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`).

Résolution. Le constructeur de la classe `Complexe` a besoin des paramètres `re` et `im` correspondant respectivement à la partie réelle et la partie imaginaire du nombre complexe. Ces valeurs sont mises dans les attributs *privés* `__re` et `__im`. Bien que ces attributs soient privés, ils peuvent être rendus accessibles *en lecture* (i.e. sous forme de *getters*) à l'aide de propriétés.

Une propriété en Python se définit à l'aide du décorateur `@property` et permet d'accéder à des attributs privés ou à des éléments calculés sur base de ces mêmes attributs en les rendant *virtuellement* publiques :

Pour les opérations demandées (i.e. addition et multiplication), nous allons définir les méthodes `__iadd__` et `__imul__` qui *surchargent* les opérateurs `+=` et `*=`, i.e. pour deux variables `x` et `y` de type `Complexe`, l'instruction `x += y` est équivalente à `x.__iadd__(y)` (pour peu que cette méthode soit définie). Sur base de ces deux méthodes, nous pouvons définir les méthodes `__add__` et `__mul__` qui surchargent les opérateurs `+` et `*`.

Remarque. *Les méthodes* `__iadd__` *et* `__imul__` *(et par extension* `__isub__` *et* `__idiv__` *qui surchargent respectivement les opérateurs* `-=` *et* `/=`*) doivent renvoyer* `self` *et les méthodes* `__add__` *et* `__mul__` *(et par extension* `__sub__` *et* `__div__` *qui surchargent les opérateurs* `-` *et* `/`*) doivent renvoyer une instance de* `Complexe` *:*

```python
class Complexe:
    def __init__(self, re=0., im=0.):
        #attributs
        self.__re = re
        self.__im = im

    @property
    def re(self):
        return self.__re

    @property
    def im(self):
        return self.__im

    def __str__(self):
        """Conversion en str"""
        return f"{self.re:g} + {self.im:g}i"

    def __iadd__(self, z):
        """Opérateur +="""
        if isinstance(z, (int, float)):
            z = Complexe(re=z, im=0)
```

```
23          if not isinstance(z, Complexe):
24              return NotImplemented
25          self.__re += z.re
26          self.__im += z.im
27          return self
28
29      def __add__(self, z):
30          """Opérateur +"""
31          ret = self.copy()
32          ret += z
33          return ret
34
35      def __imul__(self, z):
36          """Opérateur *="""
37          tmp = self.re
38          self.__re = tmp*z.re - self.im*z.im
39          self.__im = tmp*z.im + self.im*z.re
40          return self
41
42      def __mul__(self, z):
43          """Opérateur *"""
44          ret = self.copy()
45          ret *= z
46          return ret
47
48      def copy(self):
49          return Complexe(self.re, self.im)
```

Vous remarquerez que nous avons défini une méthode `copy` qui renvoie une copie de l'objet initial. Cette méthode est utilisée dans `__add__` et `__mul__` pour créer un nouvel objet identique au premier et appliquer soit `__iadd__` soi `__imul__` dessus. Nous avons également défini la méthode `__str__` qui renvoie une représentation de l'objet dans un `str`. Ce `str` suit la représentation $z = \Re(z) + \Im(z)i$. Vous pouvez bien entendu adapter cette méthode afin d'éviter que l'objet `Complexe(2, -3)` soit affiché `2 + -3i` mais bien `2 - 3i`.

Les méthodes `__iadd__` et `__imul__` devraient normalement être *sécurisée* pour éviter qu'une exception soit levée si le paramètre `z` n'est pas du bon type. Pour cela, la meilleure approche est d'utiliser la fonction `isinstance`. Si `z` n'est pas du bon type, la fonction doit renvoyer `NotImplented` afin que l'interpréteur Python puisse essayer de trouver un autre sens à l'instruction. Pour plus de détails, voir la doc officielle de Python. Cela donne (par exemple) :

```
def __iadd__(self, z):
    """Opérateur +="""
    if isinstance(z, (int, float)):
        z = Complexe(re=z, im=0)
    if not isinstance(z, Complexe):
        return NotImplemented
    self.__re += z.re
```

```
8       self.__im += z.im
```

Remarque. *En Python, on évite habituellement de vérifier le type d'un objet avec la fonction* `type` *car cela casse des principes de programmation orientée objet que vous verrez en bloc 2 en INFO-F202 et INFO-F204. On utilise donc* `if isinstance(z, Complexe)` *à la place de* `if type(z) is Complexe`.

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.

Résolution. Cet exercice requiert un accès à la classe `Complexe` définie ci-dessus (soit via un `import`, soit en ajoutant directement la classe `Complexe` dans le fichier source). La classe `MatriceComplexe` admet trois paramètres à son constructeur : `m` (le nombre de lignes), `n` (le nombre de colonnes), et `var` (la valeur par défaut des entrées de la matrice). Les deux premiers sont stockés directement sous forme d'attributs, et les propriétés associées sont écrites. La matrice en elle-même est stockée dans l'attribut `__mat` qui est une liste de listes. Ce dernier attribut n'a pas de propriété pour accéder aux éléments, cependant lire et modifier les éléments de la matrice est bien entendu nécessaire. Pour cela, la bonne approche est d'implémenter les méthodes `__getitem__` et `__setitem__` qui permettent respectivement d'accéder à un élément via la syntaxe `object[index]` et d'écrire un élément via la syntaxe `object[index] = value` (tout comme pour une liste ou un dictionnaire).

La méthode `__getitem__` doit prendre un unique paramètre correspondant à l'indice (dans le cas d'une liste), la clef (dans le cas d'un dictionnaire) ou encore la position (dans le cas d'une matrice). La méthode `__setitem__` doit en plus prendre un paramètre correspondant à la valeur que l'on veut mettre à la position donnée. Dans notre cas, l'attribut de position sera un `tuple` contenant uniquement deux éléments.

