{O}(g_1 \cdot g_2)$$

Démonstration. Par hypothèse, nous savons qu'il existe  $N_1, N_2, C_1, C_2 > 0$  tels que :

$$\begin{aligned} \forall n > N_1 : f_1(n) &\leq C_1 g_1(n), \\ \forall n > N_2 : f_2(n) &\leq C_2 g_2(n). \end{aligned}$$

Posons alors  $N := \max\{N_1, N_2\}$ . Soit  $n \in \mathbb{N}$  tel que  $n > N$ . Alors nous avons :

$$(f_1 \cdot f_2)(n) = f_1(n)f_2(n) \leq C_1 g_1(n) C_2 g_2(n) = C_1 C_2 (g_1 \cdot g_2)(n).$$

□

**Corollaire 1.12.** Soient  $f_1, f_2, g_1, g_2 : \mathbb{N} \rightarrow \mathbb{R}^+$  telles que  $f_1 = \Theta(g_1)$  et  $f_2 = \Theta(g_2)$ . Alors  $f_1 \cdot f_2 = \Theta(g_1 \cdot g_2)$ .

Démonstration. Par définition de  $\Theta$ , nous savons que  $f_1 = \mathcal{O}(g_1)$  et  $f_2 = \mathcal{O}(g_2)$ , et nous savons également que  $g_1 = \mathcal{O}(f_1)$  et  $g_2 = \mathcal{O}(f_2)$ . En appliquant deux fois la proposition précédente, nous avons  $f_1 \cdot f_2 = \mathcal{O}(g_1 \cdot g_2)$  et  $g_1 \cdot g_2 = \mathcal{O}(f_1 \cdot f_2)$ , i.e.  $f_1 \cdot f_2 = \Theta(g_1 \cdot g_2)$ . □

Cette propriété est très importante pour l'analyse de complexité des algorithmes. En effet, supposons que nous ayons une fonction `fct(i, n)` dont le nombre d'opérations dans le pire des cas est  $\mathcal{O}(n)$ . Alors il est assez clair que le code suivant nécessite  $\mathcal{O}(n^2)$  opérations dans le pire des cas :

---

```

1 for i in range(n):
2     fct(i, n)

```

---

En effet, puisque la fonction `fct` est appelée  $n = \mathcal{O}(n)$  fois, le nombre total d'opérations est :

$$\mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2).$$

**Proposition 1.13.** Soient  $f, g, h : \mathbb{N} \rightarrow \mathbb{R}^+$  telles que  $f = \mathcal{O}(g)$  et  $g = \mathcal{O}(h)$ . Alors  $f = \mathcal{O}(h)$ .

**Remarque.** On appelle cela la transitivité de la relation grand O.

Démonstration. Par hypothèse, nous savons qu'il existe  $C_f, C_g, N_f, N_g > 0$  tels que :

$$\begin{aligned} \forall n > N_f : f(n) &\leq C_f \cdot g(n), \\ \forall n > N_g : g(n) &\leq C_g \cdot h(n). \end{aligned}$$

Posons alors  $N := \max\{N_f, N_g\}$ . Soit  $n \in \mathbb{N}$  tel que  $n > N$ . Nous avons alors :

$$f(n) \leq C_f \cdot g(n) \leq (C_f \cdot C_g) \cdot h(n).$$

□

Cette proposition est à comprendre comme suit : si  $g$  borne le comportement de  $f$  asymptotiquement et si  $h$  borne le comportement de  $g$  asymptotiquement, alors  $h$  borne le comportement de  $f$  asymptotiquement. Attention cependant à toujours donner une information la plus précise possible lors d'une analyse de complexité. Prenons le code suivant :

---

```

1 mnemonic = input('Quelle est la mnémonique du meilleur TP de l\'ULB ?')
2 if mnemonic == 'INFO-F103':
3     print('Bien vu !')
4 else:
5     print('Bien essayé mais non...')

```

---

Il est clair que le nombre d'opérations est  $\mathcal{O}(1)$  (il y a toujours un `input`, une comparaison et un `print`, peu importe la valeur de `mnemonic`). Il est cependant *techniquement correct* de dire que le nombre d'opérations est  $\mathcal{O}(2^n)$  où  $n == \text{len}(\text{mnemonic})$ . En effet, une exponentielle est obligatoirement (beaucoup !) plus grande qu'une fonction constante (asymptotiquement), mais nous ne sommes absolument pas avancés car la borne supérieure est tellement loin du réel nombre d'opérations.

**Proposition 1.14.** *L'équivalence asymptotique est une relation d'équivalence sur l'ensemble des fonctions de  $\mathbb{N}$  dans  $\mathbb{R}_0^+$ .*

Démonstration. Fixons  $f, g, h : \mathbb{N} \rightarrow \mathbb{R}_0^+$  arbitrairement.

La réflexivité de  $\sim$  est assez claire :

$$\frac{f(n)}{f(n)} = 1 \xrightarrow{n \rightarrow +\infty} 1.$$

La transitivité vient du fait que la limite d'un produit correspond au produit des limites (si ces limites existent), en particulier si  $f \sim g$  et  $g \sim h$  :

$$\frac{f(n)}{h(n)} = \frac{f(n)g(n)}{g(n)h(n)} = \underbrace{\frac{f(n)}{g(n)}}_{\xrightarrow{n \rightarrow +\infty} 1} \underbrace{\frac{g(n)}{h(n)}}_{\xrightarrow{n \rightarrow +\infty} 1} \xrightarrow{n \rightarrow +\infty} 1.$$

La symétrie vient du fait que si une suite  $(x_n)_n$  converge vers une valeur non-nulle  $L$ , alors  $(x_n^{-1})_n$  converge vers  $L^{-1}$ , en particulier si  $f \sim g$  :

$$\lim_{n \rightarrow +\infty} \frac{g(n)}{f(n)} = \frac{1}{\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)}} = \frac{1}{1} = 1.$$

□

**Proposition 1.15.**  *$\Theta$  forme également une relation d'équivalence sur  $\mathbb{R}_0^{+\mathbb{N}}$ .*

Démonstration. La transitivité découle directement de celle du grand O de Landau, la réflexivité est triviale et la symétrie vient de la symétrie inhérente à  $\mathcal{O}$  et  $\Omega$ . □



| **Proposition 1.16.**  $\max\{m, n\} = \Theta(m + n)$ .

Démonstration. Cela s'observe facilement via :

$$\max\{m, n\} \leq m + n = \max\{m, n\} + \min\{m, n\} \leq 2 \max\{m, n\}.$$

□

Nous utilisons donc régulièrement la somme au lieu du max dans un  $\mathcal{O}$  ou un  $\Theta$

### 1.5.1 Quelques ordres de grandeur

**Définition 9.** Si  $P : \mathbb{R} \rightarrow \mathbb{R}$  est une fonction définie par :

$$P(x) = \sum_{k=0}^n a_k x^k$$

pour un certain  $n \in \mathbb{N}$  et  $a_0, \dots, a_n \in \mathbb{R}$ , on dit que  $P$  est un *polynôme* et appelle  $n$  son *degré* (qu'on note également  $\deg P$ ). Pour tout  $0 \leq k \leq \deg P$ , on note  $[x^k]P$  le coefficient de  $x^k$  dans  $P$ , i.e. :

$$P(x) = \sum_{k=0}^{\deg P} [x^k]P x^k.$$

| **Proposition 1.17.** Si  $P$  est un polynôme de degré  $n$ , alors  $P \sim [x^n]P x^n$ .

Démonstration. Notons  $P(x) = \sum_{k=0}^n a_k x^k$ . Calculons :

$$\frac{P(x)}{a_n x^n} = 1 + \sum_{k=1}^n a_{n-k} \frac{1}{x^k} \xrightarrow{x \rightarrow +\infty} 1.$$

□

| **Remarque.** Cette proposition implique également que tout polynôme  $P$  satisfait  $P = \mathcal{O}(x^{\deg P})$  et donc  $\mathcal{O}(x^{\deg P})$  ainsi que  $P = \Omega(x^{\deg P})$ .

| **Proposition 1.18.** Soient  $m, n \in \mathbb{R}^+$  tels que  $m \not\geq n$ . Alors  $x^m = o(x^n)$ .

Démonstration. Trivial :

$$\frac{x^m}{x^n} = \frac{1}{x^{n-m}} \xrightarrow{x \rightarrow +\infty} 0,$$

puisque  $n - m \geq 0$ .

□

| **Proposition 1.19.** Pour tout  $k \geq 0$ ,  $x^k = o(\exp(x))$ .

Démonstration. Montrons cela par récurrence sur  $k$ . Il est clair quand  $k = 0$  que :

$$\frac{x^0}{e^x} = e^{-x} \xrightarrow{x \rightarrow +\infty} 0.$$

Prenons alors  $k > 0$  et supposons que la proposition soit vraie pour tout  $k' \leq k$ . Nous savons que  $x^k$  et  $e^x$  divergent tous les deux vers  $+\infty$  quand  $x \rightarrow +\infty$ , dès lors nous pouvons appliquer le théorème de L'Hospital et déduire :

$$\lim_{x \rightarrow +\infty} \frac{x^k}{e^x} = \lim_{x \rightarrow +\infty} \frac{kx^{k-1}}{e^x} = k \lim_{x \rightarrow +\infty} \frac{x^{k-1}}{e^x} = 0$$

par l'hypothèse de récurrence. □

| **Corollaire 1.20.** Si  $k \in \mathbb{R}_0^+$ ,  $x^k = o(\exp(x))$  également.

Démonstration. Trivial par transitivité de  $o$  et parce que  $x^k = o(x^{\lfloor k+1 \rfloor})$ . □

**Remarque.** La *Proposition 1.19* peut également se démontrer en utilisant le fait que  $\exp$  est une fonction analytique, en particulier elle est égale à sa série de Taylor en  $x = 0$  sur l'entière de son domaine de définition. Dès lors pour tout  $x \in \mathbb{R}$  :

$$\exp(x) = \sum_{k \geq 0} \frac{x^k}{k!}.$$

Par la *Proposition 1.18*, nous savons donc que  $\exp(x) = \omega(x^k)$  pour tout  $k \geq 0$ .

| **Proposition 1.21.** Pour tout  $k \geq 0$  :  $(\log x)^k = o(x)$ .

Démonstration. Procédons par récurrence sur  $k$ . Il est clair que  $(\log x)^0 = 1 = o(x)$ .

Maintenant prenons  $k > 0$  et appliquons à nouveau le théorème de L'Hospital :

$$\lim_{x \rightarrow +\infty} \frac{(\log x)^k}{x} = \lim_{x \rightarrow +\infty} \frac{k(\log x)^{k-1} \frac{1}{x}}{1} = k \lim_{x \rightarrow +\infty} \frac{(\log x)^{k-1}}{x} = 0$$

par hypothèse de récurrence. □

## 1.6 Qualité d'un algorithme

En algorithmique, nous aimons déterminer, pour un maximum de problèmes, l'approche la plus efficace possible. Pour cela, il nous faut un moyen de quantifier ces performances, et c'est pour cela que nous mesurons la *complexité* des approches (et que nous utilisons les notations  $\mathcal{O}$ ,  $\Omega$ , etc.). Nous avons envie de pouvoir mettre un *ordre* sur les algorithmes, et instinctivement il faut que cet ordre dépende de la complexité. En un sens, si deux algorithmes permettent de résoudre le même problème mais que le premier nécessite un nombre *minimum* d'opérations plus grand que le nombre *maximum* de l'autre, alors cet algorithme est moins efficace. De manière plus formelle (pour pouvoir prendre en compte une paramétrisation des problèmes à résoudre), nous pouvons introduire la définition suivante :

**Définition 10.** Soient deux algorithmes  $A_1$  et  $A_2$  résolvant un même problème paramétrisé par un entier  $n \in \mathbb{N}$ . Nous disons que l'algorithme  $A_2$  est *meilleur* que l'algorithme  $A_1$  pour ce problème s'il existe deux fonctions  $f_1, f_2 : \mathbb{N} \rightarrow \mathbb{N}$  telles que :

1.  $A_1$  se résout en  $\Omega(f_1(n))$  opérations ;

2.  $A_2$  se résout en  $\mathcal{O}(f_2(n))$  opérations ;
3.  $f_2 = o(f_1)$ .

Voici la première (d'une longue liste) des divagations mathématiques au sein de ce correctif. Bon amusement !

**Remarque** 🐼. *On va un peu s'emballer ici, mais ours avec moi comme on dit outre-Atlantique.*

*On ne peut uniquement utiliser la notion de grand  $\mathcal{O}$  pour ceci. En effet cette dernière ne forme pas une relation d'ordre sur l'ensemble des fonctions de  $\mathbb{N}$  dans  $\mathbb{R}_0^+$  car elle n'est pas anti-symétrique. En effet si  $f = \mathcal{O}(g)$  et  $g = \mathcal{O}(f)$ , nous n'avons pas nécessairement  $f = g$ , mais nous avons bien  $f = \Theta(g)$ .*

*Considérons l'ensemble  $\mathbb{R}_0^{+\mathbb{N}}$  des fonctions de  $\mathbb{N}$  à valeurs dans  $\mathbb{R}_0^+$ . Nous savons maintenant que  $\Theta$  fournit une relation d'équivalence (notons-la  $\sim_\Theta$ ) sur cet ensemble, nous pouvons donc identifier tous les éléments équivalents et considérer les classes d'équivalence dans  $\mathbb{R}_0^{+\mathbb{N}} / \sim_\Theta$ .*

*Sur cet ensemble quotient, nous pouvons définir la relation d'ordre  $\preccurlyeq$  suivante : pour  $[f], [g] \in \mathbb{R}_0^{+\mathbb{N}} / \sim_\Theta$ , nous définissons  $[f] \preccurlyeq [g]$  lorsque  $f = \mathcal{O}(g)$ . En procédant de la sorte, nous avons construit une réelle relation d'ordre nous permettant de classer l'efficacité asymptotique des algorithmes permettant de résoudre un problème donné, ce qui n'est que simulé par la définition précédente qui utilise le petit  $o$  de Landau pour exprimer un grand  $\mathcal{O}$  mais pas un grand  $\Theta$  (pour arriver à une "relation d'ordre" de la forme  $>$  et pas  $\geq$ ).*

*En effet nous ne pouvions pas utiliser  $\mathcal{O}$  seul comme relation d'ordre à cause de toutes les fonctions qui seraient différentes mais qui seraient tout de même un grand  $\Theta$  de l'une l'autre. En identifiant toutes ces fonctions ensemble (donc en quotientant par la relation d'équivalence associée), nous réglons, de fait, le problème.*

## 1.7 Considérations arithmétiques

**Proposition 1.22.** Notons  $H_n := \sum_{k=1}^n \frac{1}{k}$  le  $n$ ème nombre harmonique.  $H_n \sim \log n$ .

Démonstration. Commençons par remarquer que pour tout  $k > 1$ , nous avons :

$$\int_k^{k+1} \frac{dx}{x} \leq \frac{1}{k} \leq \int_{k-1}^k \frac{dx}{x}$$

puisque sur l'intervalle  $[k, k+1]$ , la fonction  $x \mapsto \frac{1}{x}$  est bornée par au-dessus par  $\frac{1}{k}$  et sur l'intervalle  $[k-1, k]$ , elle est bornée par en-dessous par  $\frac{1}{k}$ . De plus, pour  $k = 1$ , nous pouvons écrire :

$$\int_1^2 \frac{dx}{x} \leq 1 \leq 1$$

En sommant sur  $k$ , nous obtenons :

$$\begin{aligned} \sum_{k=1}^n \int_k^{k+1} \frac{dx}{x} &\leq \sum_{k=1}^n \frac{1}{k} \leq 1 + \sum_{k=2}^n \int_{k-1}^k \frac{dx}{x} \\ \int_1^{n+1} \frac{dx}{x} &\leq H_n \leq 1 + \int_1^n \frac{dx}{x} \end{aligned}$$

$$\begin{aligned} \log(n+1) &\leq H_n \leq 1 + \log n \\ \log n + \log \frac{n+1}{n} &\leq H_n \leq 1 + \log n \\ \log \frac{n+1}{n} &\leq H_n - \log n \leq 1. \end{aligned}$$

En particulier, nous savons que  $H_n - \log n \in [0, 1]$  pour tout  $n$ , ce qui implique donc  $H_n \sim \log n$ .  $\square$

**Remarque (■).** Il est possible de montrer que la suite  $H_n - \log n$  converge (en utilisant le fait qu'elle est strictement décroissante mais bornée par le bas). La limite de cette suite est notée  $\gamma$  ( $\approx 0.577$ ) et appelée la constante d'Euler-Mascheroni. Cette constante est très intéressante car bien qu'elle soit connue dans le paysage mathématique depuis près de 300 ans, nous ne savons toujours pas si  $\gamma \in \mathbb{Q}$ . Nous vous invitons à vous renseigner sur cette fort belle constante !

**Définition 11.** On appelle suite géométrique toute suite  $(x_n)_{n \in \mathbb{N}}$  satisfaisant  $x_n = q^n x_0$  pour un certain  $q \in \mathbb{R}$ . Ce  $q$  est appelé la raison de la suite.

Pour toute suite  $(x_n)_n$ , la somme  $S_n = \sum_{k=0}^n x_k$  est appelée la  $n$ ème somme partielle de la suite  $(x_n)_n$ .

**Proposition 1.23.** Soit  $(x_n)_n$  une suite géométrique de raison  $q \neq 1$ . Pour tout  $n$ , la  $n$ ème somme partielle de  $(x_n)_n$  vaut :

$$S_n = x_0 \frac{1 - q^{n+1}}{1 - q}.$$

Démonstration. Intéressons-nous à la valeur de  $(1 - q)S_n$  :

$$(1 - q)S_n = S_n - qS_n = \sum_{k=0}^n x_0 q^k - q \sum_{k=0}^n x_0 q^k = x_0 \left( \sum_{k=0}^n q^k - \sum_{k=1}^{n+1} q^k \right) = x_0 (q^0 - q^{n+1}).$$

Puisque  $q \neq 1$ , nous savons que  $1 - q \neq 0$ , nous pouvons donc diviser de part et d'autre par cette valeur afin d'obtenir :

$$S_n = \frac{x_0(1 - q^{n+1})}{1 - q}.$$

$\square$

**Proposition 1.24.** Soient  $\alpha \in \mathbb{R}$  et  $(x_n)_n$  une suite réelle satisfaisant  $x_n = \alpha x_{n-1}$  pour tout  $n \geq 1$ . Alors pour tout  $n \geq 0$  :  $x_n = \alpha^n x_0$ .

Démonstration. Procédons par récurrence : il est trivial que  $x_0 = 1 \cdot x_0 = \alpha^0 \cdot x_0$  et si l'égalité tient pour une certaine valeur  $n$ , alors nous pouvons écrire :

$$x_{n+1} = \alpha \cdot x_n = \alpha \alpha^n x_0 = \alpha^{n+1} x_0.$$

$\square$

**Proposition 1.25.** Soient  $\alpha, \beta \in \mathbb{R}$  tels que  $\alpha \geq 1$  et  $(x_n)_n$  une suite réelle satisfaisant  $x_n = \alpha x_{n-1} + \beta$  pour tout  $n \geq 1$ . Alors pour tout  $n \geq 0$  :

$$x_n = \alpha^n x_0 + \beta \frac{1 - \alpha^n}{1 - \alpha} \sim \left( x_0 - \frac{\beta}{1 - \alpha} \right) \alpha^n.$$

Démonstration. À nouveau, procédons par récurrence : il est clair que le terme de droite vaut, pour  $n = 0$  :

$$\alpha^0 x_0 + \beta \frac{1 - \alpha^0}{1 - \alpha} = 1 \cdot x_0 + \beta \frac{0}{1 - \alpha} = x_0.$$

Supposons maintenant que l'égalité soit valable pour un certain  $n$  et calculons :

$$\begin{aligned} x_{n+1} &= \alpha x_n + \beta = \alpha \left( \alpha^n x_0 + \beta \sum_{j=0}^{n-1} \alpha^j \right) + \beta = \alpha^{n+1} x_0 + \alpha \beta \sum_{j=0}^{n-1} \alpha^j + \beta \\ &= \alpha^{n+1} x_0 + \beta \sum_{j=1}^n \alpha^j + \beta = \alpha^{n+1} x_0 + \beta \sum_{j=0}^n \alpha^j = \alpha^{n+1} x_0 + \beta \frac{1 - \alpha^{n+1}}{1 - \alpha}. \end{aligned}$$

□

**Remarque.** On peut également arriver à ce résultat en observant que :

$$\begin{aligned} x_n &= \alpha x_{n-1} + \beta = \alpha(\alpha x_{n-2} + \beta) + \beta = \alpha^2 x_{n-2} + \alpha\beta + \beta = \alpha^2(\alpha x_{n-3} + \beta) + \beta \\ &= \alpha^3 x_{n-3} + \alpha^2\beta + \alpha\beta + \beta = \dots \\ &= \alpha^k x_{n-k} + \beta \sum_{\ell=0}^{k-1} \alpha^\ell = \alpha^k x_{n-k} + \beta \frac{1 - \alpha^k}{1 - \alpha} \end{aligned}$$

pour tout  $k \in \llbracket 0, n \rrbracket$ . En particulier pour  $k = n$  :

$$x_n = \alpha^n x_0 + \beta \frac{1 - \alpha^n}{1 - \alpha}.$$

**Proposition 1.26.**

$$\log n! = \Theta(n \log n).$$

Démonstration. Il est facile de voir que  $n \log n$  est une borne supérieure de  $\log n!$  :

$$\log n! = \sum_{k=1}^n \log k \leq \sum_{k=1}^n \log n = n \log n,$$

donc  $\log n! = \mathcal{O}(n \log n)$ . Afin de voir que c'est également une borne inférieure :

$$\begin{aligned} \log n! &= \sum_{k=1}^n \log k = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} \log k + \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^n \log k \geq \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^n \log k \\ &\geq \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^n \log \frac{n}{2} = (n - \lfloor \frac{n}{2} \rfloor) \log \frac{n}{2} \sim \frac{n}{2} \log n, \end{aligned}$$

donc  $\log n! = \Omega(n \log n)$ .

□

**Remarque.** Nous pouvons faire preuve de plus de précision quant à l'estimation de  $\log n!$  en utilisant la formule de Stirling (que nous démontrerons pas ici).

**Théorème 1.27** (Formule de De Moivre-Stirling).

$$\log n! = n \log n - n + \Theta(\log n),$$

ou plus précisément :

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}.$$

Ce qui implique que  $\log n! \sim n \log n$

**Théorème 1.28** (Approximation d'une somme par une intégrale). Soit  $\varphi : \mathbb{R}^+ \rightarrow \mathbb{R}^+$  une fonction croissante et intégrable. Définissons  $\Phi$  et  $F$  par :

$$\begin{aligned} \Phi : \mathbb{R}_0^+ &\rightarrow \mathbb{R}^+ : x \mapsto \int_0^x \varphi(t) dt \\ F : \mathbb{N} &\rightarrow \mathbb{R}^+ : n \mapsto \sum_{k=0}^n \varphi(k). \end{aligned}$$

Alors pour tout  $n \in \mathbb{N}$ ,  $\varphi$ ,  $\Phi$  et  $F$  vérifient l'inégalité suivante :

$$\varphi(0) + \Phi(n) \leq F(n) \leq \Phi(n+1).$$

En particulier  $F = \Omega(\Phi)$  et  $F = \mathcal{O}(\Phi(\cdot + 1))$

**Remarque.** Il n'est pas nécessaire de supposer  $\Phi(0) = 0$  puisque si  $\Phi_1$  et  $\Phi_2$  sont deux primitives de  $\varphi$ , alors  $\Phi_1 = \Theta(\Phi_2)$  (et même  $\Phi_1 \sim \Phi_2$  si  $\lim_{x \rightarrow +\infty} \varphi(x) > 0$ , i.e. si  $\Phi_1$  et  $\Phi_2$  ne sont pas bornées) et seul l'ordre de grandeur nous intéresse ici.

Démonstration. Fixons un certain  $n \in \mathbb{N}$ . Nous pouvons réécrire :

$$F(n) = \sum_{k=0}^n \varphi(k) = \sum_{k=0}^n \int_k^{k+1} \varphi(k) dx.$$

Par monotonie de  $\varphi$  :

$$F(n) \leq \sum_{k=0}^n \int_k^{k+1} \varphi(x) dx = \int_0^{n+1} \varphi(x) dx = \Phi(n+1).$$

mais également :

$$F(n) = \varphi(0) + \sum_{k=1}^n \int_{k-1}^k \varphi(k) dx \geq \varphi(0) + \int_0^n \varphi(x) dx = \varphi(0) + \Phi(n).$$

Dès lors :

$$\varphi(0) + \Phi(n) \leq F(n) \leq \Phi(n+1).$$

□

**Corollaire 1.29.** *Sous les mêmes hypothèses que ci-dessus, si  $\Phi(\cdot + 1) = \mathcal{O}(\Phi)$ , alors  $F = \Theta(\Phi)$ . De plus si  $\frac{\Phi(n+1)}{\Phi(n)} \xrightarrow{n \rightarrow +\infty} 1$ , alors  $F \sim \Phi$ .*

*Démonstration.* Il est clair par transitivité que  $F = \mathcal{O}(\Phi)$  et donc que  $F = \Theta(\Phi)$ . De même si  $\frac{\Phi(n+1)}{\Phi(n)}$  converge vers 1, alors par positivité de  $\Phi$  (qui vient de la positivité de  $\varphi$ ) :

$$1 \leq \frac{\varphi(0) + \Phi(n)}{\Phi(n)} = 1 + o(1) \leq \frac{F(n)}{\Phi(n)} \leq \frac{\Phi(n+1)}{\Phi(n)} = 1 + o(1),$$

et donc  $F \sim \Phi$ . □

**Théorème 1.30.** *Sous les mêmes hypothèses que le théorème précédent, si  $\varphi$  est décroissante au lieu de croissante, alors :*

$$\Phi(n+1) \leq F(n) \leq \varphi(0) + \Phi(n),$$

*i.e.  $F = \Omega(\Phi(\cdot + 1))$  et  $F = \mathcal{O}(\Phi)$ . Si de plus  $\Phi = \mathcal{O}(\Phi(\cdot + 1))$  alors  $F = \Theta(\Phi)$  et si  $\Phi \sim \Phi(\cdot + 1)$ , alors  $F \sim \Phi$ .*

*Démonstration.* La démonstration de ce théorème est similaire à celle du théorème précédent. □

**Remarque.** *C'est en réalité ce résultat-ci qui a été adapté pour montrer la **Proposition 1.22**. Nous pouvons maintenant tout simplement dire que pour  $\varphi(x) = \frac{1}{x+1}$ , nous avons  $H_n = F(n-1)$  et  $\Phi(n) = \log(n+1)$ . Comme de plus :*

$$\log(n+1) \leq \log(2n) = \log n + \log 2,$$

*on sait que  $H_n = F(n-1) \sim \Phi(n-1) = \log n$ .*

*Ce théorème permet également de montrer la **Proposition 1.26** : en prenant  $\varphi(x) = \log(x+1)$ , nous avons pour  $n \geq 1$  :*

$$\Phi(n-1) = \int_0^{n-1} \log(x+1) dx = \int_1^n \log x dx = [x(\log x - 1)]_1^n = n(\log n - 1),$$

*ainsi que :*

$$\begin{aligned} \frac{\Phi(n)}{\Phi(n-1)} &= \frac{n \log(n+1) - n + \log(n+1) - 1}{n(\log n - 1)} = 1 + \frac{n \log\left(1 + \frac{1}{n}\right) + \log(n+1) - 1}{n(\log n - 1)} \\ &\leq 1 + \frac{n \log 2 + n - 1}{n(\log n - 1)} = 1 + \frac{1 + \log 2 - \frac{1}{n}}{\log n - 1} \xrightarrow{n \rightarrow +\infty} 1. \end{aligned}$$

*Dès lors nous savons que  $\log(n!) = F(n-1) \sim \Phi(n-1) = n(\log n - 1) \sim n \log n$ .*

**Corollaire 1.31.** *Si  $\varphi$  est asymptotiquement croissante (ou décroissante), i.e. s'il existe un  $N > 0$  tel que  $\varphi$  est croissante (ou décroissante) sur  $[N, +\infty)$ , alors les conclusions du **Théorème 1.28**, du **Théorème 1.30** et de leurs corollaires sont toujours vérifiées.*

*Démonstration.* Regardons ici uniquement le cas où  $\varphi$  est asymptotiquement croissante. Considérons une valeur  $N > 0$  à partir de laquelle  $\varphi$  est croissante. Notons  $\bar{\varphi}(k) := \varphi(n + k)$

$N$ ) la fonction  $\varphi$  décalée (qui est donc croissante sur  $\mathbb{R}^+$ ),  $\bar{F}(n)$  la somme des  $\bar{\varphi}(0)$  jusque  $\bar{\varphi}(n)$  et  $\bar{\Phi}$  l'intégrale de  $\bar{\varphi}$ . Par le **Théorème 1.28**, nous avons :

$$\bar{\varphi}(0) + \bar{\Phi}(k) \leq \bar{F}(k) \leq \bar{\Phi}(k+1)$$

pour tout  $k$ . En particulier pour tout  $n > 0$ , nous savons que :

$$F(n) = \sum_{k=0}^n \varphi(k) = \sum_{k=0}^{N-1} \varphi(k) + \sum_{k=N}^n \varphi(k) = F(N-1) + \sum_{k=0}^{n-N} \bar{\varphi}(k) = F(N-1) + \bar{F}(n-N),$$

ainsi que :

$$\Phi(n) = \int_0^n \varphi(x) dx = \int_0^N \varphi(x) dx + \int_N^n \varphi(x) dx = \Phi(N) + \int_0^{n-N} \bar{\varphi}(x+N) dx,$$

i.e. :

$$\Phi(n) = \Phi(N) = \bar{\Phi}(n-N).$$

Dès lors à  $n > N$  fixé :

$$F(n) = F(N-1) + \bar{F}(n-N) \leq F(N-1) + \bar{\Phi}(n-N+1) = F(N) + \Phi(n+1) - \Phi(N),$$

et :

$$F(n) = F(N-1) + \bar{F}(n-N) \geq F(N-1) + \bar{\varphi}(0) + \bar{\Phi}(n-N) = \varphi(N) + F(N-1) + \Phi(n) - \Phi(N).$$

Notons alors  $\delta(N) := F(N-1) - \Phi(N)$  (l'erreur d'approximation sur la partie non-monotone). Nous pouvons écrire pour tout  $n > N$  :

$$\varphi(N) + \delta(N) + \Phi(n) \leq F(n) \leq \delta(N) + \Phi(n+1),$$

i.e. nous conservons la même inégalité que précédemment mais décalée verticalement de la constante  $\delta(N)$ . Les arguments concernant les bornes restent donc valides et les équivalences asymptotiques sont toujours respectées.  $\square$

**Lemme 1.32.** Si  $\varphi : \mathbb{R} \rightarrow \mathbb{R}$  est une fonction telle que  $\varphi(x) \xrightarrow{x \rightarrow +\infty} +\infty$  (resp.  $-\infty$ ), alors elle est asymptotiquement positive (respectivement négative).

Démonstration. Par définition d'une limite divergente : pour tout  $M > 0$ , il existe un  $N > 0$  tel que pour tout  $x > N$  :  $\varphi(x) > M$  (resp.  $\varphi(x) < -M$ ).  $\square$

**Corollaire 1.33.** Tout polynôme est asymptotiquement monotone.

Démonstration. Prenons  $P$  un polynôme de degré  $n$  et de coefficients  $a_0, \dots, a_n$ . Si  $n = 0$ , alors  $P$  est constant et donc monotone sur l'entièreté de son domaine. Si par contre  $n > 0$ , nous savons que  $P$  est  $\mathcal{C}^\infty$ , donc en particulier dérivable. Notons  $Q := \frac{d}{dx}P$  sa dérivée et observons :

$$Q(x) = \sum_{k=1}^n a_k \frac{dx^k}{dx} \Big|_x = \sum_{k=0}^{n-1} (k+1)a_{k+1}x^k.$$

Dès lors :

$$\lim_{x \rightarrow +\infty} Q(x) = \lim_{x \rightarrow +\infty} n a_n x^{n-1} = \lim_{x \rightarrow +\infty} a_n x^{n-1} = \text{sign}(a_n) \infty.$$



En particulier si  $a_n > 0$ , alors  $Q$  diverge vers  $+\infty$  et est donc asymptotiquement positive, ce qui veut dire que  $P$  est asymptotiquement croissante. Si  $a_n < 0$ , alors  $Q$  diverge vers  $-\infty$  et est donc asymptotiquement négative, ce qui veut dire que  $P$  est asymptotiquement décroissante. Comme  $a_n$  ne peut pas être égal à 0 (sinon  $P$  ne serait pas de degré  $n$ ), nous savons que  $P$  est asymptotiquement monotone.  $\square$

| **Lemme 1.34.** Si  $P$  est un polynôme, alors  $P(\cdot + 1) \sim P$ .

*Démonstration.* Soit  $P$  un polynôme de degré  $n$  et de coefficients  $a_0, \dots, a_n$ . Si  $n = 0$ , alors  $P$  est constant et donc  $\frac{P(\cdot+1)}{P}$  est la fonction  $\mathbb{R} \rightarrow \mathbb{R} : x \mapsto 1$ .

Si  $n > 0$ , nous savons que par la **Proposition 1.17** que  $P(x) \sim a_n x^n$  (et donc  $P(x+1) \sim a_n(x+1)^n$ ). Dès lors :

$$P(x+1) \sim a_n(x+1)^n = a_n \sum_{k=0}^n \binom{n}{k} x^k \sim a_n \binom{n}{n} x^n = a_n x^n \sim P(x).$$

$\square$

| **Corollaire 1.35.** Si  $P$  est un polynôme unitaire de degré  $n$ , alors :

$$\sum_{k=0}^m P(k) \sim \frac{1}{n+1} m^{n+1},$$

où l'équivalence asymptotique est prise pour  $m \rightarrow \infty$ , i.e. :

$$\lim_{m \rightarrow +\infty} \frac{(n+1)}{m} \sum_{k=0}^m \left(\frac{k}{m}\right)^n = 1.$$

*Démonstration.* Nous savons que  $P$  est une fonction asymptotiquement croissante par le **Corollaire 1.33**. De plus par le **Lemme 1.34**, nous savons que  $P(\cdot + 1) \sim P$ . Nous pouvons donc appliquer le **Corollaire 1.31** afin d'avoir l'équivalence suivante (par le **Corollaire 1.29** du **Théorème 1.28**) :

$$\sum_{k=0}^m P(k) \sim \int_0^m P(x) dx.$$

Or puisque  $P$  est unitaire, par la **Proposition 1.17**, nous savons que  $P(x) \sim x^n$ . Dès lors si  $Q(x)$  est un polynôme tel que :

$$Q(x) = \int_0^x P(t) dt,$$

alors  $Q$  est de degré  $n+1$  et  $[x^{n+1}]Q = \frac{1}{n+1}$ . Finalement nous en déduisons :

$$\sum_{k=0}^m P(k) \sim Q(m) \sim \frac{1}{n+1} m^{n+1}.$$

$\square$

**Proposition 1.36.** Soient  $f$  et  $g$  deux fonctions intégrables. Si  $g$  est positive et croissante et si  $f \sim g$ , alors  $F \sim G$  où  $F$  et  $G$  sont définies par :

$$F : \mathbb{R} \rightarrow \mathbb{R} : x \mapsto \int_0^x f(t) dt,$$

$$G : \mathbb{R} \rightarrow \mathbb{R}^+ : x \mapsto \int_0^x g(t) dt.$$

*Démonstration.* Définissons  $h := f - g$ . Par hypothèse nous savons que  $h = o(g)$ . Si  $H(x)$  est l'intégrale de  $h$  sur  $[0, x]$ , montrons que  $H = o(G)$ . Fixons  $\varepsilon > 0$ . Puisque  $h = o(g)$ , nous savons qu'il existe un certain  $N_0 > 0$  tel que si  $x > N_0$ , alors  $h(x) < \frac{\varepsilon}{2}g(x)$ . Pour un tel  $x > N_0$  :

$$H(x) = H(N_0) + \int_{N_0}^x h(t) dt < H(N_0) + \frac{\varepsilon}{2} \int_{N_0}^x g(t) dt = H(N_0) - \frac{\varepsilon}{2}G(N_0) + \frac{\varepsilon}{2}G(x).$$

De plus puisque  $g$  est croissante et positive, nous savons que  $G$  diverge vers  $+\infty$ . En particulier pour toute valeur  $y > 0$ , il existe une valeur  $M > 0$  telle que  $G(x) > y$  pour tout  $x > M$ . Prenons donc  $N_1 > N_0 > 0$  tel que  $G(x) > \frac{2}{\varepsilon}H(N_0) - G(N_0)$  pour tout  $x > N_1$ .

