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 2022-2023

Table des matières

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 premier chapitre 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 3), la notation $\mathcal{O}$ de Landau et ses variations (c.f. la section 5), ou encore sur les méthodes de lecture et d'écriture de preuves en mathématique (c.f. la section 2). La section 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 second chapitre 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 troisième chapitre 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 quatrième et dernier chapitre 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, retenez 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. Dès lors, soyez curieux, soyez curieuses, et aimez l'algorithmique.

Chapitre 1

Rappels, notations, prérequis et digressions

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) :

```
class Node:
    def __init__(self, next_node, data):
        self.next_node = next_node
        self.data = data
```

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

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

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.

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. □

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 $z_n \xrightarrow[n\to+\infty]{} x + y$.

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 :

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

□

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*

négation, 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 $P \vee \neg P$ [a]).

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

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 :

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

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 $b_0, \ldots, b_{p-1} \in \{0, 1\}$ (ce qui est donc sa représentation en base 2). Sur base de cette expression, définissons la représentation de n par $c_0, \ldots, c_p \in \{0, 1\}$ 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).*

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. □

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 ?*

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 ≥ 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. □

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 et 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. □

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

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).

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 jargonneux.

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'infos, 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.

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 :

```
from datetime import datetime

start = datetime.now()

def log_to_stdout(function):
    def wrapper(*args, **kwargs):
        offset = datetime.now() - start
        print(f'[{offset}] called function "{function.__name__}"')
        return function(*args, **kwargs)
    return wrapper
```

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

```
@decorator_function
def wrapped_function(...):
    ...
```

qui en réalité correspond à :

```
def wrapped_function(...):
    ...
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` :

```
@log_to_stdout
def print_sum(array):
    print(sum(array))

@log_to_stdout
def main():
    array = []
    for _ in range(5):
        array.append(int(input('Entier: ')))
    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 :

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

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 :

```
def decorator(param1, param2, ...):
    def decorator(func):
        def wrapper(*args, **kwargs):
            ... # utilise les paramètres de decorator
            ret = func(*args, **kwargs)
            ... # utilise les paramètres de decorator
            return ret
        return wrapper
    return decorator
```

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

```
import functools
def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        ...
    return wrapper
```

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.

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'expliciter 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 section 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) :

```
class Time:
    def __init__(self, h=0, m=0, s=0):
        self.hour_ = h
        self.minute_ = m
        self.second_ = s

    def tick(self):
        self.second_ += 1
        if self.second_ == 60:
            self.second_ = 0
            self.minute_ += 1
        if self.minute_ == 60:
            self.minute_ = 0
            self.hour_ += 1
        if self.hour_ == 24:
            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 :

```
    def get_hour(self):
        return self.hour_
    def get_minute(self):
        return self.minute_
    def get_second(self):
        return self.second_

    def set_hour(self, new_h):
        if 0 <= new_h < 24:
            self.hour_ = new_h
    def set_minute(self, new_m):
        if 0 <= new_m < 60:
            self.minute_ = new_m
    def set_second(self, new_s):
        if 0 <= new_s < 60:
            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 section 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 !

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
```

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 section 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.

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 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 :

```
    def __len__(self):
        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`) :

```
for b in (8, 1, 0, -2):
    I = IntegerInterval(0, b)
    print(repr(I))
    print(I)
    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 section 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) [1] 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.

```python
def gcd(a, b):
    while b > 0:
        a, b = b, a%b
    return a

class Rational:
    def __init__(self, a=0, b=1):
        self.num_ = a
        self.den_ = b
        self.reduce()

    def reduce(self):
        d = gcd(self.num_, self.den_)
        self.num_ //= d
        self.den_ //= d

    def __str__(self):
        return f'{self.num_} / {self.den_}'

    def __add__(self, other):
        num = self.num_ * other.den_ + other.num_ * self.den_
        den = self.den_ * other.den_
        return Rational(num, den)

if __name__ == '__main__':
    x = Rational(3, 5)
    y = Rational(1, 10)
    z = x+y
    print(type(z))
    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 différente.

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` :

```
class Rational:
    def __init__(self, a=0, b=1):
        self.num_ = a
        self.den_ = b
        self.reduce()

    def reduce(self):
        d = gcd(self.num_, self.den_)
        self.num_ //= d
        self.den_ //= d

    def __str__(self):
        return f'{self.num_} / {self.den_}'

    def __add__(self, other):
        if isinstance(other, int):
            return Rational(self.num_ + other * self.den_, self.den_)
        elif isinstance(other, Rational):
            num = self.num_ * other.den_ + other.num_ * self.den_
            den = self.den_ * other.den_
            return Rational(num, den)
        else:
            return NotImplemented

    def __radd__(self, other):
        return self + other # notre somme est commutative

if __name__ == '__main__':
    x = Rational(3, 5)
    y = Rational(1, 10)
    print(f'({x}) + ({y}) == ({x + y})')
    print(f'({x}) + 1 == ({x + 1})')
    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 :

```
def addition(a, b):
   ret = NotImplemented
   if hasattr(a, '__add__'):
      ret = a.__add__(b)
   if ret is NotImplemented:
      if hasattr(b, '__radd__'):
         ret = b.__radd__(a)
   if ret is NotImplemented:
      raise TypeError(
         f"unsupported operand type(s) for +: '{type(a)}' and '{type(b)}'"
      )
   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) :

```
temp = addition(x, y)
del x
x = temp
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 section 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 section 3.11. Voici un exemple d'implémentation de cette méthode sur la classe `Rational` :

```
class Rational:
    def __init__(self, a=0, b=1):
        self.num_ = a
        self.den_ = b
        self.reduce()

    def reduce(self):
        d = gcd(self.num_, self.den_)
        self.num_ //= d
        self.den_ //= d

    def __str__(self):
        return f'{self.num_} / {self.den_}'

    def __add__(self, other):
        if isinstance(other, int):
            return Rational(self.num_ + other * self.den_, self.den_)
        elif isinstance(other, Rational):
            num = self.num_ * other.den_ + other.num_ * self.den_
            den = self.den_ * other.den_
            return Rational(num, den)
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, int):
            self.num_ += other*self.den_
        elif isinstance(other, Rational):
            self.num_ = self.num_ * other.den_ + other.num_ * self.den_
            self.den_ *= other.den_
            self.reduce()
        else:
            return NotImplemented
        return self # très important

    def __radd__(self, other):
        return self + other # notre somme est commutative

if __name__ == '__main__':
    x = Rational(3, 5)
    y = Rational(1, 10)
    print(x)
    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 méthoe `__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 oéprateurs `:-` (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'*indiçage* 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):
```