Il ne reste plus qu'à implémenter la multiplication par un scalaire qui se fait alors en itérant simplement sur tous les éléments de la matrice et en les multipliant par le scalaire donné en paramètre.

```
1  class MatriceComplexe:
2      def __init__(self, m=1, n=1, val=Complexe(0, 0)):
3          self.__m = m
4          self.__n = n
5          self.__mat = [[val.copy() for j in range(self.n)] for i in
               range(self.m)]
6
7      @property
8      def m(self):
9          return self.__m
10
11     @property
12     def n(self):
13         return self.__n
14
15     def __str__(self):
```

```
        """Conversion en str"""
        ret = ""
        for i in range(self.m):
            for j in range(self.n):
                ret += f"{self[i,j]} "
            ret += "\n"
        return ret

    def __getitem__(self, pos):
        """Opérateur []"""
        row, col = pos
        return self.__mat[row][col]

    def __setitem__(self, pos, z):
        """Assignation self[pos[0],pos[1]] = z"""
        row, col = pos
        self.__mat[row][col] = z

    def __imul__(self, scalar):
        """Opérateur *="""
        for row in range(self.m):
            for col in range(self.n):
                self[row,col] *= scalar
        return self

    def copy(self):
        ret = MatriceComplexe(self.m, self.n)
        for i in range(self.m):
            for j in range(self.n):
                # équivalent à
                # ret.__setitem__((i, j), self.__getitem__((i, j)))
                ret[i,j] = self[i,j]
        return ret
```

Vous pouvez y ajouter les méthodes `__mul__`, `__add__`, `__iadd__`, etc. en faisant bien attention à la dimension des différentes matrices pour vous assurer que les opérations soient bien définies.

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)
```

Résolution. Les attributs privés `next` et `previous` de la classe `Node` doivent être obligatoirement accessibles en lecture puisque c'est ainsi que l'on parcourt la liste. Ils doivent également être accessibles en écriture puisque ces attributs sont modifiés lors d'un ajout ou d'une suppression. Il faut cependant faire attention à l'encapsulation lorsque l'on permet à des attributs d'être réécrits : il faut s'assurer que l'on ne casse pas l'état interne de l'objet. En particulier ici, il faut s'assurer que les attributs `previous` et `next` soient bien soit `None`, soit une instance de la classe `Node`. Cette vérification se fait donc dans les `setters`. On lance ici simplement une exception de type `TypeError` si ce n'est pas le cas.

La propriété `previous_data` est simple à écrire puisque les propriétés `previous` et `data` existent déjà :

```
class Node:
    def __init__(self, init_data, init_previous=None, init_next=None):
        self.__data = init_data
        self.__next = init_next
        self.__previous = init_previous

    @property
    def data(self):
        return self.__data

    @data.setter
    def data(self, new_data):
        self.__data = new_data

    @property
    def next(self):
        return self.__next

    @next.setter
    def next(self, new_next):
        if not (isinstance(new_next, Node) or new_next is None):
            raise TypeError()
        self.__next = new_next

    @property
    def previous(self):
        return self.__previous
```

```
28
29      @previous.setter
30      def previous(self, new_previous):
31          if not (isinstance(new_previous, Node) or new_previous is None):
32              raise TypeError()
33          self.__previous = new_previous
34
35      @property
36      def previous_data(self):
37          return self.previous.data
38
39      def __str__(self):
40          return str(self.data)
```

La classe `UnorderedList` s'est vue modifiée comme suit :

```
1   class UnorderedList:
2       def __init__(self):
3           self.head = Node(None)
4           self.head.next = self.head
5           self.head.previous = self.head
6
7       def is_empty(self):
8           return len(self) == 0
9           #alternative : return self.head.next is self.head
10
11      def __len__(self):
12          """len(self)"""
13          current = self.head.next
14          count = 0
15          while current is not self.head:
16              count += 1
17              current = current.next
18          return count
19
20      def add(self, value):
21          self.add_after(self.head, value)
22
23      def add_after(self, node, value):
24          temp = Node(value)
25          temp.previous = node
26          temp.next = node.next
27          if node.next is not None:
28              node.next.previous = temp
29          node.next = temp
30
31      def __contains__(self, value):
32          """value in self"""
33          return self.search_node_of(value) is not None
```

```
34
35      def search_node_of(self, value):
36          """returns the node containing value, or None if not found"""
37          current = self.head.next
38          found = False
39          while current is not self.head and not found:
40              if current.data == value:
41                  found = True
42              else:
43                  current = current.next
44          return current if found else None
45
46      def remove(self, value):
47          node = self.search_node_of(value)
48          if node is not None:
49              self.remove_node(node)
50
51      def remove_node(self, node):
52          if not self.is_empty():
53              node.previous.next = node.next
54              node.next.previous = node.previous
55
56      @property
57      def last(self):
58          return self.head.previous_data
59
60      def __str__(self):
61          current = self.head.next
62          res = "[ "
63          while current is not self.head:
64              res += str(current) + " "
65              current = current.next
66          res += f"] (length = {len(self)})"
67          return res
68
69      def __iter__(self):
70          current = self.head.next
71          while current is not self.head:
72              yield current.data
73              current = current.next
74
75      def __reversed__(self):
76          current = self.head.previous
77          while current is not None and current is not self.head:
78              yield current.data
79              current = current.previous
```

Vous remarquerez que la méthode `search` est devenue `__contains__`, ce qui permet d'utiliser la syntaxe Python `element in l` (qui correspond donc à `l.__contains__(element)`).

La propriété `last` doit juste appeler la propriété `previous_data` sur l'attribut `head` puisque la liste est doublement chaînée.

Afin de pouvoir utiliser la syntaxe Python `for element in l`, il faut définir la méthode `__iter__`. Plus précisément, cette méthode permet d'utiliser la fonction Python `iter`, et `for element in l` est équivalent à `for element in iter(l)`. La fonction `iter` doit renvoyer un *itérateur*, i.e. un objet qui a une méthode `__next__`, ce qui est le cas des générateurs qui sont définis à l'aide du mot-clef `yield` ; mais vous pouvez définir vos propres classes d'itérateurs si vous le désirez.

Pour faire un générateur, il vous faut une fonction qui contient (au moins) un `yield`. Ce `yield` se comporte virtuellement comme un `return`, sauf que lorsque la fonction est rappelée, elle reprend là où elle s'était arrêtée au dernier `yield`. Ainsi, notre méthode `__iter__` se place d'abord sur le premier élément, et tant que cet élément n'est pas à nouveau la tête de la liste, on *renvoie* la donnée associée et on passe au suivant. Le générateur s'arrête lorsque la fonction arrive à une instruction `return` ou à la fin du code de la fonction. Plus précisément, cela mène à un `raise StopIteration` qui signale à l'interpréteur qu'il ne faut plus appeler la méthode `__next__`. Pour plus d'informations, voir la documentation officielle de Python.

Pour le bonus, la fonction `reversed` appelle la méthode `__reversed__` qui renvoie également un itérateur (comme `__iter__`). Il suffit juste de parcourir la chaîne de la fin vers le début (donc en passant par `previous` au lieu de `next`).

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.

Résolution. Pour résoudre cet exercice, nous allons utiliser un `Stack` qui va contenir les éléments de la première partie de la chaîne de caractères, et une fois que le caractère central (i.e. `*`) sera lu, on poppera les éléments du stack au fur et à mesure que les nouveaux caractères seront rentrés. S'il y a discordance entre l'élément retiré de la pile et l'élément entré, on sait que la chaîne n'est pas un palindrome. De plus, si le stack est vide mais que des caractères sont encore entrés ou si le caractère `#` est entré alors que le stack n'est pas vide, alors on sait également que la chaîne n'est pas un palindrome.

Le code du stack est le même que celui vu au cours (si ce n'est pour la méthode `__len__` comme dans l'exercice précédent) :