Nous avons donc pour  $x > N_1$  :

$$H(x) < \frac{\varepsilon}{2} \left( \frac{2}{\varepsilon}H(N_0) - G(N_0) + G(x) \right) < \frac{\varepsilon}{2} (G(x) + G(x)) = \varepsilon G(x),$$

i.e.  $H = o(G)$ . Nous en déduisons finalement que  $F \sim G$  puisque :

$$F(x) - G(x) = \int_0^x f(t) dt - \int_0^x g(t) dt = \int_0^x h(t) dt = H(x) = o(G(x)).$$

□

**Corollaire 1.37.** Soient  $\varphi_1, \varphi_2 : \mathbb{R}^+ \rightarrow \mathbb{R}^+$  deux fonctions croissantes et intégrables telles que  $\varphi_1 \sim \varphi_2$ . Sous les notations du **Théorème 1.28**, si  $\Phi_1(n+1) \sim \Phi_1(n)$ , alors :

$$F_1(n) \sim \Phi_2(n).$$

*Démonstration.* Les hypothèses du **Théorème 1.28** sont vérifiées pour  $\varphi_1$ , dès lors  $F_1 \sim \Phi_1$ . De plus par la **Proposition 1.36**, nous savons que  $\Phi_1 \sim \Phi_2$ . Par transitivité nous déduisons  $F_1 \sim \Phi_2$ . □

## Chapitre 2

# Rappels théoriques

### 2.1 Les ADTs

Pour plus d'informations sur l'implémentation des ADTs en Python, voir la [Section 1.3](#).

#### 2.1.1 Listes chaînées

Une *liste* (ou plus couramment *liste chaînée*) est une structure de donnée qui permet de représenter une séquence ordonnée d'éléments, dans le sens où chaque élément peut être associé à un *indice*. Elle se compose d'une suite de *nœuds* liés les uns aux autres. Les différents types de listes chaînées se distinguent dans la manière de gérer ce chaînage et exhibent des propriétés différentes.

**Attention :** le terme *ordonné* (et donc *non-ordonné*) est utilisé à la fois pour cette notion d'*indiquabilité* et pour une notion de relation d'ordre. Afin de ne pas confondre les deux et donc de ne pas être ambigu, nous parlerons de structure *ordonnée* dans le premier cas et de structure *triée* dans le second cas. Le livre de référence de Miller et Ranum utilisé au cours théorique parle donc de la classe `UnorderedList` pour parler d'une liste chaînée non-nécessairement triée, dans ce syllabus d'exercices, nous appellerons cette structure `List` ou `LinkedList`.

La classe `Node` peut s'implémenter comme suit :

---

```
1 class Node:
2     def __init__(self, data, next_node=None):
3         self.data = data
4         self.next_node = next_node
```

---

Et la classe `LinkedList` peut s'écrire comme suit :

---

```
1 class LinkedList:
2     def __init__(self):
3         self.head = None
4     def push_front(self, data):
5         self.head = Node(data, self.head)
6     def add_next(self, node: Node, new_node: Node):
7         previous_next = node.next_node
```

---

```

8     node.next_node = new_node
9     new_node.next_node = previous_next
10    def pop_front(self) -> Node:
11        if self.is_empty(): raise ValueError('Pop from empty list')
12        ret = self.head
13        self.head = ret.next_node
14        ret.next_node = None
15        return ret
16    def is_empty(self) -> bool:
17        return self.head is None

```

---

Toutes les méthodes ici s'exécutent en  $\mathcal{O}(1)$  puisque la taille de la liste chaînée n'intervient à aucun moment. Cependant il n'est pas possible ici de retirer efficacement un nœud et le mieux que l'on puisse faire est la méthode suivante :

```

1    def remove(self, node: Node):
2        current = self.head
3        previous = None
4        while current is not None and current is not Node:
5            previous = current
6            current = current.next_node
7        if current is None:
8            raise ValueError('Removing a node not in the list')
9        previous.next_node = current.next_node
10       current.next_node = None

```

---

Mais nous voyons bien que dans le pire des cas, nous devons parcourir l'entièreté de la liste avant de trouver le nœud en question. Afin de compenser cela, nous pouvons avoir une liste doublement chaînée dans laquelle en plus d'être lié au nœud suivant, chaque nœud est également lié au nœud précédent. La classe Node devient donc :

```

1    class Node:
2        def __init__(self, data, prev_node: Node=None, next_node: Node=None):
3            self.data = data
4            self.prev_node = prev_node
5            self.next_node = next_node

```

---

La classe LinkedList devient donc DoubleLinkedList (ou parfois DoubleEndedLinkedList) :

```

1    class DoubleLinkedList:
2        def __init__(self):
3            self.head = None
4
5        def push_front(self, data):
6            self.head = Node(data, None, self.head)
7
8        def add_next(self, node: Node, new_node: Node):
9            new_node.next_node = node.next_node
10           new_node.next_node.prev_node = new_node

```

```

11     node.next_node = new_node
12     new_node.prev_node = node
13
14     def pop_front(self) -> Node:
15         if self.is_empty(): raise ValueError('Pop from an empty list')
16         ret = self.head
17         self.head = ret.next_node
18         if self.head is not None:
19             self.head.prev_node = None
20         ret.next_node = None
21         return ret
22
23     def is_empty(self) -> bool:
24         return self.head is None
25
26     def remove(self, node: Node):
27         prev_node = node.prev_node
28         next_node = node.next_node
29         if next_node is not None:
30             next_node.prev_node = prev_node
31         if prev_node is not None:
32             prev_node.next_node = next_node

```

---

Ici, les insertions sont bien toujours en  $\mathcal{O}(1)$ , mais la suppression se fait également en  $\mathcal{O}(1)$ . En effet, il n'est plus nécessaire de trouver quel était le nœud précédent en parcourant la liste puisque le nœud précédent est connu de chaque nœud. Attention cependant, nous avons donc ici retiré la vérification que le nœud appartient bien à la liste de laquelle on cherche à le retirer. Notez que ceci peut aisément être contourné si on accepte de payer un espace  $\mathcal{O}(1)$  supplémentaire par nœud : en plus d'une référence vers le nœud suivant et d'une référence vers le nœud précédent, nous pouvons ajouter une référence vers la liste chaînée qui contient le nœud.<sup>1</sup>

Une liste doublement chaînée peut également être *circulaire*, c'est-à-dire que la liste ne contient plus réellement de début et de fin et donc que l'élément qui précède la tête est le dernier élément de la liste. De plus, afin de ne pas s'embêter avec des vérifications `if self.head is not None` comme dans les codes ci-dessus, il est également possible d'utiliser un *élément bidon*

### 2.1.2 Piles (*stacks*)

Un *stack* (ou *pile* en français, mais on utilisera le mot *stack* dans ce document) est un conteneur supportant les opérations suivantes en temps constant (i.e. ne dépendant pas de la taille du conteneur) :

1. insérer un élément ;
2. consulter l'élément le plus récent (i.e. le dernier inséré) ;
3. retirer l'élément le plus récent ;

---

1. À nouveau, ça peut être embêtant si on manipule des listes chaînées qui peuvent se partager des nœuds, mais dans ce cas-là, la vérification est non-triviale, quoi qu'on fasse.

4. récupérer la taille ;
5. vérifier s'il est vide.

Un autre nom pour ce type de conteneur est LIFO (acronyme de *Last In, First Out*). Un stack peut être facilement implémenté en utilisant une liste chaînée où le TOS (*Top Of Stack*) est la tête de la liste. Puisque l'insertion et la suppression en tête se fait en temps constant dans une liste chaînée, nous avons la garantie que les trois premières opérations s'exécutent bien en temps  $\mathcal{O}(1)$ .

### 2.1.3 Files (*queues*)

Une *queue* (ou *file*, voire *file d'attente* en français, mais on utilisera le mot *queue* dans ce document) est un conteneur supportant les opérations suivantes en temps constant :

1. insérer un élément ;
2. consulter l'élément le plus ancien ;
3. retirer l'élément le plus ancien ;
4. récupérer sa taille ;
5. vérifier s'il est vide.

Si un stack s'appelle également un LIFO, par symétrie, une queue s'appelle également un FIFO (First In, First Out). De plus, à l'instar du stack, la queue peut être implémentée via une liste chaînée, mais qui nécessite deux points d'accès : le *début* et la *fin*. Cela peut se faire soit en maintenant une deuxième référence *tail* (en opposition à *head*), soit en utilisant une liste doublement chaînée dans laquelle les insertions et les suppressions/consultations se font de part et d'autre de la tête.

### 2.1.4 Files à double extrémité (*deques*)

Un (ou une) *deque* (*Double Ended Queue* en anglais) est un ADT qui généralise simultanément les stacks et les queues. En effet un tel conteneur supporte les opérations suivantes (toujours en temps constant) :

1. insérer un élément au *début* ;
2. insérer un élément à la *fin* ;
3. consulter l'élément au *début* ;
4. consulter l'élément à la *fin* ;
5. supprimer l'élément au *début* ;
6. supprimer l'élément à la *fin* ;
7. récupérer sa taille ;
8. vérifier s'il est vide.

Il est commun d'utiliser un deque comme implémentation des stacks et des queues dans la lib standard des langages les supportant (e.g. C++), ou de simplement proposer un type Deque et de se passer des types Stack et Queue (e.g. Python).

### 2.1.5 ADTs et complexité

En fonction des auteurs ou autrices, les ADTs sont parfois définis uniquement par les opérations supportées, mais indépendamment de la complexité de ces opérations. Si nous relaxons la contrainte de temps constant sur les opérations pour les trois ADTs présentés ci-dessus, nous remarquons qu'ils peuvent également être implémentés par des vecteurs (donc des tableaux de taille variable), mais dans le pire des cas, cela force les opérations à s'exécuter en temps linéaire en la taille du conteneur, ce qui est clairement suboptimal.

#### Optimisation des dequeues (🐢)

Il est commun de ne pas utiliser de *simples* listes chaînées pour implémenter des dequeues mais d'utiliser des listes chaînées dans lesquelles chaque nœud contient un buffer (de taille fixe !), ce qui permet de ne pas devoir réallouer de la mémoire à chaque insertion. Comme la taille des buffers est fixée à l'avance, cela ne change pas le comportement asymptotique (donc en terme de  $\mathcal{O}$ ) du coût des opérations, mais permet en pratique de les rendre plus rapide d'un facteur constant.

## 2.2 Récursivité

Du point de vue mathématique, un concept est *récuratif* s'il est défini par rapport à une version "plus simple" de lui-même. Par exemple souvenez vous de la *factorielle* d'un nombre qui peut être définie de manière récursive ou non. En effet nous pouvons définir :

$$n! := \prod_{k=1}^n k = 1 \cdot 2 \dots (n-1) \cdot n,$$

mais nous pouvons également définir :

$$n! := \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n \geq 1. \end{cases}$$

Il est assez clair que ces deux définitions sont équivalentes, mais conceptuellement différentes.

En informatique, nous parlerons de *fonction récursive* pour toute fonction qui, pendant son exécution, peut s'appeler elle-même, potentiellement avec des paramètres différents. Le lien avec le sens mathématique est assez clair puisque nous pouvons calculer des objets définis récursivement via des fonctions récursives. En effet la fonction suivante calcule la factorielle d'un nombre  $n$  donné en paramètre :

---

```

1 def factorial(n):
2     if n == 0: return 1
3     else: return n * factorial(n-1)

```

---

Notons que, tant d'un point de vue mathématique que d'un point de vue informatique, une définition récursive contient toujours deux éléments fondamentaux :

1. un cas de base ;
2. et un cas récursif.

En effet, puisque le principe de la récursivité est de définir un objet par une version “plus simple” de lui-même, il faut bien une version qui ne peut pas être simplifiée sinon la définition ne s’arrête jamais (et manipuler des objets mal définis a tendance à être rapidement problématique en mathématique. . .) Il faut donc un cas de base pour éviter ce problème. Notons que le cas de base est potentiellement un *misnomer* et peut induire en erreur puisqu’il n’est pas nécessairement unique. En effet prenons l’exemple de la suite de Fibonacci qui est définie comme ceci :

$$F_n := \begin{cases} n & \text{si } 0 \leq n \leq 1 \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2. \end{cases}$$

Nous voyons que nous avons besoin de *deux* définitions dans notre cas de base pour garantir que la définition de  $F_2$  est bien définie puisque nous avons besoin des éléments  $F_0$  et  $F_1$ .

Notons, pour faire le lien avec la [sous-section 1.2.3](#), que la notion de récursivité est intrinsèquement liée à la notion d’induction, et par extension les preuves de complexité de fonctions récursives font presque systématiquement appel à des relations de récurrence et/ou des preuves par induction.

### 2.2.1 Récursivité croisée

Faisons bien attention à l’existence de la *récursivité croisée*, i.e. une situation dans laquelle il existe deux fonctions  $f$  et  $g$  telles que  $f$  appelle  $g$  et  $g$  appelle  $f$  (bon en réalité pour rester formel, il faudrait plutôt dire qu’on a des fonctions  $f_1$  jusque  $f_k$  pour une constante  $k$  et où chaque  $f_i$  appelle un certain  $f_j$ , avec potentiellement  $i = j$ , mais on a l’idée).

### 2.2.2 Contexte des fonctions

Lorsqu’une fonction est exécutée, elle manipule probablement des variables locales. Il est impératif que, lors de la création d’un langage supportant la récursivité<sup>2</sup>, ces variables locales soient *réellement* locales et non partagées par toutes les exécutions de la fonction. On appelle *contexte* l’ensemble des variables locales d’une telle exécution de la fonction. Le comportement du contexte en ASM, et par extension en C(++), sera vu au cours INFO-F105 – Langages de Programmation. De manière générale, l’élément essentiel est la gestion d’un stack des appels, qui peut même être écrit explicitement dans le code si on veut dérécurifier une fonction (c.f. la [section 2.9](#)).

### 2.2.3 Diviser pour régner

L’approche récursive pour résoudre un problème consiste en le fait de résoudre dans un premier temps une version plus simple du problème afin de résoudre le problème entier plus tard. Nous pouvons cependant également choisir de subdiviser le problème initial en plus d’un seul sous-problème, de résoudre tous ces sous-problèmes dans un premier temps, et d’ensuite utiliser ces sous-solutions pour construire la solution générale. Nous parlons de *diviser pour régner* (ou *Divide and Conquer* en anglais, parfois abrégé *DC*, ou

2. (🐣) Ce qui n’était pas nécessairement le cas pour les premiers langages de programmation. En effet ALGOL60 ne spécifiait pas explicitement le potentiel récursif des *fonctions* et les compilateurs pouvaient donc tout à fait assigner des zones mémoires fixes pour toutes les variables locales, ce qui est bien entendu terrible. Ce fut un sujet très actif dans les années 60, et Donald Knuth a même proposé un petit bout de code en ALGOL60 qui permettait de savoir si le compilateur utilisé supportait la récursivité ou non.



*D & C*) lorsque les sous-problèmes ont tous (approximativement) la même taille. Bien souvent (tout du moins dans les algorithmes relativement simples que nous verrons dans ce cours), le nombre de tels sous-problèmes est fixé à l'avance et ne dépend pas de la taille de l'entrée. Notons qu'il n'est parfois pas nécessaire de résoudre *tous* les sous-problèmes pour résoudre le problème initial (e.g. la recherche dans un BST, c.f. la [section 2.5](#)).

## 2.3 Backtracking

Le *backtracking* (*to backtrack* se traduit littéralement par *revenir en arrière*) est une technique récursive de résolution de problèmes pour tenter d'éviter de générer l'entièreté des solutions potentielles au problème posé. En effet le problème avec une *recherche exhaustive* est le suivant : habituellement l'ensemble des objets parcourus a tendance à croître de manière au moins exponentielle en la taille du problème posé. En effet si nous cherchons par exemple s'il existe un vecteur  $v$  binaire de taille  $n$  ayant une certaine propriété  $P(v)$  et que nous devons pour ce faire regarder individuellement chaque tel vecteur et tester s'il satisfait la propriété  $P$ , nous remarquons aisément que nous allons devoir générer tous les vecteurs binaires de taille  $n$ . Or combien de tels vecteurs existe-t-il ? Nous savons que pour chaque indice  $0 \leq i \leq n - 1$ , nous avons  $v_i = 0$  ou  $v_i = 1$ , i.e. nous avons deux possibilités pour chaque position, ce qui veut dire qu'il y a  $2^n$  vecteurs binaires de taille  $n$ . Même en supposant que tester la propriété  $P$  se fait en temps constant (i.e. ne dépendant pas de  $n$  :  $\mathcal{O}(1)$ ), si aucun vecteur ne satisfait la propriété, nous aurons tout de même  $\Theta(2^n)$  opérations à exécuter, ce qui est malheureusement infaisable pour  $n$  pas très grand (par exemple  $n = 100$ ).

Nous voulons alors une manière d'éviter, si possible, de regarder des pans entiers de l'espace des objets à analyser. C'est ici que le backtracking intervient. Attention : de manière générale, dans le pire des cas, un backtracking restera exponentiel dans le pire des cas, mais sous certaines conditions et avec une bonne analyse *a priori* du problème étudié, nous pouvons garantir nombre d'opérations subexponentiel (par exemple en choisissant bien dans quel ordre nous générons les potentielles solutions). Notons que le backtracking peut également servir d'implémentation relativement simple à un algorithme exhaustif, et donc dans un tel contexte, l'intérêt du backtracking n'est pas *l'élagage* (donc la possibilité d'ignorer les morceaux de l'espace ne pouvant pas mener à une solution) mais bien une procédure de génération exhaustive.

L'idée fondamentale du backtracking est la suivante : nous voulons regarder les solutions possibles de taille  $n$ , et pour cela, nous allons regarder toutes les *solutions partielles* de taille  $n' < n$  (habituellement  $n' = n - 1$ ), et nous allons *augmenter* chacune de ces solutions pour peu qu'elles puissent mener à une solution complète. Mais évidemment nous allons devoir augmenter chaque solution partielle de plusieurs manières (par la nature exponentielle de l'espace de recherche), et nous allons faire chacune de ces augmentations itérativement. C'est de là que le backtracking tire son nom : en supposant que nous avons une solution partielle  $S'$ , nous allons tenter de construire  $S_1$  sur base de  $S'$  (une première augmentation), explorer le plus loin possible, et puis revenir en arrière à  $S'$  et construire  $S_2$  (une deuxième augmentation possible), explorer le plus loin possible, etc.

Le canevas général d'une approche par backtracking est le suivant :

---

```

1 def solve(subsize):
2     if done():
3         if found():
4             update_solution()
5     else:
6         for augmentation in possible_augmentations():
7             augment_current_solution(augmentation)
8             solve(subsize+1)
9             remove_augmentation(augmentation)

```

---

Notons que si `possible_augmentations` renvoie toutes les possibilités d'augmentation de manière générale (i.e. sans se poser la question de la possibilité de trouver une solution optimale dans cette branche), alors cette méthode fera une génération exhaustive des résultats. Par exemple si nous utilisons ce canevas pour générer tous les vecteurs binaires, nous obtenons quelque chose comme suit :

---

```

1 def binary_vectors(k, v):
2     if k == len(v): # if done and found
3         print(v)
4         return
5     v[k] = 0
6     binary_vectors(k+1, v)
7     v[k] = 1
8     binary_vectors(k+1, v)

```

---

Cette fonction peut ensuite être appelée comme ceci : `binary_vectors(0, [None]*4)`.

### 2.3.1 Élagage (*pruning*)

Lors de l'étape de recherche des augmentations possibles pour la solution partielle actuelle, il est parfois possible de se rendre compte que la sous-solution active ne permettra pas de générer de solutions recherchées. Par exemple si au lieu de chercher *tous* les vecteurs binaires, nous cherchons *uniquement* les vecteurs binaires qui ne contiennent pas de 1 isolé, alors nous pouvons adapter le code ci-dessus comme suit :

---

```

1 def binary_vectors(k, v):
2     if k == len(v): # if done
3         found = True
4         if len(v) >= 2 and v[0] == 1 and v[1] == 0:
5             found = False
6         i = 2
7         while i < len(v) and found:
8             if v[i] == 0 and v[i-1] == 1 and v[i-2] == 0:
9                 found = False
10                i += 1
11            if found:
12                print(v)
13            return
14     v[k] = 0

```

---

```

15     binary_vectors(k+1, v)
16     v[k] = 1
17     binary_vectors(k+1, v)

```

---

Cependant ici, même si seulement une partie des vecteurs binaires satisfait notre propriété (le nombre de tels vecteurs est  $< 1.754^n$ , nous vous épargnons le raisonnement), nous devons tout de même générer tous les vecteurs binaires de taille  $n$ . Nous devons donc générer  $2^n$  vecteurs, mais nous savons que seuls moins de  $1.754^n$  tels vecteurs satisferont notre propriété, i.e. nous générerons de l'ordre de  $2^n - (1.754/2)^n \sim 2^n$  vecteurs en trop, ce qui est fondamentalement déraisonnable.

Nous pouvons donc faire mieux et décider de ne pas continuer le backtracking dès qu'on insère le pattern  $[0, 1, 0]$  dans le vecteur afin de ne générer que les vecteurs demandés. Il y en a toujours un nombre exponentiel, mais de base plus petite :

```

1 def binary_vectors(k, v):
2     if k == len(v):
3         print(v)
4         return
5     if (k != 1 or v[0] != 1) and (k < 2 or v[k-2] != 0 or v[k-1] != 1):
6         v[k] = 0
7         binary_vectors(k+1, v)
8     v[k] = 1
9     binary_vectors(k+1, v)

```

---

### 2.3.2 Recherche de solution optimale et *Branch and Bound*

Le backtracking apparaît régulièrement dans des problèmes d'optimisation, i.e. des problèmes dans lesquels nous pouvons ordonner les solutions (dans le sens où nous pouvons dire qu'une solution est meilleure qu'une autre). En effet, dans cette famille de problème, si nous n'avons pas d'algorithme efficace connu, nous devons nous résoudre à faire une recherche exhaustive. Cependant, le principe de *pruning* vu ci-dessus peut nous aider à ne pas devoir explorer l'entièreté de l'espace des solutions possibles. En effet, puisque nous construisons notre solution au fur et à mesure, nous pouvons bien souvent détecter à un certain moment si la branche explorée peut mener à une solution meilleure que la solution trouvée jusqu'à présent.

Notons que puisque nous parlons ici de solution actuelle (la solution partielle construite), de solution optimale, nous allons devoir gérer plusieurs variables simultanément et il est donc commun d'utiliser une classe pour stocker le tout. La résolution du problème se fera donc dans une méthode de la classe. Le canevas est donc le suivant :

```

1 class Solver:
2     def __init__(self, params):
3         self.params = params
4         self.init_current_sol()
5         self.init_best_sol()
6
7     def solve(self):
8         self._solve()

```

```

9         return self.best_solution
10
11     def _solve(self, step=0):
12         if done:
13             if self.current_solution >= self.best_solution:
14                 self.update_best_solution()
15         else:
16             for choice in self.possible_choices():
17                 if self.can_find_better_solution(choice):
18                     self.augment_solution(choice)
19                     self._solve(step+1)
20                     self.restore_previous_solution()

```

---

Typiquement, si nous cherchons une solution de coût minimum, lors de l'exécution de la méthode `_solve`, si la solution partielle actuelle `self.current_solution` est associée à un coût qui est supérieur à celui de `self.best_solution` et que nous savons que le coût de la solution ne peut qu'augmenter, il est inutile de continuer d'explorer les solutions basées sur `current_solution` puisque nous savons qu'aucune de ces solutions ne pourrait satisfaire la condition de la ligne 13. C'est le rôle de la méthode `can_find_better_solution` appelée à la ligne 17 de déterminer si ce critère est vérifié ou non.

Cette approche est appelée *Branch and Bound* puisque dans l'arbre des appels récursifs, nous utilisons des bornes sur la solution optimale pour déterminer quelles branches seront visitées et quelles branches seront ignorées (ou élaguées).

## 2.4 Arbres

Un *arbre* est une structure de donnée généralisant la notion de liste chaînée dans laquelle chaque nœud peut avoir un nombre arbitraire de successeurs (potentiellement 0). Notons que les nœuds sont ici parfois également appelés sommets.<sup>3</sup>

Dans les arbres que nous considérerons ici, il existe systématiquement un nœud particulier appelé *racine* qui est le *premier* nœud de l'arbre.<sup>4</sup> Les nœuds n'ayant aucun successeur sont appelés *feuilles*. Les arbres ont pour trait particulier qu'entre toute paire de nœuds, il existe *un unique* chemin les reliant. La *distance* entre deux nœuds est la *longueur* de cet unique chemin, i.e. le nombre d'arêtes qu'il contient. La *profondeur* (parfois également appelé *le niveau*) d'un nœud  $x$ , notée  $p(x)$  est la distance entre  $x$  et la racine. La profondeur d'un arbre  $T$  (parfois également appelé *la hauteur de l'arbre*), notée  $h(T)$ , est la profondeur maximale d'un nœud de l'arbre.

Considérons un nœud  $x$  d'un arbre  $T$  de racine  $r$ . Si  $y$  est lié par une arête à  $x$ , alors soit (i)  $y$  est sur le chemin entre  $r$  et  $x$ , en quel cas  $y$  est le *nœud parent* (parfois dit *nœud père* ou simplement *père*) de  $x$ ; soit (ii)  $y$  n'est pas sur ce chemin, en quel cas  $y$  est un *nœud enfant* (parfois dit *nœud fils* ou simplement *fils*) de  $x$ . Notez que si  $y$  est le parent de  $x$ , alors  $p(x) = p(y) + 1$ . De plus, tous les nœuds, sauf la racine, ont un unique parent. Cela

3. Nous essayerons, tant bien que mal, dans ce document de réserver le mot *nœud* aux arbres et le mot *sommet* aux graphes. Même s'il n'y a pas de différence fondamentale entre les deux, cela nous permettra de clarifier le contexte.

4. Notez que de manière générale, la notion d'arbre est définie sur base de la notion de graphe (c.f. la [section 2.8](#)) et les arbres ne sont, en général, pas enracinés, sauf si explicitement mentionné.

nous permet de garantir qu'un arbre à  $n$  nœuds possède exactement  $n - 1$  arêtes. Pour un certain nœud  $x$  de  $T$ , nous appellerons *ancêtres* de  $x$  tous les nœuds se trouvant sur le chemin entre  $r$  et  $x$ . En particulier le parent de  $x$  est un ancêtre, mais le parent du parent est également un ancêtre, etc. De manière symétrique, si  $y$  est un ancêtre de  $x$ , nous dirons que  $x$  est un *descendant* de  $y$ . Les descendants de  $y$  sont donc tous les nœuds  $x$  tels que  $y$  se trouve sur l'unique chemin entre  $x$  et  $r$ . Il est tout à fait légitime de se demander si dans cette définition nous considérons qu'un nœud est son propre descendant, et donc par extension son propre ancêtre. Il n'y a pas de consensus sur la question, mais nous considérerons que oui ici. Pour deux nœuds  $x$  et  $y$  de  $T$ , nous appellerons le *plus petit ancêtre commun* entre  $x$  et  $y$  dans  $T$ , noté  $\text{LCA}_T(x, y)$  (pour *Lowest Common Ancestor*), i.e. le nœud  $z$  de profondeur maximal tel que  $z$  est un ancêtre de  $x$  et de  $y$ .

Un arbre dont tous les nœuds ont au plus  $m$  enfants est appelé arbre  $m$ -aire. En particulier, nous porterons un certain intérêt aux arbres binaires et ternaires (donc dans lesquels tout nœud a au plus 2 voire 3 nœuds). Un arbre  $m$ -aire est dit *parfait* lorsque toutes les feuilles ont la même profondeur et tous les nœuds internes (donc les nœuds n'étant pas des feuilles) ont exactement  $m$  enfants. On dit qu'un arbre  $T$  de profondeur  $h(T)$  est *complet* lorsque tous les niveaux  $< h(T)$  sont complets (i.e.  $T$  contient  $2^h$  nœuds de profondeur  $h$  pour  $0 \leq h < h(T)$ ) et le dernier niveau (i.e. le niveau  $h(T)$ ) n'est pas nécessairement saturé, mais ne peut pas avoir de trou (i.e. si un nœud a un enfant droit, alors il doit avoir un enfant gauche; et si un nœud n'a pas deux enfants, alors tous les nœuds à sa droite n'ont aucun enfant).

Parfois les arbres que nous avons appelé *parfaits* sont dits *complets*, et les arbres que nous avons appelé *complets* sont donc appelés *presque-complets* (ou *semi-complets* ou encore *pseudo-complets*). Même si nous essayons d'être explicites autant que possible dans ce document, il est possible que les deux conventions soient utilisées. Si c'est le cas, toutes nos confuses !

Si  $T$  est un arbre enraciné en  $r$ , et que nous regardons un autre nœud  $x \neq r$  de  $T$ , nous pouvons parler du *sous-arbre induit par  $x$*  qui est le *sous-arbre* de  $T$  dont la racine est  $x$ , i.e. l'arbre obtenu en ne conservant que  $x$  et ses descendants.

Dans certains cas, nous allons considérer un *ordre* sur les enfants de chaque nœud, i.e. nous parlerons du *premier enfant*, du *deuxième enfant*, etc. Dans le cas des arbres binaires, nous parlerons plutôt d'*enfant gauche* et d'*enfant droit*. Dans le cas des arbres ternaires, nous ajouterons l'*enfant du milieu*. Notez bien que cela implique que dans ce cas-là, nous ferons une différence entre les deux arbres ci-dessous :



En effet, même si dans les deux cas, l'arbre contient simplement une racine qui a un unique enfant, dans le premier cas, cet enfant est un enfant gauche alors que dans le second cet enfant est un enfant droit. Nous considérerons généralement de tels arbres dans ce cours, sauf si mentionné explicitement. Cela sera particulièrement important lorsque nous parlerons de BSTs (c.f. la [section 2.5](#)).

### 2.4.1 Implémentation

Il existe deux manières classiques de représenter des arbres :

1. via un vecteur ;
2. via des nœuds et des références/pointeurs.

Le premier cas est très utile lorsque nous considérons des arbres  $m$ -aires dont le nombre de nœuds est connu à l'avance. En effet, nous pouvons représenter un arbre  $m$ -aire (supposons le complet pour l'instant) à  $n$  nœuds via un vecteur dont la racine est stockée à la position d'indice 0, et où les  $m$  enfants du nœud à l'indice  $i$  se trouvent aux indices  $im + 1 \dots im + m$ . En particulier pour les arbres binaires, nous aurons la racine en position 0, l'enfant gauche de la racine en position 1, l'enfant droit de la racine en position 2, l'enfant gauche de l'enfant gauche de la racine en position 3, l'enfant droit de l'enfant gauche en position 4, l'enfant gauche de l'enfant droit en position 5, etc. Cette représentation sera utilisée lorsque nous parlerons de files à priorité (c.f. la [section 2.6](#)).

Le second cas est utilisé lorsque soit (i) le nombre total de nœuds que va contenir l'arbre n'est pas connu à l'avance ; soit (ii) les nœuds peuvent avoir un nombre arbitraire d'enfants. Nous allons voir deux manières équivalentes de représenter un arbre de cette manière, et pour comprendre ces deux possibilités, il faut d'abord remarquer qu'un arbre est composé de nœuds reliés entre eux, et que nous pouvons tout à fait identifier le nœud  $x$  avec  $T[x]$ , le sous-arbre de racine  $x$ . Nous pouvons dès lors choisir entre une représentation récursive ou une représentation itérative. Nous regarderons la situation récursive uniquement dans le cas des arbres binaires : dans ce cas un arbre binaire est composé de (i) une racine ; (ii) un sous-arbre gauche ; et (iii) un sous-arbre droit. Cela peut être implémenté de la façon suivante :

---

```

1 class BinaryTree:
2     def __init__(self, root, left=None, right=None):
3         self.root_ = root
4         self.left_ = left
5         self.right_ = right

```

---

Dans l'option itérative, il nous faut une classe `Node` qui représentera un nœud de l'arbre, et une classe `BinaryTree` qui contiendra une instance de `Node` représentant la racine. En voici une implémentation possible :

---

```

1 class Node:
2     def __init__(self, value):
3         self.value_ = value
4         self.children_ = []
5
6     def add_child(self, node: Node):
7         self.children_.append(node)
8
9     def __iter__(self):
10        return iter(self.children_)
11
12 class BinaryTree:
13     def __init__(self, root):
14         self.root_ = root

```

---

### 2.4.2 Relations d'ordre (📌)

Les listes chaînées induisent un ordre total (ou ordre linéaire) sur les éléments contenus puisque nous pouvons dire  $x \prec y$  lorsque  $x$  apparaît en position  $i$ ,  $y$  apparaît en position  $j$ , et  $i < j$ . En effet cet ordre est bien total puisque chaque élément se trouve à une position bien définie (en supposant que tous les éléments du conteneur soient distincts), et que toute paire d'indices est comparable.

En ce sens, les arbres (peu importe leur arité) induisent un ordre *partiel* sur les éléments contenus :  $x \prec y$  lorsque  $x$  est un ancêtre de  $y$ . De plus, tout comme un ordre total est un cas particulier d'ordre partiel (i.e. le cas où en réalité tous les éléments sont deux à deux comparables), les listes chaînées sont un cas particulier d'arbre : le cas où l'arbre est unaire (i.e. d'arité 1, ou encore chaque nœud a un unique enfant, ou encore le cas où l'arbre est un *chemin* au sens de la théorie des graphes). Dans ce cas, la relation d'ordre partiel induite sur l'arbre (qui n'est autre que la relation d'ancêtre) donne bien un ordre total.

## 2.5 Séquences triées

L'ADT *Séquence triée* est un type de conteneur trié (comme son nom l'indique) dans le sens où il est défini par les opérations suivantes :

- récupérer le plus petit élément (`get_first`) ;
- récupérer le plus grand élément (`get_last`) ;
- insérer un élément dans le conteneur (`insert`) ;
- trouver un élément s'il appartient au conteneur (`find`) ;
- retirer un élément du conteneur (`remove`) ;
- sur base d'un élément  $x$  donné, récupérer l'élément après  $x$  dans le conteneur (`get_next`) ou récupérer l'élément avant  $x$  dans le conteneur (`get_previous`).

Notons ici qu'il est implicite que les éléments placés dans un tel conteneur sont munis d'une relation d'ordre totale, i.e. ils peuvent toujours être ordonnés du plus petit au plus grand. Il est possible de regarder ce qu'il est possible de faire si nous relaxons cette contrainte et que donc nous retirons `get_first`, `get_last`, `get_next` et `get_previous` de la définition de l'ADT (c.f. la [section 2.7](#)).