```
        del self.params_[i]

if __name__ == '__main__':
    callback = Callback(print)
    callback.add_param(0)
    # ah non je ne veux pas afficher 0
    del callback[0] # appelle callback.__delitem__(0)
    callback.add_param(1)
    # oups, je voulais dire -1
    callback[0] = -1 # appelle callback.__setitem__(0, -1)
    callback.add_param(2)
    # c'était quoi le premier paramètre encore ?
    print(callback[0]) # appelle callback.__getitem__(0)
    callback() # et on affiche le tout
```

Nous pouvons également mentionner la méthode spéciale `__contains__` qui permet d'utiliser l'opérateur `in` : `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. section 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 :

```
import functools
@functools.total_ordering
class Rational:
    #...
    def __eq__(self, other):
        return self.num_ * other.den_ == self.den_ * other.num_

    def __lt__(self, other):
        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 com-

paraison. 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__`.

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 :

```
class C:
    def __init__(self):
        self.__x = 1

    @property
    def x(self):
        return self.__x

    @property
    def twice_x(self):
        return self.__x*2

c = C()
assert c.x == 1 # appelle la propriété x
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 :

```
class C:
    def __init__(self):
        self.__x = 1

    @property
    def x(self):
        return self.__x

```

```
    @property
    def twice_x(self):
        return self.__x*2

    @twice_x.setter
    def twice_x(self, two_x):
        self.__x = two_x // 2

c = C()
assert c.x == 1
c.twice_x = 4
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 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 :

```
class Solver:
    def __init__(self):
        ...
        self.solution_ = None

    def solve(self):
        ...
        self.solution_ = ...

    @property
    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'expliciterons pas le sujet ici.

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`. La comportement de la fonction `iter` se paramétrise via la méthode spéciale `__iter__` (c.f. la section 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 `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-même 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 parmètre 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 commençant 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 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 section 3.8) `__reversed__`. Par exemple la méthode `list.__reversed__` peut ressembler à ceci :

```
def __reversed__(self):
    for i in range(len(self)-1, -1, -1):
        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 2.7 pour plus d'informations.

3.11 Adresses et cache de références interne à CPython

Cette section est importante pour la section 12.1 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 ambigüité 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 :

```
for i in range(1000000):
    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 :

```
>>> a = 1
>>> old_id = id(a)
>>> a += 1
>>> new_id = id(a)
>>> old_id == new_id
False
```

La raison est la suivante : les entiers sont *imuuables*. 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 équi-

valent 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 :

```
>>> a = 'a'
>>> old_id = id(a)
>>> a += a
>>> new_id = id(a)
>>> old_id == new_id
False
```

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

```
>>> a = 'a'
>>> b = (a+a)[len(a):]
>>> a == b
True
>>> a is b
True
>>> a = 'abcdefghijklmnopqrstuvwxyz'
>>> b = (a+a)[len(a):]
>>> a == b
True
>>> a is b
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 du 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 fonctionnement 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 cecla, 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 opérandes 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.

4 Notions ensemblistes

Définition 3. Nous utilisons les notation suivantes pour désigner les intervalles entiers (pour $a, b \in \mathbb{Z}$) :
- $[\![a, b]\!] \coloneqq \{m \in \mathbb{Z} \text{ s.t. } a \leq m \leq b\} = \{a, a+1, \dots, b-1, b\}$;
- $[\![a, b[\![\coloneqq \{m \in \mathbb{Z} \text{ s.t. } a \leq m < b\} = \{a, a+1, \dots, b-1\}$;
- $]\!]a, b]\!] \coloneqq \{m \in \mathbb{Z} \text{ s.t. } a < m \leq b\} = \{a+1, \dots, b-1, b\}$;
- $]\!]a, b[\![\coloneqq \{m \in \mathbb{Z} \text{ s.t. } a < m < b\} = \{a+1, \dots, b-1\}$.

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

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

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

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

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 $E \cap F \coloneqq \{x \in E \text{ s.t. } x \in E \text{ et } x \in F\}$;
- l'*union* de E et F, notée $E \cup F$, par l'ensemble $E \cup F \coloneqq \{x \text{ s.t. } x \in E \text{ ou } x \in F\}$;
- 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 $\varphi : E \to F$; $|E| \geq |F|$ s'il existe une application surjective $\phi : E \to F$ et $|E| = |F|$ s'il existe une application bijective $\psi : E \to F$. On peut même uniquement se réduire au cas $|E| \leq |F|$ lorsqu'il existe une application injective $E \to F$ puisque l'existence d'une telle application implique l'existence d'une application surjective $\phi : F \to E$ (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 $\phi : E \to E$ et $\psi : F \to E$ sont injectives, alors il existe $\theta : E \to F$ bijective.*

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

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

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 $\theta : E \to F : x \mapsto \theta(x)$ 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), $\phi\big|_{[x]_\sim}$ est une bijection donc $\theta(x) = \phi(x)$ et dans le cas (b), $\psi^{-1}\big|_{[x]_\sim}$ 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 θ est bien définie sur E, et par construction θ est bien inversible. □

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 $E = \{x_1, \ldots, x_n\}$ et $F = \{y_1, \ldots, y_m\}$ sont eux ensembles disjoints, alors :

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

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 :

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

□

5 Notions asymptotiques de Landau

Définition 7. Soient deux fonctions $f, g : \mathbb{N} \to \mathbb{R}$. 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 $f, g : \mathbb{N} \to \mathbb{R}$, $f = \mathcal{O}(g)$ 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 ?) :

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

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

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

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

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

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

Proposition 1.11. *Soient $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^+$ 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 :

$$\forall n > N_1 : f_1(n) \leq C_1 g_1(n),$$
$$\forall n > N_2 : f_2(n) \leq C_2 g_2(n).$$

Posons alors $N \coloneqq \max\{N_1, N_2\}$. 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 $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}^+$ 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 Θ, 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 $f, g, h : \mathbb{N} \to \mathbb{R}^+$ 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 :

$$\forall n > N_f : f(n) \leq C_f \cdot g(n),$$
$$\forall n > N_g : g(n) \leq C_g \cdot h(n).$$

Posons alors $N \coloneqq \max\{N_f, N_g\}$. 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 mnemonique = input('Quelle est la mnémonique du meilleur TP de l\'ULB ?')
2 if mnemonique == '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 `mnemonique`). Il est cependant *techniquement correct* de dire que le nombre d'opérations est $\mathcal{O}(2^n)$ où n `== len(mnemonique)`. 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 $f, g, h : \mathbb{N} \to \mathbb{R}_0^+$ arbitrairement.