```
1  class Stack:
2      def __init__(self):
3          self.items = []
4
5      def is_empty(self):
6          return len(self) == 0
7
8      def push(self, item):
9          self.items.append(item)
10
```

```
    def pop(self):
        return self.items.pop()

    def top(self):
        return self.items[len(self.items)-1]

    def __len__(self):
        return len(self.items)

    def __str__(self):
        res = "| "
        for i in self.items:
            res += str(i) + " | "
        return res
```

La fonction `palindrome` se définit alors comme suit :

```
def palindrome():
    stack = Stack()
    c = input("Premier caractère: ")
    while c != "*":
        stack.push(c)
        c = input("Caractère suivant (* au milieu): ")
    identique = True
    c = input("Caractère suivant (# à la fin): ")
    while c != "#" and identique:
        if stack.is_empty() or stack.pop() != c:
            identique = False
            break
        c = input("Caractère suivant (# à la fin): ")
    if identique and stack.is_empty():
        print("Vous avez introduit un palindrome.")
    else:
        print("Vous n'avez pas introduit de palindrome.")
```

Proposition 2.1. *Cet algorithme permet de déterminer si une séquence de longueur n est un palindrome en $\mathcal{O}(n)$ comparaisons.*

Démonstration. En effet les m premiers caractères (avant `*`) sont tous comparés à `*` pour trouver le potentiel milieu et $m < n$ (puisque la séquence est de longueur n). Les $n-m-1$ caractères qui suivent le `*` sont comparés au plus deux fois (une première fois avec `#` et une seconde avec le *top of stack*). Le nombre total de comparaisons dans le pire des cas est donc :

$$m + 2(n - m - 1) = 2n - m - 2 \leq 2n = \mathcal{O}(n).$$

□

Remarque. *Nous pouvons utiliser un argument encore plus simple pour cela : il y a au plus n caractères qui sont lus sur l'input, et chacun de ces caractères est comparé $\mathcal{O}(1)$ fois*

(i.e. le nombre de comparaisons pour chacun de ces caractères est borné par une quantité ne dépendant pas de n), dès lors le nombre total de comparaisons est $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

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.

Résolution. Afin de résoudre cet exercice, nous reprenons la classe `Stack` de l'exercice précédent, et nous allons l'utiliser comme suit :

- si le caractère actuel est une parenthèse ouvrante (i.e. `(`, `[` ou `{`), alors on l'ajoute sur le stack ;
- si le caractère actuel est une parenthèse fermante (i.e. `)`, `]` ou `}`), alors on pop un élément du stack et on vérifie qu'ils correspondent bien ;
- si le caractère actuel est `#`, alors il suffit de vérifier que le stack est bien vide.

Ce qui donne donc le code suivant :

```
1  def parentheses():
2      stack = Stack()
3      parenthese_ok = True
4      c = input("Premier caractère (# à la fin): ")
5
6      matching_parentheses = {')': '(', ']': '[', '}': '{'}
7      while c != "#" and parenthese_ok:
8          if c in "({[":
9              stack.push(c)
10             c = input("Caractère suivant (# à la fin): ")
11         else:
12             if stack.is_empty():
13                 parenthese_ok = False
14             else:
15                 c2 = stack.pop()
16                 if c2 == matching_parentheses.get(c):
17                     c = input("Caractère suivant (# à la fin): ")
18                 else:
19                     parenthese_ok = False
20     if parenthese_ok and stack.is_empty():
21         print("L'expression est valide")
22     else:
23         print("L'expression n'est pas valide")
```

Notez que la méthode `get` est appelée sur `matching_parentheses` au lieu de `__getitem__` car cette dernière lève une `KeyError` si la clef n'existe pas dans le conteneur, et donc le programme planterait si l'utilisateur rentre autre chose qu'une parenthèse fermante.

Proposition 2.2. *Cet algorithme permet de déterminer si une expression est bien parenthésée en $\mathcal{O}(n)$ comparaisons.*

Démonstration. De manière similaire à l'exercice précédent : au plus n caractères sont lus et subissent $\mathcal{O}(1)$ comparaisons, pour un total de $\mathcal{O}(n)$ comparaisons. □

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`.

Résolution. Pour résoudre cet exercice, il est important que chaque nombre n'apparaisse qu'une seule fois. Un petit lemme s'impose :

Lemme 2.3. *Soit $n \in \mathbb{N}$, un entier. La somme des puissances de 2 inférieures à 2^n est inférieure à 2^n, i.e. :*

$$\sum_{k=0}^{n-1} 2^k < 2^n.$$

Démonstration. On se rend facilement compte de ce lemme en utilisant l'écriture binaire d'un nombre : la somme des puissances de 2 inférieures à 2^n correspond au nombre dont l'écriture binaire est `111.....1`, composé de n bits. Ce nombre vaut $2^n - 1$ qui est bien inférieur à 2^n. □

Nous allons donc utiliser ce résultat de la manière suivante : nous savons que la suite de nombres entrés par l'utilisateur ne contient que des puissances de 2 distinctes et nous voulons extraire une sous-suite *strictement décroissante*. Dès lors nous savons que le premier élément de la sous-suite doit obligatoirement être la plus grande puissance de 2 donnée dans la sous-suite. En effet si la sous-suite extraite ne contient pas la plus grande puissance de 2 donnée (disons 2^N), alors pour 2^K la plus grande puissance de 2 de la sous-suite extraite, on sait que la somme des éléments de la sous-suite est nécessairement inférieure à 2^N par le lemme.

Nous procédons alors de la manière suivante : on garde dans notre sous-séquence tous les nombres entrés tant qu'ils sont décroissant, et dès qu'un nombre entré est plus grand que le nombre précédent, on retire de la sous-séquence tous les nombres plus petits que le dernier entré, et on ajoute ensuite ce même nombre. Pour cela nous utilisons un `stack` puisqu'à chaque instant, seul le `top` nous intéresse pour nous assurer du bon fonctionnement de l'algorithme.