La représentation usuelle d'une séquence triée est à l'aide d'un arbre binaire de recherche, ou BST (pour *Binary Search Tree*),<sup>5</sup> mais nous verrons d'abord une manière *naïve* de représenter une séquence triée via un vecteur.

### 2.5.1 Implémentation par vecteur

Si nous représentons le conteneur par un vecteur trié, il est clair que nous pouvons implémenter la classe suivante :

---

```

1 class Sequence:
2     def __init__(self):
3         self.vec_ = []
4     def __len__(self):
5         return len(self.vec_)
```

---

5. Notons que *binary search* désigne une recherche dichotomique en anglais, et donc BST veut plutôt dire *arbre de recherche dichotomique* (ou *arbre de recherche binaire*) au lieu d'*arbre binaire de recherche*, mais que voulez-vous, une fois qu'un nom est établi, il est compliqué de le changer.

```

6     def is_empty(self):
7         return len(self) == 0
8     def get_first(self):
9         return self.vec_[0]
10    def get_last(self):
11        return self.vec_[-1]
12    def insert(self, v):
13        if not self.is_empty() and v < self.get_first():
14            self.vec_.insert(0, v)
15        else:
16            self.vec_.insert(self._find_idx(v)+1, v)
17    def find(self, v):
18        if self.is_empty() or v > self.get_last() or v < self.get_first():
19            raise KeyError()
20        idx = self._find_idx(v)
21        if self.vec_[idx] == v: return self.vec_[idx]
22        else: raise KeyError()
23    def remove(self, v):
24        idx = self._find_idx(v)
25        if self.vec_[idx] == v:
26            self.vec_ = self.vec_[:idx] + self.vec_[idx+1:]
27        else:
28            raise KeyError()
29    def get_next(self, x):
30        if x == self.get_last(): raise KeyError()
31        idx = self.find_idx(x)
32        if self.vec_[idx] == x:
33            idx += 1
34        return self.vec_[idx]
35    def get_previous(self, x):
36        if x == self.get_first(): raise KeyError()
37        idx = self.find_idx(x)
38        return self.vec_[idx]
39    def _find_idx(self, v):
40        ''' Return i s.t. vec_[i] == v if such an index exists
41            and the largest value i such that vec_[i] < v otherwise '''
42        lo, hi = 0, len(self)
43        while lo < hi:
44            mid = (lo+hi) // 2
45            if self.vec_[mid] > v:
46                hi = mid
47            elif self.vec_[mid] == v:
48                hi = lo = mid
49            else:
50                lo = mid+1
51        return lo

```

---

L'intérêt de cette implémentation est que la recherche d'un élément peut se faire de manière très efficace : la recherche dichotomique nécessite au plus  $\sim \log_2 n$  comparaisons



(pour  $n$  la taille du conteneur). L'insertion fait également usage de la recherche dichotomique afin de trouver où insérer, donc le nombre de comparaisons reste intéressant. Cependant, dans le pire des cas, l'élément inséré est le plus petit du conteneur et donc tous les éléments déjà insérés doivent être déplacés. De plus, une insertion nécessite en moyenne  $\sim \frac{n}{2}$  opérations.

### 2.5.2 Implémentation par liste chaînée bidirectionnelle

Si nous voulons nous débarrasser de ce nombre linéaire d'opérations pour les insertions/-suppressions d'éléments, nous pouvons utiliser une liste chaînée bidirectionnelle. Cependant, il n'est pas possible d'effectuer une recherche dichotomique sur une liste chaînée puisque la notion centrale est l'accès à un élément quelconque (l'élément *mid*) en temps constant, ce qui n'est pas possible dans une liste chaînée.<sup>6</sup>

### 2.5.3 Implémentation par BST

Il est possible d'implémenter une séquence triée via un *arbre binaire de recherche* (BST) afin de garantir que toutes les opérations (donc insertion, suppression et recherche) requièrent  $\Theta(\log n)$  opérations dans le pire des cas (ou enfin presque, nous reparlerons d'équilibrage des BSTs à la fin de cette section).

Un BST est *simplement* un arbre binaire dont chaque nœud contient une entrée du conteneur et satisfaisant la propriété suivante : si  $x$  est un nœud de l'arbre, et que  $y$  est un descendant (au sens défini dans la [section 2.4](#)) de  $x$ , alors l'entrée en  $x$  est strictement supérieure à celle de  $y$  si et seulement si  $y$  est dans le sous-arbre gauche de  $x$ , et donc l'entrée en  $x$  est bornée par celle de  $y$  si et seulement si  $y$  est dans le sous-arbre droit de  $x$ .

Nous allons écrire la classe `BinarySearchTree` sur base de la classe `BstNode` (qui, comme son nom l'indique subtilement, représente un nœud du BST) :

---

```

1 class BstNode:
2     def __init__(self, value, parent: 'BstNode', left: 'BstNode'=None,
3         ↪ right: 'BstNode'=None):
4         self.value_ = value
5         self.parent_ = parent
6         self.left_ = left
7         self.right_ = right

```

---

Nous supposons que nous avons des propriétés en lecture et en écriture pour chacun des 4 attributs de la classe `BstNode`. Notons que nous gardons ici une référence au nœud `parent` car nous en aurons besoin pour écrire la suppression (même s'il est possible de faire sans). La classe `BinarySearchTree` est définie par son unique attribut `root` :

---

```

1 class BinarySearchTree:
2     def __init__(self):

```

---

6. En réalité, il est tout à fait possible d'avoir une structure qui est *globalement* une liste chaînée avec quelques fioritures qui permet une insertion en temps constant et une recherche en temps logarithmique *en moyenne* : les *skip-lists* qui seront vues à la fin de ce cours d'Algorithmique I. Il est également possible de *déterminiser* les *skip-lists* (qui sont une structure aléatoire par essence) afin de garantir une recherche en  $\Theta(\log n)$  dans le pire des cas, où  $n$  est la taille du conteneur. Cette notion (dans sa version probabiliste) a été introduite en 1989 par William Pugh, donc près de 30 ans après les BSTs.

---

```
3     self.root = None
```

---

Avec une telle structure, il est possible de faire un sorte de recherche dichotomique. En effet, pour savoir si la valeur  $x$  est stockée dans l'arbre, il faut partir de la racine, comparer la valeur de la racine (disons  $y$ ) à  $x$ , et agir selon les trois possibilités :

1. si  $x = y$ , alors nous avons trouvé l'élément  $x$ , il est donc dans l'arbre ;
2. si  $x < y$ , alors  $x$ , s'il est bien dans l'arbre, se trouve dans le sous-arbre gauche ;
3. si  $x > y$ , alors  $x$ , s'il est bien dans l'arbre, se trouve dans le sous-arbre droit.

Cela peut s'implémenter comme suit :

---

```
1     def __contains__(self, x) -> bool:
2         return self.find_node(x) is not None
3
4     def find_node(self, x) -> BstNode:
5         node = self.root
6         while node is not None and node.value != x:
7             if x < node.value:
8                 node = node.left
9             else:
10                node = node.right
11        return node
```

---

Dans le pire des cas, la recherche doit descendre tout l'arbre en partant de la racine, i.e. le nombre d'opérations (et donc de comparaisons) est au plus  $h(T)$  (qui est la hauteur de  $T$ ).

En partant d'une valeur, nous voulons également pouvoir trouver la valeur qui suit dans le conteneur. La méthode `get_next` s'implémente facilement si la valeur de laquelle nous cherchons le successeur a un fils droit (l'existence de valeurs plus grandes est donc garantie et nous savons où les trouver), mais dans le cas contraire, il faut d'abord remonter dans l'arbre pour pouvoir redescendre derrière.

---

```
1     def get_next_node(self, node) -> BstNode:
2         if node is None:
3             raise ValueError()
4         if node.right is not None:
5             node = node.right
6             while node.left is not None:
7                 node = node.left
8         else:
9             previous = node
10            node = node.parent
11            while node is not None and previous is node.right:
12                previous = node
13                node = node.parent
14        return node
```

---

De manière symétrique, il est clair que nous pouvons trouver la valeur précédente en inversant les enfants gauches/droits.

L'insertion d'un élément se fait de manière similaire à la différence près que lorsqu'on arrive sur une feuille, au lieu de renvoyer `False`, il faut créer un nouveau sous-arbre et l'insérer au bon endroit (notons qu'il faut d'abord choisir si on accepte d'avoir plusieurs fois la même valeur dans l'arbre ou non, les deux options sont tout à fait valides et sont utilisées dans des contextes différents, nous accepterons les valeurs dupliquées ici) :

---

```

1  def insert(self, x) -> None:
2      if self.root is None:
3          self.root = BstNode(x, None)
4      else:
5          child = self.root
6          while child is not None:
7              node = child
8              if x < child.value:
9                  child = child.left
10             else:
11                 child = child.right
12         if x < node.value:
13             node.left = BstNode(x, node)
14         else:
15             node.right = BstNode(x, node)

```

---

À nouveau, puisque nous insérons toutes les valeurs comme de nouvelles feuilles, il nous faut partir de la racine et trouver le bon chemin vers la feuille que nous allons insérer, i.e. le nombre de comparaisons nécessaires est donc au proportionnel à  $h(T)$ .

La suppression demande plus de travail puisque nous voulons impérativement maintenir la structure de BST, en particulier nous ne voulons pas de trou dans l'arbre (ce qui le déconnecterait) et nous voulons garder la relation d'ordre sur les arcs parent-enfant. Nous allons procéder récursivement pour cela. Observons les trois cas fondamentaux :

1. si le nœud à supprimer est une feuille, elle peut être simplement retirée sans traitement ajouté ;
2. si le nœud à supprimer a un unique enfant (qu'il soit enfant gauche ou droit), le nœud peut être *remplacé* par cet enfant ;
3. si le nœud à supprimer a un enfant gauche *et* un enfant droit, alors il faut réfléchir. Afin de conserver la relation d'ordre entre les valeurs stockées dans les nœuds, nous voulons conserver le nœud, mais changer sa valeur par une valeur suffisamment proche de la sienne pour ne pas causer de problème. Pour cela, nous allons prendre son successeur (attention : nous parlons ici de la valeur qui suit celle du nœud, et pas d'un nécessairement d'un enfant), remplacer la valeur du nœud par celle du successeur et récursivement supprimer le successeur.

Une fois cette distinction faite, et en utilisant la méthode `get_next_node`, nous pouvons maintenant écrire la suppression (nous supposons ici que `BstNode` dispose d'une propriété `nb_children` en lecture seule) :

---

```

1  def __delitem__(self, x):
2      node = self.find_node(x)
3      self.remove(node)
4
5  def remove(self, node):
6      if node is None:
7          raise ValueError()
8      if node.nb_children == 2:
9          next_node = self.get_next_node(node)
10         node.value = next_node.value
11         self.remove(next_node)
12         return node
13     else:
14         if node is self.root:
15             if node.left is not None:
16                 self.root = node.left
17             else:
18                 self.root = node.right
19                 self.root.parent = None
20         else:
21             parent = node.parent
22             if node.left is not None:
23                 child = node.left
24             else:
25                 child = node.right
26             node_is_left_child = parent.left is node
27             if node_is_left_child:
28                 parent.left = child
29             else:
30                 parent.right = child
31             if child is not None:
32                 child.parent = parent

```

---

### BSTs équilibrés

Les différentes opérations montrées ci-dessus nécessitent toutes  $\Theta(h(T))$  opérations dans le pire des cas. Mais notons qu'il est possible (relativement facilement d'ailleurs) d'avoir un BST de  $n$  éléments et de hauteur  $\Theta(n)$  (par exemple en insérant les éléments dans l'ordre croissant ou dans l'ordre décroissant). Nous savons que nous ne pourrions pas avoir mieux que des opérations logarithmiques en temps dans le pire des cas ici puisque la profondeur d'un arbre binaire de  $n$  éléments est au moins  $\sim \log_2 n$ . Par contre, si nous garantissons que le BST créé est *équilibré*<sup>7</sup>, alors cette borne inférieure devient également une borne supérieure. Les deux constructions classiques de BST équilibrés sont les AVLs (vu dans ce cours, mais pas au TP), et les RBTs (arbres rouge-noirs) qui seront vus en INFO-F203 — Algorithmique II. Les AVLs ont une profondeur au plus  $\sim 1.44 \log_2 n$  alors que les RBTs

---

7. Ce concept n'est pas simple à définir, dès lors il existe plusieurs notions d'équilibre qui ne sont pas équivalentes, mais qui atteignent toutes le même objectif : une structure d'arbre reposant sur cette définition a une profondeur  $\mathcal{O}(\log n)$  dans le pire des cas.

ont une profondeur au plus  $2 \log_2 n$ . Il n'y a pas de *meilleure* manière d'équilibrer un BST puisque tout gain dans une opération est compensé dans une autre, et le choix entre les deux (s'il doit être fait) est décidé en fonction de l'utilisation, voire en faisant des *benchmarks*, mais nous nous emballons.

## 2.6 Files à priorité (Heaps)

### 2.6.1 File à priorité

Une *file à priorité* (ou *priority queue* en anglais) est une généralisation des piles (c.f. la [sous-section 2.1.2](#)) et des files (c.f. la [sous-section 2.1.3](#)). En effet ces deux structures de données permettent d'insérer des éléments et de les consulter ou de les extraire, mais uniquement de manière linéaire (soit dans l'ordre d'insertion soit dans l'ordre inverse). Dans une file à priorité, nous assignons à chaque élément du conteneur une... priorité, et nous pouvons toujours consulter/extraire l'élément de priorité maximale.

Plus précisément une file à priorité est définie par les opérations suivantes :

- insérer un élément avec une priorité donnée (*insert*) ;
- consulter l'élément de priorité maximale (*top* ou parfois *peek*) ;
- retirer l'élément de priorité maximale (*pop*).

### 2.6.2 Implémentation via un heap

Une implémentation classique des files à priorité est via un *heap* (ou *tas*), i.e. un arbre binaire presque-complet satisfaisant la propriété suivante : tout nœud  $x$  de l'arbre binaire est au moins aussi grand que ses enfants (ou de manière équivalente, par transitivité, au moins aussi grand que tous ses descendants). Notons que cette structure est également appelée *max-heap* puisque l'élément maximal se situe toujours à la racine, mais nous pouvons définir, de manière totalement symétrique, un *min-heap* dans lequel chaque nœud est borné par ses enfants (ou descendants). Puisque  $\max\{x, y\} = -\min\{-x, -y\}$ , tout ce que nous pourrions dire sur les max-heaps tiendra également sur les min-heaps, et nous nous concentrerons donc sur les max-heaps (que nous appellerons *heaps*).

Puisqu'un heap est presque-complet, il ne peut pas avoir de trous, et donc la représentation d'arbre binaire via un tableau ou un vecteur est très intéressante (entre autres pour des raisons de caching, c.f. INFOF-102 — Fonctionnement des ordinateurs), surtout si le nombre d'éléments que va contenir le heap est connu à l'avance.<sup>8</sup> Nous pouvons alors écrire :

---

```

1 class Heap:
2     def __init__(self, n: int=1):
3         self.size_ = 0
4         self.array_ = [None] * n
5
6     def __len__(self):
7         return self.size_
8
9     def parent_idx(self, i: int) -> int:
10        return (i-1) // 2

```

---

8. En réalité, puisqu'il est possible de faire des vecteurs très efficaces en terme de coût des réallocations, même si cette taille n'est pas bornée, les heaps se comportent tout de même très bien.

```

11
12     def left_child_idx(self, i: int) -> int:
13         return 2*i + 1
14
15     def right_child_idx(self, i: int) -> int:
16         return 2*i + 2

```

---

Puisque l'élément de plus haute priorité est toujours à la racine, la méthode `top` peut s'écrire très facilement (avec une vérification pour s'assurer qu'on ne prend pas le top d'un heap vide) :

```

1     def top(self) -> object:
2         if len(self) == 0: raise ValueError()
3         return self.array_[0]

```

---

Pour insérer un élément, nous allons l'insérer dans la seule position disponible pour conserver le fait que l'arbre est pseudo-complet, i.e. la première feuille libre ; et pour garantir la relation d'ordre de heap, nous allons faire *remonter* cet élément tant qu'il est plus grand que son parent (et donc bien sûr qu'on n'a pas atteint la racine en quel cas, la notion de parent est mal définie). La fonction en charge de faire remonter un élément s'appelle ici `swim` (ou parfois `priority_up`, ou encore `increase_key`) :

```

1     def insert(self, priority: object) -> None:
2         if self.size_ < len(self.array_):
3             self.array_[self.size_] = priority
4         else:
5             self.array_.append(priority)
6             self.swim(self.size_)
7             self.size_ += 1
8
9     def swim(self, idx: int) -> None:
10        if idx == 0:
11            return
12        parent_idx = self.parent_idx(idx)
13        if self.array_[idx] > self.array_[parent_idx]:
14            self.swap(idx, parent_idx)
15            self.swim(parent_idx)

```

---

La méthode `swap` s'occupe, comme son nom l'indique, simplement d'échanger deux entrées du vecteur interne :

```

1     def swap(self, i: int, j: int) -> None:
2         self.array_[i], self.array_[j] = self.array_[j], self.array_[i]

```

---

Le problème en supprimant la racine est qu'on crée un trou qui doit être comblé. Or afin de maintenir le fait que l'arbre est pseudo-complet, la seule possibilité pour combler ce trou est de prendre la dernière feuille existante et de la mettre à la racine. Il faut ensuite s'assurer que l'ordre de heap est bien maintenu et donc de faire descendre cette entrée tant qu'elle est plus petite qu'un de ses deux enfants. Il faut tout de même faire attention à bien

choisir la branche dans laquelle on fait descendre le nœud puisque si par exemple le nœud qu'on fait descendre a une priorité  $x$ , son enfant gauche a priorité  $y$  et son enfant droit a priorité  $z$  et si ces valeurs satisfont  $x < y < z$ , nous devons impérativement descendre dans la branche droite sinon on fait remonter  $y$ , mais maintenant  $y$  ne satisfait pas la relation d'ordre et nous avons un problème. Il faut donc trouver lequel des enfants (s'ils existent !) a la priorité la plus haute, et faire descendre le nœud dans cette branche-là. La fonction en charge de faire descendre un nœud s'appelle `sink` (ou parfois `priority_down`, voire `decrease_key`) :

---

```

1  def pop(self) -> None:
2      if len(self) == 0: raise ValueError()
3      self.array[0] = self.array[self.size_-1]
4      self.sink(0)
5      self.size_ -= 1
6
7  def sink(self, idx: int) -> None:
8      left_idx = self.left_child_idx(idx)
9      right_idx = self.right_child_idx(idx)
10     if left_idx >= len(self):
11         return
12     max_idx = left_idx
13     if right_idx < len(self):
14         if self.array[right_idx] > self.array[left_idx]:
15             max_idx = right_idx
16     if self.array[max_idx] > self.array[idx]:
17         self.swap(idx, max_idx)
18         self.sink(max_idx)

```

---

Bien sûr, les méthodes `swim` et `sink` ont été implémentées de manière récursive ici, mais elles peuvent tout à fait être écrites de manière itérative avec une boucle `while`.

### 2.6.3 Complexité

L'implémentation d'une file à priorité via un tas nous permet d'avoir une insertion et une suppression en temps au plus logarithmique en la taille du conteneur puisqu'il faut, dans le pire des cas, remonter (ou redescendre) un chemin liant une feuille à la racine, mais puisque l'arbre est équilibré, un tel chemin a longueur au plus  $\sim \log_2 n$ . De plus, la consultation de l'élément de priorité maximale se fait en temps constant puisque cet élément est toujours à la racine de l'arbre binaire, i.e. à la position 0 du vecteur sous-jacent.

### 2.6.4 Heapsort

Il est assez clair qu'une file à priorité permet toujours de trier un vecteur, et de plus si les trois opérations s'exécutent en  $\mathcal{O}(\log n)$  (comme c'est le cas avec les heaps), alors nous pouvons trier un vecteur de taille  $n$  en temps  $\mathcal{O}(n \log n)$  et en nécessitant au plus  $\Theta(n)$  en mémoire :

---

```

1 def sort(vector):
2     n = len(vector)
3     pq = PriorityQueue(n)
4     for x in vector:
5         pq.insert(x)
6     for i in reversed(range(n)):
7         vector[i] = pq.top()
8         pq.pop()

```

---

Cependant, si nous savons que la file à priorité est implémentée via un heap, nous pouvons faire mieux que ça : nous savons qu'un heap est représenté dans un vecteur, et nous avons déjà un vecteur sur lequel nous travaillons, il serait bien de se débrouiller avec celui-là et ainsi ne pas avoir besoin d'espace supplémentaire. De plus, dans l'exemple ci-dessus, la création de la file à priorité (l. 4-5) nécessite  $\Theta(n \log n)$  comparaisons dans le pire des cas (e.g. si les éléments sont insérés dans l'ordre croissant), mais nous pouvons en réalité le faire en temps linéaire. Bien sûr, la seconde partie du tri devra nécessairement se faire en temps  $\Omega(n \log n)$  dans le pire des cas (nous verrons cela plus en détail en INFO-F203 — Algorithmique II), mais tout gain est bon à prendre.

Nous allons d'abord *transformer* le vecteur reçu en un heap (une opération qu'on appelle *heapify* ou *heapification*) et ensuite, nous allons itérativement, retirer la racine, ce qui diminue la taille du heap en libérant une place à droite et placer la valeur qui était à la racine à cet endroit. Puisque nous retirons les éléments du plus grand au plus petit et que nous remplissons le vecteur de droite à gauche, il finit bien trié dans l'ordre croissant.

---

```

1 def heapsort(vector):
2     size = n = len(vector)
3     # heapify
4     for idx in reversed(range(n//2 + 1)):
5         sink(vector, idx, size)
6     # sort
7     for i in range(n-1):
8         swap(vector, 0, n-1-i)
9         size -= 1
10        sink(vector, 0, size)
11 def sink(v, i, n):
12     left = 2*i + 1
13     right = 2*i + 2
14     if left >= n:
15         return
16     if right < n and v[right] > v[left]:
17         max_idx = right
18     else:
19         max_idx = left
20     if v[max_idx] > v[i]:
21         swap(v, max_idx, i)
22         sink(v, max_idx, n)
23 def swap(v, i, j):
24     v[i], v[j] = v[j], v[i]

```

---



## 2.7 Les ensembles dynamiques et les dictionnaires

Un *ensemble dynamique* (ou simplement *ensemble* ou *set*) est un ADT défini par les opérations suivantes :

- ajouter un élément dans le conteneur (`insert`) ;
- déterminer si un élément appartient à l'ensemble (`contains`) ;
- supprimer un élément du conteneur (`remove`).

Nous allons voir deux manières d'implémenter un ensemble dynamique :

- via un BST équilibré (e.g. un RBT ou un AVL) ;
- via une table de hachage.

Bien sûr une manière naïve d'implémenter un ensemble dynamique serait via un vecteur ou une liste chaînée, mais ces trois opérations seraient linéaires en la taille du conteneur dans le pire des cas, ce qui n'est pas envisageable en pratique.

Nous utiliserons les notations suivantes dans la suite de cette section :  $U$  est l'ensemble des éléments possibles pour le conteneur et  $n$  est le nombre d'éléments stockés dans l'ensemble considéré.

### 2.7.1 Implémentation via un BST

Nous savons qu'il est possible d'implémenter un arbre binaire de recherche dont la profondeur est toujours au plus logarithmique en le nombre d'éléments contenus (c.f. la [sous-section 2.5.3](#)). Dès lors, si nous implémentons l'ADT *set* via un tel BST, nous savons que les trois opérations nécessaires peuvent s'exécuter en temps au plus logarithmique en la taille du conteneur.

Cependant un BST requiert une relation d'ordre sur les éléments à insérer, ce qui ne fait pas partie de la définition de l'ADT *set*. Dans ce cas, nous pouvons toujours fixer arbitrairement un ordre en prenant une injection de l'ensemble des éléments possibles vers  $\mathbb{N}$  (ce qui est toujours possible si cet ensemble est au plus dénombrable). Bien sûr, habituellement, nous désirons choisir cette fonction de manière à ce qu'elle soit facilement calculable (i.e. en temps au plus linéaire en la taille de l'encodage de ces éléments).

Le *hachage* est une extension de cette idée : nous allons en effet associer à chaque élément en entier (mais cette fois-ci pas nécessairement unique) et utiliser cet entier pour *indicer* notre conteneur (de manière interne bien sûr : le *monde extérieur* n'aura aucune idée cet indiciage).

### 2.7.2 Implémentation via adressage direct

Supposons maintenant que nous avons associé un entier à chaque élément possible pour le conteneur. De manière formelle, nous avons une *fonction de hachage*  $h : U \rightarrow K$  où  $U$  est l'ensemble des éléments possibles et  $K \subset \mathbb{N}$  est l'ensemble des nombres associés. Par convention, nous notons  $m := |K|$  le nombre d'entiers que nous pouvons potentiellement associer.

Si  $h$  est une bijection (i.e. chaque nombre est associé à exactement *un unique* élément  $x \in U$ ) et que donc  $K = \llbracket 0, m-1 \rrbracket = \{0, \dots, m-1\}$  où  $m$  est *raisonnable* (dans le sens  $m \ll S$  où  $S$  est la mémoire disponible), alors nous pouvons utiliser un vecteur que nous indiquerons par ces entiers.

---

```

1 class Set:
2     def __init__(self):
3         self.T = [None] * m
4     def insert(self, x):
5         k = h(x)
6         T[k] = x
7     def remove(self, x):
8         k = h(x)
9         T[k] = None
10    def contains(self, x):
11        k = h(x)
12        return T[k] is not None

```

---

En supposant que le temps d'exécution de la fonction  $h$  ne dépend ni de  $m$ , ni du nombre d'éléments dans le conteneur (ce qui n'est pas aberrant comme hypothèse, je vous rassure), alors les trois opérations désirées s'exécutent toutes en temps  $\mathcal{O}(1)$ , ce qui est très efficace !

Cette approche a tout de même deux *gros* problèmes qui la rendent inutilisable en pratique :

1. l'usage mémoire est  $\Theta(m)$  dans tous les cas, même si  $n \ll m$  en quel cas, cette approche *gaspille* énormément de mémoire ;
2.  $m$  est rarement raisonnable.

Une bonne manière de se convaincre du point 2 est la suivante. Supposons que nous voulons stocker des mots de passe sous forme de chaînes de caractères en ASCII (bon alors c'est une très mauvaise chose, mais nous ne parlons pas de cybersécurité ici, une chose à la fois) et que nous savons que (i) tous ces mots de passe ont une longueur entre 5 et 10 caractères et (ii) les seules caractères autorisés sont les lettres minuscules, majuscules et les chiffres de 0 à 9. Nous savons donc que chaque caractère peut prendre  $62 = 26 + 26 + 10$  valeurs possibles. Le nombre de chaînes possible est donc (par la **Proposition 1.23**) :

$$\sum_{\ell=5}^{10} 62^\ell = 62^5 \sum_{\ell=0}^5 62^\ell = 62^5 \cdot \frac{62^6 - 1}{62 - 1} = 853\,058\,371\,851\,163\,296 \approx 0.853 \times 10^{18},$$

à savoir presque 1 milliard de milliards d'éléments. Dès lors même en supposant que chaque entrée de  $T$  ne prend qu'un byte, stocker ce vecteur nécessite plus de 853 000 TB de stockage. . . Et nous n'avons pas considéré les caractères spéciaux, ni les chaînes de longueur 11 !

À partir d'ici, nous noterons  $\alpha := \frac{n}{m}$  le *facteur de charge* (ou *load factor* en anglais), i.e. la proportion du vecteur  $T$  qui est remplie. La notation  $\alpha$  est utilisée dans le CLRS, mais Miller et Ranum utilisent plutôt la notation  $\lambda$  de manière équivalente dans leur textbook.

### 2.7.3 Implémentation via une table de hachage

Si par contre la fonction  $h : U \rightarrow K$  associe un entier à chaque élément de  $U$ , mais pas de manière unique (i.e.  $h$  n'est pas injective) et où  $m$  est raisonnable, alors les deux points mentionnés ci-dessus sont réglés. Nous pouvons toujours nous réduire à ce cas-là :

si nous avons une fonction  $h_1 : U \rightarrow \mathbb{N}$ , alors pour tout  $m \in \mathbb{N}^*$ , nous pouvons définir  $h_2 : U \rightarrow \llbracket 1, m \rrbracket$  par  $h_2 : x \mapsto h_1(x) \bmod m$ , et donc nous pouvons choisir  $m$  au préalable. Cependant nous avons maintenant un nouveau problème : si les valeurs  $x$  et  $y$  dans  $U$  sont telles que  $h(x) = h(y) = k$ , où  $x \neq y$ , alors comment représenter simultanément  $x$  et  $y$  dans le conteneur ?

Une telle paire  $(x, y)$  d'éléments distincts mais étant associées au même entier est appelée une *collision*. Il existe deux manières principales pour gérer les collisions dans une table de hachage :

1. la gestion par chaînage ;
2. la gestion par adressage ouvert.

### Gestion par chaînage

Nous pouvons gérer les collisions en créant un vecteur de taille  $m$  dont chaque élément est une liste chaînée. En procédant de la sorte, pour insérer un élément, il suffit de l'insérer dans la liste chaînée à l'indice associé. La recherche et la suppression fonctionnent de la même manière.

---

```

1 class Set:
2     def __init__(self):
3         self.T = [LinkedList() for _ in range(m)]
4     def insert(self, x):
5         k = h(x)
6         self.T[k].insert(x)
7     def remove(self, x):
8         k = h(x)
9         self.T[k].remove(x)
10    def contains(self, x):
11        k = h(x)
12        return self.T[k].contains(x)

```

---

Notons ici que dans le pire des cas, tous les éléments insérés sont associés à la même clef, et donc cette structure se réduit à une liste chaînée, i.e. la recherche, suppression et insertion prennent un temps linéaire dans le pire des cas. Il est toutefois possible d'utiliser les skip-lists pour conserver un temps logarithmique dans le pire des cas (c.f. la note <sup>6</sup>, page 73). De plus, dans ce contexte-ci,  $n < m$  et donc  $\alpha$  peut être supérieur à 1. D'ailleurs il peut être arbitrairement grand (si  $U = \llbracket 1, m \rrbracket$  et  $K = \{0\}$ , alors le facteur de charge peut être tout nombre entre 0 et  $m$ , bornes incluses).

L'hypothèse de hachage uniforme simple (ou SUHA pour *Simple Uniform Hashing Assumption*) est une hypothèse faite sur les fonctions de hachage et dit que les fonctions  $h : U \rightarrow K$  considérées sont telles que  $\mathbb{P}_x[h(x) = k] = \frac{1}{m}$ , i.e. en considérant une distribution uniforme sur les éléments de  $U$ , aucune case du vecteur  $T$  n'est favorisée. Bien qu'elle ne soit pas strictement vérifiée en pratique (même s'il y a différents degrés de divergence possibles par rapport à cette distribution uniforme des indices), elle est très pratique pour l'analyse de complexité.

Nous supposons également que le calcul de  $h(x)$  s'effectue en temps  $\mathcal{O}(1)$ .<sup>9</sup>

---

9. Plus précisément, nous supposons que le calcul de  $h(x)$  ne dépend ni de  $m$ , ni de  $n$ , mais peut dépendre

**Proposition 2.1.** *Sous l'hypothèse de hachage uniforme simple, en moyenne, pour  $x$  pris uniformément dans  $U$ , la recherche d'un élément nécessite  $\Theta(1 + \alpha)$  opérations.*

*Démonstration.* Notons ici  $L_k$  la longueur de la liste chaînée en position  $k$ . En prenant une de ces listes au hasard, sa longueur moyenne est  $\alpha$ . En effet :

$$\mathbb{E}_k[L_k] = \sum_{\ell=0}^{m-1} \mathbb{P}[k = \ell] L_\ell = \frac{1}{m} \sum_{\ell=0}^{m-1} L_\ell = \frac{1}{m} n = \alpha.$$

Supposons d'abord que la recherche est infructueuse (donc que l'élément recherché *n'est pas* dans l'ensemble). Dans ce cas, lors de la recherche, l'indice dans le vecteur est  $k = h(x)$  (qui est distribué uniformément par SUHA) est le nombre de comparaisons est exactement  $L_k$  (puisque l'élément doit être comparé à chacun des éléments de la liste chaînée). Le nombre moyen de comparaisons correspond donc à la longueur moyenne de ces listes chaînées et vaut donc  $\alpha$ . Le nombre total d'opérations est donc (en comptant le calcul de  $h(x)$ )  $\Theta(1 + \alpha)$ .

Supposons maintenant que la recherche est fructueuse (donc que l'élément recherché est dans l'ensemble). Dans ce cas, lors de sa recherche, en supposant que  $x$  dans la liste chaînée à l'indice  $h(x) = k$ , et que  $x$  est en position  $j$  dans cette liste chaînée, il faudra exactement  $j$  comparaisons afin de trouver  $x$ . En moyenne, cet élément sera en position  $\frac{1}{2}(L_k + 1)$  dans la liste puisque :

$$\frac{1}{L_k} \sum_{j=1}^{L_k} j = \frac{1}{L_k} \frac{L_k(L_k + 1)}{2} = \frac{1}{2}(L_k + 1).$$

Dès lors, puisque cet élément peut se trouver dans chacune des listes chaînées avec la même probabilité, le nombre de comparaisons est donc :

$$\sum_{k=0}^{m-1} \frac{1}{m} \frac{1}{2}(L_k + 1) = \frac{1}{2}(1 + \alpha).$$

En comptant le temps de calcul de  $h(x)$ , le nombre moyen d'opérations est  $\Theta(1 + \frac{1}{2}(1 + \alpha)) = \Theta(1 + \alpha)$ .  $\square$

**Remarque.** *Pour une preuve plus détaillée, voir les théorèmes 11.1 et 11.2 dans le CLRS.*

Contrairement à l'adressage direct, ici l'espace nécessaire pour représenter l'ensemble est linéaire en  $n + m$  et donc dépend de  $n$  en plus de dépendre de  $m$ . De plus, si nous pouvons garantir que  $\alpha = O(1)$  (et donc  $m = \Omega(n)$ ), alors nous pouvons garantir que les différentes opérations nécessitent toutes  $\Theta(1)$  opérations, ce qui est très efficace (mais attentions : cette hypothèse veut dire que la taille moyenne d'une liste chaînée doit être bornée par une constante, et donc  $m$  ne peut pas être choisi arbitrairement, ce qui n'est pas systématiquement le cas en pratique).

---

de la taille de l'encodage de  $x$ . Cette distinction est importante, et de plus cette notion de complexité selon la taille de l'encodage de l'input sera revue en INFO-F302 — Informatique Fondamentale et nous permettra de définir les classes de complexité P et NP, mais une fois encore : chaque chose en son temps.

### Gestion par adressage ouvert

Si soit (i) nous ne voulons pas utiliser de stockage en dehors du vecteur  $T$  (en effet l'utilisation d'une liste chaînée comme dans le cas précédent aura un gros impact sur la possibilité d'utiliser les systèmes de cache mémoire, c.f. INFO-F102 — Fonctionnement des ordinateurs) ; soit (ii) nous ne voulons pas autoriser des chaînes de longueur  $\omega(m)$ , nous devons changer de stratégie.

L'adressage ouvert est une variante de la gestion des collisions par chaînage dans laquelle nous allons faire un *pseudo-chaînage* au sein du vecteur directement, et dans lequel plusieurs chaînes peuvent s'intersecter en toute position.