La réflexivité de $\sim$ est assez claire :

$$\frac{f(n)}{f(n)} = 1 \xrightarrow[n\to+\infty]{} 1.$$

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 $g \sim h$:

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

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 $f \sim g$:

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

□

Proposition 1.15. *Θ 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 Ω. □

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 Θ

5.1 Quelques ordres de grandeur

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

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

pour un certain $n \in \mathbb{N}$ et $a_0, \ldots, a_n \in \mathbb{R}$, 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 $P \sim [x^n]Px^n$.*

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

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

□

Remarque. *Cette proposition implique également que tout polynôme P satisfait $P = \Theta(x^{\deg P})$ 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 $m \lneqq n$. Alors $x^m = o(x^n)$.*

Démonstration. Trivial :

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

puisque $n - m \gneqq 0$. □

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 :

$$\frac{x^0}{e^x} = e^{-x} \xrightarrow[x\to+\infty]{} 0.$$

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

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

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'entièreté 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 :

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

par hypothèse de récurrence. □

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 $\mathcal{O}$, Ω, 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 $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ 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 Θ 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 Θ (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 Θ 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.

7 Considérations arithmétiques

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

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

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

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 :

$$\int_1^2 \frac{\mathrm{d}x}{x} \leq 1 \leq 1$$

En sommant sur k, nous obtenons :

$$\begin{aligned}
\sum_{k=1}^{n} \int_k^{k+1} \frac{\mathrm{d}x}{x} \leq & \quad \sum_{k=1}^{n} \frac{1}{k} & \leq 1 + \sum_{k=2}^{n} \int_{k-1}^{k} \frac{\mathrm{d}x}{x} \\
\int_1^{n+1} \frac{\mathrm{d}x}{x} \leq & \quad H_n & \leq 1 + \int_1^n \frac{\mathrm{d}x}{x} \\
\log(n+1) \leq & \quad H_n & \leq 1 + \log n \\
\log n + \log \frac{n+1}{n} \leq & \quad H_n & \leq 1 + \log n \\
\log \frac{n+1}{n} \leq & H_n - \log n \leq 1.
\end{aligned}$$

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

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 γ (≈ 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 $(1-q)S_n$:

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

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 :

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

□

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 :

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

□

Proposition 1.25. *Soient $\alpha, \beta \in \mathbb{R}$ tels que $\alpha \gneqq 1$ 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 $k \in [\![0, n]\!]$. 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 :

$$\log n! = \sum_{k=1}^{n} \log k = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} \log k + \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} \log k \geq \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} \log k$$

$$\geq \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} \log \frac{n}{2} = (n - \left\lfloor \frac{n}{2} \right\rfloor) \log \frac{n}{2} \sim \frac{n}{2} \log n,$$

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* $\varphi : \mathbb{R}^+ \to \mathbb{R}^+$ *une fonction croissante et intégrable. Définissons* Φ *et* F *par :*

$$\Phi : \mathbb{R}_0^+ \to \mathbb{R}^+ : x \mapsto \int_0^x \varphi(t) \mathrm{d}\, t$$

$$F : \mathbb{N} \to \mathbb{R}^+ : n \mapsto \sum_{k=0}^{n} \varphi(k).$$

Alors pour tout $n \in \mathbb{N}$*,* φ, Φ *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* Φ_1 *et* Φ_2 *sont deux primitives de* Φ*, alors* $\Phi_1 = \Theta(\Phi_2)$ *(et même* $\Phi_1 \sim \Phi_2$ *si* $\lim_{x \to +\infty} \varphi(x) > 0$*, i.e. si* Φ_1 *et* Φ_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 :

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

Par monotonie de φ :

$$F(n) \leq \sum_{k=0}^{n} \int_{k}^{k+1} \varphi(x)\mathrm{d}x = \int_{0}^{n+1} \varphi(x)\mathrm{d}x = \Phi(n+1).$$

mais également :

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

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* $\frac{\Phi(n+1)}{\Phi(n)} \xrightarrow[n\to+\infty]{} 1$*, 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 Φ (qui vient de la positivité de φ) :

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

et donc $F \sim \Phi$. □

Théorème 1.30. *Sous les mêmes hypothèses que le théorème précédent, si* φ *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* $n \geq 1$ *:*

$$\Phi(n-1) = \int_{0}^{n-1} \log(x+1)\mathrm{d}x = \int_{1}^{n} \log x\mathrm{d}x = [x(\log x - 1)]_{1}^{n} = n(\log n - 1),$$

ainsi que :

$$\frac{\Phi(n)}{\Phi(n-1)} = \frac{n\log(n+1) - n + \log(n+1) - 1}{n(\log n - 1)} = 1 + \frac{n\log\left(1+\frac{1}{n}\right) + \log(n+1) - 1}{n(\log n - 1)}$$
$$\leq 1 + \frac{n\log 2 + n - 1}{n(\log n - 1)} = 1 + \frac{1 + \log 2 - \frac{1}{n}}{\log n - 1} \xrightarrow[n\to+\infty]{} 1.$$

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

Démonstration. Regardons ici uniquement le cas où φ est asymptotiquement croissante. Considérons une valeur $N > 0$ à partir de laquelle φ est croissante. Notons $\overline{\varphi}(k) \coloneqq \varphi(n+N)$ la fonction φ décalée (qui est donc croissante sur $\mathbb{R}^+$), $\overline{F}(n)$ la somme des $\overline{\varphi}(0)$ jusque $\overline{\varphi}(n)$ et $\overline{\Phi}$ l'intégrale de $\overline{\varphi}$. Par le théorème 1.28, nous avons :