```
1 def sous_suite():
2     stack = Stack()
3     nb = int(input("First number (-1 to stop): "))
4     while nb != -1:
5         while not stack.is_empty() and nb > stack.top():
6             stack.pop()
7         stack.push(nb)
8         nb = int(input("Next number (-1 to stop): "))
9     display(stack)
```

```
10
11 def display(stack):
12     while not stack.is_empty():
13         print(stack.pop())
```

La fonction `display` sert juste à afficher la sous-suite extraite. Attention : puisque le top of stack contient toujours le plus petit élément du stack, la fonction `display` affiche les éléments par ordre croissant !

Proposition 2.4. *La fonction* `sous_suite` *détermine la sous-séquence décroissante de somme maximale d'une suite de longueur n en $\Theta(n)$ comparaisons.*

Démonstration. La boucle principale `while nb != 1` (l. 4) est exécutée exactement n fois puisque n est la longueur de la suite entrée sur l'input. Remarquons d'abord que l'instruction `stack.push(nb)` (l. 7) est exécutée exactement n fois également. Dès lors nous savons que le stack stockera n éléments au total, et donc nous pouvons en déduire que l'instruction `stack.pop()` (l. 7) ne pourra être exécutée qu'au plus n fois également ; et donc la seconde boucle `while` ne sera exécutée qu'au plus n fois. Nous en déduisons donc qu'il y a au moins n comparaisons (l. 4) et au plus $2n$ comparaisons (l. 4 et l. 5). Le nombre total de comparaisons est donc compris entre n et $2n$, ce qui revient à dire que le nombre total de comparaisons est $\Theta(n)$. □

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.

Résolution. Si le stack est représenté par une `list`, alors on peut inverser en place son contenu à l'aide de la méthode `list.reverse`, mais nous n'utilisons pas les propriétés d'un stack et nous *cassons* l'encapsulation. Nous supposons donc que nous n'avons accès qu'aux méthodes intrinsèques des stacks (i.e. `push`, `pop` et `is_empty`). Nous pourrions être tenté d'écrire quelque chose comme ceci :

```
1 def inversion(stack):
2    temp = Stack()
3    while not stack.is_empty():
4       temp.push(stack.pop())
5    while not temp.is_empty():
6       stack.push(temp.pop())
```

mais cela revient à inverser deux fois l'ordre du stack initial, autrement dit, ça ne sert à rien...

L'idée est donc de faire itérativement descendre le `top` à sa bonne position tout en laissant la partie supérieure du stack dans son ordre initial jusqu'à ce que tout le stack ait été inversé :

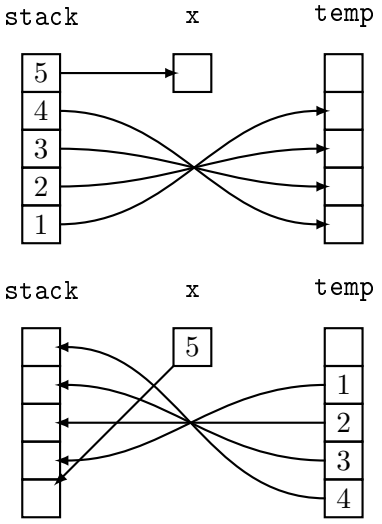
```
1 def inversion(stack):
2     n = len(stack)
3     temp = Stack()
4     for i in range(n, 1, -1):
```