Nous allons ici utiliser une fonction de hachage de la forme :

$$h : U \times \mathbb{N} \rightarrow K : (x, k) \mapsto h(x, k).$$

En particulier, nous ne hachons plus uniquement l'élément  $x \in U$ , mais nous le hachons de manière paramétrée par un entier  $k$ . L'idée va être d'utiliser ce deuxième paramètre de manière incrémentale pour représenter le nombre de fois qu'on essaye de trouver une position libre dans le tableau  $T$ . Lors d'une insertion, nous allons donc commencer par regarder à l'indice  $k_0 = h(x, 0)$ . Soit cette position est libre dans le vecteur  $T$  en quel cas l'insertion est faite, soit elle ne l'est pas, en quel cas on regarde l'indice  $k_1 = h(x, 1)$ , etc. Notons bien qu'il faut faire bien attention en choisissant la fonction  $h$  afin que pour une table de taille  $m$ , la séquence d'indices  $(h(x, 0), h(x, 1), \dots, h(x, m-1))$  soit une permutation des indices  $\{0, \dots, m-1\}$  afin de toujours pouvoir insérer un élément s'il existe une position disponible.

Lors de la recherche d'un élément, nous allons procéder de la même manière : en commençant par l'indice  $h(x, 0)$ , puis en regardant à l'indice  $h(x, 1)$ , etc. jusqu'à trouver l'élément recherché. Notons cependant que, si l'élément recherché n'est pas dans le conteneur, il est possible de s'en rendre compte *relativement vite* parfois : si à une certaine étape, l'indice  $h(x, i)$  est vide dans le vecteur  $T$ , nous savons que  $x$  ne peut pas être dans le conteneur. En effet s'il avait été dans le conteneur, lors de l'insertion, la première position non-vide (donc le plus petit  $j$  tel que  $h(x, j)$  soit un indice valide) aurait été choisi ; or ici la recherche a trouvé un plus petit  $i$  valide pour insérer, mais l'élément n'a pas été trouvé aux indices  $(h(x, 0), \dots, h(x, i-1))$  et donc ne peut pas être dans le conteneur.

La raison pour laquelle l'adressage ouvert peut être vu comme une forme de chaînage interne est donc la suivante : la *chaîne des collisions* pour l'objet  $x$  est la chaîne donnée par les indices  $(h(x, 0), \dots, h(x, i))$  où  $i$  est le plus petit entier où  $x$  a pu être inséré. Cette chaîne n'est pas stockée explicitement et est bien interne puisqu'un conteneur chaîné (comme une liste chaînée) n'a dû être ajouté au conteneur : tout est dans la table  $T$ . De plus, deux chaînes ici peuvent tout à fait avoir une ou plusieurs intersections. Par exemple supposons que nous avons deux éléments  $x$  et  $y$  tels que  $h(x, 0) = k_0 \neq \ell_0 = h(y, 0)$  mais tels que les indices  $k_0$  et  $\ell_0$  soient déjà occupés lors de la tentative d'insertion. Il faut alors regarder les indices  $k_1 = h(x, 1)$  et  $\ell_1 = h(y, 1)$ . À ce moment, il est tout à fait possible que  $k_1 = \ell_1$ .

Voici une implémentation possible de l'ADT *set* avec une résolution des collisions par adressage ouvert (où on suppose que la fonction  $h(x, i)$  est déjà définie quelque part) :

---

```

1 class Set:
2     def __init__(self, m: int):
3         self.T_ = [None] * m
4         self.n_ = 0
5
6     def __len__(self) -> int:
7         return self.n_
8
9     def insert(self, x):
10        if self.n_ == len(self.T_):
11            raise ValueError('Unable to insert in a full set')
12        i = 0
13        inserted = False
14        while not inserted and i < len(self.T_):
15            k_i = h(x, i) % len(self.T_)
16            if self.T_[k_i] is None:
17                self.T_[k_i] = x
18                self.n_ += 1
19                inserted = True
20            else:
21                i += 1
22
23    def __contains__(self, x):
24        i = 0
25        while i < len(self.T_):
26            k_i = h(x, i) % len(self.T_)
27            if self.T_[k_i] is None:
28                return False
29            elif self.T_[k_i] == x:
30                return True
31            else:
32                i += 1
33        return False

```

---

**Remarque.** Notez bien que nous n'avons pas parlé de la suppression d'un élément dans un tel conteneur, ni de comment contourner le problème du conteneur plein lors de l'insertion. Ces deux opérations font l'objet des exercices supplémentaires 13.7 et 13.8.

Le choix de la fonction de hachage  $h : U \times \mathbb{N}$  est très important pour trouver des propriétés intéressantes. Nous pouvons aisément dire que si la fonction  $h$  est séparable en  $x$  et  $i$ , nous risquons fortement d'avoir des problèmes de *clustering*. Plus précisément, si la fonction  $h$  satisfait :

$$h(x, i) = f(x) + g(i),$$

pour toute paire  $(x, i)$ , alors dès qu'une collision apparaît entre deux éléments  $x, y \in U$  la séquence de sondage sera exactement la même. Dès lors, les éléments entrant en collision formeront des chaînes de collisions internes, et ces collisions seront de plus en plus longues à résoudre.

Nous voulons donc qu'il existe une interaction entre la valeur de  $x$  et celle de  $i$  lors de

l'évaluation de  $h(x, i)$ . Une option pour cela est le *double hachage* dans lequel notre fonction  $h$  est définie sur base de deux fonctions  $h_1, h_2 : U \rightarrow K$  (où  $h_1 \neq h_2$ ) de la manière suivante :

$$h : U \times \mathbb{N} \rightarrow K : (x, i) \mapsto h_1(k) + i \times h_2(x).$$

En procédant de la sorte, s'il y a collision entre les éléments  $x$  et  $y$  dans  $U$ , c'est que  $h_1(x) = h_1(y)$ , mais sous l'hypothèse que  $h_1$  et  $h_2$  sont indépendantes l'une de l'autre,<sup>10</sup> alors la probabilité que la collision se poursuive jusque  $h(x, 1) = h(y, 1)$ , et décroît exponentiellement vite en  $k$  pour  $h(x, k) = h(y, k)$ .

Notons bien que dans le cas de l'adressage ouvert (peu importe à quoi ressemble la fonction  $h$ ), le facteur de charge est toujours  $0 \leq \alpha \leq 1$  puisqu'aucun élément ne peut apparaître en dehors de la table  $T$ .

**Proposition 2.2.** *Dans le pire des cas, une recherche ou une insertion dans une table à adressage ouvert nécessite  $\sim n$  comparaisons.*

*Démonstration.* Dans le pire des cas, l'insertion ou la recherche doit passer par toutes les cases avant d'en trouver soit une vide, soit une qui contient l'élément recherché.  $\square$

L'hypothèse de hachage uniforme dans le cas de l'adressage ouvert peut s'exprimer comme ceci : toute permutation de  $\{0, \dots, m-1\}$  a la même probabilité d'être la séquence de sondage d'un élément  $x$ . Autrement dit :

$$\forall k \in \mathfrak{S}([0, m-1]) : \mathbb{P}_x [\forall 0 \leq i < m : h(x, i) = k_i] = \frac{1}{m!}.$$

**Proposition 2.3.** *Sous l'hypothèse de hachage uniforme, le nombre moyen de comparaisons pour une recherche infructueuse (et donc en particulier une insertion) est au plus  $\frac{\alpha}{1-\alpha}$ . Le nombre moyen de comparaisons pour une recherche fructueuse est au plus  $\frac{1}{\alpha} \log \frac{1}{1-\alpha}$ .*

Pour démontrer cette proposition, nous allons utiliser les deux petits lemmes techniques suivants.

**Lemme 2.4.** *Si  $x, y \in \mathbb{R}_0^+$  et  $x \leq y$ , alors :*

$$\forall \delta > 0 : \frac{x - \delta}{y - \delta} \leq \frac{x}{y} \leq 1.$$

*Démonstration.* Il est facile d'observer que  $\frac{x-\delta}{y-\delta} \leq \frac{x}{y}$  est vrai ssi  $(x-\delta)y \leq (y-\delta)x$ , et donc  $\delta x \leq \delta y$ , ce qui est vrai pour tout  $\delta \geq 0$  puisque  $x \leq y$ .  $\square$

**Lemme 2.5.** *Si  $X$  est une variable aléatoire prenant ses valeurs dans  $[1, N]$ , notons  $\mathbb{P}[X \geq t] = p_t$ . L'égalité est toujours vérifiée :*

$$\mathbb{E}[X] = \sum_{t=1}^N p_t,$$

où  $\mathbb{E}[X]$  est l'espérance de  $X$ .

10. Dans le sens où  $\mathbb{P}_{x,y}[h_1(x) = h_1(y), h_2(x) = h_2(y)] = \mathbb{P}_{x,y}[h_1(x) = h_1(y)]\mathbb{P}_{x,y}[h_2(x) = h_2(y)]$

*Démonstration.* Puisque  $\mathbb{P}[X = t] = \mathbb{P}[X \geq t] - \mathbb{P}[X \geq t + 1] = p_t - p_{t+1}$  (du fait que  $X$  est à valeurs dans un intervalle entier), nous avons le raisonnement suivant (en utilisant le fait que  $p_t = 0$  pour tout  $t > N$ ) :

$$\begin{aligned} \mathbb{E}X &= \sum_{t=1}^N t\mathbb{P}[X = t] = \sum_{t=1}^N t(p_t - p_{t+1}) = \sum_{t=1}^N tp_t - \sum_{t=1}^N tp_{t+1} = \sum_{t=1}^N tp_t - \sum_{t=2}^{N+1} (t-1)p_t \\ &= p_1 + \sum_{t=2}^N tp_t - \sum_{t=2}^N (t-1)p_t + Np_{N+1} = p_1 + \sum_{t=2}^N (t - (t-1))p_t + 0 = \sum_{t=1}^N p_t. \end{aligned}$$

□

*Démonstration de la Proposition 2.3.* Commençons par le cas d'une recherche infructueuse, et montrons que la probabilité qu'une insertion nécessite au moins  $t$  sondages est bornée par  $\alpha^{t-1}$ . En effet, si  $(k_0, \dots, k_{t-1})$  est la séquence des  $t$  premiers sondages, nous savons par SUHA qu'il y a  $\binom{m}{t}$  telles séquences toutes équiprobables. Cependant, il y a  $\binom{n}{t}$  séquences possibles de sondages tous occupés parmi les  $n$  éléments insérés dans le conteneur. La probabilité que la séquence  $(k_0, \dots, k_{t-1})$  ne donne que des indices déjà occupés est donc :

$$\frac{\binom{n}{t}}{\binom{m}{t}} = \frac{\frac{n!}{t!(n-t)!}}{\frac{m!}{t!(m-t)!}} = \frac{n!(m-t)!}{m!(n-t)!} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-t+1}{m-t+1}.$$

Maintenant, par le Lemme 2.4, nous savons que  $\frac{n-j}{m-j} \leq \frac{n}{m} = \alpha$  pour tout  $j \geq 0$ . Nous en déduisons que la probabilité que les  $t$  premiers sondages soient infructueux est au plus  $\alpha^t$ , et donc la probabilité qu'une insertion nécessite au moins  $t$  sondages est la probabilité que les  $t-1$  premiers sondages soient infructueux et est bornée par  $\alpha^{t-1}$ .

Observons finalement que le nombre moyen de sondages est, par le Lemme 2.5, au plus (par la Proposition 1.23) :

$$\sum_{t=1}^m \alpha^{t-1} = \sum_{t=0}^{m-1} \alpha^t = \frac{1 - \alpha^m}{1 - \alpha} \leq \frac{1}{1 - \alpha}.$$

Maintenant pour le cas d'une recherche fructueuse, nous savons que la séquence de sondage lors de la recherche sera la même que lors de l'insertion. Dès lors, en supposant que tous les éléments ont la même probabilité d'être recherchés, nous savons que si l'élément recherché (appelons-le  $x$ ) était le  $t$ ème élément inséré dans la table, alors il a fallu au plus  $\frac{1}{1 - \frac{t-1}{m}} = \frac{m}{m+1-t}$  sondages pour l'insérer. Sans présupposer quand  $x$  a été inséré dans la table, nous pouvons dire que le nombre moyen de sondages nécessaires pour son insertion est donc au plus :

$$\frac{1}{n} \sum_{t=1}^n \frac{m}{m+1-t} = \frac{m}{n} \sum_{t=1}^n \frac{1}{m+1-t} = \frac{m}{n} \sum_{m+1-n}^m \frac{1}{t} \leq \frac{m}{n} \log \frac{m}{m-n} = \frac{1}{\alpha} \log \frac{1}{1-\alpha},$$

où nous avons utilisé le fait que  $\frac{m}{n} = \frac{1}{\alpha}$ , et où nous avons (comme dans la Proposition 1.22) utilisé le fait que :

$$\sum_{t=a+1}^b \frac{1}{t} \leq \int_a^b \frac{1}{x} dx = \log b - \log a = \log \frac{b}{a}.$$

□



### 2.7.4 Dictionnaires comme cas particuliers

Notons que l'ADT *dictionnaire* (ou *map*) est une variante de l'ADT *set* dans lequel les éléments considérés ont tous la forme (key, value), et où il est classiquement demandé que pour chaque valeur de k, il existe au plus une entrée (key, value) du conteneur telle que `key == k`.

Cet ADT peut, tout comme un *set*, être implémenté par une table de hachage. La seule différence est que seule la valeur de `key` va être hachée, et que la valeur de `value` va être ignorée. En procédant de la sorte, la clef permet de donner la position dans la table T, et la valeur `value` est juste gardée en plus mais peut être renvoyée lors d'un appel à `__getitem__`.

## 2.8 Graphes

**Définition 12.** Soit  $V$  un ensemble de sommets.

- Un ensemble  $E \subset \binom{V}{2}$  (i.e. un ensemble  $E$  de la forme  $\{\{v_{i_1}, v_{j_1}\}, \dots, \{v_{i_m}, v_{j_m}\}\}$  où pour tout  $k : i_k \neq j_k$ ) est appelé *ensemble d'arêtes*.
- Un ensemble  $E \subset V \times V$  (i.e. un ensemble  $E$  de la forme  $\{(v_{i_1}, v_{j_1}), \dots, (v_{i_m}, v_{j_m})\}$ ) est appelé *ensemble d'arcs*.
- Si  $E$  est un ensemble d'arêtes, on appelle la paire  $G = (V, E)$  un *graphe non-dirigé*, et si  $E$  est un ensemble d'arcs, on appelle la paire  $G = (V, E)$  un *graphe dirigé* (ou *digraphe*).
- On appelle *graphe pondéré* tout triplet  $G = (V, E, \gamma)$  tel que  $(V, E)$  est un (di)graphe et où  $\gamma : E \rightarrow \mathbb{R}$  est appelée *application de poids*, i.e. le *poids* d'une arête (ou d'un arc)  $e \in E$  est  $\gamma(e)$ .
- Un (di)graphe (pondéré ou non) est dit *simple* si aucune arête (ou aucun arc) ne joint un sommet à lui-même.
- Un *chemin* dans  $G$  est une séquence de sommets  $(v_{i_0}, \dots, v_{i_k})$  telle qu'il existe une arête entre  $v_{i_j}$  et  $v_{i_{j+1}}$  pour tout  $0 \leq j < k$ . La *longueur* de ce chemin est le nombre d'arêtes présentes (ou de manière équivalente, un de moins que le nombre de sommets).
- La *distance* entre deux sommets  $u$  et  $v$  dans  $G$  est la plus petite longueur d'un chemin entre  $u$  et  $v$ . S'il n'existe pas de tel chemin, la distance est  $+\infty$ .

**Remarque.** Il existe une famille de graphes plus large appelée *multigraphes* dans laquelle il peut exister plusieurs arêtes (ou plusieurs arcs) joignant une paire de sommets donnés, ce qui n'est pas permis dans la définition donnée ci-dessus. Nous ne nous intéressons pas à ces graphes dans ce cours-ci.

**Définition 13.** Soit un graphe dirigé pondéré simple  $G = (V, E, \gamma)$  tel que  $V = \{v_1, \dots, v_n\}$  et  $E = \{e_1, \dots, e_m\}$ .

- La matrice d'adjacence de  $G$  est une matrice  $A \in \mathbb{R}^{n \times n}$  telle que :

$$A_{ij} = \begin{cases} \gamma(v_i, v_j) & \text{si } (v_i, v_j) \in E, \\ 0 & \text{sinon.} \end{cases}$$

- La matrice d'incidence de  $G$  est une matrice  $I \in \mathbb{R}^{n \times m}$  telle que :

$$I_{ij} = \begin{cases} \gamma(e_j) & \text{si } \exists v \in V \text{ s.t. } e_j = (v_i, v) \\ -\gamma(e_j) & \text{si } \exists v \in V \text{ s.t. } e_j = (v, v_i) \\ 0 & \text{sinon.} \end{cases}$$

**Remarque.** La matrice d'incidence est habituellement définie par  $\pm\sqrt{\gamma(e)}$  au lieu de  $\pm\gamma(e)$  pour des raisons techniques d'analyse de graphes (pour tout graphe, on peut définir une matrice appelée matrice de Laplace du graphe qui doit satisfaire, entre autres,  $L = I^\top I$  et  $L_{ij} = -\gamma(v_i, v_j)$  si  $v_i$  et  $v_j$  sont adjacents). Nous garderons ici la définition sans racine carrée pour des raisons de lisibilité (et parce que nous n'utilisons pas le Laplacien).

**Remarque.** Tout graphe non pondéré  $G = (V, E)$  peut être vu comme un graphe pondéré  $G' = (V, E, \mathbf{1})$  où l'application de poids  $\mathbf{1} : E \rightarrow \{1\} : e \mapsto 1$  assigne un poids de 1 à toutes les arêtes (ou tous les arcs). Les définitions de matrice d'adjacence et matrice d'incidence peuvent donc être étendues aux graphes non-dirigés en posant  $\gamma(e) = 1$  pour tout  $e \in E$ .

**Définition 14.** Soit un graphe non-dirigé  $G = (V, E)$ . Pour tout  $v \in V$ , on définit le degré de  $v$  par le nombre de sommets dans  $V$  adjacents à  $v$ . Si  $G$  est dirigé, pour tout  $v \in V$ , on définit le degré entrant de  $v$  par le nombre de sommets  $w$  dans  $V$  tels que  $(w, v) \in E$ , et le degré sortant de  $v$  par le nombre de sommets  $w$  dans  $V$  tels que  $(v, w) \in E$ . Lorsque l'on parle du degré d'un sommet dans un graphe non-dirigé sans préciser s'il s'agit du degré entrant ou sortant, il est habituellement question du degré sortant. Le degré d'un sommet  $v$  est noté  $\deg_G(v)$ , et si le graphe  $G$  en question est non-ambigu, on note alors  $\deg(v)$ .

**Proposition 2.6.** (Lemme des poignées de main) Si  $G = (V, E)$  est un graphe non-dirigé à  $n$  sommets et  $m$  arêtes, alors l'égalité suivante est vérifiée :

$$\sum_{v \in V} \deg(v) = 2m.$$

Si  $G$  est dirigé, alors l'égalité suivante est vérifiée :

$$\sum_{v \in V} \deg(v) = m.$$

Démonstration. Si  $G$  est dirigé, alors nous pouvons écrire :

$$m = |E| = \left| \bigcup_{v \in V} \{(v_i, v_j) \in E \text{ s.t. } v_i = v\} \right| = \left| \bigsqcup_{v \in V} \{(v_i, v_j) \in E \text{ s.t. } v_i = v\} \right|,$$

où  $\sqcup$  désigne une union disjointe. Dès lors :

$$m = \sum_{v \in V} |\{(v_i, v_j) \in E \text{ s.t. } v_i = v\}| = \sum_{v \in V} |\{w \in V \text{ s.t. } (v, w) \in E\}| = \sum_{v \in V} \deg(v).$$

Si  $G$  est non-dirigé, alors l'union n'est pas disjointe et nous ne pouvons donc pas écrire la même chose. Cependant en sommant les degrés de tous les sommets, chaque arête est comptée deux fois, d'où l'égalité. De manière plus formelle :

$$\begin{aligned} \sum_{v \in V} \deg(v) &= \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E\}| \\ &= \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E, v < w\} \sqcup \{w \in V \text{ s.t. } \{v, w\} \in E, v > w\}| \\ &= \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E, v < w\}| + \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E, v > w\}| \end{aligned}$$

$$\begin{aligned}
&= \left| \bigsqcup_{v \in V} \{w \in V \text{ s.t. } \{v, w\} \in E, v < w\} \right| + \left| \bigsqcup_{v \in V} \{w \in V \text{ s.t. } \{v, w\} \in E, v > w\} \right| \\
&= |E| + |E| = 2m.
\end{aligned}$$

□

**Corollaire 2.7.** Si  $G$  est un graphe non-dirigé, le nombre de sommets de degré impair est pair.

*Démonstration.* Supposons par l'absurde que le sous-ensemble  $U = \{v \in V \text{ s.t. } \deg v = 1 \pmod{2}\}$  est de cardinalité impaire. Nous savons donc que :

$$\sum_{v \in U} \deg v = 1 \pmod{2}.$$

De plus nous savons que :

$$\sum_{v \in V \setminus U} \deg v = 0 \pmod{2}.$$

Dès lors nous déduisons que (par la **Proposition 2.6**) :

$$2|E(G)| = \sum_{v \in V} \deg v = \sum_{v \in U} \deg v + \sum_{v \in V \setminus U} \deg v = 1 + 0 \pmod{2} = 1 \pmod{2},$$

ce qui est une contradiction car  $2|E(G)|$  est pair. Dès lors  $U$  est de cardinalité paire. □

### 2.8.1 Représentation des graphes

Il existe deux manières principales de représenter un graphe : soit de manière statique, soit de manière dynamique. Ici, la dénomination *statique* vs *dynamique* fait référence à la possibilité de redimensionner le graphe ou non. En effet, dans le cas où le graphe est représenté par sa matrice d'adjacence (ou sa matrice d'incidence), si on veut ajouter des sommets dans le graphe, il faut réallouer une matrice et la re-remplir, nous sommes donc dans le cas d'une représentation statique. Par contre si le graphe est représenté par une *liste de voisins* (ou *liste de successeurs* ou *liste d'adjacences*), c'est-à-dire que le graphe est représenté par une liste de listes (attention, listes algorithmique, pas au sens de Python !), alors ajouter un sommet se fait simplement par l'ajout d'un nouveau nœud dans la liste chaînée principale.

**Proposition 2.8.** Considérons un graphe  $G = (V, E)$  fixé avec  $n$  sommets et  $m$  arêtes.

- Représenter  $G$  par sa matrice d'adjacence requiert  $\Theta(n^2)$  en espace.
- Représenter  $G$  par sa matrice d'incidence requiert  $\Theta(nm)$  en espace.
- Représenter  $G$  par une liste d'adjacence requiert  $\Theta(m + n)$  en espace.

### 2.8.2 Matrice d'accessibilité

**Définition 15.** La matrice d'accessibilité du graphe  $G = (V, E)$  où  $V = \{v_1, \dots, v_n\}$  est la matrice booléenne  $M \in \mathbb{B}^{n \times n}$  telle que  $M_{ij} = \top$  si et seulement si il existe un chemin entre  $v_i$  et  $v_j$  dans  $G$ .

**Définition 16.** La *fermeture transitive* d'une matrice booléenne  $A \in \mathbb{B}^{n \times n}$  est la matrice :

$$A^* := \bigvee_{k \geq 0} A^k.$$

**Théorème 2.9** ((plus connu ici sous le nom de *Théorème de Maréchal* pour des raisons historiques propres à ce cours d'algo I)). Soit  $G = (V, E)$  un graphe à  $n$  sommets et soit  $A \in \mathbb{B}^{n \times n}$  sa matrice d'adjacence vue comme matrice booléenne. Si  $M$  est la matrice d'accessibilité du graphe  $G$  et si  $A^*$  désigne la fermeture transitive de  $A$ , alors :

$$M = A^* = \bigvee_{k=0}^{n-1} A^k.$$

Pour des raisons de complétude, nous montrerons ce résultat ici, mais cette preuve est donnée à titre indicatif. Nous aurons besoin des résultats intermédiaires suivants pour prouver le **Théorème 2.9**.

**Proposition 2.10.** Soit  $A \in \mathbb{B}^{n \times n}$  la matrice d'adjacence d'un graphe  $G$  vue comme matrice booléenne.  $A_{ij}^k = \top$  si et seulement si il existe un chemin  $P$  dans  $G$  de longueur  $k$  liant  $v_i$  et  $v_j$  et  $A_{ij}^k = \perp$  sinon.

*Démonstration.* Montrons cela par récurrence sur  $k$ . Comme cas de base, prenons  $k = 0$  : dans ce cas-là,  $A^k = I$ , la matrice identité. Dès lors  $A_{ij}^0 = \top$  ssi  $i = j$  et en effet les seuls chemins de longueur 0 dans un graphe sont les chemins d'un sommet vers lui-même.

Par récurrence, montrons la double implication séparément : supposons qu'il existe un chemin

$$P = (v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_{k-1}}, v_{i_k})$$

de longueur  $k$  dans  $G$  tel que  $i_0 = i$  et  $i_k = j$ . En particulier  $\bar{P} = (v_{i_0}, \dots, v_{i_{k-1}})$  est un chemin dans  $G$  depuis  $v_i$  vers  $v_{i_{k-1}}$  de longueur  $k-1$ . De plus, nous savons que  $A_{i_{k-1}j} = \top$  puisqu'il existe une arête entre  $v_{i_{k-1}}$  et  $v_j$ .

Par hypothèse de récurrence,  $A_{ii_{k-1}}^{k-1} = \top$ . Dès lors :

$$\begin{aligned} A_{ij}^k &= \bigvee_{\ell=1}^n (A_{i\ell}^{k-1} \wedge A_{\ell j}) = \left( \bigvee_{\substack{\ell=1 \\ \ell \neq i_{k-1}}}^n A_{i\ell}^{k-1} \wedge A_{\ell j} \right) \vee (A_{ii_{k-1}}^{k-1} \wedge A_{i_{k-1}j}) \\ &= \left( \bigvee_{\substack{\ell=1 \\ \ell \neq i_{k-1}}}^n A_{i\ell}^{k-1} \wedge A_{\ell j} \right) \vee (\top \wedge \top) \\ &= \top. \end{aligned}$$

Supposons maintenant que  $A_{ij}^k = \top$ . Par l'égalité suivante :

$$A_{ij}^k = \bigvee_{\ell=1}^n (A_{i\ell}^{k-1} \wedge A_{\ell j}),$$

nous savons qu'il existe un  $\ell \in \llbracket 1, n \rrbracket$  tel que  $A_{i\ell}^{k-1} = \top$  et tel que  $A_{\ell j} = \top$ . Par hypothèse de récurrence, il existe un chemin  $\bar{P} = (v_{i_0}, \dots, v_{i_{k-1}})$  dans  $G$ , de longueur  $k-1$  tel que  $i_0 = i$  et tel que  $i_{k-1} = \ell$ . Or, puisque  $A_{\ell j} = \top$ , nous savons que  $(v_\ell, v_j) \in E(G)$ . Nous pouvons en déduire que  $P = (v_{i_0}, \dots, v_{i_{k-1}}, v_j)$  est un chemin de  $G$ , de longueur  $k$  liant  $v_i$  à  $v_j$ .  $\square$

**Lemme 2.11.** Si  $P = (v_{i_0}, \dots, v_{i_k})$  est un chemin de longueur minimale entre  $v_{i_0}$  et  $v_{i_k}$  dans le graphe  $G = (V, E)$ , alors tout  $v_{i_j}$  n'apparaît qu'une unique fois dans le chemin.

*Démonstration.* Par contraposée, si  $P$  est un chemin entre  $v_{i_0}$  et  $v_{i_k}$  tel qu'il existe deux indices distincts  $i_j$  et  $i_\ell$  tels que  $v_{i_j} = v_{i_\ell}$ , nous pouvons supposer sans perte de généralité que  $i_j < i_\ell$ . Dès lors définissons le nouveau chemin  $P'$  comme ceci :

$$P' = (v_{i_0}, \dots, v_{i_j}, v_{i_{j+1}}, \dots, v_{i_k}).$$

La longueur du chemin  $P'$  est égale à celle de  $P$  moins  $\ell - j > 0$  puisque  $j < \ell$ , et donc  $P'$  est de longueur inférieure à celle de  $P$ , ou encore  $P$  n'est pas de longueur minimale.  $\square$

**Corollaire 2.12.** Soit  $G = (V, E)$  un graphe fini. Si  $u, v \in V$  sont deux sommets joignables dans  $G$ , alors en particulier il existe un chemin de longueur strictement inférieure à  $n$  joignant  $u$  et  $v$  dans  $G$ , où  $n = |V|$ .

*Démonstration.* Soient  $u, v$  deux tels sommets joignables et prenons  $P$  un chemin de longueur minimale entre  $u$  et  $v$ . Le **Lemme 2.11** impose que tous les sommets de ce chemin n'apparaissent qu'une unique fois, donc en particulier, tout sommet de  $V$  apparaît au plus une unique fois dans  $P$ . Nous en déduisons que  $P$  a au plus  $n$  sommets et donc au plus  $n - 1$  arêtes, i.e. a longueur au plus  $n - 1$ .  $\square$

*Preuve du théorème de Maréchal (Théorème 2.9).* La **Proposition 2.10** montre bien la première égalité, à savoir  $A^* = M$ , où  $A^*$  est vue comme une disjonction infinie. Le **Corollaire 2.12** montre la seconde égalité, à savoir  $A^* = \bigvee_{k=0}^{n-1} A^k$ . En effet ce corollaire précise qu'il existe un chemin entre deux sommets si et seulement s'il existe un chemin de longueur au plus  $n - 1$ . Puisque par la **Proposition 2.10** garantit que  $A^k$  correspond à l'existence de chemins de longueur exactement  $k$ , nous savons que si  $A_{ij}^N = \top$  pour un certain  $N \geq n$ , alors il doit exister un  $k \leq n - 1$  tel que  $A_{ij}^k = \top$ , ce qui conclut la preuve.  $\square$

**Proposition 2.13.** Calculer la matrice d'accessibilité d'un graphe via la fermeture transitive de sa matrice d'adjacence requiert  $\mathcal{O}(n^4)$  opérations dans le pire des cas.

*Démonstration.* Calculer le produit entre deux matrices carrées  $n \times n$  requiert  $\mathcal{O}(n^3)$  opérations (si fait de manière naïve, c.f. **Exercice 5.6**), et le calcul de la fermeture transitive de la matrice d'adjacence requiert de calculer  $n$  telles matrices et de les sommer, pour un total de  $n \times \mathcal{O}(n^3) = \mathcal{O}(n^4)$ .  $\square$

L'algorithme de Floyd-Warshall (ou Floyd-Roy-Warshall) est un algorithme de programmation dynamique permettant de calculer la distance entre toutes les paires de sommets d'un graphe (qu'il soit dirigé ou non). L'idée de cet algorithme est en fait l'inégalité triangulaire : si  $u, v, w$  sont trois sommets de  $G$ , alors  $d(u, v) \leq d(u, w) + d(w, v)$ . L'algorithme peut s'écrire comme ceci (on fait ici l'hypothèse que les sommets du graphe sont les entiers  $\{0, \dots, n - 1\}$ ) :

---

```

1 def distances_floyd_warshall(G: Graph):
2     D = G.get_adjacency_matrix().copy() # Theta(n^2)
3     D.replace(0, float('inf'))         # Theta(n^2)
4     n = G.get_number_of_vertices()     # O(1)
5     for k in range(n):
6         for i in range(n):
7             for j in range(n):
8                 D[i][j] = min(D[i][j], D[i][k] + D[k][j])
9     return D

```

---

**Proposition 2.14.** *L'algorithme de Floyd-Warshall calcule la matrice de distances de  $G$  en temps  $\Theta(n^3)$ .*

*Démonstration.* L'aspect cubique de l'algorithme est trivial. Il faut tout de même s'assurer qu'il fait bien ce qu'il est censé faire, et que la matrice renvoyée à la fin correspond bien à la matrice de distances.

Pour cela, observons qu'à la fin de chaque tour de la première boucle (celle sur  $k$ ), on regarde la longueur d'un chemin minimal entre chaque paire de sommets, mais ne passant que par les sommets  $\{0, \dots, k\}$ . Après le dernier tour de cette boucle, tous les plus courts chemins entre les paires de sommets pouvant passer par n'importe quel sommet intermédiaire auront été vérifiés.  $\square$

Notez que cet algorithme peut être utilisé pour déterminer la matrice d'accessibilité : il suffit de remplacer les entrées finies par 1 et les entrées infinies par 0. Le problème de la matrice d'accessibilité peut donc être résolu en temps cubique (ce qui est mieux que l'algorithme quartique ci-dessus). Cependant, il y a moyen de faire bien mieux : en identifiant les composantes (fortement) connexes du (di)graphe en temps  $\Theta(n + m)$  et puis en remplissant une nouvelle matrice en temps quadratique. Nous verrons comment identifier les composantes d'un graphe en INFO-F203 — ALgorithmique II.

### 2.8.3 Parcours de graphes

Plein d'algorithmes de graphes vont nécessiter de traiter soit tous les sommets, soit une partie seulement, mais en ayant des garanties sur l'ordre dans lequel ces sommets vont être traités. Les deux algorithmes les plus communs sont le parcours *en profondeur* et le parcours *en largeur*. Le premier est une généralisation des parcours vus sur les arbres (tant préfixe, qu'infixe que postfixe), et le second est une généralisation du parcours par niveau sur les arbres.

Dans les deux cas, nous allons partitionner l'ensemble des sommets en trois catégories : ceux déjà traités ; ceux vus mais pas encore traités ; et finalement ceux qui n'ont pas encore été vus. Cette partition ne sera pas représentée explicitement, mais presque : les sommets déjà traités vont devoir être marqués pour être sûr de ne pas les traiter plusieurs fois et les sommets en attente d'être traités vont être placés dans un conteneur en l'attente de leur traitement. On appelle *frange* l'ensemble des sommets en attente d'être traités.

La forme générale est donc la suivante :

---

```

1 def operation(G: Graph, src: Vertex):
2     container = Container()
3     container.insert(src)
4     marks = [False] * G.get_number_of_vertices()
5     marks[src] = True
6     while not container.empty():
7         v = container.pop()
8         treat(v)
9         for w in G.neighbours_of(v):
10             if not marks[w]:
11                 container.insert(w)

```

---

Si la frange est représentée par une pile, alors le parcours est dit *en profondeur* et si la frange est représentée par une file, alors le parcours est dit *en largeur*.

**Proposition 2.15.** *En supposant que le traitement de chaque sommet se fait en temps  $\mathcal{O}(f(n, m))$ , un parcours de graphe (qu'il soit en largeur ou en profondeur) se fait en  $\mathcal{O}(m + nf(n, m))$  opérations.*

Attention, l'implémentation donnée ci-dessus ne peut explorer qu'une composante connexe, mais ne peut pas atteindre des sommets qui ne sont pas accessibles depuis le sommet `src`. La forme générale d'un parcours de graphe est donc :

---

```

1 def operation(G: Graph):
2     marks = [False] * G.get_number_of_vertices()
3     for v in G.V:
4         if not marks[v]:
5             marks[v] = True
6             _operation(G, v, marks)
7
8 def _operation(G: Graph, v: Vertex, marks: list[bool]):
9     container = Container()
10    container.insert(v)
11    while not container.empty():
12        v = container.pop()
13        treat(v)
14        for w in G.neighbours_of(v):
15            if not marks[w]:
16                container.insert(w)

```

---

## 2.9 Dérécursification

Le principe de la dérécursification est, comme son nom l'indique, de transformer une fonction récursive en une fonction non-récursive. Pour ce faire, il va falloir simuler à la main le fonctionnement de la pile des appels récursifs et des contextes (c.f. la [sous-section 2.2.2](#)).