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

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

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

ainsi que :

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

i.e. :

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

Dès lors à $n > N$ fixé :

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

et :

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

Notons alors $\delta(N) \coloneqq F(N-1) - \Phi(N)$ (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. □

Lemme 1.32. *Si $\varphi : \mathbb{R} \to \mathbb{R}$ est une fonction telle que $\varphi(x) \xrightarrow[x\to+\infty]{} +\infty$ (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. $\varphi(x) < -M$). □

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 $Q := \frac{\mathrm{d}}{\mathrm{d}x}P$ sa dérivée et observons :

$$Q(x) = \sum_{k=1}^{n} a_k \left.\frac{\mathrm{d}\,x^k}{\mathrm{d}\,x}\right|_x = \sum_{k=0}^{n-1}(k+1)a_{k+1}x^k.$$

Dès lors :

$$\lim_{x\to+\infty} Q(x) = \lim_{x\to+\infty} na_n x^{n-1} = \lim_{x\to+\infty} a_n x^{n-1} = \operatorname{sign}(a_n)\infty.$$

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. □

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 $\mathbb{R} \to \mathbb{R} : x \mapsto 1$.

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 :

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

□

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

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

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) :

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

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 :

$$Q(x) = \int_0^x P(t)\mathrm{d}\,t,$$

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

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

□

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 :*

$$F : \mathbb{R} \to \mathbb{R} : x \mapsto \int_0^x f(t)\mathrm{d}\,t,$$
$$G : \mathbb{R} \to \mathbb{R}^+ : x \mapsto \int_0^x g(t)\mathrm{d}\,t.$$

Démonstration. Définissons $h \coloneqq f - g$. 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 $x > N_0$:

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

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 $x > N_1$:

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

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

$$F(x) - G(x) = \int_0^x f(t)\mathrm{d}\,t - \int_0^x g(t)\mathrm{d}\,t = \int_0^x h(t) = H(x) = o(G(x)).$$

□

Corollaire 1.37. *Soient $\varphi_1, \varphi_2 : \mathbb{R}^+ \to \mathbb{R}^+$ 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 φ_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

Séances de TP

Séance 1 — Les ADT (partie 1)

Exercice 1.1. 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 1.2. 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 1.3. 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. :

```
l = UnorderedList() # on crée une liste doublement chaînée
for i in range(5): # on ajoute les éléments de 0 à 4
    l.add(i)
for element in l:  # doit itérer sur : 4, 3, 2, 1, 0
    print(element) # on affiche simplement
```

Bonus : rendez compatible la classe `UnorderedList` avec la fonction `reversed` de Python, i.e. :

```
for element in reversed(l): # doit itérer sur 0, 1, 2, 3, 4
    print(element)
```

Exercice 1.4. 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 2 — Les ADT (partie 2)

Exercice 2.1. 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 2.2. Soit une suite de puissances de 2 lue sur l'input 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 2.3. 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 2.4. Depuis Python 3.5, un opérateur dédié à la multiplication matricielle a été introduit : l'opérateur `@` (avec un **at** comme dans m**at**rices) qui se définit par la méthode spéciale `__matmul__` et a ses alternatives `__imatmul__` et `__rmatmul__`. Complétez la classe `MatriceComplexe` de l'exercice 1.2 afin de supporter la multiplication à l'aide de cet opérateur.

Exercice 2.5. 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 2.6. Écrivez une classe `Array` représentant un tableau de taille fixe supportant l'indiçage 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'indiçage 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 2.7. 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 2.6.

Exercice 2.8. 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 3 — La récursivité (partie 1)

Exercice 3.1. 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 3.2. Écrivez une fonction qui transforme, de manière récursive, un vecteur d'entiers de taille connue en son image miroir.

Exercice 3.3. 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 4 — La récursivité (partie 2)

Exercice 4.1. 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 4.2. 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. 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 4.3. Calculez récursivement le déterminant d'une matrice carrée $n+1 \times n+1$. Pour rappel, pour une matrice carrée :

$$A = \begin{bmatrix} a_{0\,0} & \cdots & a_{0\,n} \\ \vdots & \ddots & \vdots \\ a_{n\,0} & \cdots & a_{n\,n} \end{bmatrix} \in \mathbb{R}^{n+1 \times n+1},$$

la formule de Laplace nous permet d'exprimer (pour $j \in [\![0, n]\!]$ fixé) :

$$\det A = \sum_{k=0}^{n} (-1)^{j+k} a_{jk} \det A^{jk},$$

où A^{jk} est la sous-matrice :

$$A^{jk} = \begin{bmatrix} a_{0\,0} & \cdots & a_{0\,k-1} & a_{0\,k+1} & \cdots & a_{0n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{j-1\,0} & \cdots & a_{j-1\,k-1} & a_{j-1\,k+1} & \cdots & a_{j-1\,n} \\ a_{j+1\,0} & \cdots & a_{j+1\,k-1} & a_{j+1\,k+1} & \cdots & a_{j+1\,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n\,0} & \cdots & a_{n\,k-1} & a_{n\,k+1} & \cdots & a_{nn} \end{bmatrix} \in \mathbb{R}^{n \times n},$$

i.e. la matrice A sans la $j$ème ligne et la $k$ème colonne.

Exercices supplémentaires

Exercice 4.4. 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'esapce des logarithmes de x et y, i.e. $\log\sqrt{xy} = \frac{1}{2}(\log x + \log y)$). La moyenne arithmétique-géométrique de x et y, notée $\mathrm{agm}(x,y)$, 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. $\lim_{n\to+\infty} a_n = \lim_{n\to+\infty} g_n$.

$\mathrm{agm}(x,y)$ est défini par cette limite.

Écrivez une fonction récursive `recursive_agm(x, y, eps)` et une fonction non-récursive `agm(x, y, eps)` calculant $\mathrm{agm}(x,y)$, à précision `eps`.