```
5           x = stack.pop()
6           for j in range(1, i):
7               temp.push(stack.pop())
8           stack.push(x)
9           for j in range(1, i):
10              stack.push(temp.pop())
```

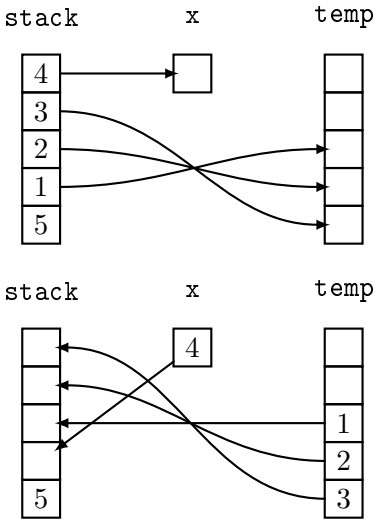
La variable `x` contient en effet le *top of stack*, et les `i-1` éléments suivants sont inversés dans le stack `temp`. `x` est ensuite ajouté dans `stack`, et les éléments inversés de `temp` sont alors ré-inversés dans `stack` (i.e. ces `i-1` éléments reprennent leur place dans le stack, mais en étant décalés d'une place vers le haut). En répétant cela `size` fois, on peut facilement se convaincre que `stack` contient toujours les mêmes éléments mais que leur ordre a été inversé.

Montrons sur un exemple ce que donne cet algorithme sur le stack `[1, 2, 3, 4, 5]` (où `5` est le *top of stack*) :

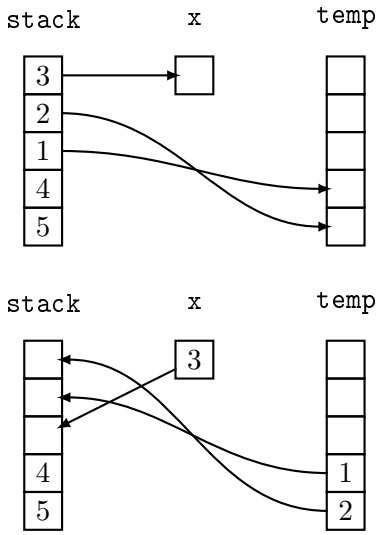
Descente de 5



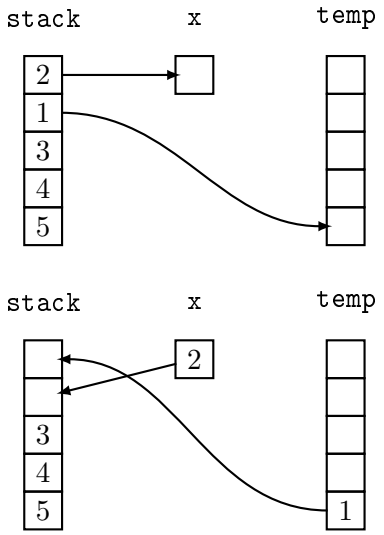
Descente de 4



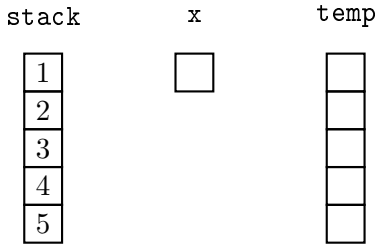
Descente de 3



Descente de 2



Descente de 1 On voit bien ici que le stack a été inversé puisque le dernier élément initial (i.e. 1) se retrouve au top :



Proposition 2.5. *Cet algorithme permet d'inverser le contenu d'un stack de taille n en place en $\Theta(n^2)$ opérations de stack (i.e.* `push` *et* `pop`*).*

Démonstration. Le `pop` de la l. 5 est exécuté exactement $n-1$ fois, idem pour le `push` de la l. 7 (car la boucle sur `i` s'exécute $n-1$ fois).

Les boucles sur `j` s'exécutent toutes deux (l. 6 et l. 9) `i-1` fois (pour `i` $\in [\![2, n]\!]$). Chacune de ces boucles exécute un `push` et un `pop`. Le nombre total d'opérations de stack est donc :

$$2(n-1)+4\sum_{i=2}^{n}(i-1) = 2(n-1)+4\sum_{i=1}^{n-1} i = 2(n-1)+4\frac{n(n-1)}{2} = 2(n+1)(n-1) = \Theta(n^2).$$

□

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.

Résolution. Il faut pour cela ajouter une méthode `__matmul__` :

```
def __matmul__(self, other):
   if self.n != other.m:
      raise ValueError('Dimension error')
   ret = MatriceComplexe(self.m, other.n)
   for i in range(self.m):
      for j in range(other.n):
         for k in range(self.n):
            ret[i,j] += self[i,k] * other[k,j]
   return ret
```

Implémentons également la méthode `__imatmul__` afin de permettre la syntaxe `A @= B` :

```
def __imatmul__(self, other):
   tmp = self @ other
   self.__m = tmp.__m
   self.__n = tmp.__n
   self.__mat = tmp.__mat
   return self
```

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 :

```
def empty_list(n):
   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.

Résolution. Cette classe `List` nécessite au moins les 3 attributs suivants : `container` (une liste), `size` (le nombre d'éléments non-nuls placés dans le conteneur), et `maxsize` (la taille physique du conteneur). En s'assurant que `maxsize` est toujours une puissance de 2, nous

pouvons garantir une complexité amortie linéaire pour les insertions (voir plus bas).

Les méthodes de base (constructeur, `__len__`, etc.) sont faciles à mettre en place ; il faut uniquement faire attention au traitement des indices (gérer les indices négatifs, l'indiçage cyclique des listes en Python et les dépassements des limites du conteneur).

```
def empty_list(n):
    return [None] * n

class List:
    def __init__(self):
        self.container_ = empty_list(1)
        self.size_ = 0
        self.maxsize_ = 1

    def __len__(self):
        return self.size_

    def __convert_index(self, idx, caller):
        msg = f'[{caller}] index out of range'
        if idx < 0:
            if idx < -len(self):
                raise IndexError(msg)
            idx += self.size_
        if idx >= self.size_:
            raise IndexError(msg)
        return idx

    def __getitem__(self, idx):
        idx = self.__convert_index(idx, 'List.__getitem__')
        return self.container_[idx]

    def __setitem__(self, idx, element):
        idx = self.__convert_index(idx, 'List.__setitem__')
        self.container_[idx] = element

    def __delitem__(self, idx):
        self.pop(idx)

    def __contains__(self, element):
        for idx in range(len(self)):
            if self.container_[idx] == element:
                return True
        return False

    def copy(self):
        ret = List()
        ret += self
        return ret
```

L'insertion d'un élément nécessite la création d'un nouveau conteneur de taille adaptée *seulement si* `size+1 >= maxsize`. En effet, c'est uniquement dans ce cas que l'élément que l'on désire insérer serait en dehors des limites physiques du conteneur. Lors de l'insertion simultanée de plusieurs éléments (par la méthode `extend`), il faut allouer un unique nouveau conteneur suffisamment grand pour contenir tous les éléments à ajouter. L'insertion d'un élément à un indice quelconque est juste une généralisation de `append`.

```
    def __resize_if_needed(self, new_size):
        if new_size > self.maxsize_:
            while self.maxsize_ < new_size:
                self.maxsize_ *= 2
            new_container = empty_list(self.maxsize_)
            for i in range(len(self)):
                new_container[i] = self.container_[i]
            self.container_ = new_container

    def append(self, element):
        self.__resize_if_needed(len(self)+1)
        self.container_[self.size_] = element
        self.size_ += 1

    def extend(self, other):
        self.__resize_if_needed(len(self) + len(other))
        for i in range(len(other)):
            self.container_[self.size_] = other[i]
            self.size_ += 1

    def insert(self, idx, element):
        if idx == len(self):
            self.append(element)
        if idx < 0:
            idx = max(0, idx+len(self))
        idx = min(len(self), idx)
        self.__resize_if_needed(len(self)+1)
        for i in range(len(self)+1, idx+1, -1):
            self.container_[i] = self.container_[i-1]
        self.container_[idx] = element
        self.size_ += 1
```