Malheureusement, aucune procédure automatique de dérécursification n'est connue à part la gestion explicite du stack, qui a tendance à ne pas donner des implémentations opti-

males : il faut encore les nettoyer *a posteriori*.

### 2.9.1 Récursivité unaire

On parle de *récursivité unaire* pour une fonction faisant un unique appel récursif. Si aucun traitement n'est fait après cet appel récursif (et qu'en particulier la valeur retournée n'est pas utilisée), on parle aussi de *fausse récursivité* puisqu'elle peut tout simplement être remplacée par une boucle, sans gestion du stack. Le code suivant montre un canevas général de fausse récursivité et d'une boucle équivalente.

---

```

1 def rec_f(state):
2     if stop(state):
3         treat_stop(state)
4     else:
5         pre_treatment(state)
6         ref_f(transform(state))
7
8 def non_rec_f(state):
9     while not stop(state):
10         pre_treatment(state)
11         state = transform(state)
12     treat_stop(state)

```

---

Si par contre la fonction récursive initiale contient un post-traitement en plus du pré-traitement, alors il faut sauver l'état avant de le transformer pour pouvoir appeler le post-traitement dessus.

---

```

1 def rec_f(state):
2     if stop(state):
3         treat_stop(state)
4     else:
5         pre_treatment(state)
6         ref_f(transform(state))
7         post_treatment(state)
8
9 def non_rec_f(state):
10     S = Stack()
11     while not stop(state):
12         pre_treatment(state)
13         S.push(state.copy())
14         state = transform(state)
15     treat_stop(state)
16     while len(S) > 0:
17         state = S.pop()
18         post_treatment()

```

---

Notez bien qu'il faut faire le pop sur le stack *avant* d'effectuer le post-traitement puisque dans la version non-récursive, (l. 7), le post-traitement est bien effectué sur `state` et non sur `transform(state)`.



**Remarque.** Parfois, si la fonction `transform` est suffisamment simple et peut être inversée (efficacement qui plus est), il est possible de se passer du `stack` puisque parcourir le `stack` revient simplement à appliquer la transformation inverse autant de fois qu'on a fait la transformation initiale.

### 2.9.2 Récursivité binaire

Dans le cas d'une récursivité binaire, la fonction récursive s'appelle deux fois avec deux transformations (potentiellement) différentes de l'état. De plus il y a un pré-traitement (avant le premier appel), un post-traitement (après le deuxième), et un traitement intermédiaire (entre les deux appels).

---

```

1 def rec_f(state):
2     if stop(state):
3         treat_stop(state)
4     else:
5         pre_treatment(state)
6         rec_f(transform_1(state))
7         intermediate_treatment(state)
8         rec_f(transform_2(state))
9         post_treatment(state)
10
11 def non_rec_f(state):
12     S = Stack()
13     S.push((0, None)) # sentinelle
14     while True:
15         while not stop(state):
16             pre_treatment(state)
17             S.push((1, state.copy())) # prépare 1e appel
18             state = transform_1(state)
19         treat_stop(state)
20         i, state = S.pop()
21         while i == 2: # retour d'un traitement intermédiaire
22             post_treatment(state)
23             i, state = S.pop()
24         if i == 0: # ne peut arriver qu'après avoir pop un i=2
25             break
26         intermediate_treatment(state)
27         S.push((2, state.copy())) # prépare le 2e appel
28         state = transform_2(state)

```

---

### 2.9.3 Récursivité $n$ -aire

Ce principe se généralise très bien à un cas où il y aurait  $n$  appels récursifs, avec systématiquement un traitement entre chaque paire d'appels, un pré-traitement avant le premier appel et un post-traitement après le dernier appel.

De manière schématique, nous pouvons supposer qu'il existe un vecteur de fonctions `transform_functions` de taille  $n$  et un vecteur de fonctions `treatment_functions`, également de taille  $n$ , et que la fonction récursive suit le canevas suivant :

---

```

1 def rec_f(state):
2     if stop(state):
3         treat_stop(state)
4     else:
5         pre_treatment(state)
6         for j in range(n):
7             rec_f(transform_functions[j](state))
8             treatment_functions[j](state)

```

---

Une telle fonction peut être dérécursiée de la sorte, toujours en gérant le stack à la main et où le stack contient des paires  $(i, \text{state})$  où  $i$  nous permet de savoir quel appel récursif a été préparé (et donc quelle est la dernière fonction de traitement qui a été appelée) :

---

```

1 def non_rec_f(state):
2     S = Stack()
3     S.push((0, None))
4     while True:
5         while not stop(state):
6             pre_treatment(state)
7             S.push((1, state.copy()))
8             state = transform_functions[0](state)
9         treat_stop()
10        i, state = S.pop()
11        while i == n:
12            treatment_functions[n-1](state)
13            i, state = S.pop()
14        if i == 0: # seulement possible après un nème appel récursif
15            break
16        for j in range(1, n):
17            if i == j:
18                treatment_functions[j-1](state)
19                S.push((j+1, state.copy()))
20                state = transform_functions[j](state)
21            break

```

---

## 2.10 Tris

Dans toute cette section, nous supposons que nous avons un vecteur d'éléments deux à deux comparables de taille  $n$ .

### 2.10.1 Tris naïfs

Nous rappellerons ici les trois algorithmes naïfs de tri que sont :

- le tri bulle ;
- le tri par sélection ;
- le tri par insertion.

**Tri bulle**

L'idée au cœur du tri bulle est la suivante : nous allons itérativement comparer l'élément à l'indice  $i$  avec l'élément à l'indice  $i + 1$ . Si le premier est plus grand que le second, nous échangeons ces deux éléments et puis nous passons à l'élément suivant. Une fois arrivé au bout du vecteur, nous recommençons depuis le départ pour un total de  $n$  passes.

---

```

1 def compare_swap(v, i, j):
2     if v[i] > v[j]:
3         v[i], v[j] = v[j], v[i]
4
5 def bubblesort(v):
6     n = len(v)
7     for i in range(n):
8         for j in range(n-1):
9             compare_swap(v, i, i+1)

```

---

Notons qu'à la fin de la première passe, le plus grand élément est obligatoirement à la fin du vecteur puisqu'à chaque fois qu'il est comparé à un autre élément, il est obligatoirement le plus grand des deux et finit donc à la position le plus à droite. Dès lors la seconde passe ne doit pas aller jusqu'au dernier élément qui ne bougera pas, mais seulement jusqu'à l'élément d'indice  $n - 1$ . À la fin de cette deuxième passe, le deuxième plus grand élément arrivera à l'avant dernière position et donc la troisième passe ne devra aller que jusqu'à l'indice  $n - 2$ , etc. Nous pouvons donc réécrire l'algorithme comme suit :

---

```

1 def bubblesort(v):
2     n = len(v)
3     for i in range(n-1):
4         for j in range(n-1-i):
5             compare_swap(v, i, i+1)

```

---

Dans le pire des cas (si le vecteur est trié dans l'ordre décroissant), chacune des comparaisons va mener à un swap, et il y a  $\sim \frac{n^2}{2}$  comparaisons lors de l'exécution de cet algorithme de tri. En effet il y en a  $n - 1$  lors de la première passe,  $n - 2$  lors de la deuxième, etc. Or :

$$(n - 1) + (n - 2) + \dots + 1 = \sum_{m=1}^{n-1} m = \frac{n(n - 1)}{2} \sim \frac{n^2}{2}.$$

Notons cependant que si, lors d'une passe sur le vecteur, aucun swap n'est effectué, c'est que le vecteur est trié et qu'il n'est pas nécessaire de continuer à itérer. En particulier, si le vecteur  $v$  donné en paramètre est déjà trié, il ne faudra faire qu'une unique passe, ce qui amènerait le temps d'exécution dans le meilleur des cas à  $\sim n$  comparaisons. Il nous faut pour ça adapter la fonction `compare_swap` pour qu'elle renvoie `True` dans le cas où un swap a été effectué et `False` sinon.

---

```

1 def compare_swap(v, i, j):
2     swap = v[i] > v[j]
3     if swap:
4         v[i], v[j] = v[j], v[i]
5     return swap

```

---

```

6
7 def bubblesort(v):
8     n = len(v)
9     for i in range(n-1):
10         swapped = False
11         for j in range(n-1-i):
12             swapped = swapped or compare_swap(v, i, i+1)
13         if not swapped:
14             break

```

---

Il n'est cependant pas possible de réduire le temps d'exécution dans le pire des cas : ce pire cas est en  $\Omega(n^2)$  de manière inhérente. Notons qu'en fait, le nombre moyen de swaps pour bubble sort (en supposant que le vecteur contient  $n$  valeurs distinctes) est  $\frac{1}{2} \binom{n}{2} \sim \frac{n^2}{4}$  (il suffit de regarder, pour une permutation  $\sigma \in \mathfrak{S}_n$  prise uniformément, le nombre moyen d'*inversions*, i.e. de paires  $(i, j)$  telles que  $i < j$  et  $\sigma(i) > \sigma(j)$ ). Dès lors le tri bulle est, par essence, fondamentalement quadratique et n'est en fait jamais utilisé en pratique car beaucoup trop inefficace.

### Tri par sélection

Le tri par sélection peut être vu comme une amélioration du tri bulle dans le sens suivant : au lieu de comparer les éléments deux à deux et de les échanger jusqu'à repousser le plus grand élément à la fin du vecteur (pour donc au plus  $\sim n$  comparaisons et swaps), nous pouvons, plus simplement, trouver le plus grand élément en une passe (pour donc  $\sim n$  comparaisons mais aucun swap), et ensuite faire un unique swap pour placer cet élément à la bonne place.

```

1 def selection_sort(v):
2     n = len(v)
3     for i in range(n-1):
4         max_idx = 0
5         for j in range(1, n-i):
6             if v[j] > v[max_idx]:
7                 max_idx = j
8         v[max_idx], v[n-i-1] = v[n-i-1], v[max_idx]

```

---

Bien que le nombre de comparaisons moyen reste quadratique, l'intérêt du tri par sélection est que le nombre de swaps est maintenant linéaire en la taille du conteneur. En pratique, il est commun de vouloir trier des objets bien plus compliqués que de simples entiers, et donc pour lesquels le coût des comparaisons est bien plus important que le temps des swaps, dès lors le tri par sélection reste très peu intéressant.

### Tri par insertion

Le principe du tri par insertion est de parcourir le vecteur de gauche à droite, et de placer l'élément actuellement visité (à l'indice  $i$ ) à la bonne place tel que le sous-vecteur entre les indices 0 et  $i$  soit trié. Pensez à la manière de trier une main de cartes de la plus petite à la plus grande !

---

```

1 def insertion_sort(v):
2     n = len(v)
3     for i in range(1, n):
4         while i > 0:
5             if v[i] >= v[i-1]:
6                 break
7             else:
8                 v[i-1], v[i] = v[i], v[i-1]
9                 i -= 1

```

---

Tout comme les deux autres tris vus juste avant, dans le pire des cas, le tri par insertion nécessite  $\Theta(n^2)$  comparaisons et swaps. Cependant si le vecteur est déjà trié (ou *presque-trié*, même si cette notion n'est pas bien définie formellement), le tri par insertion a tendance à se comporter assez bien. Cependant, le nombre moyen de comparaisons et de swaps reste quadratique, ce qui en fait un assez mauvais choix d'algorithme de tri générique. Il peut donc avoir ses utilités, mais nécessite une analyse de la situation avant de le choisir.

Remarquons également que dans le cas où nous ne cherchons pas à trier un vecteur mais plutôt une liste (doublement) chaînée, nous pouvons faire descendre le nombre de swaps dans le pire des cas à  $\sim n$  puisque, comme dans le tri par sélection il *suffit* de trouver où le placer, et ensuite d'adapter les pointeurs/références pour donc un unique swap.

Attaquons-nous maintenant aux tris vraiment utilisés en pratique (à savoir les tris qui nécessitent  $\Theta(n \log n)$  comparaisons dans le pire des cas). Nous ne parlerons pas de heapsort qui a déjà été décrit et analysé dans la [sous-section 2.6.4](#).

### 2.10.2 Tri fusion

Le tri fusion (ou *merge sort*) est un tri reposant sur le principe *D & C* (c.f. la [sous-section 2.2.3](#)) qui comporte deux parties (comme son nom le laisse deviner) : la partie de *tri* et la partie de *fusion*. Pour cela, nous créons deux sous-vecteurs `left` et `right` qui correspondent à la première et la seconde moitié, nous appelons récursivement `mergesort` sur ces deux sous-vecteurs, et ensuite nous fusionnons `left` et `right` dans `v` :

---

```

1 def mergesort(v):
2     n = len(v)
3     if n < 2:
4         return
5     mid = n // 2
6     # Partie tri
7     left = v[:mid]
8     right = v[mid:]
9     mergesort(left)
10    mergesort(right)
11    # Partie fusion
12    i = 0
13    j = 0
14    k = 0
15    while i < len(left) and j < len(right):
16        if left[i] < right[j]:

```

---

---

```

17         v[k] = left[i]
18         i += 1
19     else:
20         v[k] = right[j]
21         j += 1
22     k += 1
23     while i < len(left):
24         v[k] = left[i]
25         i += 1
26         k += 1
27     while j < len(right):
28         v[k] = right[j]
29         j += 1
30         k += 1

```

---

Notons que la première boucle (l. 15-22) parcourt simultanément les deux sous-vecteurs triés, à choisir à chaque étape l'élément minimum entre les deux. Ce faisant, il est garanti que nous savons toujours où se situe le plus petit élément stocké dans les deux sous-vecteurs. Il faut tout de même faire attention : une fois que cette boucle est effectuée, nous savons uniquement que nous avons épuisé toutes les entrées d'un de ces sous-vecteurs, mais nous n'avons aucune idée d'où nous en sommes dans le deuxième. Pour cela, nous avons les boucles (l. 23-30) pour copier les entrées oubliées. Même s'il y a deux boucles (une pour `left` et une pour `right`), une seule sera exécutée puisque soit `i == len(left)`, soit `j == len(right)` en sortant de la première boucle.

Afin d'éviter des réallocations à tout va lors de l'exécution de mergesort, nous pouvons allouer une unique fois un vecteur de la même taille que `v` et travailler sur ce vecteur auxiliaire :

---

```

1 def mergesort(v, beg=0, end=None, aux=None):
2     if end is None:
3         end = len(v)
4     if aux is None:
5         aux = [None] * len(v)
6     if end-beg < 2:
7         return
8     mid = (beg+end) // 2
9     copy_to(v, beg, end, aux)
10    mergesort(aux, beg, mid, v)
11    mergesort(aux, mid, end, v)
12    i = k = beg
13    j = mid
14    while i < mid and j < end:
15        if aux[i] < aux[j]:
16            v[k] = aux[i]
17            i += 1
18        else:
19            v[k] = aux[j]
20            j += 1
21        k += 1

```

---

```

22     while i < mid:
23         v[k] = aux[i]
24         i += 1
25         k += 1
26     while j < end:
27         v[k] = aux[j]
28         j += 1
29         k += 1

```

---

où la fonction `copy_to` sert uniquement à copier les entrées de `aux` dans `v` entre les indices donnés :

---

```

1 def copy_to(v, i, j, w):
2     for idx in range(i, j):
3         w[idx] = v[idx]

```

---

Nous pouvons assez facilement voir que le tri fusion requiert au plus  $\mathcal{O}(n \log n)$  opérations pour trier un vecteur de taille  $n$ . Si  $T(n)$  désigne le nombre d'opérations dans le pire des cas pour trier un vecteur de taille  $n$ ,  $T(n)$  doit satisfaire l'inégalité suivante :

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n).$$

Par simplicité, prenons le cas où  $n = 2^N$  pour un certain  $N$  entier (au pire si ce n'est pas le cas, nous pouvons *padder* de tableau). Dès lors :

$$\begin{aligned}
 T(n) = T(2^N) &\leq 2T(2^{N-1}) + \Theta(2^N) \\
 &\leq 4T(2^{N-2}) + \Theta(2^N) + \Theta(2 \cdot 2^{N-1}) = 2^2 T(2^{N-2}) + \Theta(2^N) + \Theta(2^N) \\
 &\leq \dots \leq 2^k T(2^{N-k}) + \sum_{j=1}^k \Theta(2^N)
 \end{aligned}$$

pour  $k$  entier tel que  $0 \leq k \leq N$ . En particulier pour  $k = N$  :

$$T(n) \leq 2^N T(1) + \sum_{j=1}^N \Theta(2^N) = 2^N T(1) + \Theta(N 2^N),$$

or  $T(1) = \Theta(1)$  est une constante. Comme  $N = \log_2 n$ , nous obtenons :

$$T(n) \leq \Theta(2^{\log_2 n}) + \Theta(\log_2 n 2^{\log_2 n}) = \Theta(n) + \Theta(n \log_2 n) = \Theta(n \log n).$$

Notons qu'en fait mergesort peut être de manière itérative et donc sans avoir besoin d'appels récursifs ! Nous verrons l'approche bottom-up (en opposition à l'approche top-down vue ici) en INFO-F203 — Algorithmique II.

### 2.10.3 Tri rapide

Le tri rapide (ou *quicksort* en anglais) peut être vu comme un cousin du tri fusion mais qui se fait en place (i.e. qui opère directement sur les données reçues, sans avoir besoin d'un

vecteur additionnel).<sup>11</sup> L'idée derrière quicksort est de déterminer un *pivot* (un élément du vecteur à trier), de placer tous les éléments inférieurs au pivot à sa gauche et tous les éléments plus grands que le pivot à sa droite, et puis de trier récursivement les parties gauches et droite du pivot. En procédant de la sorte, à chaque étape, nous pouvons garantir que le pivot est à la bonne place et nous ne devons donc plus nous en occuper. Cette étape s'appelle le *partitionnement*.

---

```

1 def quicksort(v, beg=0, end=None):
2     if end is None:
3         end = len(v)
4     if end-beg < 2:
5         return
6     idx = partition(v, beg, end)
7     quicksort(v, beg, idx)
8     quicksort(v, idx+1, end)

```

---

Il existe plein de manières d'écrire le partitionnement de quicksort, nous nous intéressons ici à la version de Robert Sedgewick (qui a fait sa thèse précisément sur ce sujet en 1975 sous la supervision de Donald Knuth – oui oui, rien que ça –, une gigantesque brique de 360 pages disponible en ligne dans une qualité visuelle discutable). L'idée ici est de fixer le pivot arbitrairement (ici le premier élément du sous-vecteur à trier)<sup>12</sup> et puis de parcourir simultanément ce sous-vecteur de gauche à droite *et* de droite à gauche afin de trouver les paires d'indices  $(i, j)$  qui doivent être échangées (i.e. telles que  $v[i] > \text{pivot} \geq v[j]$ ), de les échanger, de poursuivre jusqu'à trouver où placer le pivot, et le mettre au bon endroit. Il faut bien sûr que la fonction *partition* renvoie l'indice où a été placé le pivot afin de pouvoir savoir sur quelles plages d'indices les appels récursifs suivants doivent s'exécuter.

---

```

1 def partition(v, beg, end):
2     pivot = v[beg]
3     i = beg+1
4     j = end-1
5     while True:
6         while i <= j and v[i] <= pivot:
7             i += 1
8         while i <= j and v[j] > pivot:
9             j -= 1
10        if i >= j:
11            break
12        swap(v, i, j)
13        i, j = i+1, j-1
14    swap(v, beg, j)
15    return j

```

---

11. Plus précisément, on peut implémenter quicksort de manière à nécessiter un espace au plus logarithmique en la taille du conteneur (donc strictement sublinéaire puisque  $\log n = o(n)$ , c.f. [Proposition 1.21](#)), mais nous en reparlerons dans l'analyse de complexité plus loin.

12. En pratique aujourd'hui, le choix du pivot est randomisé pour garantir un temps  $\mathcal{O}(n \log n)$  en moyenne, mais nous verrons cela en INFO-F203 — Algorithmique II.



### Analyse de complexité

La fonction `partition` prend très clairement un temps linéaire en la taille du sous-vecteur à partitionner. Nous pouvons nous convaincre que si l'appel à `partition` place le pivot au bon endroit et que le sous-vecteur de gauche a taille  $m$  et celui de droite  $n - 1 - m$ , alors pour  $T(n)$  le nombre de comparaisons nécessaire dans le pire des cas satisfait l'inégalité :

$$T(n) \leq T(m) + T(n - 1 - m) + \Theta(n).$$

Dans le cas où  $m = \frac{n}{2}$ , nous retombons sur l'inégalité vue dans l'étude du tri fusion, et nous savons que  $T(n) = \mathcal{O}(n \log n)$ .

Cependant, ce n'est pas vrai et en réalité quicksort nécessite  $\Theta(n^2)$  comparaisons dans le pire des cas. Cependant, en supposant que les tableaux à trier sont dans un ordre aléatoire, nous pouvons nous convaincre qu'en réalité, en moyenne le pivot sera placé au milieu du sous-vecteur partitionné, et qu'on ne devrait pas être trop loin du  $n \log n$  en moyenne.

De fait, quicksort nécessite  $\mathcal{O}(n \log n)$  comparaisons en moyenne, mais l'analyse plus précise sera faite en INFO-F203 — Algorithmique II.

Cependant il est très simple de créer un exemple qui force quicksort à être quadratique : si le vecteur  $v$  est déjà trié, puisque nous prenons le pivot à la première position, nous savons qu'il n'y aura aucun swap à effectuer, dès lors à chaque fois  $m$  sera égal à 0. Dans ce cas, le nombre de comparaisons est donc :

$$T(n) = T(n - 1) + T(0) + \Theta(n),$$

or  $T(0) = 0$ , dès lors :

$$T(n) = T(n - 1) + \Theta(n) = T(n - 2) + \Theta(n) + \Theta(n - 1) = \dots = T(1) + \sum_{k=1}^n \Theta(k) = \Theta(n^2).$$

On pourrait alors se dire que prendre le pivot au milieu est une bonne approche, mais pas du tout. En réalité, peu importe l'indice utilisé pour trouver le pivot, il est toujours possible de créer un vecteur qui nécessitera un nombre quadratique de comparaisons.

### Complexité en espace (📌)

Tel qu'écrit ici, l'algorithme nécessite un espace linéaire supplémentaire dans le pire des cas. Certes, il n'y a jamais aucune allocation de sous-vecteur, tout se fait en place, mais pourtant il y a des appels récursifs. Et comme vu dans la [sous-section 2.2.2](#), chaque appel récursif a besoin de son propre contexte sur le stack d'exécution. Et sur un vecteur déjà trié, puisqu'à chaque fois la taille du sous-vecteur à trier n'est diminuée que de 1, il y aura  $\sim n$  appels les uns sur les autres sur le stack des appels récursifs. Hors chacun de ces appels a besoin de  $\Theta(1)$  en espace, dès lors, cet algorithme a besoin de  $\Theta(n)$  en espace dans le pire des cas. Pourtant je vous ai promis un espace sublinéaire au début de la section. Pour cela, il ne faut pas faire plus d'algorithmique mais faire confiance à votre compilateur préféré (oui ça risque d'être plus technique dans un langage interprété, mais de toute façon, quand on parle efficacité, on ne parle pas de Python...) En effet, nous pouvons écrire la fonction quicksort de manière à systématiquement faire d'abord l'appel récursif sur le plus petit des deux sous-vecteurs, et ensuite seulement en faisant le grand :

---

```

1 def quicksort(v, beg, end):
2     if end-beg < 2: return
3     idx = partition(v, beg, end)
4     if idx-beg < end-(idx+1):
5         quicksort(v, beg, idx)
6         quicksort(v, idx+1, end)
7     else:
8         quicksort(v, idx+1, end)
9         quicksort(v, beg, idx)

```

---

Ici, nous n'avons rien changé à la profondeur de l'arbre des appels récursifs, mais si le compilateur fait bien son travail, il va réserver un traitement spécial au deuxième appel récursif. En effet, il n'y a rien après le deuxième appel récursif. Dès lors, le contexte ne sert plus à rien. De là, le compilateur peut s'arranger pour ne pas créer un nouveau contexte, mais tout simplement modifier le contexte de l'appel en cours pour sauver un appel sur le stack. Dès lors nous savons qu'à chaque fois qu'un nouveau contexte est créé, et que donc l'appel récursif n'est pas écrasé (le terme technique est *tail-call elimination*), la taille du sous-vecteur à trier est au plus la moitié de ce qu'il fallait trier à l'étape d'avant. Il ne peut dès lors y avoir qu'au plus  $\sim \log_2 n$  appels sur le stack et donc l'espace nécessaire est réduit à  $\Theta(\log n) \times \Theta(1) = \Theta(\log n)$ .

#### 2.10.4 Tri Shell

Le tri Shell (qui tient son nom de son inventeur : Donald Shell, rien à voir avec une coquille) a un statut particulier du fait qu'il est en réalité paramétrisé par une *gap sequence*, et que le nombre de comparaisons à effectuer dans le pire des cas dépend fortement de cette séquence. Dès lors l'analyse de cet algorithme (que ce soit en moyenne, dans le pire des cas ou dans le meilleur des cas) n'est valide que pour une séquence donnée (ou pour une famille de séquences ayant certaines propriétés). À l'heure actuelle, aucune telle séquence permettant d'atteindre  $\mathcal{O}(n \log n)$  comparaisons dans le pire des cas n'est connue ; la meilleure borne supérieure connue est  $\mathcal{O}(n \log^2 n)$ , mais ce tri est tout de même intéressant.

Le tri Shell est en fait une généralisation du tri par insertion. En effet, l'idée fondamentale est d'effectuer plusieurs tris sur des sous-vecteurs, mais non-consécutifs ! L'objectif est plutôt de trier les sous-vecteurs définis par les indices sous la forme  $i \cdot \text{gap} + k$  pour  $i$  qui varie et à  $\text{gap}$  fixé, et pour toutes les valeurs de  $k < \text{gap}$ . Après avoir fait ça une première fois, le vecteur est déjà *plus ordonné* qu'avant (bien que ça ne soit pas très formel). Maintenant il faut réitérer sur différentes valeurs de  $\text{gap}$ , en les prenant de plus en plus petites.

---

```

1 def shellsort(v, gaps):
2     n = len(v)
3     for gap in gaps:
4         for k in range(gap):
5             idx = gap + k
6             while idx < n:
7                 j = idx
8                 while j >= gap and v[j] < v[j-gap]:

```

---

```

9         swap(v, j, j-gap)
10        j -= 1
11        idx += gap

```

---

Notons que pour peu que la valeur 1 apparaisse comme dernier gap, nous savons que le vecteur sera trié : en effet puisque shellsort effectue des tris par insertion sur des sous-vecteurs dont les indices sont espacés de la valeur gap, lorsque  $\text{gap} == 1$ , il se réduit à un tri par insertion, ce qu'il y a de plus classique. Maintenant, si on fait de toute façon un tri par insertion (et on en fait même plusieurs), comment peut-on faire mieux que  $\Theta(n^2)$  dans le pire des cas alors que le tri par insertion est déjà quadratique ? Pour répondre à cette question, il faut se rappeler que sur un tableau (presque-)trié, le tri par insertion est (presque-)linéaire. Dès lors, si on s'arrange pour faire des tris par insertion sur des *petits* sous-vecteurs pour commencer, on peut se rapprocher d'un vecteur déjà trié afin que les tris suivants ne soient pas quadratiques.

**Proposition 2.16.** *Si la séquence de gaps est  $(\frac{n}{2}, \frac{n}{4}, \dots, 2, 1)$ , alors le tri Shell est quadratique dans le pire des cas sur un vecteur de taille  $n$ .*

*Démonstration.* Le tri Shell fait itérativement des tris par insertions, et nous savons qu'un tri par insertion nécessite  $\mathcal{O}(m^2)$  comparaisons dans le pire des cas. Supposons par simplicité que  $n = 2^N$  pour un  $N$  entier. Si la séquence de gaps est définie par  $g_k = n/2^k$ , alors à l'étape  $k$ , il y a exactement  $g_k$  sous-vecteurs à trier par insertion, tous de taille  $n/g_k = 2^k$ . Dès lors le nombre de comparaisons le tri Shell est borné par :

$$\sum_{k=1}^N g_k \mathcal{O}\left(\left(\frac{n}{g_k}\right)^2\right) = \mathcal{O}\left(\sum_{k=1}^N \frac{n}{2^k} 2^k 2^k\right) = \mathcal{O}\left(n \sum_{k=1}^N 2^k\right) = \mathcal{O}(n 2^{N+1}) = \mathcal{O}(n^2).$$

Maintenant que nous savons que le tri Shell est *au plus* quadratique dans le pire des cas, construisons une famille d'exemples qui atteint cette borne quadratique. Pour  $n = 2^N$  avec  $N$  entier, considérons les valeurs  $\{1, \dots, 2^N\}$  et créons le vecteur  $v^{(n)} = (v_1^{(n)}, \dots, v_n^{(n)})$  comme ceci (pour  $1 \leq i \leq n$ ) :

$$v_i = \begin{cases} k+1 & \text{si } i = 2k+1, \\ \frac{n}{2} + k & \text{si } i = 2k. \end{cases}$$

Ce vecteur  $v$  contient les valeurs  $\{1, 2, \dots, 2^{N-1}\}$  dans l'ordre croissant aux indices impairs et les valeurs  $\{2^{N-1}+1, 2^{N-1}+2, \dots, 2^N\}$  dans l'ordre aux indices pairs. Dès lors, pour les étapes  $1 \leq k < N$ , les sous-vecteurs considérés dans le tri Shell seront déjà triés, et ils nécessiteront au  $\sim n/g_k$  comparaisons pour un total de :

$$\sum_{k=1}^{N-1} g_k \frac{n}{g_k} = Nn = n \log_2 n$$

comparaisons. Et finalement, la dernière passe sera quadratique puisque la valeur à l'indice  $2k+1$  vont devoir *descendre* de  $k$  positions. Le nombre total de comparaisons est donc de l'ordre de :

$$\sim n \log_2 n + \sum_{k=1}^{n/2} k \sim n \log_2 n + \frac{1}{8} n^2 = \Omega(n^2).$$

Nous en déduisons que le tri Shell est quadratique dans le pire des cas sur la *gap sequence*  $g_k = \lfloor \frac{n}{2^k} \rfloor$ .  $\square$

**Proposition 2.17.** *Avec la séquence de gaps définie par les nombres sous la forme  $2^p 3^q$  pour  $p$  et  $q$  entiers, le tri Shell nécessite  $\mathcal{O}(n \log^2 n)$  comparaisons.*

Cette séquence de gaps est actuellement celle qui propose le meilleur comportement asymptotique, mais en pratique, il existe des séquences de gaps qui se comportent très bien (au moins sur des vecteurs de taille *raisonnable*). C'est un sujet de recherche encore très actif, et une des meilleures séquences trouvée actuellement date de 2023.

**Remarque.** *Nous savons que tout tri par comparaison nécessite au moins  $\sim n \log n$  comparaisons dans le pire des cas. Attention, cela ne veut pas dire que tous les tris peuvent atteindre cette borne (e.g. le tri par sélection qui est quadratique même dans le meilleur des cas). En particulier, cette borne inférieure s'applique au tri Shell. Cependant, en 1993, Robert Cypher a démontré que cette borne  $n \log n$  ne pouvait pas être atteinte avec une séquence de gaps. Plus précisément, il a montré que le tri Shell nécessite  $\Omega\left(n \log n \times \frac{\log n}{\log \log n}\right)$  comparaisons dans le pire des cas. Même si la fonction  $n \mapsto \frac{\log n}{\log \log n}$  croît très lentement, d'un point de vue théorique, ce résultat place le tri Shell dans la catégorie des tris non-optimaux asymptotiquement.*

## 2.11 Programmation dynamique

L'approche *Divide and Conquer* (c.f. la [sous-section 2.2.3](#)) est une manière de résoudre un problème en le découpant en sous-problèmes *disjoints*. Il n'est cependant pas toujours possible (ou intuitif) de trouver une telle séparation. Dans ce cas, on peut essayer de chercher une séparation qui ne soit pas disjointe, par exemple via une définition récursive.

Par exemple déterminer le  $n$ ème nombre de Fibonacci peut aisément se formuler de cette manière par la relation de récurrence qui définit cette suite :

$$F_n = \begin{cases} n & \text{si } n \leq 1 \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Cette définition peut donner à première vue l'impression d'être une séparation en sous-problèmes disjoints : pour calculer le  $n$ ème nombre, je dois calculer le  $(n-1)$ ème et le  $(n-2)$ ème, mais en réalité il y a énormément de redondance. En effet, pour calculer le  $(n-1)$ ème nombre, il faudra également calculer le  $(n-2)$ ème nombre...

L'idée fondamentale derrière la programmation dynamique est de pouvoir trouver une séparation en sous-problèmes (potentiellement un relativement grand nombre de tels sous-problèmes) de manière à pouvoir reconstituer la solution (ou plutôt *une solution*) au problème initial depuis les solutions des sous-problèmes. Il faut également s'assurer que chaque sous-problème n'est résolu qu'au plus une fois (en particulier on ne recalcule *jamaïs* quelque chose qui a déjà été calculé).

Pour cela, il existe deux visions différentes de la DP : l'approche *top-down* et l'approche *bottom-up*. La première revient à partir du problème initial et d'appeler récursivement les sous-problèmes en effectuant une sauvegarde des solutions aux sous-problèmes qui ont déjà été résolus. De cette manière, dès qu'un sous-problème est attaqué, il faut d'abord regarder s'il a déjà été résolu (en quel cas, on peut renvoyer la solution en  $\mathcal{O}(1)$ ), soit

le résoudre, et ensuite stocker le résultat pour les fois suivantes. Pour Fibonacci, nous pourrions par exemple l'écrire comme ceci :

---

```

1 def F(n):
2     Fns = [None] * (n+1)
3     return _F(n, Fns)
4
5 def _F(n, Fns):
6     if Fns[n] is not None:
7         return Fns[n]
8     if n <= 1:
9         Fn = n
10    else:
11        Fn = _F(n-1, Fns) + _F(n-2, Fns)
12    Fns[n] = Fn
13    return Fn

```

---

Nous pouvons aisément nous convaincre que le  $n^{\text{ème}}$  nombre de Fibonacci est ici trouvé en temps  $\Theta(n)$ . Il y a tout de même un aspect un peu embêtant ici, duquel on aimerait bien pouvoir se débarrasser : l'implémentation reste fondamentalement récursive, et il y a donc un certain prix à payer, tant en temps qu'en espace, pour la gestion des contextes.

Il y a moyen de remédier à ce problème en implémentant la solution de manière itérative (mais sans procéder à une *simple* dérécursification du problème, c.f. la [section 2.9](#)). En effet, si nous connaissons l'ordre dans lequel les sous-problèmes vont être résolus dans l'implémentation *top-down*, nous pouvons, de manière itérative, résoudre les sous-problèmes dans ce même ordre et remplir notre table de sauvegardes au fur et à mesure, en utilisant les potentielles valeurs trouvées précédemment. Cette approche est appelée *bottom-up* puisqu'on part du *bas* (des sous-problèmes les plus petits possible) et qu'on remonte vers le *gros* problème initial.