Exercice 4.5. Une relation de récurrence linéaire est une équation de la forme :

$$x_n = \alpha_1 x_{n-1} + \alpha_2 x_{n-2} + \ldots + \alpha_k x_{n-k} = \sum_{j=1}^{k} \alpha_j x_{n-j} \qquad \text{pour } n > k,$$

pour un certain k entier, muni de conditions initiales $(x_1, \ldots, x_k)$.

Écrivez une fonction `linear_rec_rel(N, alpha, x)` prenant en paramètre un entier `N` ≥ 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 4.6. 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 $\mathcal{O}(n^{2.807\ldots})$ 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 $2^{N-1} \times 2^{N-1}$:

$$A = \begin{bmatrix} A^1 & A^2 \\ A^3 & A^4 \end{bmatrix} \qquad \text{et} \qquad B = \begin{bmatrix} B^1 & B^2 \\ B^3 & B^4 \end{bmatrix}.$$

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 5 — Le backtracking (partie 1)

Exercice 5.1. Écrivez une fonction qui engendre récursivement tous les sous-ensembles d'un ensemble d'entiers positifs de taille connue.

Exercice 5.2. 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 5.3. 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 5.4. 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 :

$$a_0x_0 + a_1x_1 + \ldots + a_{n-1}x_{n-1} = y. \tag{2.1}$$

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 $[\![b', b]\!]$ au lieu de $[\![0, b]\!]$?

Séance 6 — Le backtracking (partie 2)

Exercice 6.1. 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 6.2. 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 6.3. 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)`.

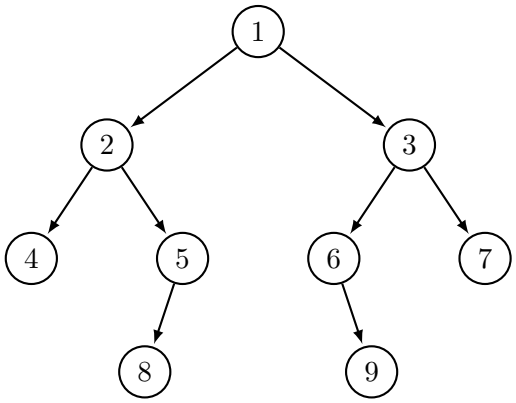
Exercice 6.4. Soit un vecteur `v`, écrivez un programme qui détermine s'il existe trois indices `i`, `j`, `k` tels que `v[i] + v[j] + v[k] = 0`.

Exercices supplémentaires

Exercice 6.5. 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.

Séance 7 — Arbres (partie 1)

Exercice 7.1. Donnez l'ordre de parcours préfixé, infixé et suffixé des nœuds de l'arbre suivant :



Exercice 7.2. Écrivez un morceau de code qui construit l'arbre binaire de l'exercice précédent en utilisant la classe `BinaryTree` donnée ci-dessous :

```
class BinaryTree:
    def __init__(self, root_obj, left=None, right=None):
        self.key = root_obj
        self.left = left
        self.right = right

    @property
    def root_value(self):
        return self.key

    @property
    def left_child(self):
        return self.left

    @property
    def right_child(self):
        return self.right
```

Exercice 7.3. 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 7.4. 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 8 — Arbres (partie 2)

Exercice 8.1. 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.

```
class BinaryTreeFather(BinaryTree):
    def __init__(self, root_obj, left=None, right=None, father=None):
        BinaryTree.__init__(self, root_obj, left, right)
        self.__father = father

    @property
    def father(self):
        return self.__father
```

Exercice 8.2. Écrivez une fonction récursive `contains(tree, value)` qui renvoie `True` si `value` est dans `tree` (un arbre binaire) et `False` sinon.

Exercice 8.3. Écrivez une fonction récursive qui teste si deux arbres binaires sont égaux.

Exercice 8.4. 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.

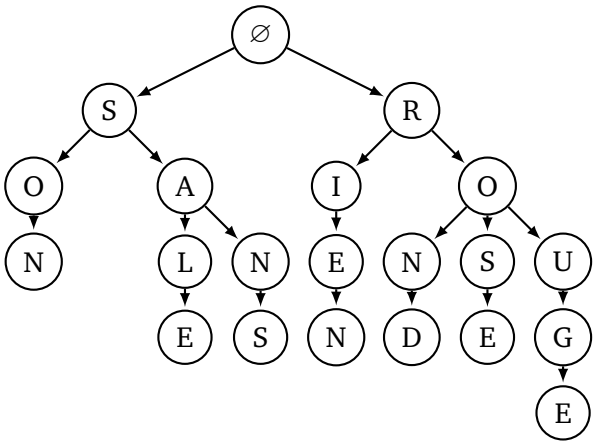
Exercices supplémentaires

Exercice 8.5. Considérons un alphabet quelconque Σ (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 Σ 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 ℓ, ces trois méthodes doivent s'exécuter en $\mathcal{O}(\ell)$ opérations dans le pire des cas.



Séance 9 — 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 9.1. 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 9.2. 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 9.3. 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 9.4. 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 10 — Files à priorité

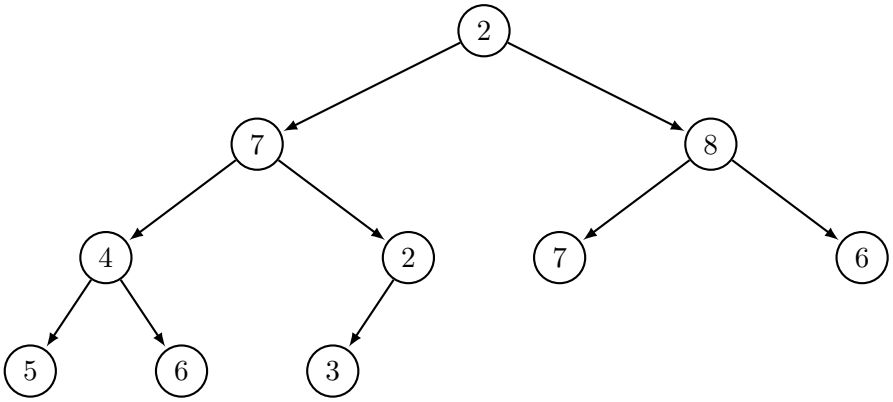
Définition 12. Soit T un arbre binaire.
- si tout sommet v de T satisfait la propriété suivante : la valeur de v est *supérieure* ou égale à la valeur de ses fils, alors T est un *max-tas* (ou *max-heap*) ;
- si tout sommet v de T satisfait la propriété suivante : la valeur de v est *inférieure* ou égale 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 10.1. 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 10.2. 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.