La suppression d'un élément ne nécessite pas de réallocation du conteneur (il est possible de libérer de la mémoire si `size` devient trop petit par rapport à `maxsize` (e.g. `2*size < maxsize`) mais n'est généralement pas implémenté pour éviter un traitement inutile).

```
    def __pop_last(self):
        if len(self) == 0:
            raise IndexError('Pop from empty list')
        self.size_ -= 1
        ret = self.container_[self.size_]
```

```
        self.container_[self.size_] = None
        return ret

    def pop(self, idx=-1):
        if idx == -1:
            return self.__pop_last()
        idx = self.__convert_index(idx, 'List.pop')
        ret = self.container_[idx]
        for i in range(idx+1, len(self)):
            self.container_[i-1] = self.container_[i]
        self.size_ -= 1
        return ret
```

Le reste de l'implémentation sert uniquement à faciliter l'utilisation de la classe :

```
    def __iadd__(self, other):
        self.extend(other)
        return self

    def __add__(self, other):
        ret = self.copy()
        ret += other
        return ret

    def __imul__(self, count):
        backup = self.copy()
        self.__resize_if_needed(len(self) * count)
        for k in range(1, count):
            self += backup
        return self

    def __mul__(self, count):
        ret = self.copy()
        ret.__resize_if_needed(len(ret) * count)
        for k in range(1, count):
            ret += self
        return ret

    def __iter__(self):
        return (element for idx, element in enumerate(self.container_) \
                        if idx < self.size_)

    def __eq__(self, other):
        if len(self) != len(other):
            return False
        for x, y in zip(self, other):
            if x != y:
                return False
        return True
```

```
35
36      def __neq__(self, other):
37          return not (self == other)
38
39      def __str__(self):
40          return str(self.container_[:len(self)])
```

Remarque : nous pouvons remarquer que `List.insert` n'effectue pas le même traitement d'indices que `List.__getitem__` (ou `List.__setitem__`). Nous voulons ici reproduire le comportement d'une liste Python, même si nous vous accordons largement le fait que CPython fait des conditions surprenantes pour considérer les indices inférieurs à `-len(self)` comme étant 0 ou ceux étant supérieurs à `len(self)` comme étant `len(self)`.

Proposition 2.6.

1. *L'insertion d'un élément dans une* `List` *de taille* n *prend* $\Theta(n)$ *opérations dans le pire des cas mais s'effectue en* $\Theta(1)$ *en moyenne ;*
2. *l'extension d'une* `List` *de taille* m *dans une* `List` *de taille* n *prend* $\Theta(m+n)$ *opérations dans le pire des cas et* $\Theta(m)$ *opérations en moyenne.*

Démonstration. Il est clair que dans le pire des cas, `List.append` va allouer un nouveau conteneur de taille $2n = \Theta(n)$ et le remplir. Afin de déterminer la complexité moyenne, il faut établir une mesure de probabilité sur la taille du conteneur (que l'on veut considérer uniforme). Il faut donc fixer une borne supérieure sur la taille d'un tel conteneur. Supposons donc qu'il existe un certain N entier (possiblement très grand) et considérons une instance de `List` de taille n quelconque entre 0 et $2^N - 1$.

L'exécution de `List.append` nécessite donc $\Theta(n)$ opérations dans le pire des cas (comme vu ci-dessus) mais nécessite $\Theta(1)$ opérations dans la plupart des cas (en effet, il y a réallocation seulement si n est une puissance de 2). Notons $C(n)$ le nombre d'opérations nécessaires à l'exécution de `List.append` sur une `List` de taille n. Si C désigne le nombre d'opérations en moyenne, nous avons donc :

$$2^N C = \sum_{n=0}^{2^N-1} C(n) = \sum_{j=0}^{N-1} \Theta(2^j) + \sum_{j \text{ s.t. } j \neq 2^k} \Theta(1) = \Theta(2^N) + \Theta(2^N - N) = \Theta(2^N).$$

De là, nous déduisons que $C = \Theta(1)$.

Il est clair que l'insertion itérative de m éléments est en $\Omega(m)$ opérations puisque cela correspond à :

$$C(n,m) \coloneqq C(n) + C(n+1) + \ldots + C(n+m-1) = \underbrace{\Omega(1) + \Omega(1) + \ldots + \Omega(1)}_{n+m-1-n+1=m} = \Omega(m).$$

Dans le pire des cas, il y a des réallocations dont la première et la dernière insertion. Il faut donc que n soit une puissance de 2 (disons $n = 2^k$) et $n+m-1$ est une puissance de 2 (disons $n+m-1 = 2^\ell$). Le nombre total d'opérations est donc :

$$\begin{aligned} C(n) + C(n+1) + \ldots + C(n+m-1) &= \left(C(2^k) + \ldots + C(2^\ell)\right) + (m - (\ell - k + 1))\Theta(1) \\ &= \Theta(2^{\ell+1}) = \Theta(m+n). \end{aligned}$$

Par un raisonnement similaire à ci-dessus, un appel à `List.extend` pour ajouter m éléments à une liste de n éléments requiert $\Theta(m+n)$ opérations dans le pire des cas. En effet, dans le pire des cas, une réallocation est nécessaire et il existe un K entier tel que $m+n \leq 2^K \lneqq 2(m+n)$ (qui implique donc $2^K = \Theta(m+n)$) ; or l'exécution de `extend` effectue une allocation d'une nouvelle liste de taille 2^K et la remplit pour donc $\Theta(2^K) = \Theta(m+n)$ opérations. S'il n'y a pas de réallocation, alors l'appel à `List.extend` requiert $\Theta(m)$ opérations (la copie des m valeurs dans un espace déjà pré-alloué).