Par exemple, dans l'exemple des nombres de Fibonacci, les nombres sont calculés dans l'ordre : en effet puisque  $F_n$  a besoin de connaître  $F_{n-1}$  et  $F_{n-2}$ , nous savons que ces deux sous-problèmes doivent être résolus avant de trouver la solution pour  $F_n$ . Dès lors, nous pouvons dérécursifier notre fonction  $F$  comme suit :

---

```

1 def F(n):
2     values = [None] * (n+1)
3     values[0] = 0
4     values[1] = 1
5     for k in range(2, n+1):
6         values[k] = values[k-1] + values[k-2]
7     return values[n]

```

---

De plus, une fois l'implémentation *bottom-up* réalisée, il est bien souvent possible de l'améliorer en ne conservant que ce qui est fondamentalement nécessaire. Par exemple, toujours pour Fibonacci, l'entrée  $i$  du tableau n'est utilisée que pour calculer les entrées  $i+1$  et  $i+2$ , et est totalement ignorée après. En particulier, nous pouvons déterminer que seules 3 valeurs sont nécessaires à chaque instant : la dernière valeur calculée, l'avant dernière valeur calculée, et la valeur actuelle. Nous pouvons alors nous passer de la table et ne demander

qu'un espace  $\mathcal{O}(1)$  :

---

```
1 def F(n):
2     if n <= 1:
3         return n
4     F_nm2 = 0
5     F_nm1 = 1
6     for k in range(2, n+1):
7         F_n = F_nm1 + F_nm1
8         F_nm2, F_nm1 = F_nm1, F_n
9     return F_n
```

---

De manière générale, pour résoudre un problème avec de la programmation dynamique, il vous faut les étapes suivantes :

1. trouver une définition récursive du problème (potentiellement en ajoutant des paramètres) ;
2. déterminer une structure dans laquelle stocker les résultats intermédiaires (presque toujours un tableau à  $m$  dimensions) ;
3. déterminer un ordre de parcours de cette table tel que tous la solution à chacun des sous-problèmes nécessaires soit disponible à chaque instant (dans le cas d'une implémentation *bottom-up*).

Cette dernière étape n'est pas *nécessaire* pour parler de programmation dynamique mais peut s'avérer assez importante pour les performances (surtout dans un langage interprété comme Python qui a un *overhead* assez important sur plus ou moins tout et n'importe quoi, en particulier la récursivité).

## **Chapitre 3**

### **Séances de TP**

## Séance 1 — Analyse de complexité

**Exercice 1.1.** Exprimez et justifiez la complexité des fonctions suivantes :

```
1 def f1(vector: list[int]) -> int:
2     ret = 0
3     for i in range(len(vector)):
4         ret += vector[i]
5     return ret
6
7 def f2(vector: list[int]) -> int:
8     ret = 0
9     for e in reversed(vector):
10         ret += e
11     return ret
12
13 def f3(n: int) -> int:
14     ret = 0
15     for i in range(n):
16         ret += i
17     return ret
```

**Exercice 1.2.** Étudiez les fonctions suivantes. Quelles sont les complexités dans le meilleur des cas, en moyenne ainsi que dans le pire cas ? Ces fonctions effectuent-elles bien le traitement attendu ?

```
1 def belongs_to1(vector: list[int], x: int) -> bool:
2     for e in vector:
3         if e == x:
4             return True
5     return False
6
7 def belongs_to2(vector: list[int], x: int) -> bool:
8     for e in vector:
9         return e == x
10
11 def belongs_to3(vector: list[int], x: int) -> bool:
12     i = 0
13     ret = False
14     while i < len(vector):
15         if vector[i] == x:
16             ret = True
17         i += 1
18     return ret
```

**Exercice 1.3.** Analysez la complexité des fonctions suivantes :

```
1 def g1(n: int) -> int:
```



```

2     ret = 0
3     while n > 0:
4         n //= 2
5         ret += 1
6     return ret
7
8 def g2(n: int) -> int:
9     ret = 0
10    k = 1
11    while k <= n:
12        for j in range(k):
13            ret += j
14        k *= 2
15    return ret

```

**Exercice 1.4.** Complexifions un peu l'[Exercice 1.2](#) en remplaçant les entiers par des chaînes de caractères, toutes de longueur  $m$ . Quelle est la complexité de la fonction suivante ?

```

1 def belongs_to(vector: list[str], x: str) -> bool:
2     for s in vector:
3         if x == s:
4             return True
5     return False

```

**Exercice 1.5.** Comme vu dans l'exercice précédent, analyser la complexité d'un algorithme basé sur des comparaisons est compliqué car il faut analyser *simultanément* l'algorithme en tant que tel et ses interactions avec la complexité des tris. Pour cette raison, lors de l'analyse d'un algorithme de tri<sup>a</sup>, il est commun de parler du nombre de comparaisons et pas du nombre d'opérations effectuées.

Analysez le nombre de comparaisons nécessaires dans le meilleur cas et dans le pire cas des deux implémentations de l'algorithme du *tri à bulle* ci-dessous. Trouvez également, pour chaque valeur de  $n$ , la longueur du vecteur, un exemple de vecteur atteignant ce nombre de comparaisons.

**Bonus :** déterminez également le nombre de *swaps* dans les deux cas.

```

1 def bubblesort(vector: list[int]) -> None:
2     n = len(vector)
3     for _ in range(n):
4         for i in range(1, n):
5             if vector[i-1] > vector[i]:
6                 vector[i-1], vector[i] = vector[i], vector[i-1]
7
8 def bubblesort(vector: list[int]) -> None:
9     n = len(vector)
10    is_sorted = False
11    while not is_sorted:

```

```

12     is_sorted = True
13     for i in range(1, n):
14         if vector[i-1] > vector[i]:
15             vector[i-1], vector[i] = vector[i], vector[i-1]
16             is_sorted = False

```

a. Uniquement lorsqu'il est question d'un algorithme de tri *par comparaisons* bien entendu.

**Exercice 1.6.** Le code suivant est une implémentation de l'algorithme appelé *crible d'Ératosthène* et permet de déterminer tous les nombres premiers inférieurs à une certaine quantité  $N$ . Son fonctionnement est le suivant : partons de l'ensemble de tous les nombres entiers entre 2 et  $N$  (compris). Ensuite, tant que cet ensemble n'est pas vide, sortons-en le plus petit et d'abord (i) ajoutons-le à l'ensemble des nombres premiers ; ensuite (ii) retirons de l'ensemble de départ tous les multiples de ce nombre. Cela fonctionne bien puisqu'un nombre est premier lorsque ses uniques diviseurs entiers sont 1 et lui-même, donc à chaque étape, lorsque nous retirons le plus petit nombre de l'ensemble, nous savons que puisqu'il n'a pas été retiré au préalable, il n'a pas d'autre diviseur que 1 ou lui-même. De plus, tous les nombres que nous retirons de l'ensemble sont composites puisqu'il sont multiples d'un nombre strictement inférieur à eux.

En voici une implémentation en Python (l'utilisation d'un vecteur pour marquer les nombres est plus efficace que de passer par des ensembles, et de plus nous n'avons pas encore vu comment implémenter un ensemble via un ADT, ce qui se fait habituellement soit à l'aide d'un arbre binaire de recherche équilibré, soit à l'aide d'une table de hachage). Quelle est la complexité de cette fonction ?

```

1 def eratosthenes(N):
2     is_prime = [True] * (N+1)
3     primes = []
4     for p in range(2, N+1):
5         if not is_prime[p]:
6             continue
7         primes.append(p)
8         k = 2
9         while k*p <= N:
10             is_prime[k*p] = False
11             k += 1
12     return primes

```

Notons que  $k$  pourrait être initialisé à  $p$  plutôt que 2 (l. 8) puisque  $kp$  (pour  $2 \leq k < p$ ) aura déjà été marqué lors des tours de boucle précédents. Nous garderons cependant cette implémentation ici pour notre analyse.

## Exercices supplémentaires

**Exercice 1.7** (Inspiré du problème 1.1 du CLRS). Considérons que nous exécutons un algorithme qui nécessite  $f(n)$  opérations pour un input de taille  $n$ , et que la machine sur laquelle nous exécutons cet algorithme effectue  $10^6$  opérations par seconde (peu importe l'opération considérée). Pour les différentes fonctions  $f$  données ci-dessous, déterminez la plus grande valeur  $n$  telle que le programme peut s'exécuter en 1 seconde, 1 heure et 1 siècle :

- $f(n) = \log_2 n$  ;
- $f(n) = n$  ;
- $f(n) = n \log_2 n$  ;
- $f(n) = n^2$  ;
- $f(n) = 2^n$  ;
- $f(n) = n!$ .

## Séance 2 — Les ADT (partie 1)

**Exercice 2.1.** Implémentez, sous forme de type de données abstrait, une classe représentant des nombres complexes ainsi que les opérations d'addition et de multiplication.

**Indice :** utilisez les propriétés (`@property`).

**Exercice 2.2.** Implémentez, sous forme de type de données abstrait, une classe représentant une matrice de nombres complexes, et qui propose une méthode réalisant la multiplication de la matrice par un scalaire complexe.

**Exercice 2.3.** Adaptez les classes `Node` et `UnorderedList` en utilisant des propriétés. Ajoutez-y les méthodes suivantes :

- dans la classe `Node`, une propriété `previous_data` qui permet d'accéder à la donnée du nœud précédent;
- dans la classe `UnorderedList`, une propriété `last` qui permet d'accéder à la dernière donnée de la liste.

Ajoutez-y le nécessaire pour que la classe `UnorderedList` puisse être parcourue à l'aide d'une boucle `for`, i.e. :

---

```

1 l = UnorderedList() # on crée une liste doublement chaînée
2 for i in range(5): # on ajoute les éléments de 0 à 4
3     l.add(i)
4 for element in l: # doit itérer sur : 4, 3, 2, 1, 0
5     print(element) # on affiche simplement

```

---

**Bonus :** rendez compatible la classe `UnorderedList` avec la fonction `reversed` de Python, i.e. :

---

```

1 for element in reversed(l): # doit itérer sur 0, 1, 2, 3, 4
2     print(element)

```

---

**Exercice 2.4.** Soit une chaîne de caractères lue, caractère par caractère, sur l'input. Cette chaîne se termine par un `#` et contient un `*`. Écrivez un programme qui vérifie, au fur et à mesure de la lecture de la chaîne de caractères, si les caractères se trouvant avant le `*` constituent l'image miroir de ceux se trouvant entre le `*` et le `#`. Par exemple : `AB*BA#` respecte cette règle. Par contre, `ABA*BA#` ne la respecte pas.

## Séance 3 — Les ADT (partie 2)

**Exercice 3.1.** Soit une chaîne de caractères lue, caractère par caractère, sur l'input. Cette chaîne se termine par un # et est composée de parenthèses de différents types : (, ), [, ], {, }. Écrivez un programme qui vérifie, au fur et à mesure de la lecture de la chaîne de caractères, si l'expression lue est correctement parenthésée, c'est-à-dire si :

- pour chaque type, le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes ;
- à chaque fois que l'on rencontre une parenthèse fermante d'un certain type, la dernière parenthèse ouvrante non encore fermée est du même type.

**Exercice 3.2.** Soit une suite de puissances de 2 lue sur l'input, nombre par nombre, se terminant par -1, dans laquelle chaque nombre n'apparaît qu'une seule fois. Trouvez la sous-suite partielle ordonnée strictement décroissante et non nécessairement contiguë de somme maximale. Par exemple : pour 128 8 4 16 64 512 2 32 -1 nous obtiendrions comme résultat 512 32.

**Exercice 3.3.** Écrivez une fonction recevant une pile et qui inverse, en place, la pile à l'aide d'une deuxième pile de travail.

## Exercices supplémentaires

**Exercice 3.4.** Depuis **Python 3.5**, un opérateur dédié à la multiplication matricielle a été introduit : l'opérateur `@` (avec un `at` comme dans `matrices`) qui se définit par la méthode spéciale `__matmul__` et a ses alternatives `__imatmul__` et `__rmatmul__`. Complétez la classe `MatriceComplexe` de l'**Exercice 2.2** afin de supporter la multiplication à l'aide de cet opérateur.

**Exercice 3.5.** Une `list` en Python (tout comme un `std::vector` en C++) est un conteneur dynamique (i.e. de taille variable) ayant pour particularité que l'insertion d'un élément dans une liste de taille  $n$  se fait en  $\Theta(n)$  opérations dans le pire des cas, mais  $m$  insertions successives dans une liste de taille initiale  $n$  se fait en  $\Theta(n + m)$  dans le pire des cas (et non  $\Theta(m(m + n))$  comme l'approche triviale).

Écrivez une classe `List` utilisable comme une `list` et satisfaisant les mêmes contraintes (c.f. <https://wiki.python.org/moin/TimeComplexity#list>). Vous n'avez pas le droit de manipuler directement des listes (excepté via les crochets pour accéder à un élément), mais vous avez droit à la fonction suivante :

---

```
1 def empty_list(n):
2     return [None] * n
```

---

**Hint :** il ne faut pas recréer un conteneur entier à chaque insertion mais assurer qu'un redimensionnement ne sera exécuté qu'un nombre de fois logarithmique en la taille du conteneur.

**Exercice 3.6.** Écrivez une classe `Array` représentant un tableau de taille fixe supportant l'indigage par un entier *et une slice* en lecture et en écriture telle que si `a` est un `Array` et que `b` est le résultat d'indigage de `a` par une *slice*, alors les valeurs contenues dans `b` et `a` partagent *la même zone mémoire*. En particulier :

---

```
N = 10
a = Array(N)
for i in range(N):
    a[i] = i
evens = a[::2]
odds = a[1::2]
odds[:] = -1 # on met les éléments impairs à -1
evens[::2] = 0 # on met les multiples de 4 à 0
backwards = a[::-1]
backwards[1::3] = 3
assert list(a) == [0, -1, 3, -1, 0, 3, 6, -1, 3, -1]
```

---

Une telle structure s'appelle une *vue* (ou *view* en anglais). Il faut que la méthode `Array.__getitem__` s'exécute en  $\mathcal{O}(1)$  opérations, même si une *slice* est passée en paramètre et que `Array.__setitem__` s'exécute en  $\mathcal{O}(m)$  opérations, pour  $m$  la taille de la *slice*. Vous ne pouvez pas utiliser la classe `range` en dehors des boucles `for` classiques.

**Exercice 3.7.** Écrivez une classe `Range` se comportant comme la classe `range` de Py-

thon. En particulier il doit utiliser un espace mémoire en  $\mathcal{O}(1)$  et à  $i$  fixé, l'expression `i in Range(start, stop, step)` doit s'exécuter en  $\mathcal{O}(1)$ . Bien sûr vous n'avez pas le droit d'utiliser `range`.

`range` est une *séquence* et doit dès lors satisfaire la structure générique définie par `la doc`. Remarquons tout de même que les opérateurs `+` et `*` ne sont pas supportés sur le type `range` car l'existence d'une instance de `range` comme concaténation de deux instances `r1` et `r2` n'est garantie que si les deux ont la même valeur pour `step` et si la `start` de la seconde équivaut au `stop` de la première (c.f. `ceci`). De plus, la méthode `index` existe en deux *versions* : une prenant uniquement la valeur dont on cherche l'indice (i.e. `Seq.index(self, x)`) et une prenant en plus les indices de début et (potentiellement) de fin de la plage possible pour les indices (i.e. `Seq.index(self, x, i[, j])`). `range` ne supporte que la première (la raison étant qu'une valeur ne peut apparaître qu'une unique fois dans une instance de `range`, il n'est pas nécessaire d'implémenter la deuxième version).

Notons également que nous n'implémenterons pas ici la gestion de l'indçage par *slices* car le sujet a déjà été traité dans l'`Exercice 3.6`.

**Exercice 3.8.** Écrivez une structure de données `MinStack` qui supporte les opérations `push` et `pop` de stack mais qui permet également de *consulter* la valeur minimale, le tout en  $\Theta(1)$ .

## Séance 4 — La récursivité (partie 1)

**Exercice 4.1.** Écrivez une fonction qui recherche, de manière récursive (sans utiliser la technique *diviser pour résoudre*) le plus grand élément d'un vecteur d'entiers de taille connue, et qui renvoie la valeur de cet élément.

**Exercice 4.2.** Écrivez une fonction qui transforme, de manière récursive, un vecteur d'entiers de taille connue en son image miroir.

**Exercice 4.3.** Dans le problème des tours de Hanoï, on dispose de 3 tours A, B et C, ainsi que de  $n$  disques de tailles différentes. Initialement, tous les disques sont placés sur la tour A, triés, en partant de la base de la tour, dans l'ordre décroissant par rapport à leur taille. Écrivez une fonction qui indique quels sont les déplacements de disques à effectuer pour que les disques de la tour A se trouvent dans la tour C, tout en respectant les règles suivantes :

- on ne peut déplacer qu'un seul disque à la fois ;
- un disque de taille plus grande ne peut jamais être placé sur un disque de taille plus petite.

Par exemple, pour  $n = 3$  nous obtiendrions :

```
Déplacer un disque de A à C
Déplacer un disque de A à B
Déplacer un disque de C à B
Déplacer un disque de A à C
Déplacer un disque de B à A
Déplacer un disque de B à C
Déplacer un disque de A à C
```



## Séance 5 — La récursivité (partie 2)

**Exercice 5.1.** Soit une suite de  $n$  entiers positifs lus, un à un, sur l'input. Écrivez une fonction récursive qui imprime cette suite de nombres en ordre inverse. Cette fonction ne devra pas utiliser de structure de travail intermédiaire pour stocker les nombres. Par exemple, pour  $n = 8$  et après avoir lu :

2 8 5 9 13 11 46 51

nous obtiendrions :

51 46 11 13 9 5 8 2.

**Exercice 5.2.** Soit une suite croissante de  $n$  entiers positifs lus, un à un, sur l'input. Écrivez une fonction qui imprime la sous-suite des nombres pairs de façon croissante et la sous-suite des nombres impairs de façon décroissante. Cette fonction ne devra pas utiliser de structure de travail intermédiaire pour stocker les nombres. Par exemple, pour  $n = 8$  et après avoir lu :

2 5 8 9 11 13 46 51

nous obtiendrions :

2 8 46 et 51 13 11 9 5.

**Exercice 5.3.** Calculez récursivement le déterminant d'une matrice carrée  $n+1 \times n+1$ . Pour rappel, pour une matrice carrée :

$$A = \begin{bmatrix} a_{00} & \dots & a_{0n} \\ \vdots & \ddots & \vdots \\ a_{n0} & \dots & a_{nn} \end{bmatrix} \in \mathbb{R}^{n+1 \times n+1},$$

la formule de Laplace nous permet d'exprimer (pour  $j \in \llbracket 0, n \rrbracket$  fixé) :

$$\det A = \sum_{k=0}^n (-1)^{j+k} a_{jk} \det A^{jk},$$

où  $A^{jk}$  est la sous-matrice :

$$A^{jk} = \begin{bmatrix} a_{00} & \dots & a_{0\,k-1} & a_{0\,k+1} & \dots & a_{0n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{j-1\,0} & \dots & a_{j-1\,k-1} & a_{j-1\,k+1} & \dots & a_{j-1\,n} \\ a_{j+1\,0} & \dots & a_{j+1\,k-1} & a_{j+1\,k+1} & \dots & a_{j+1\,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n0} & \dots & a_{n\,k-1} & a_{n\,k+1} & \dots & a_{nn} \end{bmatrix} \in \mathbb{R}^{n \times n},$$

i.e. la matrice  $A$  sans la  $j$ ème ligne et la  $k$ ème colonne.

## Exercices supplémentaires

**Exercice 5.4.** Soient deux nombres réels positifs  $x$  et  $y$ . La moyenne arithmétique de  $x$  et  $y$  est définie par  $\frac{1}{2}(x + y)$ , la moyenne géométrique de  $x$  et  $y$  est définie par  $\sqrt{xy}$  (la moyenne géométrique correspond à la moyenne arithmétique dans l'espace des logarithmes de  $x$  et  $y$ , i.e.  $\log \sqrt{xy} = \frac{1}{2}(\log x + \log y)$ ). La moyenne arithmético-géométrique de  $x$  et  $y$ , notée  $\text{agm}(x, y)$ , est définie de la manière suivante : considérons les deux suites  $(a_n)_n$  et  $(g_n)_n$  suivantes :

$$\begin{aligned} a_0 &= x \\ g_0 &= y \\ a_n &= \frac{1}{2}(a_{n-1} + g_{n-1}) \text{ si } n \geq 1 \\ g_n &= \sqrt{a_{n-1}g_{n-1}} \text{ si } n \geq 1 \end{aligned}$$

(i.e.  $a_n$  est la moyenne arithmétique de  $a_{n-1}$  et  $g_{n-1}$  alors que  $g_n$  est la moyenne géométrique de  $a_{n-1}$  et  $g_{n-1}$ ). Il est possible de montrer que pour tout  $n$ , on a :

1.  $g_n \leq a_n$  ;
2. cette inégalité est stricte ssi  $a_0 \neq g_0$  ;
3. la suite  $(a_n)_{n \geq 1}$  est décroissante alors que la suite  $(g_n)_{n \geq 1}$  est croissante ;
4.  $\lim_{n \rightarrow +\infty} a_n = \lim_{n \rightarrow +\infty} g_n$ .

$\text{agm}(x, y)$  est défini par cette limite.

Écrivez une fonction récursive `recursive_agm(x, y, eps)` et une fonction non-récursive `agm(x, y, eps)` calculant  $\text{agm}(x, y)$ , à précision `eps`.

**Exercice 5.5.** Une relation de récurrence linéaire est une équation de la forme :

$$x_n = \alpha_1 x_{n-1} + \alpha_2 x_{n-2} + \dots + \alpha_k x_{n-k} = \sum_{j=1}^k \alpha_j x_{n-j} \quad \text{pour } n > k,$$

pour un certain  $k$  entier, muni de conditions initiales  $(x_1, \dots, x_k)$ .

Écrivez une fonction `linear_rec_rel(N, alpha, x)` prenant en paramètre un entier  $N \geq 0$  et deux tuples `alpha` et `x` de taille  $k$  et renvoyant la valeur de  $x_N$  si  $(x_n)_n$  est une solution de la relation de récurrence linéaire associée.

**Exercice 5.6.** L'algorithme classique de multiplication de deux matrices de taille  $n \times n$  requiert  $\Theta(n^3)$  opérations. Volker Strassen a montré que cette approche n'est pas optimale en proposant un algorithme en  $\mathcal{O}(n^{2.807\dots})$  que l'on peut formuler comme suit.

Soient deux matrices  $A, B \in \mathbb{R}^{2^N \times 2^N}$  (pour un certain  $N > 0$  entier). Nous pouvons séparer ces matrices en *blocs* de taille  $2^{N-1} \times 2^{N-1}$  :

$$A = \begin{bmatrix} A^1 & A^2 \\ A^3 & A^4 \end{bmatrix} \quad \text{et} \quad B = \begin{bmatrix} B^1 & B^2 \\ B^3 & B^4 \end{bmatrix}.$$

Notons  $C \in \mathbb{R}^{2^N \times 2^N}$  le produit  $AB$  et admettant la même séparation en blocs. Introduisons les 7 matrices suivantes :

- $M^1 := (A^1 + A^4)(B^1 + B^4)$ ;
- $M^2 := (A^3 + A^4)B^1$ ;
- $M^3 := A^1(B^2 - B^4)$ ;
- $M^4 := A^4(B^3 - B^1)$ ;
- $M^5 := (A^1 + A^2)B^4$ ;
- $M^6 := (A^3 - A^1)(B^1 + B^2)$ ;
- $M^7 := (A^2 - A^4)(B^3 + B^4)$ .

Alors les égalités suivantes sont vérifiées :

$$C = \begin{bmatrix} M^1 + M^4 - M^5 + M^7 & M^3 + M^5 \\ M^2 + M^4 & M^1 - M^2 + M^3 + M^6 \end{bmatrix}.$$

Implémentez cet algorithme.

## Séance 6 — Le backtracking (partie 1)

**Exercice 6.1.** Écrivez une fonction qui engendre récursivement tous les sous-ensembles d'un ensemble d'entiers positifs de taille connue.

**Exercice 6.2.** Adaptez le code de la génération de sous-ensembles pour ne générer que les sous-ensembles de taille  $k$  de l'ensemble des entiers allant de 1 à  $n$ .

Par exemple, pour  $k = 2$  et  $n = 4$ , nous obtiendrons :

```
{1 2}
{1 3}
{1 4}
{2 3}
{2 4}
{3 4}
```

**Exercice 6.3.** Générez tous les sous-ensembles de taille  $k$  de l'ensemble des entiers allant de 1 à  $n$ , mais en n'utilisant qu'un vecteur de travail de taille  $k$  (au lieu de  $n$ ). Utile si, par exemple, nous avons  $n = 1000000000$  et  $k = 25$ .

**Exercice 6.4.** Écrivez un algorithme, de manière récursive et puis de manière non récursive, qui affiche toutes les solutions entières d'une équation linéaire à  $n$  inconnues, dont les solutions sont dans  $[0, b]$ . Une équation linéaire est de la forme :

$$a_0x_0 + a_1x_1 + \dots + a_{n-1}x_{n-1} = y. \quad (3.1)$$

On suppose que  $\forall 0 \leq k < n : a_k > 0$ .

**Bonus :** comment devez-vous adapter votre code afin d'assurer que les variables  $x_k$  sont dans  $\llbracket b', b \rrbracket$  au lieu de  $\llbracket 0, b \rrbracket$  ?

## Séance 7 — Le backtracking (partie 2)

**Exercice 7.1.** Écrivez un algorithme qui calcule un parcours d'un cavalier sur un échiquier  $n \times m$ , tel que le cavalier partant de la case  $(0, 0)$  passe exactement une fois par chaque case et arrive à la case  $(x, y)$  donnée en paramètre. Pour rappel, un cavalier se déplace en L.

**Exercice 7.2.** Soit un labyrinthe, représenté par une matrice  $n \times m$  dans laquelle les murs sont marqués par des `x` et les passages par des `_`. Trouvez un chemin, s'il en existe, reliant les coordonnées  $(0, 0)$  aux coordonnées  $(n-1, m-1)$ .

**Exercice 7.3.** Adaptez votre solution de l'exercice précédent pour trouver le chemin *le plus court* entre les coordonnées  $(0, 0)$  et les coordonnées  $(n-1, m-1)$ .

**Exercice 7.4.** Soit un vecteur  $v$ , écrivez un programme qui détermine s'il existe trois indices distincts  $i, j, k$  tels que  $v[i] + v[j] + v[k] = 0$ .

## Exercices supplémentaires

**Exercice 7.5.** Soient  $n$  et  $k$  deux nombres entiers positifs. Considérons un plateau  $n \times n$ . Écrivez un programme qui détermine s'il existe une configuration de  $k$  reines blanches et  $k$  reines noires sur le plateau telles qu'aucune reine noire n'attaque une reine blanche et inversement.

**Exercice 7.6.** Écrivez un programme qui compte le nombre de matrices booléennes  $n \times n$  telles que chaque ligne et chaque colonne contient exactement  $k$  entrées à  $\top$ . Par exemple pour  $n = 5$  et  $k = 1$ , il y a  $5! = 120$  telles matrices (qui correspondent aux permutations de l'identité), et pour  $n = 4$  et  $k = 2$ , il y a 90 telles matrices.

**Exercice 7.7.** Considérons les variables booléennes  $x_1, \dots, x_n$ . Un *littéral* est une expression de la forme  $x_i$  ou  $\neg x_i$ . Une clause est une disjonction de littéraux, i.e. un OU logique entre plusieurs littéraux. Nous appelons ici une *formule sous forme normale conjonctive (FNC)* toute expression étant une conjonction de plusieurs clauses. Par exemple dans les formules suivantes,  $\phi_1$  et  $\phi_2$  sont des formules sous FNC alors que  $\phi_3$  ne l'est pas :

$$\phi_1 \equiv (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

$$\phi_2 \equiv (x_1 \vee x_2) \vee (x_1 \vee \neg x_3)$$

$$\phi_3 \equiv (x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_2 \wedge x_4 \wedge \neg x_5)$$

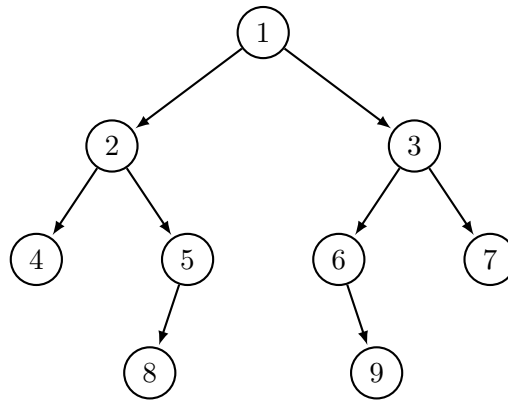
Une formule est dite *satisfaisable* s'il est possible d'assigner à chaque variable booléenne une valeur (donc soit  $\top$  soit  $\perp$ ) telle que la formule en entier est évaluée à  $\top$ . Par exemple la formule  $\phi_1$  ci-dessus est satisfaisable : il suffit de mettre tous les  $x_i$  à  $\perp$ .

Écrivez un programme qui détermine si une formule sous FNC donnée en entrée est satisfaisable ou non. La formule est reçue en paramètre sous la forme d'une liste de listes où le littéral  $x_k$  est représenté par  $k$  et le littéral  $\neg x_k$  est représenté par  $-k$ . Par exemple la formule  $\phi_1$  ci-dessus serait représentée comme ceci :

```
[[1, 3, -4], [2, -3, 4]]
```

## Séance 8 — Arbres (partie 1)

**Exercice 8.1.** Donnez l'ordre de parcours préfixé, infixé et suffixé des nœuds de l'arbre suivant :



**Exercice 8.2.** Écrivez un morceau de code qui construit l'arbre binaire de l'exercice précédent en utilisant la classe `BinaryTree` donnée ci-dessous :

```

1 class BinaryTree:
2     def __init__(self, root_obj, left=None, right=None):
3         self.key = root_obj
4         self.left = left
5         self.right = right
6
7     @property
8     def root_value(self):
9         return self.key
10
11    @property
12    def left_child(self):
13        return self.left
14
15    @property
16    def right_child(self):
17        return self.right
  
```

**Exercice 8.3.** Écrivez une fonction non récursive qui réalise le parcours préfixé d'un arbre binaire respectant l'interface de la classe `BinaryTree` sans père vue dans ce chapitre.

**Exercice 8.4.** Écrivez une fonction non récursive qui réalise le parcours infixé d'un arbre binaire respectant l'interface de la classe `BinaryTree` sans père vue dans ce chapitre.

## Séance 9 — Arbres (partie 2)

**Exercice 9.1.** Sur base de l'interface de la classe `BinaryTreeFather` ci-dessous, écrivez :

- une fonction `first(tree)` qui renvoie le premier nœud de l'arbre `tree` visité lors d'un parcours préfixé;
- une fonction `next(node)` qui renvoie le nœud qui sera traité juste après `node` lors d'un parcours préfixé.

---

```

1 class BinaryTreeFather(BinaryTree):
2     def __init__(self, root_obj, left=None, right=None, father=None):
3         BinaryTree.__init__(self, root_obj, left, right)
4         self.__father = father
5
6     @property
7     def father(self):
8         return self.__father

```

---

**Bonus :** si maintenant, `first` ne prend plus en paramètre une racine mais un nœud quelconque de l'arbre, adaptez votre solution pour qu'elle renvoie le premier nœud traité lors du parcours préfixé de l'arbre contenant ce nœud.

**Exercice 9.2.** Écrivez une fonction récursive `contains(tree, value)` qui renvoie `True` si `value` est dans `tree` (un arbre binaire) et `False` sinon.

**Bonus :** adaptez votre fonction pour qu'elle renvoie `None` si l'arbre ne contient pas `value` et qu'elle renvoie la référence vers le sommet ayant cette valeur si un tel nœud existe.

**Exercice 9.3.** Écrivez une fonction récursive qui teste si deux arbres binaires sont égaux.

**Exercice 9.4.** Écrivez une fonction récursive `mirror(tree)` qui renvoie un nouvel arbre binaire, étant celui-ci l'image miroir d'un arbre binaire reçu en paramètre.



## Exercices supplémentaires

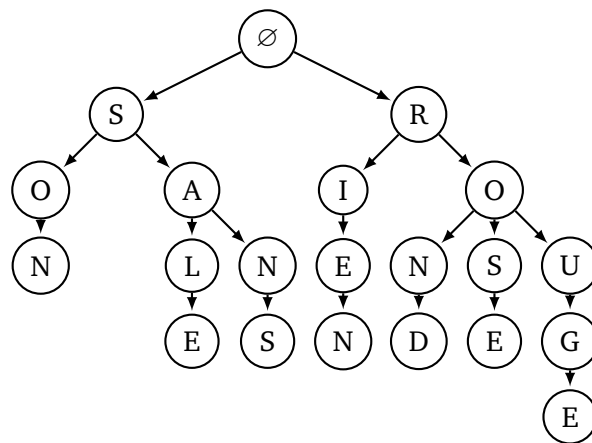
**Exercice 9.5.** Considérons un alphabet quelconque  $\Sigma$  (par exemple l'ensemble des lettres latines), et prenons un ensemble de clefs  $K \subset \Sigma^*$ . Un *trie* (également appelé *arbre à préfixes*) est une structure de données dynamique permettant de stocker (et possiblement localiser) des clefs de  $K$ .

Pour ce faire, nous conservons un arbre (pas nécessairement binaire) dont la racine correspond à la clef vide, chaque sommet contient un unique caractère de  $\Sigma$  et le chemin vers chaque feuille détermine les clefs stockées.

Implémentez un tel type de données avec ses méthodes :

- `insert(key)` qui insère la clef `key` dans la structure ;
- `contains(key)` qui renvoie `True` ou `False` en fonction de si `key` apparaît dans le conteneur ou non ;
- `delete(key)` qui retire la clef `key` du conteneur.

Si `key` est une clef de longueur  $\ell$ , ces trois méthodes doivent s'exécuter en  $\mathcal{O}(\ell)$  opérations dans le pire des cas.



## Séance 10 — Séquences triées

**Remarque.** *Au cours des exercices de ce chapitre nous travaillerons avec des arbres binaires de recherche tels que les éléments présents dans le sous-arbre gauche sont strictement plus petits que la racine et ceux présents dans le sous-arbre droit sont plus grands ou égaux à la racine. Afin de pouvoir manipuler les arbres binaires de recherche en tant qu'arbres binaires, nous travaillerons avec la classe `BinaryTree` cachant tout détail d'implémentation.*

**Exercice 10.1.** Écrivez une fonction qui vérifie, récursivement et puis non récursivement, si un arbre binaire d'entiers respectant l'interface de la classe `BinaryTree` est un arbre binaire de recherche.

**Exercice 10.2.** Écrivez une fonction non-récursive qui vérifie si deux arbres binaires de recherche donnés sont équivalents, c'est-à-dire s'ils contiennent les mêmes éléments.

**Exercice 10.3.** Écrivez une fonction qui, étant donné un entier  $x$ , découpe un arbre binaire de recherche en deux arbres binaires de recherche tels que les éléments du premier (resp. second) arbre soient strictement inférieurs (resp. supérieurs ou égaux) à  $x$ .

**Exercice 10.4.** Écrivez une fonction qui, étant donné un arbre binaire de recherche et un entier  $x$  se trouvant dans l'arbre, construit un nouvel arbre binaire de recherche contenant les mêmes éléments que l'arbre de départ, mais dont  $x$  est la racine.