Séance 11 — 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 section 12.1 pour plus de détails sur le fonctionnement interne de ces notions.

Exercice 11.1. 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(++)).

```python
def h1(x):
    return sum(x)

def h2(x):
    return len(x) + sum(x)

def h3(x):
    ret = 5381
    for item in x:
        ret = 33*ret + item
    return ret
```

Exercice 11.2. Le code suivant lancera-t-il une exception ?

```python
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
S = set()
p = Pair(1, 2)
S.add(p)
assert p in S
p.x = -1
assert p not in S
```

Redéfinissez la méthode `__hash__` afin que de sorte à ce que :

$$h(x, y) = 2^{32}x + (y \mod 32).$$

Le code ci-dessus s'exécutera-t-il correctement ?

Exercice 11.3. 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 ← P
3:     for every k in t do
4:         x ← x + h(k) · N
5:         x ← rotate_left(x, 31)
6:         x ← x · M
7:     end for
8:     return x + (L xor P xor X)
9: end procedure
```

où L est la longueur de t, et où M, N, P, X sont les constantes suivantes :

$$M = 11400714785074694791$$
$$N = 14029467366897019727$$
$$P = 2870177450012600261$$
$$X = 3527539$$

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 11.4. 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 11.5. 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 11.6. 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 12 — Hachage (partie 2)

Exercice 12.1. 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 12.2. Supposons que l'on utilise le double hachage pour gérer les collisions ; autrement dit, on utilise la fonction de hachage suivante :

$$h(k, j) = (h_1(k) + jh_2(k)) \mod m.$$

Montrez que, si m et $h_2(k)$ ne sont pas premiers entre eux (i.e. $d = \text{GCD}(m, h(k)) \gneqq 1$) 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 12.3. 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) :

```
class Set:
    def __init__(self, m): # Theta(m)
        # ...
    def insert(self, x: object) -> None: # O(1)
        # ...
    def find(self, x: object) -> Node: # O(m)
        # ...
    def remove(self, x: object) -> None: # O(m)
        # ...
    def delete(self, node: Node) -> None: # O(1)
        # ...
```

Exercice 12.4. Impléméntez un dictionnaire via une table de hahchage à adressage ouvert similaire à celle implémentée dans CPython (c.f. section 12.1). Vous pouvez supposer que votre table est de taille fixe et définie lors de l'initialisation.

Exercices supplémentaires

Exercice 12.5. On considère une variante de la méthode de la division dans laquelle $h(k) = k \mod m$, 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 12.6. On dit qu'une famille $\mathcal{H}$ de fonctions de hachage reliant un ensemble fini U à un ensemble fini B est ε-universelle si, pour toute paire d'éléments distincts k et ℓ 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 ε-universelle de fonctions de hachage doit vérifier :

$$\varepsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

12.1 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 :

```
class object:
    # ...
    def __hash__(self):
        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 $h(k) \mod m$ 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) :

```
class object:
    # ...
    def __hash__(self):
        h = hex(id(self))[2:]
        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 hashs 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. :

```
class C:
    def __hash__(self):
        return -1
c = C()
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 11.3, 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 psuedo-aléatoire (voir ces explications pour plus d'informations). Le raisonnement est celuci : 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 $\mod 2^k$ 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 `hash(x)%m == hash(y)%m`, alors en particulier les hashs 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 `hash(x) % m` (et pas `hash(x)`) ne permet pas de savoir à l'avance quel va être l'ordre de parcours des indices sondés.

Séance 13 — Graphes (partie 1)

Définition 13. Soit V un ensemble de sommets.

— Un ensemble $E \subset \binom{V}{2}$ (i.e. un ensemble E de la forme $\{\{v_{i_1}, v_{j_1}\}, \ldots, \{v_{i_m}, v_{j_m}\}\}$ 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 $\{(v_{i_1}, v_{j_1}), \ldots (v_{i_m}, v_{j_m})\}$) 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 $\gamma : E \to \mathbb{R}$ 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.

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 14. Soit un graphe dirigé pondéré simple $G = (V, E, \gamma)$ tel que $V = \{v_1, \ldots, v_n\}$ et $E = \{e_1, \ldots e_m\}$.

— 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 :

$$I_{ij} = \begin{cases} \gamma(e) & \text{si } \exists v \in V \text{ s.t. } e_j = (v_i, v) \\ -\gamma(e) & \text{si } \exists v \in V \text{ s.t. } e_j = (v, v_i) \\ 0 & \text{sinon.} \end{cases}$$

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 $\mathbf{1} : E \to \{1\} : e \mapsto 1$ 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 15. 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 2.1. *(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 :

$$\begin{aligned}
\sum_{v \in V} \deg(v) &= \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E\}| \\
&= \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E, v < w\} \sqcup \{w \in V \text{ s.t. } \{v, w\} \in E, v > w\}| \\
&= \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E, v < w\}| + \sum_{v \in V} |\{w \in V \text{ s.t. } \{v, w\} \in E, v > w\}| \\
&= \left| \bigsqcup_{v \in V} \{w \in V \text{ s.t. } \{v, w\} \in E, v < w\} \right| + \left| \bigsqcup_{v \in V} \{w \in V \text{ s.t. } \{v, w\} \in E, v > w\} \right| \\
&= |E| + |E| = 2m.
\end{aligned}$$

□

Corollaire 2.2. *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 $U = \{v \in V \text{ s.t. } \deg v = 1 \mod 2\}$ est de cardinalité impaire. Nous savons donc que :

$$\sum_{v \in U} \deg v = 1 \mod 2.$$

De plus nous savons que :

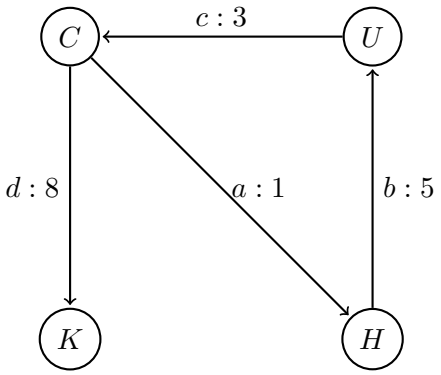
$$\sum_{v \in V \setminus U} \deg v = 0 \mod 2.$$

Dès lors nous déduisons que (par la proposition 2.1) :

$$2\,|E(G)| = \sum_{v\in V} \deg v = \sum_{v\in U} \deg v + \sum_{v\in V\setminus U} \deg v = 1 + 0 \mod 2 = 1 \mod 2,$$

ce qui est une contraction car $2\,|E(G)|$ est pair. Dès lors U est de cardinalité paire. □

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 13.1. Construisez la matrice d'adjacence et la matrice d'incidence du graphe G_1.