Si nous notons $m = 2^L + k$, alors il y a réallocation si et seulement si n n'est pas sous la forme $2^K + j$ où $0 < j < 2^L(2^{K-L}-1) - k = 2^K - m$. Dès lors nous pouvons observer qu'à K fixé (tel que $K > L$), il y a $\sim m$ valeurs de j qui vont forcer une réallocation (donc un traitement en $\Theta(n) = \Theta(2^K)$) et qu'il y en a $\sim 2^K - m$ qui ne causeront pas de réallocation (donc pour un traitement en $\Theta(m)$).

Il y a donc $\Theta\left((2^K - m)m\right)$ opérations pour les cas sans réallocation et il y a :

$$\sum_{j=2^K-1-m}^{2^K} \Theta\left(2^K + j\right) = \Theta\left(m2^K\right) + \sum_{j=2^K-1-m}^{2^K} \Theta(j) = \Theta\left(m2^K\right) + \Theta\left((2^K - m)m\right)$$

opérations pour les cas avec réallocation. En sommant sur K (notons que le cas $K < L$ ne donne que des réallocations, mais dans un contexte asymptotique, les "petites valeurs" ne nous intéressent pas), nous obtenons un total de :

$$\Theta\left(m2^N\right) + \Theta\left(m(2^N - m)\right)$$

opérations. Afin d'avoir le nombre moyen d'opérations, il faut encore normaliser par 2^N, ce qui donne une moyenne de $\Theta(m)$ opérations (en utilisant le fait que $m < 2^N$). □

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.

Résolution. Adaptons la fonction `empty_list` de l'exercice précédent (de manière à gérer une valeur par défaut) :

```
def empty_list(N, v=0):
    return [v for _ in range(N)]
```

Nous voulons ensuite écrire une classe `Array` qui aurait en attribut une instance d'une classe `View`. Cette dernière doit maintenir une référen ce vers la liste servant de buffer ainsi que les indices de départ, d'arrivée et le pas entre les éléments. Lorsqu'une copie (même partielle) d'un objet de type `Array` sera faite, une nouvelle instance de la classe `View` devrai être créée, avec ses propres indices de départ, d'arrivée et de pas, mais avec la *même* référence vers le buffer.

Commençons par écrire la classe `View`. Nous voulons pouvoir en construire une instance soit sur base d'une taille (et initialiser le buffer à 0) soit sur base d'une autre instance de `View` (pour une copie). Nous avons donc le constructeur suivant :

```
class View:
    def __init__(self, param):
        if type(param) is int:
            self.buffer_ = empty_list(param)
            self.indices_ = (0, param, 1)
        elif isinstance(param, View):
            self.buffer_ = param.buffer_
            self.indices_ = tuple(param.indices_)
        else:
            raise TypeError('Unknown initialisation')
```

L'attribut `buffer_` est donc une référence vers une liste servant de buffer, et l'attribut `indices_` est un tuple contenant trois entiers : (i) l'indice de départ ; (ii) le premier indice non valide ; (iii) le pas. Nous avons gardé ici l'ordre et le sens des paramètres de `range`.

Nous voulons permettre l'indiçage en lecture par un entier ou par une slice. Nous devons alors, dans la méthode `__getitem__`, discriminer selon le type du paramètre reçu :

```
    def __getitem__(self, idx):
        if type(idx) is int:
            return self.__get_value(idx)
        elif isinstance(idx, slice):
            return self.__get_subview(
                *idx.indices(len(self))
            )
        else:
            raise TypeError(f'Expected int or slice. Got {type(idx)}')
```

Le cas où `idx` est un entier est simple à gérer :

```
    def __get_value(self, idx):
```

```
        idx = self.__convert_index(idx)
        self.__check_index(idx)
        return self.buffer_[idx]
```

où les méthodes `__check_index` et `__convert_index` sont définies comme ceci :

```
    def __check_index(self, idx):
        if not (0 <= idx < len(self)):
            raise IndexError('Out of Array')

    def __convert_index(self, idx):
        start, stop, step = self.indices_
        idx = start + idx*step
        return idx
```

La première s'assure que la valeur donnée en paramètre est bien un indice valide *pour le buffer* alors que la seconde traduit un indice *pour la vue* en un indice *pour le buffer*.

Le cas d'indiçage par un slice est légèrement plus long puisque nous devons retourner une autre instance de la classe `View`. Pour cela nous créons une copie de la vue actuelle (en passant `self` au constructeur) et puis nous adaptons les indices sur base des indices initiaux ainsi que de la slice :

```
    def __get_subview(self, start, stop, step):
        ret = View(self)
        ret.indices_ = (
            self.__convert_index(start),
            self.__convert_index(stop),
            self.indices_[2]*step
        )
        return ret
```

Il faut ensuite gérer l'indiçage en écriture. Bien que nous voulions pouvoir indicer un `Array` avec une slice, nous pouvons ne supporter que l'assignation d'une `View` par un indice entier :

```
    def __setitem__(self, idx, value):
        if type(idx) is not int:
            raise TypeError(f'Unknown index type: {type(idx)}')
        self.__check_index(idx)
        idx = self.__convert_index(idx)
        self.buffer_[idx] = value
```

Ajoutons tout de même une méthode `__iter__` basée sur une méthode `__iter_indices` afin de pouvoir convertir aisément un `Array` en un autre type de conteneur (e.g. via `list(a)`) :

```
    def fill(self, value):
        for idx in self.__iter_indices():
```

```
            self.buffer_[idx] = value

    def __iter_indices(self):
        start, stop, step = self.indices_
        idx = start
        for i in range(len(self)):
            yield idx
            idx += step
        # ou yield from range(*self.indices_)
```

Ajoutons également une méthode `fill` qui nous servira pour la classe `Array`. Cette méthodde doit remplacer toutes les valeurs de la vue par une valeur donnée en paramètre :

```
    def __iter__(self, value):
        for idx in self.__iter_indices():
            yield self.buffer_[idx]
```