## Séance 11 — Files à priorité

**Définition 17.** Soit  $T$  un arbre binaire.

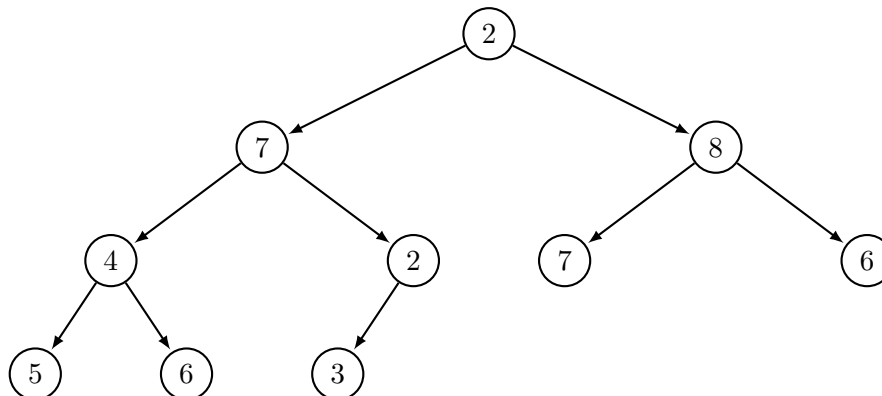
- si tout sommet  $v$  de  $T$  satisfait la propriété suivante : la valeur de  $v$  est *supérieure* ou égale à la valeur de ses fils, alors  $T$  est un *max-tas* (ou *max-heap*) ;
- si tout sommet  $v$  de  $T$  satisfait la propriété suivante : la valeur de  $v$  est *inférieure* ou égale à la valeur de ses fils, alors  $T$  est un *min-tas* (ou *min-heap*).

Habituellement, un *tas* désigne un max-tas, si ce n'est pas précisé.

**Exercice 11.1.** Soit le vecteur  $v$  suivant :  $v = [8, 1, 9, 4, 7, 6, 10]$ .

1. Simulez la création d'un max-tas en insérant itérativement les éléments de  $v$ .
2. Simulez la création d'un min-tas en insérant itérativement les éléments de  $v$ .
3. Simulez le fonctionnement de l'algorithme heapsort sur le vecteur  $v$ .

**Exercice 11.2.** Soit un tas particulier décrit par un arbre binaire complet dont tout élément appartenant à un niveau pair est plus petit ou égal que tous ses descendants et tout élément appartenant à un niveau impair est plus grand ou égal que tous ses descendants. Le numéro de niveau de la racine étant 0. Voici un exemple d'un tel tas, dans lequel chaque élément est représenté par un nombre entier :



Écrivez un algorithme permettant d'insérer un élément dans ce type de structure.

## Exercices supplémentaires

**Exercice 11.3.** Implémentez une file à priorité en utilisant un BST et non un heap.  
Quelle est la complexité des différentes opérations ?

## Séance 12 — Hachage (partie 1)

Les exercices de ce chapitre viennent en partie du chapitre 11 du livre *Introduction to algorithms* de Cormen, Leiserson, Rivest et Stein (livre également appelé *Le CLRS*, du nom de ses auteurs).

Ces exercices font intervenir les spécificités de Python vis à vis du hachage, référez-vous à la [sous-section 13.1](#) pour plus de détails sur le fonctionnement interne de ces notions.

**Exercice 12.1.** Voici trois fonctions de hachage fonctionnant sur des conteneurs d'entiers (e.g. des tuples). Pour chacune d'entre elles, déterminez deux clefs différentes qui entrent en collision, et trouvez une clef qui donne le hash suivant : 177583. Vous pouvez supposer que toutes ces fonctions renvoient en réalité `ret & 0xFFFFFFFF` (i.e. renvoient un `uint32_t` en C(++)).

```

1 def h1(x):
2     return sum(x)
3
4 def h2(x):
5     return len(x) + sum(x)
6
7 def h3(x):
8     ret = 5381
9     for item in x:
10         ret = 33*ret + item
11     return ret

```

**Exercice 12.2.** Le code suivant lancera-t-il une exception ?

```

1 class Pair:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5 S = set()
6 p = Pair(1, 2)
7 S.add(p)
8 assert p in S
9 p.x = -1
10 assert p not in S

```

Redéfinissez la méthode `__hash__` afin que de sorte à ce que :

$$h(x, y) = 2^{32}x + (y \bmod 32).$$

Le code ci-dessus s'exécutera-t-il correctement ?

**Exercice 12.3.** En Python (tout du moins selon l'interpréteur CPython, c.f. [ceci](#)), les tuples sont hachés à l'aide de l'algorithme suivant (qui est une variation de l'algorithme `xxHash`) :

---

```

1: procedure xxHASH(tuple  $t$ )
2:    $x \leftarrow P$ 
3:   for every  $k$  in  $t$  do
4:      $x \leftarrow x + h(k) \cdot N$ 
5:      $x \leftarrow \text{rotate\_left}(x, 31)$ 
6:      $x \leftarrow x \cdot M$ 
7:   end for
8:   return  $x + (L \text{ xor } P \text{ xor } X)$ 
9: end procedure

```

---

où  $L$  est la longueur de  $t$ , et où  $M, N, P, X$  sont les constantes suivantes :

$$\begin{aligned}
 M &= 11400714785074694791 \\
 N &= 14029467366897019727 \\
 P &= 2870177450012600261 \\
 X &= 3527539
 \end{aligned}$$

Implémentez cet algorithme et vérifiez le résultat.

**Remarque :** Cet algorithme suppose que les entiers sont encodés sur 64 bits (i.e. un `uint64_t` en C(++)). Toutes les opérations se font donc modulo  $2^{64}$ .

**Exercice 12.4.** Un vecteur de bits est tout simplement un tableau de bits (0 et 1). Un vecteur de bits de longueur  $m$  prend beaucoup moins d'espace qu'un tableau de  $m$  pointeurs/références. Décrivez comment on pourrait utiliser un vecteur de bits pour représenter un ensemble dynamique d'éléments distincts sans donnée satellite. Les opérations d'insertion, de recherche et de suppression devront s'exécuter dans un temps  $\mathcal{O}(1)$ .

**Exercice 12.5.** On souhaite implémenter un ensemble dynamique en utilisant l'adressage direct sur un très grand tableau. Au départ, les entrées du tableau peuvent contenir des données quelconques. L'initialisation complète du tableau s'avère peu pratique, à cause de sa taille. Décrivez un schéma d'implémentation de dictionnaire via adressage direct sur un très grand tableau. Chaque objet stocké devra consommer un espace  $\mathcal{O}(1)$ . Les opérations `search`, `insert` et `delete` devront prendre chacune un temps  $\mathcal{O}(1)$  et l'initialisation des structures de données devra se faire en un temps  $\mathcal{O}(1)$  également.

**Exercice 12.6.** Montrez comment on réalise l'insertion des éléments suivants dans une table de hachage où les collisions sont résolues par chaînage :

5, 28, 19, 15, 20, 33, 12, 17, 10

On suppose que la table contient 9 alvéoles et que la fonction de hachage est :

$$h(k) = k \mod 9.$$

## Séance 13 — Hachage (partie 2)

**Exercice 13.1.** Supposons qu'on souhaite parcourir une liste chaînée de longueur  $n$ , dans laquelle chaque élément contient une clé  $k$  en plus d'une valeur de hachage  $h(k)$ . Chaque clé est une longue chaîne de caractères. Comment pourrait-on tirer parti des valeurs de hachage pendant la recherche d'un élément de clé donnée ?

**Exercice 13.2.** Supposons que l'on utilise le double hachage pour gérer les collisions ; autrement dit, on utilise la fonction de hachage suivante :

$$h(k, j) = (h_1(k) + jh_2(k)) \bmod m.$$

Montrez que, si  $m$  et  $h_2(k)$  ne sont pas premiers entre eux (i.e.  $d = \text{GCD}(m, h_2(k)) \geq 1$ ) pour une certaine clé  $k$ , alors une recherche infructueuse de la clé  $k$  examine une proportion  $\frac{1}{d}$  de la table de hachage avant de revenir à l'alvéole  $h_1(k)$ . Donc, quand  $m$  et  $h_2(k)$  sont premiers entre eux (i.e.  $d = 1$ ), la recherche risque de balayer toute la table de hachage.

**Exercice 13.3.** Le principe de l'adressage ouvert est de résoudre les collisions par un *chaînage interne* de la table dans lequel les cases visitées successivement lors d'une opération correspondent à une chaîne. Implémentez un ensemble (donc se comportant comme la classe `set` en Python) via une table de hachage dans laquelle ce chaînage est explicite et qui, de plus, utilise une liste chaînée des cases libres. Nous imposons de plus les contraintes suivantes :

1. l'insertion doit s'exécuter en  $\mathcal{O}(1)$  ;
2. la suppression d'un nœud doit s'exécuter en  $\mathcal{O}(1)$ .

Attention la suppression ici ne contient pas la recherche qui, elle, doit s'exécuter en  $\mathcal{O}(m)$  où  $m$  est la taille de la table. Voici un squelette de code (la classe `Set` doit contenir une table dont tous les éléments sont des nœuds d'une liste chaînée) :

```

1 class Set:
2     def __init__(self, m): # Theta(m)
3         # ...
4     def insert(self, x: object) -> None: # O(1)
5         # ...
6     def find(self, x: object) -> Node: # O(m)
7         # ...
8     def remove(self, x: object) -> None: # O(m)
9         # ...
10    def delete(self, node: Node) -> None: # O(1)
11    # ...

```

**Exercice 13.4.** Implémentez un dictionnaire via une table de hachage à adressage ouvert similaire à celle implémentée dans CPython (c.f. la [sous-section 13.1](#)). Vous pouvez supposer que votre table est de taille fixe et définie lors de l'initialisation.

## Exercices supplémentaires

**Exercice 13.5.** On considère une variante de la méthode de la division dans laquelle  $h(k) = k \bmod m$ , où  $m = 2^p - 1$  et  $k$  est une chaîne de caractères interprétée en base  $2^p$ . Montrez que si la chaîne  $x$  peut être déduite de la chaîne  $y$  par permutation de ses caractères, alors  $x$  et  $y$  ont même valeur de hachage. Donnez un exemple d'application pour laquelle cette propriété de la fonction de hachage serait indésirable.

**Exercice 13.6.** On dit qu'une famille  $\mathcal{H}$  de fonctions de hachage reliant un ensemble fini  $U$  à un ensemble fini  $B$  est  $\varepsilon$ -universelle si, pour toute paire d'éléments distincts  $k$  et  $\ell$  de  $U$ , on a :

$$\mathbb{P}_{\mathcal{H}}[h(k) = h(\ell)] \leq \varepsilon,$$

où la probabilité est définie par le tirage aléatoire (uniforme) de la fonction de hachage  $h$  dans la famille  $\mathcal{H}$ . Montrez qu'une famille  $\varepsilon$ -universelle de fonctions de hachage doit vérifier :

$$\varepsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

**Exercice 13.7.** Adaptez l'implémentation des ensembles dynamiques via des tables de hachage avec gestion des collisions par adressage ouvert de manière à permettre la suppression d'éléments.

**Exercice 13.8.** Adaptez l'implémentation des ensembles dynamiques via des tables de hachage avec gestion des collisions par adressage ouvert de manière ne plus avoir de restriction sur la taille du conteneur (donc la table interne doit être dynamique).



## 13.1 Remarques sur le fonctionnement du hachage en Python

### Fonction de hachage

Python propose une fonction *built-in* appelée `hash`. Tout comme `len`, `str`, `repr`, etc., cette fonction appelle la méthode spéciale associée sur son paramètre, i.e. `hash(x)` renvoie systématiquement `x.__hash__()` (donc `type(x).__hash__(x)`, avec une seule exception que nous verrons ci-dessous).

Il est donc tout à fait possible de choisir comment doivent être hachées les classes définies en donnant une implémentation de cette fonction. De plus, tout comme `__repr__` admet une implémentation par défaut (celle de `object.__repr__`), la méthode spéciale `__hash__` admet également une implémentation par défaut. La seule restriction sur le fonctionnement de cette fonction est que si deux objets `x` et `y` sont tels que `x == y` est évalué à `True`, alors `hash(x) == hash(y)` doit également être évalué à `True`. Nous allons cependant voir que cette fonction est une *très mauvaise* fonction de hachage.

Jusqu'en Python 3.1 compris, le hash par défaut de tout objet était son identifiant (qui, dans CPython, correspond à l'adresse en mémoire de l'objet représenté), i.e. cette méthode pouvait être vue comme :

---

```

1 class object:
2     # ...
3     def __hash__(self):
4         return id(self)

```

---

Cependant, depuis Python 3.2 (c.f. cette [discussion](#)), cette fonction a été légèrement modifiée afin de diminuer le nombre de collisions. Le raisonnement était le suivant : puisque le hash n'est autre que l'adresse mémoire de l'objet mais que les adresses mémoires sont alignées sur certaines puissances de 2, les quelques derniers bits vont presque systématiquement être à 0. Or si l'indice dans la table de hachage d'un élément  $k$  est  $h(k) \bmod m$  où  $m$  est la taille de la table, alors si  $m$  est une puissance de 2, une certaine proportion de la table n'est pas accessible.

Afin de remédier à cela, la fonction de hachage par défaut a été modifiée afin de ne pas donner d'importance aux 4 derniers bits de `id(self)` en regardant uniquement la quantité `id(self) // 16`. Pour tout de même permettre le hachage instructif d'un élément qui n'aurait pas quatre 0 à la fin de son adresse, les 4 bits de poids faible sont déplacés et sont mis en 4 bits de poids fort (donc une rotation droite de 4 positions) :

---

```

1 class object:
2     # ...
3     def __hash__(self):
4         h = hex(id(self))[2:]
5         return int(h[-1] + h[:-1], 16)

```

---

La fonction `hash` peut tout à fait renvoyer une valeur négative (e.g. `hash(-10) == -10`), ce qui peut paraître étrange dans certains cas, mais ce choix a été fait afin de simplifier la gestion des hashes des valeurs numériques (c.f. cette [discussion](#)).

Il est important de noter que la fonction `hash` ne peut par contre jamais renvoyer la valeur `-1` (puisque CPython est codé en C, langage qui ne permet pas la gestion d'er-

reurs par exceptions, la valeur -1 est réservée par l'interpréteur pour signaler que le hash est soit impossible, soit s'est mal passé). Si pour une raison quelconque la méthode `type(x).__hash__(x)` renvoie -1, la fonction `hash` interprètera ce résultat comme ambigu et le transformera en -2, e.g. :

---

```

1 class C:
2     def __hash__(self):
3         return -1
4 c = C()
5 print((hash(c), c.__hash__()))

```

---

affichera (-2, -1) car `C.__hash__` renverra bien -1, mais cette valeur n'est pas acceptable pour la fonction `hash`.

Bien qu'une fonction de hachage par défaut existe (comme vu ci-dessus), certains types ont une fonction de hachage prédéfinie. C'est en particulier le cas des tuples qui fonctionnent sur une version modifiée de l'algorithme xxHash ; des types numériques (donc `bool`, `int` et `float`) ; ou encore des chaînes de caractères (que ce soit `str` ou `bytes`). Le cas des tuples est vu dans l'[Exercice 12.3](#), le fonctionnement sur les types numériques est défini [dans la documentation](#), et les chaînes de caractère étaient précédemment (avant Python 3.4) hachées via une adaptation de l'algorithme FNV, sont hachées via l'algorithme SipHash 2-4 jusqu'en Python 3.10 compris) et le passage à SipHash 1-3 est officialisé à partir de Python 3.11 (c.f. [ceci](#)).

### Tables de hachage en Python

Les types `set` et `dict` sont tous deux implémentés par des tables de hachage à adressage ouvert. Cependant ces tables ne peuvent utiliser le double hachage car cela nécessiterait l'implémentation de deux méthodes `__hash__` différentes, ce qui est irréaliste en pratique (et qui s'opposerait fondamentalement avec la mentalité *Simple is better than complex* si chère à [PEP20](#)). L'implémentation repose donc sur un sondage pseudo-aléatoire (voir [ces explications](#) pour plus d'informations). Le raisonnement est celui-ci : si  $j$  (donc à comprendre comme `hash(x) % m`) est l'indice d'une entrée dans la table, au lieu de regarder itérativement les indices  $j+1$ ,  $j+2$ , etc. (comme dans le sondage linéaire) l'indice de départ est  $j$  et puis on applique la relation de récurrence suivante :

$$j_n = 5j_{n-1} + 1 \mod m,$$

où  $j_0 = j$  et  $m$  est la taille du conteneur (avec  $m = 2^k$  pour un certain  $k$  entier).

Cependant, cette approche reste assez similaire à un sondage linéaire puisque si deux entrées  $x$  et  $y$  sont traitées l'une après l'autre et qu'elles satisfont `hash(x) % m == hash(y) % m`, alors les mêmes cases vont être visitées dans le même ordre. Il a dès lors été décidé d'ajouter une composante *pseudo-aléatoire* au sondage : en reprenant le fonctionnement ci-dessus (avec la relation de récurrence sur  $j_n$ ), on définit à nouveau  $j_0 = j$  mais on initialise également une quantité  $p_0$  qui est initialisée à `hash(x)` (i.e.  $j = p \mod m$ ) et on définit la relation de récurrence suivante :

$$p_n = \left\lfloor \frac{p_{n-1}}{2^5} \right\rfloor.$$

La relation sur  $j_n$  devient donc :

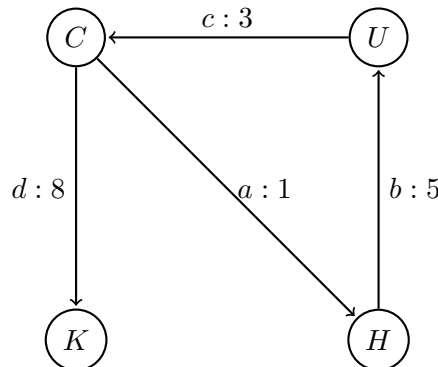
$$j_n = (5j_{n-1} + 1 + p_n) \mod m.$$

Puisque la taille de la table est obligatoirement une puissance de 2, effectuer le  $\text{mod } 2^k$  correspond à faire un masque ne conservant que les  $k$  derniers bits (donc bits de poids faible) de la représentation binaire de la quantité  $j_n$ . Cela implique en particulier que les bits de poids faible n'ont qu'un petit rôle à jouer dans la première version du chaînage proposée. Hors, si  $x$  et  $y$  satisfont  $\text{hash}(x) \% m == \text{hash}(y) \% m$ , alors en particulier les hashes coïncident sur les bits de poids faible, i.e. les bits de poids fort vont être cruciaux pour déterminer l'ordre de parcours et garantir un bon comportement du sondage. C'est pour cela que les  $p_n$  sont exponentiellement dégressifs (et correspondent en fait à un SHIFT droit) : ils garantissent qu'en seulement quelques itérations, tous les bits de poids fort auront été considérés et auront permis de former l'ordre des indices sondés.

Notons que le nom *pseudo-aléatoire* ici vient du fait que simplement connaître  $\text{hash}(x) \% m$  (et pas  $\text{hash}(x)$ ) ne permet pas de savoir à l'avance quel va être l'ordre de parcours des indices sondés.

## Séance 14 — Graphes (partie 1)

Pour cette séance, considérons le graphe  $G_1$  suivant :



Considérons également la matrice d'adjacence du graphe  $G_2$  donne ci-dessous :

	A	B	C	D	E	F	G
A	0	4	0	0	0	0	0
B	3	0	1	3	0	0	0
C	0	0	0	2	2	0	0
D	0	0	0	0	7	0	0
E	8	0	0	0	0	0	0
F	0	0	0	0	0	0	3
G	0	0	0	0	0	0	1

**Exercice 14.1.** Construisez la matrice d'adjacence et la matrice d'incidence du graphe  $G_1$ .

**Exercice 14.2.** Que deviennent les matrices d'adjacence et d'incidence de  $G_1$  si on retire tous les poids des arcs ?

**Exercice 14.3.** Construisez la structure dynamique (listes de successeurs) qui représente le graphe  $G_1$ .

**Exercice 14.4.** Dessinez le graphe  $G_2$  représenté par la matrice d'adjacence ci-dessus.

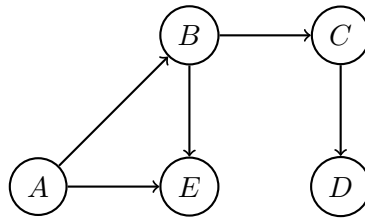
**Exercice 14.5.** Il existe deux manières principales de représenter un graphe simple dans une structure de données : à l'aide de sa matrice d'adjacence, ou grâce à une structure dynamique utilisant des listes de successeurs. On vous demande d'écrire deux fonctions :

- l'une recevra un graphe selon la première représentation et retournera le même graphe mais selon la seconde représentation ;
- l'autre aura l'effet inverse.

**Bonus :** peut-on faire la même chose avec une matrice d'incidence à la place d'une matrice d'adjacence ? Si non, quelle(s) condition(s) devons-nous imposer sur  $G$  ?

## Séance 15 — Graphes (partie 2)

Considérons le graphe  $G$  suivant :



**Exercice 15.1.** Donnez la matrice d'adjacence de  $G$  et la liste des sommets accessibles depuis chaque sommet.

**Exercice 15.2.** Exécutez l'algorithme de Roy-Warshall sur  $G$ .

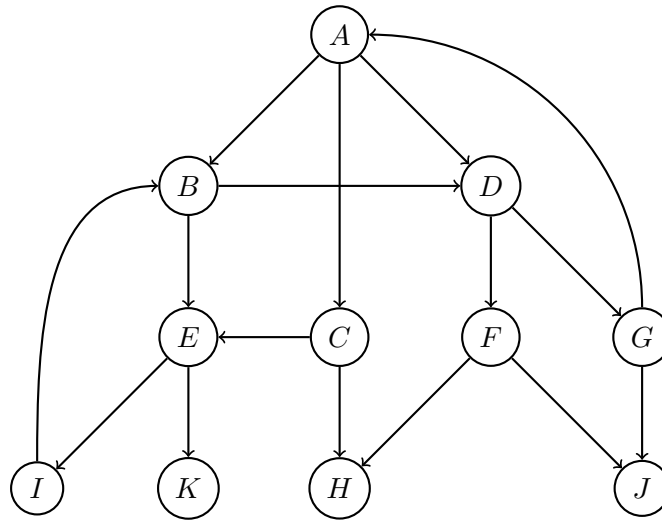
**Exercice 15.3.** Construisez la fermeture transitive ( $M^*$ ) de  $G$  en utilisant la formule suivante ( $M$  est la matrice d'adjacence) :

$$M^* = M^0 \vee M^1 \vee \dots \vee M^{n-1} = \bigvee_{k=0}^{n-1} M^k.$$

**Exercice 15.4.** Comparez les résultats des trois questions précédentes.

## Séance 16 — Graphes (partie 3)

Considérons le graphe suivant :



**Exercice 16.1.** Donnez la séquence des sommets parcouru si on utilise l'algorithme *depth-first* (parcours en profondeur) pour parcourir le graphe ci-dessus en commençant par le sommet A.

**Exercice 16.2.** Donnez la séquence des sommets parcouru si on utilise l'algorithme *breadth-first* (parcours en largeur) pour parcourir le graphe ci-dessus en commençant par le sommet A.

**Exercice 16.3.** Donnez la séquence des sommets parcouru si on utilise l'algorithme *breadth-first* (parcours en largeur) pour parcourir le graphe ci-dessus en commençant par le sommet I.

**Exercice 16.4.** Considérez un graphe  $G = (V, E)$  non-dirigé et non-pondéré représenté par sa matrice d'adjacence  $M$ . Soient  $u, v \in V$ , deux sommets de  $G$ . Écrivez `shortest_path`, une méthode qui trouve le plus court chemin de  $u$  à  $v$  avec une méthode de backtracking. Cette fonction doit renvoyer un tuple (`longueur`, `chemin`) contenant respectivement la longueur du plus court chemin et la liste des différents sommets du chemin (le premier étant  $u$ , le dernier étant  $v$ ). S'il n'existe pas de chemin entre  $u$  et  $v$ , alors `shortest_path` renverra  $(-1, [])$ .

À quel type de parcours cet algorithme correspond-il ? Peut on faire mieux ?

**Exercice 16.5.** Un graphe peut être utilisé pour représenter un réseau social dans lequel les utilisateurs peuvent définir des amis. On considère qu'un sommet représente une personne et qu'il existe une arrête entre deux sommets si les personnes représentées par ces sommets sont amies.

Dans cet exercice, chaque sommet du graphe contient au minimum les informations suivantes :

- le nom de la personne ;
- son score à *Tetris* ;

— une liste d'amis (autres sommets du graphe).

On vous demande d'écrire une classe `Graph` représentant un graphe dont le constructeur reçoit la matrice d'adjacence lui correspondant ainsi que la liste des sommets du graphe, et contenant une méthode `ranking(d, v)` affichant le classement à Tetris de l'ensemble des joueurs se trouvant à une distance inférieure ou égale à `d` du joueur représenté par `v`.

## Exercices supplémentaires

### Exercice 16.6.

1. Déterminez une famille de graphes connexes à  $n$  sommets ayant  $\Theta(n)$  arêtes ;
2. déterminez une famille de graphes connexes à  $n$  sommets ayant  $\Theta(n \log n)$  arêtes ;
3. déterminez une famille de graphes connexes à  $n$  sommets ayant  $\Theta(n^{3/2})$  arêtes ;
4. déterminez une famille de graphes connexes à  $n$  sommets ayant  $\Theta(n^2)$  arêtes.

**Exercice 16.7.** Montrez que dans un arbre  $T$ , pour chaque paire de sommets  $(u, v)$ , il existe un unique chemin  $P$  joignant  $u$  et  $v$ .

**Exercice 16.8.** Montrez que si  $G$  est un graphe connexe cyclique, alors il existe un sous-graphe strict de  $G$  qui est connexe.

**Exercice 16.9.** Montrez que si  $G$  est un graphe non-connexe, alors  $G^c$  est connexe.

**Exercice 16.10.** Soit  $G$  un graphe à  $n$  sommets.

1. Si  $G$  est connexe, quel est le nombre minimal d'arêtes de  $G$  ?
2. Si  $G$  n'est pas connexe, quel est le nombre maximal d'arêtes de  $G$  ?

**Hint :** Utilisez les deux exercices précédents.

**Exercice 16.11.** Montrez que si  $T$  est un arbre à  $n \geq 2$  sommets, alors  $T$  admet au moins deux feuilles.

**Exercice 16.12.** Soit  $T$  un arbre et notons  $\Delta(T)$  le degré maximal de ses sommets. Montrez que  $T$  a au moins  $\Delta(T)$  feuilles.

**Bonus :** trouvez (i) une famille d'arbres  $(T_n)$  ayant exactement  $\Delta(T_n)$  feuilles ; (ii) une famille d'arbres à  $n$  sommets ayant de degré maximal  $\mathcal{O}(1)$  mais avec  $\Theta(n)$  feuilles.

**Définition 18.** Soient deux graphes dirigés simples finis  $G_1$  et  $G_2$  (les définitions qui suivent peuvent être étendues, mais concentrons-nous sur un cas simple). On définit :

- le *produit cartésien* entre  $G_1$  et  $G_2$  par le graphe  $G_1 \square G_2$  tel que  $V(G_1 \square G_2) = V(G_1) \times V(G_2)$  et  $(u_1, u_2) \sim_{\square} (v_1, v_2)$  ssi soit (i)  $u_1 = v_1$  et  $u_2 \sim v_2$  dans  $G_2$ , soit (ii)  $u_1 \sim v_1$  dans  $G_1$  et  $u_2 = v_2$  ;
- le *produit tensoriel* (ou *produit de Kronecker*) entre  $G_1$  et  $G_2$  par le graphe  $G_1 \times G_2$  tel que  $V(G_1 \times G_2) = V(G_1) \times V(G_2)$  et  $(u_1, u_2) \sim_{\times} (v_1, v_2)$  ssi  $u_1 \sim v_1$  dans  $G_1$  et  $u_2 \sim v_2$  dans  $G_2$  ;
- le *produit fort* entre  $G_1$  et  $G_2$  par le graphe  $G_1 \boxtimes G_2$  tel que  $V(G_1 \boxtimes G_2) = V(G_1) \times V(G_2)$  et  $(u_1, u_2) \sim_{\boxtimes} (v_1, v_2)$  ssi soit (i)  $(u_1, u_2) \sim_{\square} (v_1, v_2)$ , soit  $(u_1, u_2) \sim_{\times} (v_1, v_2)$ .

**Exercice 16.13.** Pour chacun des produits ci-dessus, déterminez le nombre de sommets et d'arêtes du graphe produit en fonction du nombre de sommets et d'arêtes des graphes  $G_1$  et  $G_2$ . Implémentez ensuite une fonction prenant deux instances de `DynamicGraph` en paramètre et retournant une nouvelle instance de `DynamicGraph` correspondant au produit des inputs.

Sans grande surprise, le produit fort, qui est l'union (disjointe, notons-le) des produits cartésien et de Kronecker. Bien que nous puissions écrire une méthode `union` dans la classe `Graph`, nous préférons ici expliciter les deux parties du produit fort :



---

```
1 def strong_product(G1, G2):
2     ret = Graph(G1.V*G2.V)
3     as_id = lambda u, v : u*G1.V + v
4     for u1, v1 in G1:
5         for u2 in range(G2.V):
6             u1u2 = as_id(u1, u2)
7             v1u2 = as_id(v1, u2)
8             ret.add_edge(u1u2, v1u2)
9         for u2, v2 in G2:
10            ret.add_edge(as_id(u1, u2), as_id(v1, v2))
11            ret.add_edge(as_id(v1, u2), as_id(u1, v2))
12    for u2, v2 in G2:
13        for u1 in range(G1.V):
14            u1u2 = as_id(u1, u2)
15            u1v2 = as_id(u1, v2)
16            ret.add_edge(u1u2, u1v2)
17    return ret
```

---

## Séance 17 — Dérécursification

**Exercice 17.1.** L'algorithme d'Euclide permet de trouver le plus grand commun diviseur (noté GCD) entre deux nombres entiers par les règles suivantes :

$$\begin{cases} \text{GCD}(a, 0) &= a \\ \text{GCD}(a, b) &= \text{GCD}(b, a - b \lfloor \frac{a}{b} \rfloor) \end{cases}$$

L'algorithme d'Euclide étendu est une variante de l'algorithme d'Euclide qui permet, de calculer leur plus grand commun diviseur mais aussi deux entiers  $x$  et  $y$  tels que  $ax + by = \text{GCD}(a, b)$ .

Vérifiez que les deux algorithmes donnent bien le résultat attendu et écrivez deux fonctions dérécursiées des fonctions récursives présentées dans le fichier `GreatestCommonDivisorEmp.py` :

---

```

1 def extended_gcd_rec(a, b):
2     if b == 0:
3         return (1, 0, a) # 1*a + 0*b == a
4     else:
5         q, r = divmod(a, b) # a = q*b + r
6         x, y, g = extended_gcd_rec(b, r)
7         z = x - q*y
8         return y, z, g

```

---

**Exercice 17.2.** Écrivez une fonction dérécursiée de la fonction `quicksort` vue au cours.

**Exercice 17.3.** Écrivez une version non-récursive de l'exercice [Exercice 4.3](#) (tours de Hanoi).

## Exercices supplémentaires

**Exercice 17.4.** La fonction 91 de McCarthy est définie comme ceci :

$$M : \mathbb{N} \rightarrow \mathbb{N} : n \mapsto \begin{cases} n - 10 & \text{si } n > 100, \\ M(M(n + 11)) & \text{sinon.} \end{cases}$$

Écrivez une version récursive et une version non-récursive calculant  $M(n)$ . Quelle est la complexité de chaque approche, pouvez-vous faire mieux ?

## Séance 18 — Tris

**Exercice 18.1.** On vous demande d'implémenter une variante de l'algorithme Quicksort, appelée l'algorithme *du drapeau tricolore*, qui consiste à diviser le tableau en trois parties :

- les éléments strictement inférieurs au pivot ;
- les éléments qui sont égaux au pivot ;
- les éléments qui sont strictement supérieurs au pivot.

Même si cette partition en trois parties alourdit le traitement de la fonction partition, elle peut réduire le nombre d'appels récurifs si le tableau contient de nombreux éléments de même valeurs.

On ne trie alors que les deux sous-listes gauche et droite, on ne touche plus à la sous-liste centrale qui contient les éléments égaux au pivot).

**Exercice 18.2.** Le tri rapide peut servir à la recherche de l'élément qui serait en position  $k$  si le tableau était trié, et ce, sans devoir trier le tableau en entier.

Pour ce faire, après partition, on itère simplement sur la sous-liste contenant la position  $k$ . Il faudra toutefois veiller à maintenir la liste inchangée après l'appel de la fonction.

**Exercice 18.3.** Écrivez une fonction `quicksort(array, key)` prenant en paramètre une liste à trier et une fonction définissant la *valeur* de chaque entrée, i.e. l'appel `quicksort(array, key)` doit permuter la liste `array` de manière à ce que la liste `[key(e) for e in array]` soit triée.

**Exercice 18.4.** Écrivez une fonction `argsort(array)` qui renvoie une liste indices d'entiers telle que `[array[indices[i]] for i in range(len(array))]` est une liste triée.

**Bonus :** en quoi cet exercice est un cas particulier de l'exercice précédent ?

## Exercices supplémentaires

**Exercice 18.5.** Malgré le fait qu'en moyenne, quicksort requiert  $\mathcal{O}(n \log n)$  comparaisons, il reste quadratique dans le pire des cas. En particulier, pour toute méthode déterministe du choix du pivot, il est possible de trouver un exemple qui va nécessiter  $\Theta(n^2)$  comparaisons. Une méthode assez efficace (en pratique) de contourner le problème est de ne pas choisir le pivot de manière déterministe mais bien en le choisissant de manière uniforme. Procéder de la sorte garantit un nombre de comparaisons en  $\mathcal{O}(n \log n)$  en moyenne mais reste en  $\Theta(n^2)$  dans le pire des cas (mais les cas pathologiques sont *très rares* (pour peu que l'on puisse y associer un sens rigoureusement), ce qui permet de considérer quicksort comme étant un algorithme en  $n \log n$ ). Ceci est vu en détails au cours *Algorithmique 2* (INFO-F203) et dépasse le cadre de ce cours-ci).

Certaines implémentations préfèrent ne pas devoir générer de nombres (pseudo-)aléatoires pour des raisons d'efficacité et préfèrent choisir un pivot de manière heuristique mais déterministe. La solution classique est de choisir, comme pivot pour le sous-vecteur induit par les indices  $i$  et  $j$ , la médiane des valeurs aux positions  $i$ ,  $j$ , et  $(i+j)/2$ .

Adaptez l'algorithme quicksort vu en cours afin d'implémenter ce choix de pivot.

## Séance 19 — Programmation dynamique (partie 1)

**Exercice 19.1.** Écrivez une fonction `get_change` prenant les paramètres :

- `coins`, une liste triée des valeurs de pièces possibles;
- `value`, un entier,

et renvoyant le plus petit nombre de pièces nécessaire pour sommer à `value`. Par exemple :

```
get_change([1, 2, 5], 5) == 1
get_change([1, 3, 4], 6) == 2
get_change([1, 4], 19) == 7
```

Notez que pour garantir l'existence d'une solution, il faut que 1 soit dans la liste `coins`. Vous pouvez supposer que chaque pièce est disponible en quantité infinie.

**Bonus :** Adaptez votre solution pour qu'elle renvoie une liste contenant la quantité de chaque pièce nécessaire au lieu du nombre de pièces. Par exemple :

```
get_change([1, 2, 5], 5) == [0, 0, 1]
get_change([1, 3, 4], 6) == [0, 2, 0]
get_change([1, 4], 19) == [3, 4]
```

**Exercice 19.2.** Étant données deux chaînes de caractères `s1` et `s2`, trouvez la sous-séquence commune la plus longue. Attention : les lettres de la sous-séquence ne sont pas nécessairement contiguës dans les chaînes. Par exemple, pour `s1 = "mylittlekitten"` et `s2 = "yourlittlemitten"`, la plus longue sous-séquence commune est :

```
LCS(s1, s2) == "ylittleitten".
```

## Séance 20 — Programmation dynamique (partie 2)

**Exercice 20.1.** Étant donnée une séquence de valeurs réelles  $x = (x_1, \dots, x_n)$ , trouvez une sous-séquence contiguë de somme maximale. Par exemple, la séquence suivante :

$$x = (1, 2, 4, -4, 10, -5, -6, -12)$$

admet la sous-séquence de somme maximale suivante :

$$x' = (1, 2, 4, -4, 10).$$

**Exercice 20.2.** Trouvez la distance d'édition entre deux chaînes de caractères (aussi appelée distance de Levenshtein), c-à-d le nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre.