Exercice 13.2. Que deviennent les matrices d'adjacence et d'incidence de G_1 si on retire tous les poids des arcs ?

Exercice 13.3. Construisez la structure dynamique (listes de successeurs) qui représente le graphe G_1.

Exercice 13.4. Dessinez le graphe G_2 représenté par la matrice d'adjacence ci-dessus.

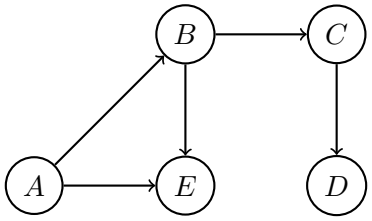
Exercice 13.5. Il existe deux manières principales de représenter un graphe simple dans une structure de données : à l'aide de sa matrice d'adjacence, ou grâce à une structure dynamique utilisant des listes de successeurs. On vous demande d'écrire deux fonctions :

- 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 14 — Graphes (partie 2)

Considérons le graphe G suivant :



Exercice 14.1. Donnez la matrice d'adjacence de G et la liste des sommets accessibles depuis chaque sommet.

Exercice 14.2. Exécutez l'algorithme de Roy-Warshall sur G.

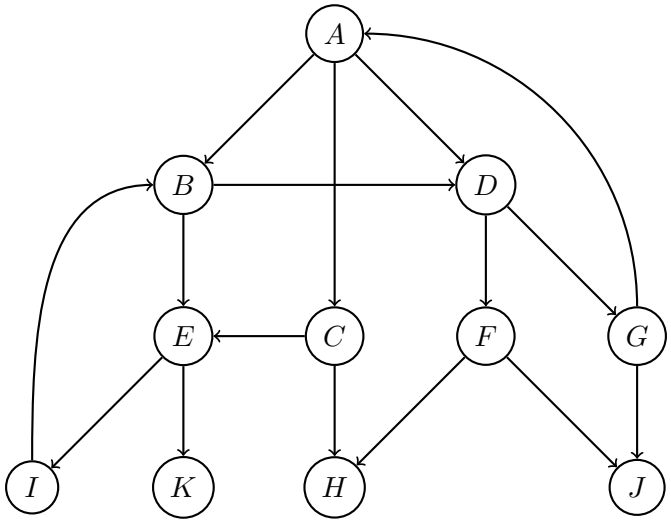
Exercice 14.3. Construisez la fermeture transitive (M^*) de G en utilisant la formule suivante (M est la matrice d'adjacence) :

$$M^* = M^0 \vee M^1 \vee \ldots \vee M^n = \bigvee_{k=0}^{n} M^k.$$

Exercice 14.4. Comparez les résultats des trois questions précédentes.

Séance 15 — Graphes (partie 3)

Considérons le graphe suivant :



Exercice 15.1. Donnez la séquence des sommets parcouru si on utilise l'algorithme *depth-first* (parours en profondeur) pour parcourir le graphe ci-dessus en commencent par le sommet `A`.

Exercice 15.2. Donnez la séquence des sommets parcouru si on utilise l'algorithme *breadth-first* (parours en largeur) pour parcourir le graphe ci-dessus en commencent par le sommet `A`.

Exercice 15.3. Donnez la séquence des sommets parcouru si on utilise l'algorithme *breadth-first* (parours en largeur) pour parcourir le graphe ci-dessus en commencent par le sommet `I`.

Séance 16 — Graphes (partie 4)

Exercice 16.1. 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 16.2. 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 16.3.

1. Déterminez une famille de graphes connexes à n sommets ayant $\Theta(n)$ sommets ;
2. déterminez une famille de graphes connexes à n sommets ayant $\Theta(n \log n)$ sommets ;
3. déterminez une famille de graphes connexes à n sommets ayant $\Theta(n^{3/2})$ sommets ;
4. déterminez une famille de graphes connexes à n sommets ayant $\Theta(n^2)$ sommets.

Exercice 16.4. Montrez que dans un arbre T, pour chaque paire de sommets (u, v), il existe un unique chemin P joignant u et v.

Exercice 16.5. Montrez que si G est un graphe connexe cyclique, alors il existe un sous-graphe strict de G qui est connexe.

Exercice 16.6. Montrez que si G est un graphe non-connexe, alors $G^{\complement}$ est connexe.

Exercice 16.7. 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 16.8. Montrez que si T est un arbre à $n \geq 2$ sommets, alors T admet au moins deux feuilles.

Exercice 16.9. 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 16. 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 16.10. 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 :

```
def strong_product(G1, G2):
    ret = Graph(G1.V*G2.V)
    as_id = lambda u, v : u*G1.V + v
    for u1, v1 in G1:
        for u2 in range(G2.V):
            u1u2 = as_id(u1, u2)
            v1u2 = as_id(v1, u2)
            ret.add_edge(u1u2, v1u2)
        for u2, v2 in G2:
            ret.add_edge(as_id(u1, u2), as_id(v1, v2))
            ret.add_edge(as_id(v1, u2), as_id(u1, v2))
    for u2, v2 in G2:
        for u1 in range(G1.V):
            u1u2 = as_id(u1, u2)
            u1v2 = as_id(u1, v2)
            ret.add_edge(u1u2, u1v2)
    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 :

$$\begin{cases} \mathrm{GCD}(a,0) & = a \\ \mathrm{GCD}(a,b) & = \mathrm{GCD}(b, a - b\left\lfloor \frac{a}{b} \right\rfloor) \end{cases}$$

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 $ax + by = \mathrm{GCD}(a,b)$.

Vérifiez que les deux algorithmes donnent bien le résultat attendu et écrivez deux fonctions dérécursifiées des fonctions récursives présentées dans le fichier `GreatestCommonDivisorEmp.py` :

```
def extended_gcd_rec(a, b):
    if b == 0:
        return (1, 0, a) # 1*a + 0*b == a
    else:
        q, r = divmod(a, b) # a = q*b + r
        x, y, g = extended_gcd_rec(b, r)
        z = x - q*y
        return y, z, g
```

Exercice 17.2. Écrivez une fonction dérécursifiée de la fonction `quicksort` vue au cours.

Exercice 17.3. Écrivez une version non-récursive de l'exercice 3.3 (tours de Hanoï).

Exercices supplémentaires

Exercice 17.4. La fonction 91 de McCarthy est définie comme ceci :

$$M : \mathbb{N} \to \mathbb{N} : n \mapsto \begin{cases} n - 10 & \text{si } n > 100, \\ M(M(n + 11)) & \text{sinon.} \end{cases}$$

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 — Quicksort

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 récursifs 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 quickosrt 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.

Exercice 18.6. Une autre approche est de ne pas placer le pivot à une position fixe mais de déterminer une position intéressante pour le pivot. Idéalement, on voudrait que le pivot soit systématiquement la médiane des éléments à trier, mais c'est en pratique trop long à calculer. Nous allons donc chercher à approximer la médiane (en passant par une heuristique) pour choisir notre pivot. Procédons de la sorte : (i) séparons le vecteur à trier en $\frac{n}{5}$ sous-tableaux de taille 5 (on suppose ici n divisible par 5 pour plus de facilité), (ii) déterminons (en $\Theta(1)$) la médiane de chacun de ces sous-tableaux, (iii) calculons récursivement l'approximation de la médiane de ces $\frac{n}{5}$ médianes.

> **Remarque.** *Cette méthode de choix de pivot a été proposée (il y a 40 ans) par Blum, Floyd, Pratt, Rivest et Tarjan (plein de grands noms dont vous n'avez pas fini d'entendre parler), d'où le nom (obsolète aujourd'hui) de cet algorithme : BFPRT ; que l'on appelle maintenant* median of medians*.*

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 sommet 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 chaines 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 chaines. 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 $x = (x_1, \ldots, x_n)$, 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, 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 $[\![1, n]\!] = \{1, 2, \ldots, n\}$. 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]`.

Chapitre 3

Exercices cotés

Exercice coté 1. Soit `n` une 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 $V = \{v_1, \ldots, v_n\}$. Le *mycielskien* de G est le graphe $M(G) = (V', E')$ défini par :

— $V' = V \cup \{v'_1, \ldots, v'_n, w\}$;
— $\forall i, j \in [\![1, n]\!] : \{v_i, v_j\} \in E' \iff \{v_i, v_j\} \in E$;
— $\forall i, j \in [\![1, n]\!] : \{v'_i, v_j\} \in E' \iff \{v_i, v_j\} \in E$;
— $\forall i \in [\![1, n]\!] : \{v'_i, w\} \in E'$.

Sans modifier les classes `DynamicUndirectedGraph` et `Vertex` suivantes, écrivez une fonction `mycielski(n)` qui renvoie le $n$ème graphe de Mycielski $M^n(K_2)$:

```python
class Vertex:
    def __init__(self, vertex_idx):
        self.idx_ = vertex_idx
        self.neighbours_ = []

    def add_edge(self, v):
        self.neighbours_.append(v)

    @property
    def idx(self):
        return self.idx_

    @property
    def neighbours(self):
        return iter(self.neighbours_)

    @property
    def nb_neighbours(self):
        return len(self.neighbours_)

class DynamicUndirectedGraph:
    def __init__(self, n):
        self.vertices_ = [Vertex(i) for i in range(n)]
        self.nb_edges_ = 0

    @property
    def n(self):
        return len(self.vertices_)

    @property
    def m(self):
        return self.nb_edges_

    def add_vertex(self):
        self.vertices_.append(Vertex(self.n))

    def vertex(self, i):
        return self.vertices_[i]

    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 6.2 :

```
def solve_maze_rec(maze, x=0, y=0):
    W = len(maze[0])
    H = len(maze)
    over = x == W-1 and y == H-1
    if over or not (0 <= x < W) or not (0 <= y < H) or maze[y][x] !=
        EMPTY:
        if over:
            maze[y][x] = '.'
            return True
        else:
            return False
    else:
        maze[y][x] = '.'
        for i in range(4):
            xp, yp = x+DELTA_X[i], y+DELTA_Y[i]
            if solve_maze_rec(maze, xp, yp):
                return True
        maze[y][x] = EMPTY
        return False
```

Écrivez une version dérécursifiée de cette même fonction.

Chapitre 4

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) `inserer(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

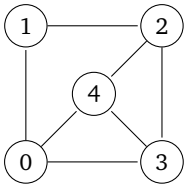
Question d'examen (Q3 août 2021)**.** Soit G un graphe connexe non-dirigé d'ordre n représenté par sa matrice d'adjacence A. On appelle *cycle hamiltonien* un cycle passant une et une seule fois par chacun des sommets de G (sans toutefois obligatoirement passer par toutes les arêtes de G). On dit qu'un 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 :

$$0 \to 1 \to 2 \to 3 \to 4 \to 0$$

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 :

$$f(x) = x^n = \begin{cases} 1 & \text{si } n = 0. \\ \left(x^2\right)^{\frac{n}{2}} & \text{si } n \text{ est pair.} \\ x \cdot x^{n-1} & \text{si } n \text{ est impair} \end{cases}$$

Nous vous demandons d'écrire une **fonction non-récursive** 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 ≥ 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 — 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.

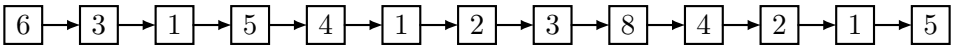
Session — Août 2022

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.