**Exercice 20.3.** Étant donnée une chaîne de caractères, trouvez une sous-séquence de taille maximale qui est un palindrome. Par exemple, une sous-séquence de `bacbaca` qui est un palindrome est `acbca`.

## Exercices supplémentaires

**Exercice 20.4.** Pour un entier  $n$  fixé, nous avons vu qu'il existe exactement  $2^n$  sous-ensembles de l'intervalle  $\llbracket 1, n \rrbracket = \{1, 2, \dots, n\}$ . Écrivez une fonction qui, pour  $n$  fixé, détermine la quantité  $D(n, k)$  pour chaque  $0 \leq k < n$ , à savoir combien de ces  $2^n$  sous-ensembles contiennent des éléments dont la somme est égale à  $k$  modulo  $n$ . Votre fonction doit satisfaire la signature suivante : `def D(n: int) -> list[int]`

Par exemple pour  $n = 4$ , voici les valeurs attendues pour les différentes valeurs de  $k$  :

$k$	$D(n, k)$	
0	4	$\emptyset, \{4\}, \{1, 3\}, \{1, 3, 4\}$
1	4	$\{1\}, \{1, 4\}, \{2, 3\}, \{2, 3, 4\}$
2	4	$\{2\}, \{2, 4\}, \{1, 2, 3\}, \{1, 2, 3, 4\}$
3	4	$\{3\}, \{1, 2\}, \{3, 4\}, \{1, 2, 4\}$

Donc l'appel `D(4)` doit renvoyer la liste `[4, 4, 4, 4]`.

**Exercice 20.5.** L'algorithme de Frame-Stewart permet de trouver une borne supérieure sur le nombre de déplacements à effectuer afin de résoudre le problème des tours de Hanoï à  $n$  disques et  $m$  tours. Il a été prouvé (mais nous n'en parlerons pas ici) qu'en fixant  $m$ , cette borne supérieure se comporte asymptotiquement comme  $2^{\Theta(n^{\frac{1}{m-2}})} = 2^{\Theta(\frac{n}{\sqrt[m-2]{n}})}$ , en particulier pour  $m = 3$ , on a bien  $2^{\Theta(n)}$ .

Cet algorithme fonctionne comme ceci : afin de déplacer  $n$  disques de la tour  $i$  vers la tour  $j$ , nous voulons trouver une valeur de  $k < n$  telle que nous déplaçons (i) d'abord les  $k$  plus petits disques de la tour  $i$  sur une des  $m - 2$  tours restantes (disons  $\ell$ ) ; ensuite (ii) les  $n - k$  disques restants de la tour  $i$  sur la tour  $j$  ; et finalement (iii) les  $k$  disques de la tour  $\ell$  vers la tour  $j$ .

Bien évidemment en fonction du choix de  $k$ , la quantité trouvée sera différente, et nous voulons la valeur la plus petite. Il vous est demandé de formuler précisément la relation de récurrence définie et de l'implémenter.

**Bonus 1 :** Sur base de cette borne, écrivez un algorithme qui effectue ces déplacements et qui résout le problème de Hanoï à  $m$  tours en respectant la borne.

**Bonus 2 :** Le comportement asymptotique sur  $n$  à  $m$  fixé a été mentionné ci-dessus. Quel est le comportement asymptotique de la solution optimale sur  $m$  à  $n$  fixé ?



## Chapitre 4

### Exercices cotés

**Exercice coté 1.** Soit  $n$  un entier positif. Écrivez une fonction récursive qui détermine la longueur de la plus longue séquence de zéros consécutifs dans l'écriture binaire de  $n$ .

**Exercice coté 2.** Écrivez une version récursive et non-récursive d'un parcours préfixé et suffixé d'un arbre  $d$ -aire.

**Exercice coté 3.** Soit un graphe non dirigé  $G = (V, E)$  de sommets  $V = \{v_1, \dots, v_n\}$ . Le *mycielskien* de  $G$  est le graphe  $M(G) = (V', E')$  défini par :

- $V' = V \cup \{v'_1, \dots, v'_n, w\}$ ;
- $\forall i, j \in \llbracket 1, n \rrbracket : \{v_i, v_j\} \in E' \iff \{v_i, v_j\} \in E$ ;
- $\forall i, j \in \llbracket 1, n \rrbracket : \{v'_i, v'_j\} \in E' \iff \{v_i, v_j\} \in E$ ;
- $\forall i \in \llbracket 1, n \rrbracket : \{v'_i, w\} \in E'$ .

Sans modifier les classes `DynamicUndirectedGraph` et `Vertex` suivantes, écrivez une fonction `mycielski(n)` qui renvoie le  $n$ ème graphe de Mycielski  $M^n(K_2)$  :

```

1 class Vertex:
2     def __init__(self, vertex_idx):
3         self.idx_ = vertex_idx
4         self.neighbours_ = []
5
6     def add_edge(self, v):
7         self.neighbours_.append(v)
8
9     @property
10    def idx(self):
11        return self.idx_
12
13    @property
14    def neighbours(self):
15        return iter(self.neighbours_)
16
17    @property
18    def nb_neighbours(self):
19        return len(self.neighbours_)
20
21 class DynamicUndirectedGraph:
22     def __init__(self, n):
23         self.vertices_ = [Vertex(i) for i in range(n)]
24         self.nb_edges_ = 0
25
26     @property
27     def n(self):
28         return len(self.vertices_)
29
30     @property
31     def m(self):
32         return self.nb_edges_
33
34     def add_vertex(self):
35         self.vertices_.append(Vertex(self.n))
36
37     def vertex(self, i):
38         return self.vertices_[i]
39
40     def link(self, i, j):

```

```
41     self.vertex(i).add_edge(self.vertex(j))
42     self.vertex(j).add_edge(self.vertex(i))
43     self.nb_edges_ += 1
```

---

**Exercice coté 4.** Implémentez les méthodes d'insertion et de recherche d'une table de hachage à adressage ouvert utilisant le double hachage  $h(k, j) = h_1(k) + jh_2(k)$  où  $h_1$  est la fonction K&R et  $h_2$  est la fonction DJB2.

**Exercice coté 5.** Voici une version modifiée de la solution de l'**Exercice 7.2** :

```
1 def solve_maze_rec(maze, x=0, y=0):
2     W = len(maze[0])
3     H = len(maze)
4     over = x == W-1 and y == H-1
5     if over or not (0 <= x < W) or not (0 <= y < H) or maze[y][x] !=
6         ↪ EMPTY:
7         if over:
8             maze[y][x] = '.'
9             return True
10        else:
11            return False
12    else:
13        maze[y][x] = '.'
14        for i in range(4):
15            xp, yp = x+DELTA_X[i], y+DELTA_Y[i]
16            if solve_maze_rec(maze, xp, yp):
17                return True
18        maze[y][x] = EMPTY
19        return False
```

Écrivez une version dérécursifiée de cette même fonction.

## Chapitre 5

# Anciens examens

### Session — Juin 2020

**Question d'examen.** Un *vecteur creux* est un tableau trié qui contient des paires (`idx`, `value`) où tous les `idx` sont des entiers apparaissant une unique fois, et dans l'ordre croissant, et où les `value` sont les valeurs correspondant à un indice donné. Une telle structure est très intéressante si le nombre d'entrées qui doivent être enregistrées est largement inférieur à l'indice maximal. En effet le vecteur creux suivant :

```
vec = VecteurCreux(  
    [(2, False), (3, False), (5, True), (7, False), (11, True), (13, False)]  
)
```

n'a que 6 entrées mais stocke des données d'indice jusque 13. La première composante de chaque paire est l'indice et le second est la valeur, tels que `vec[2]` renvoie `False` alors que `vec[13]` retourne `True` et `vec[9]` retourne `None` puisque l'indice 9 n'apparaît pas dans le vecteur creux.

Nous pouvons alors aisément définir une matrice creuse comme étant un tableau de références vers des vecteurs creux.

Considérons la classe `Sommet` suivante :

```
classe Sommet  
    identifiant (int) # numéro du sommet (entre 0 et n-1)  
    voisins (list) # liste de sommets  
    marque (bool) # sert de marquage pendant un parcours du graphe
```

Considérons ensuite la classe `Graphe` contenant une liste d'instances de la classe `Sommet` définie comme suit :

```
classe Graphe  
    sommets (list) # liste de sommets  
    n (int) # nombre de sommets  
    m (int) # nombre d'arêtes
```

Les trois choses suivantes vous sont demandées :

1. Complétez la classe `VecteurCreux` de ce même fichier en implémentant le constructeur et les deux méthodes suivantes :

- (a) `insérer(self, j, v)` qui correspond à l'assignation  $V[j] = v$ ;
  - (b) `rechercher(self, j)` qui renvoie la valeur associée à l'indice  $j$  si elle existe et qui renvoie `None` sinon.
2. Écrivez la fonction `distances(G)` prenant en paramètre  $G$ , une instance de `Graphe`, et qui retourne  $D$ , une instance de `MatriceCreuse` de taille  $n \times n$  (où  $n$  est le nombre de sommets de  $G$ ) dont l'entrée  $(i, d)$  est une liste contenant les identifiants de tous les sommets à distance  $d$  du sommet d'identifiant  $i$  et dont l'identifiant a la même parité que l'identifiant de  $i$  si de tels sommets existent.
- Les distances doivent se calculer en utilisant uniquement des parcours de graphe. Vous pouvez directement utiliser les attributs de la classe `Graphe` sans passer par des getters/setters pour un code plus concis. Vous n'avez cependant pas le droit d'utiliser de variables globales ni d'accéder directement aux vecteurs creux contenus dans `MatriceCreuse`.
3. Justifiez la complexité de votre approche dans un commentaire au début du fichier. Si vous pensez qu'une approche différente de celle que vous avez implémentée permettrait une meilleure complexité, mentionnez-le dans ce même commentaire, et justifiez.



## Session — Août 2021

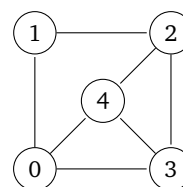
**Question d'examen** (Q3 août 2021). Soit  $G$  un graphe connexe non-dirigé d'ordre  $n$  représenté par sa matrice d'adjacence  $A$ . On appelle *cycle hamiltonien* un cycle passant une et une seule fois par chacun des sommets de  $G$  (sans toutefois obligatoirement passer par toutes les arêtes de  $G$ ). On dit qu'un graphe  $G$  est hamiltonien s'il contient au moins un cycle hamiltonien. Nous vous demandons d'écrire une fonction `hamilton(A)` qui prend une matrice d'adjacence  $A$  en paramètre et qui renvoie `True` si le graphe représenté par  $A$  est hamiltonien et `False` sinon. Vous pouvez créer d'autres fonctions si vous le souhaitez.

Exemple : le graphe  $G$  illustré ci-dessous et représenté par la matrice d'adjacence  $A$  est hamiltonien. Un cycle hamiltonien de  $G$  est par exemple :

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 0$$

$A :$

	0	1	2	3	4
0	0	1	0	1	1
1	1	0	1	0	0
2	0	1	0	1	1
3	1	0	1	0	1
4	1	0	1	1	0



**Question d'examen** (Q4 août 2021). L'exponentiation est une opération mathématique bien connue que nous pouvons également exprimer par la relation de récurrence suivante :

$$f(x) = x^n = \begin{cases} 1 & \text{si } n = 0. \\ (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair.} \\ x \cdot x^{n-1} & \text{si } n \text{ est impair} \end{cases}$$

Nous vous demandons d'écrire une **fonction non-réursive** implémentant la relation de récurrence ci-dessus permettant une telle exponentiation et de **justifier la complexité de votre approche**. Votre fonction devra prendre deux nombres ( $x$  et  $n$ ) en paramètres et renvoyer le résultat de l'exponentiation  $x^n$ . Vous pouvez supposer que  $n$  est un nombre entier  $\geq 0$ .

**Attention :** nous insistons sur le fait que votre solution doit **obligatoirement implémenter la relation de récurrence** et que toute autre proposition de solution n'implémentant pas cette relation ne sera pas considérée.

**Question d'examen** (Q5 août 2021). Tout ensemble de  $n$  éléments (avec  $n > 0$ ) peut être séparé en  $k$  sous-ensembles (avec  $0 < k \leq n$ ) disjoints (partitions) non-vides. Le nombre de tels partitionnements s'appelle nombre de *Stirling* de deuxième espèce de  $n$  en  $k$  et se note  $S(n, k)$ . Il est possible de montrer que :

$$S(n, k) = \begin{cases} 0 & \text{si } n = 0 \text{ ou } k = 0. \\ 1 & \text{si } n = 0 \text{ et } k = 0. \\ k \cdot S(n-1, k) + S(n-1, k-1) & \text{sinon} \end{cases}$$

Vous noterez que la condition du premier cas est un *ou exclusif*.

Nous vous demandons d'écrire une fonction `stirling(n)` qui prend en paramètre un entier positif  $n$  et qui calcule  $S(2n, n)$  avec une complexité de  $\mathcal{O}(n^2)$  **en utilisant la programmation dynamique**. Attention : toute proposition de solution n'utilisant pas la programmation dynamique ne sera pas considérée.

## Session — Printemps 2022

### Question d'examen.

#### Contexte

Miaouss ayant été blessé par un Pokémon sauvage, James et Jessie veulent organiser des courses relais afin de récolter des fonds pour payer les soins au centre Pokémon. Pour chaque course, l'ensemble des  $n$  Pokémon inscrits doivent être répartis en  $k$  équipes. Afin de maintenir le public en haleine, Jessie souhaite créer des équipes aussi équitables que possibles. Il est donc décidé de répartir les Pokémon en équipes en considérant leur score de vitesse.

#### Question 1

En pratique, nous utiliserons une liste  $v$ , contenant les scores de vitesse des Pokémon, et  $k$ , le nombre d'équipes désiré. Écrivez une classe, appelée `Distribution`, qui permettra à James et Jessie d'obtenir la répartition en  $k$  équipes des  $n$  Pokémon inscrits minimisant la différence maximale entre les sommes des scores des vitesses de chaque équipe. Le constructeur de la classe `Distribution` prendra deux paramètres ( $v$  et  $k$ ). Cette classe définira deux autres méthodes : `compute_max_diff` et `solve`.

- `compute_max_diff` calcule la différence maximum des sommes de chaque équipe.
- `solve` trouve et retourne la solution sous forme d'une liste de longueur  $n$  contenant le numéro de l'équipe de chaque Pokémon. `solution[i]` correspond à l'équipe du Pokémon ayant la vitesse  $v[i]$ . Les numéros d'équipe sont des entiers compris entre 0 et  $n - 1$  dont l'ordre est arbitraire (l'équipe 0 n'est pas forcément l'équipe ayant la plus petite somme). Lorsque plusieurs solutions coexistent, une seule doit être retournée.

#### Question 2

Décrivez brièvement quelle(s) modification(s) vous apporteriez à votre code pour minimiser la différence maximale entre les moyennes des scores de vitesse de chaque équipe (2-3 phrases maximum).

#### Question 3

Quelle est la complexité de cet algorithme ? Soyez le plus précis possible.

## Session — Juin 2022

**Question d'examen** (Q4 juin 2022). Il y a bien longtemps, dans une galaxie lointaine, très lointaine... c'est une époque de guerre civile où les différentes fédérations galactiques s'affrontent pour le contrôle de la galaxie. Afin de mettre un terme à cette guerre et trouver un accord entre les différentes fédérations, il vous a été demandé de trouver une solution équitable répartissant le contrôle de toutes les planètes entre les différentes fédérations. Pour mener à bien cette mission, il vous est demandé de trouver une solution qui consiste à affecter une fédération à chaque planète de sorte que deux planètes adjacentes n'appartiennent pas à la même fédération.

Nous vous demandons décrire une classe `NouvelEspoir` dont le constructeur prend en paramètre :

- $G$ , un graphe connexe non-dirigé d'ordre  $n$  dont les sommets représentent les planètes de la galaxie et dont chaque arête représente l'adjacence entre deux planètes ;
- $k$ , le nombre total de fédérations galactiques.

Votre classe `NouvelEspoir` doit contenir une méthode `save_galaxy` qui renvoie la première solution valide au problème d'affectation. Aucune contrainte supplémentaire n'est imposée concernant la validité d'une solution (par exemple, il se pourrait qu'aucune planète ne soit affectée à une ou plusieurs fédérations). Si, par malheur, aucune solution possible n'a pu être trouvée, votre méthode doit également afficher un message correspondant.

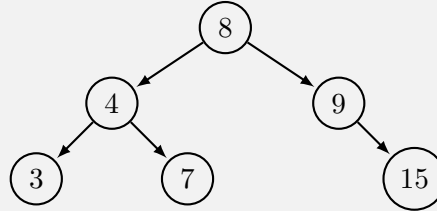
Vous pouvez créer d'autres méthodes au sein de la classe `NouvelEspoir` si nécessaire. Vous pouvez importer et utiliser les classes vues lors des cours théoriques et des travaux pratiques sans les réécrire mais en précisant clairement (en commentaire) leur origine ainsi que la complexité des opérations de base associées.

En outre, nous vous demandons également de donner et justifier la complexité de l'algorithme résolvant ce problème d'affectation.

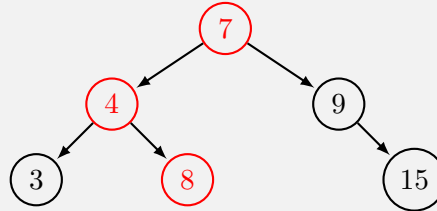
**Question d'examen** (Q5 juin 2022). Implémentez une fonction inversant les valeurs de tous les nœuds entre la racine et un nœud donné d'un arbre binaire de recherche (binary search tree, BST en anglais). Vous ne pouvez pas utiliser de variables globales. Cette fonction s'appellera `reverse_path` et prendra deux paramètres :

- `root`, la racine de l'arbre binaire de recherche de type `BinaryTree` (donc sans référence vers le sommet parent) ;
- `data`, la valeur du nœud jusqu'auquel on veut inverser le chemin.

On considère l'arbre binaire de recherche T illustré ci-dessous.



L'appel à la fonction `reverse_path(T, 7)` **modifiera** l'arbre tel que le chemin jusqu'au nœud de valeur 7 inclus soit inversé.



À noter que l'arbre binaire de recherche, une fois modifié n'en sera plus nécessairement un.

## Session — Août 2022

**Question d'examen** (Q3 Août 2022). Adaptez l'algorithme de tri fusion (*merge sort*) de manière à ce qu'il fonctionne sur une liste simplement chaînée (et non sur un vecteur). Nous voulons également que la liste chaînée triée retournée ne contienne chaque nombre qu'une seule fois, peu importe le nombre d'occurrences dans le tableau initial. Écrivez une fonction `mergesort_ll(l: Node) -> Node` qui implémente votre adaptation. Attention : vous devez vous assurer qu'aucun doublon n'est présent à *chaque étape* de l'algorithme et pas une unique fois à la fin. Vous pouvez ajouter des paramètres supplémentaires à cette fonction et vous pouvez également écrire d'autres fonctions si nécessaire. Vous n'avez cependant pas le droit d'utiliser de variables globales et vous ne pouvez pas convertir votre liste chaînée en vecteur.

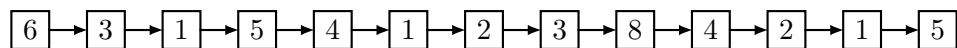
La liste chaînée est implémentée via l'utilisation de la classe `Node` suivante à laquelle vous ne devez pas ajouter des getters/setters/propriétés mais **que vous ne pouvez pas modifier** :

---

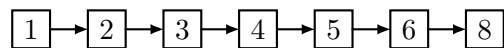
```
class Node:
    def __init__(self, key, next=None):
        self.key = key
        self.next = next
```

---

Par exemple, prenons la liste chaînée suivante :



Le résultat attendu de la fonction `mergesort_ll` est le suivant :



**Question d'examen** (Q4 Août 2022). Soit  $G = (V, E)$  un graphe connexe non-dirigé d'ordre  $n$ . La distance entre deux sommets  $u$  et  $v$  de  $G$  est le nombre d'arêtes parcourues par un plus court chemin entre  $u$  et  $v$ . L'excentricité d'un sommet  $v$ , notée  $\epsilon(v)$ , est la plus grande distance entre ce sommet un n'importe quel autre sommet de  $G$ , i.e.:

$$\epsilon(v) = \max_{u \in V} d(u, v).$$

Le rayon  $r$  d'un graphe  $G$  est l'excentricité minimale de n'importe quel sommet  $v$  de  $G$ , i.e.:

$$r = \min_{v \in V} \epsilon(v) = \min_{v \in V} \max_{u \in V} d(u, v).$$

Nous vous demandons d'écrire deux fonctions :

- `excentricite(G, v)` qui prend en paramètre un graphe  $G$  et un sommet  $v \in V(G)$  et qui retourne  $\epsilon(v)$ ;
- `rayon(G)` qui prend en paramètre un graphe  $G$  et qui retourne le rayon de  $G$ .

Nous vous demandons également de donner et de justifier la complexité du calcul du rayon de  $G$ .

Informations importantes : le graphe  $G$  est obligatoirement représenté par une structure dynamique. Vous pouvez créer d'autres fonctions si nécessaire. Vous pouvez importer et

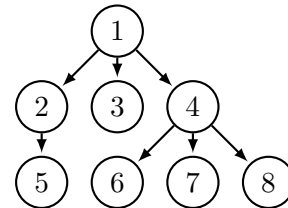
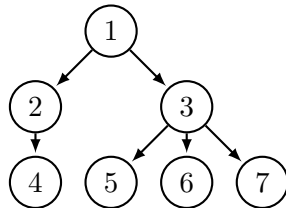
utiliser les classes vues au cours théorique ou lors de séances d'exercices sans les réécrire mais en précisant clairement leur origine ainsi que la complexité des opérations associées.

Conseil : calculer l'excentricité de  $v$  en calculant individuellement les distances entre chaque paire de sommets  $(u, v)$  n'est pas l'approche la plus efficace.

## Session — Printemps 2023

**Question d'examen** (Interrogation de printemps 2023). Considérons un arbre  $T$  enraciné à  $n$  nœuds. Chacun de ces nœuds peut avoir un nombre arbitraire de fils. Nous disons que  $T$  est *équilibré* lorsque la longueur du chemin entre la racine et toutes les feuilles a la même longueur.

Par exemple parmi les deux arbres ci-dessous, celui de gauche est bien équilibré puisque tout chemin entre la racine et une feuille a longueur 2 alors que dans l'arbre de droite, il existe une feuille de hauteur 1 alors que les autres ont hauteur 2 :



Écrivez une classe `Tree` représentant un arbre dont le nombre de fils de chaque nœud peut être quelconque. Attention, il ne vous est pas permis de maintenir une référence vers le nœud parent !

Écrivez ensuite une fonction récursive `is_balanced(tree: Tree) -> tuple[bool, int]` qui prend en paramètre une instance de cette classe et qui renvoie un tuple contenant un booléen et un entier. Ce booléen doit valoir `True` si et seulement si l'arbre `tree` est équilibré et `False` sinon. L'entier doit quant à lui contenir la longueur maximale entre la racine et une feuille.

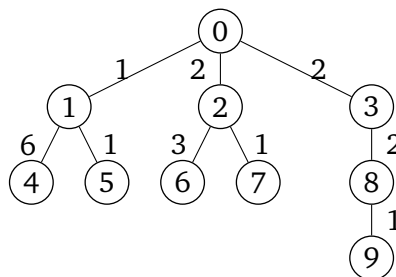
Sur les arbres ci-dessus, la fonction `is_balanced` doit renvoyer respectivement `(True, 2)` et `(False, 2)`.

Donnez la complexité de votre approche et écrivez-en une brève justification.



## Session — Juin 2023

**Question d'examen** (Q3 juin 2023). Partant d'un arbre dont les arrêtes sont pondérées, on peut calculer la matrice de distances  $M$ , où  $M_{i,j}$  donne la distance entre les sommets  $i$  et  $j$ . Cette distance correspond à la somme des poids des arêtes sur l'unique chemin qui relie  $i$  et  $j$ . Dans l'exemple ci-dessous, l'unique chemin entre 3 et 4 est  $P = (3, 0, 1, 4)$ , dont la somme des poids est  $2 + 1 + 6 = 9$ . On a donc  $M_{3,4} = 9$ .



Utilisez l'interface définie par les classes suivantes pour manipuler un arbre. Vous ne pouvez pas modifier ces classes, mais vous pouvez utiliser directement leurs attributs, sans passer par des getters ou setters. Vous pouvez considérer qu'un arbre utilisant cette interface existe déjà.

---

```

1 class Arete:
2     def __init__(self, poids, destination):
3         self.poids = poids
4         self.destination = destination
5
6 class Sommet:
7     def __init__(self, idx, areteParent=None):
8         self.idx = idx
9         self.areteParent = areteParent
10        self.arettesEnfants = []

```

---

1. Écrivez une fonction `distances_sommets(racine, n)` qui crée et renvoie la matrice des distances  $M$  pour un arbre enraciné au sommet `racine` et contenant `n` sommets. Cette fonction doit être la plus efficace possible en terme de complexité temporelle. Pour cela, utilisez la programmation dynamique et un parcours de l'arbre depuis sa racine.
2. Expliquez la complexité en temps de votre approche et justifiez qu'elle est optimale.

### Question d'examen (Q4 juin 2023 — Énumération des $k$ -uplets de Ramanujan). Contexte

Le mathématicien Srinivasa Ramanujan, un des très grands noms des mathématiques, en particulier de la théorie des nombres, a étudié une séquence de nombres particulière que nous regarderons dans cette question.

Nous définissons le nombre  $T(k)$  comme étant le plus petit nombre qui peut s'écrire comme une somme de 2 cubes d'au moins  $k$  manières différentes. Cette suite (qui croît très rapidement) est appelée suite *Taxicab*.

$k$	$T(k)$	
1	2	$= 1^3 + 1^3$
2	1 729	$= 1^3 + 12^3 = 9^3 + 10^3$
3	87 539 319	$= 228^3 + 423^3 = 167^3 + 436^3 = 255^3 + 414^3$
4	$\simeq 6\,963 \times 10^9$	
5	$\simeq 49 \times 10^{15}$	

Le tableau ci-dessus montre les premières valeurs de cette suite :

Nous noterons ici  $T(k, n)$  le  $n$ ème nombre (dans l'ordre croissant) ayant cette propriété. En particulier  $T(k, 1) = T(k)$  puisque c'est le premier nombre dans l'ordre croissant qui peut s'écrire comme la somme de deux cubes d'au moins  $k$  manières différentes ;  $T(k, 2)$  est le deuxième plus petit nombre pouvant s'écrire comme la somme de deux cubes d'au moins  $k$  manières distinctes ;  $T(k, 3)$  est le troisième plus petit nombre pouvant s'écrire d'une telle manière, etc. Le tableau suivant donne les premières valeurs des suites  $T(2, n)$  et  $T(3, n)$  :

$n$	$T(2, n)$	$T(3, n)$
1	1 729	87 539 319
2	4 104	119 824 488
3	13 832	143 604 279
4	20 683	175 959 000
5	32 832	327 763 000

## Questions

### 1. Adaptation de heap

Un *heap* (ou file à priorité) requiert une relation d'ordre sur ses éléments (il faut pouvoir déterminer sur toute paire d'éléments si l'un est plus grand que l'autre). Expliquez comment modifier l'implémentation (cela inclut potentiellement la modification des signatures de méthodes) de la classe `Heap` de manière à pouvoir spécifier une fonction `key` qui détermine la relation d'ordre, i.e. la racine contient l'élément  $x$  tel que `key(x)` est minimal. Notez que ce fonctionnement est similaire au paramètre `key` de la fonction `sorted` ou encore `list.sort`.

La complexité des opérations usuelles sur un heap est-elle modifiée ? Justifiez votre réponse.

### 2. Énumération des $T(k, n)$

En utilisant un tel heap, écrivez une fonction `taxicab(k: int, N: int) -> None` qui affiche tous les nombres  $T(k, n)$  dans l'ordre croissant et tels que  $T(k, n) \leq N$ . Faites bien attention à ne pas générer plusieurs fois la même paire de cubes !

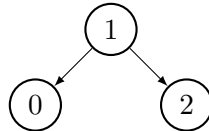
### 3. Complexité

Que pouvez-vous dire de la complexité de votre approche ? Justifiez votre réponse.

## Session — Août 2023

**Question d'examen** (Q3 Août 2023). Considérons un arbre binaire de recherche (BST)  $T$  de racine  $r$ . Soient deux nœuds  $x$  et  $y$  de  $T$ . Nous disons que  $x$  est un *ancêtre* de  $y$  dans  $T$  si  $x$  se trouve sur l'unique chemin entre  $y$  et  $r$ . Si  $x$  est un ancêtre de  $y$ , nous disons également que  $y$  est un *descendant* de  $x$ . En particulier la racine est un ancêtre de tous les nœuds ; et tout nœud est ancêtre de lui-même.

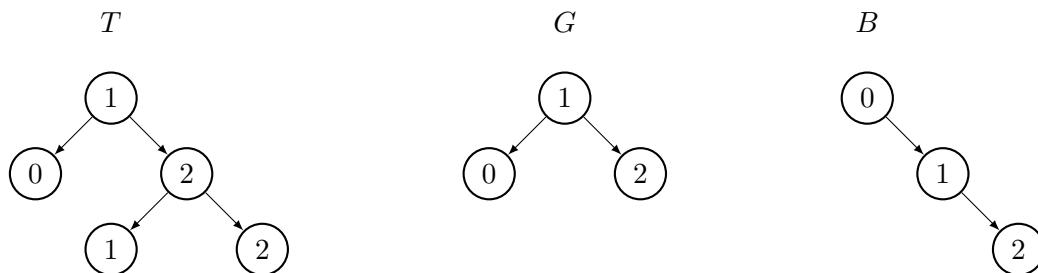
Par exemple sur l'arbre suivant, 1 est un ancêtre de 0 et de 2, mais 0 n'est pas un ancêtre de 2.



Écrivez une fonction `remove_duplicates(tree: BinaryTree, *) -> BinaryTree` qui prend en paramètre un BST  $T$  et qui renvoie un *autre* BST  $T'$  tel que :

1.  $T'$  contient les mêmes valeurs que  $T$  ;
2.  $T'$  ne contient chaque valeur qu'une unique fois ;
3. si  $x$  est un ancêtre de  $y$  dans  $T'$ , alors il existe un  $x$ , ancêtre d'un  $y$  dans  $T$ .

La figure suivante donne un exemple de BST  $T$  et la solution attendue (notée  $G$  pour *good*) ainsi qu'un exemple de solution non-valide (notée  $B$  pour *bad*) :



En effet, bien que l'ensemble des valeurs apparaissant dans les arbres soient  $\{0, 1, 2\}$  à chaque fois, les seules relations d'ancêtre dans  $G$  sont 1 et 0 ainsi que 1 et 2 qui existent toutes deux également dans  $T$ . Par contre dans  $B$ , les relations d'ancêtre sont 0 et 1 ; 1 et 2 ainsi que 0 et 2. Or 0 n'est pas un ancêtre de 2 dans  $T$ . Cette solution n'est donc pas acceptable.

De plus votre fonction doit s'exécuter en temps linéaire en le nombre de nœuds de  $T$ . Justifiez cette complexité ainsi que la validité de votre solution. Vous avez le droit d'ajouter des paramètres à la fonction `remove_duplicates` mais pensez bien à expliquer pourquoi vous en avez besoin. Vous pouvez supposer que toutes les valeurs stockées dans les nœuds sont des entiers.

**Question d'examen** (Q4 août 2023). Riley et Eric souhaitent planifier leur prochaine tournée légendaire et, connaissant vos talents pour l'algorithmique, ils font appel à vous afin de déterminer l'itinéraire qui leur permettra de jouer dans un maximum de villes, devant un maximum de fans en un nombre limité de jours et dans un budget fixé.

Pour mener à bien votre tâche, nous vous demandons d'écrire une classe `Tournee` dont le constructeur prend en paramètre :

- $G$  : un graphe dynamique, connexe et dirigé d'ordre  $n$  dont les sommets représentent des villes, possédant chacune (i) un *nombre de fans* ainsi que (ii) un *coût d'organisation d'un concert*, et les arêtes sont pondérées par le *coût du trajet* entre les 2 sommets qu'elles relient ;
- `depart` : l'*indice* du sommet de départ de la tournée ;
- `longueur_tournee` : la longueur *maximale* de la tournée ;
- `budget` : le budget *maximal* de la tournée.

Votre classe `Tournee` doit contenir une méthode `trouver_itineraire` qui détermine l'itinéraire qui maximise le nombre de fans pouvant assister aux concerts tout en ne dépassant pas le budget fixé ainsi que la longueur maximale du séjour. De plus, votre itinéraire doit obligatoirement commencer et se terminer au sommet de départ passé en paramètre au constructeur de la classe `Tournee`.

En outre, votre méthode `trouver_itineraire` doit renvoyer les éléments suivants :

- l'itinéraire sous forme d'une liste Python contenant les indices des villes, dans l'ordre de visite ;
- le nombre total de fans pouvant assister aux concerts ;
- le budget total de l'itinéraire ,
- la longueur totale de l'itinéraire.

**Précisions importantes** : un concert ne peut être organisé qu'*une seule fois* dans une ville, même si l'itinéraire passe plusieurs fois par celle-ci (par exemple, il se pourrait que l'on doive repasser par une ville déjà visitée pour pouvoir revenir à la ville de départ). L'organisation d'un concert dans une ville ne dure qu'un jour (trajet vers la prochaine ville inclus). Aucun concert n'est organisé dans la ville de départ (pour simplifier, nous prendrons l'hypothèse que le nombre de fans et que le coût d'organisation d'un concert dans la ville de départ sont fixés à 0).

De plus, nous vous demandons également de *donner et justifier la complexité de l'algorithme* résolvant ce problème.

Pour vous simplifier l'implémentation, nous vous demandons de vous baser sur la maquette de la classe `Graph` suivante :

---

```

1 class Graph:
2     def __init__(self, n: int):
3         # ...
4
5     def voisins_de(self, v: int) -> list[int]:
6         # retourne la liste des indices des sommets reliés à v
7
8     def nombre_de_fans_dans(self, v: int) -> int:
9         # retourne le nombre de fans dans v
10
11    def cout_concert_dans(self, v: int) -> int:
12        # retourne le coût de l'organisation d'un concert dans v
13
14    def cout_trajet_entre(self, v: int, w: int) -> int:
15        # retourne le coût du trajet entre v et w
16
17    def __len__(self) -> int:
18        # retourne le nombre de sommets du graphe

```

---

Vous pouvez faire appel aux méthodes de cette maquette de la classe Graph dans votre algorithme sans devoir la recopier.

Vous pouvez également créer d'autres méthodes au sein des classes Tournee et Graph si nécessaire. Veillez cependant à le mentionner clairement dans votre réponse. Vous pouvez importer et utiliser les classes vues lors des cours théoriques et des travaux pratiques sans les réécrire mais en précisant clairement (en commentaire) leur origine ainsi que la complexité des opérations de base associées.