Finalement, la longueur de la vue peut également se calculer sur base des indices car elle correspond au nombre de valeurs `k` possibles telles que `0 <= k*step <= stop - start` (si `step > 0`) :

```
    def __len__(self):
        start, stop, step = self.indices_
        if (stop-start) % step == 0:
            return (stop - start) // step
        else:
            return 1 + (stop - start) // step
```

Concernant la classe `Array`, son constructeur doit également supporter une initialisation par un paramètre entier et par une instance de `View` (tut comme la classe `View`), et la longueur d'un `Array` correspond à la longueur de la `View` sous-jacente :

```
class Array:
    def __init__(self, param):
        if type(param) is int:
            self.view_ = View(param)
        elif isinstance(param, View):
            self.view_ = param

    def __len__(self):
        return len(self.view_)
```

La méthode `__getitem__` se base entièrement sur la méthode `__getitem__` de `View` :

```
    def __getitem__(self, idx):
        if type(idx) is int:
            return self.view_[idx]
        elif isinstance(idx, slice):
            return Array(self.view_[idx])
```

```
        else:
            raise TypeError(f'Expected int or slice. Got {type(idx)}')
```

La méthode `__setitem__` se base sur la méthode `__setitem__` de `View` pour l'indiçage par un indice entier et sur la méthode `__getitem__` suivie d'un appel à `fill` sur la vue créée temporairement :

```
    def __setitem__(self, idx, value):
        if type(value) is not int:
            raise TypeError(f'Unknown value type: {type(value)}')
        if type(idx) is int:
            self.view_[idx] = value
        elif isinstance(idx, slice):
            self.view_[idx].fill(value)
        else:
            raise TypeError(f'Expected int or slice. Got {type(idx)}')
```

Remarque. *Dans l'exercice 1.1, nous vous avions précisé que vérifier le type d'une variable devait se faire via la fonction* `isinstance` *et pas via la fonction* `type` *pour des raisons liées à l'*héritage*, notions qui sera vue en détail en bloc 2. Cependant, pour une raison, fort discutable, Python considère que le type* `bool` *est une version* spécialisée *du type* `int` *(ce qui est une aberration). Dès lors* `isinstance(True, int)` *est évalué à* `True`*, or nous ne voulons pas tolérer un indiçage par un booléen (car soyons honnêtes, ça n'a aucun sens). Pour savoir si une variable est un entier, nous utilisons donc la fonction* `type`*.*

Exercice 2.7. Écrivez une classe `Range` se comportant comme la classe `range` de Python. 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.

Résolution. La classe `Range` doit maintenir 3 attributs en lecture seule (donc propriétés) : `start`, `stop` et `step` et doit égalment prendre en compte les cas où `start` est donné, ou non, au constructeur :

```
class Range:
    def __init__(self, a, b=None, step=1):
        if step == 0:
            raise ValueError('Range() arg 3 must not be zero')
        if b is None:
            self.start_ = 0
            self.stop_ = a
            self.step_ = step
        else:
            self.start_ = a
            self.stop_ = b
            self.step_ = step

    @property
    def start(self):
        return self.start_

    @property
    def stop(self):
        return self.stop_

    @property
    def step(self):
        return self.step_
```

Puisque `range` ests une séquence, il nous faut implémenter les méthodes `count` et `index` :

```
        return int(value in self)

    def index(self, value):
        if self.count(value) == 0:
            raise ValueError(f'{value} is not in range')
        return (value - self.start) // self.step
```

La première doit renvoyer un entier déterminant combien de fois la valeur `value` apparaît dans la séquence, or comme toutes les valeurs qui apparaissent sont celles sous la forme `start + i*step` telles que `i*step < stop - start`, soit la valeur apparaît une unique fois, soit elle n'apparait pas. Il est donc suffisant d'utiliser la méthode `__contains__` et de convertir le résultat (booléen donc) en entier. La méthode contains doit juste vérifier s'il existe un `i` tel que `value` satisfait la forme ci-dessus :

```
    def __contains__(self, value):
        return self.start <= value < self.stop \
          and (value - self.start) % self.step == 0
```

`__getitem__` doit supporter un paramètre entier qui est positif ou négatif car `range` supporte la syntaxe `r[-k]` pour désigner `r[len(r)-k]`. Il est donc suffisant de gérer le cas où l'indice est entre `-1` (compris) et `-len(self)` (non-compris) et y ajouter `len(self)` :

```
    def __getitem__(self, index):
        if index < 0:
            index += len(self)
        if index < 0 or index >= len(self):
            raise IndexError('Range object index out of range')
        return self.start + index*self.step
```

Afin d'implémenter `__iter__`, écrivons un itérateur adapté : `RangeIterator`. En effet, nous ne pouvons pas écrire quelque chose comme

```
def __iter__(self):
    return (self[i] for i in range(len(self)))
```

puisque nous n'avons pas accès à la classe `range`. Voici la classe `RangeIterator` :

```
class RangeIterator:
    def __init__(self, r):
        self.r = r
        self.i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.i >= len(self.r):
            raise StopIteration
        self.i += 1
        return self.r[self.i-1]
```

`__iter__` s'écrit dès lors simplement de la sorte :

```
    def __iter__(self):
        return RangeIterator(self)
```

Maintenant, puisque `__getitem__` et `RangeIterator` ont tous deux besoin de `__len__`, écrivons cette méthode. La seule subtilité est qu'i lfaut gérer les cas où `step > 0` et `step < 0`. Ces deux cas peuvent être traités séparément (e.g. dans un grand `if/else`) mais il est possible de tout gérer en un sur base d'une variable `sign` qui nous signifie si `step`est positif ou négatif :

```
    def __len__(self):
        sign = -1 if self.step < 0 else 1
        if self.stop * sign <= self.start * sign:
            return 0
        return 1 + (self.stop - sign - self.start) // self.step
```

Il nous reste à implémenter `__repr__` et `__str__`, qui sont triviales ainsi que `__reversed__`. Cependant, nous pouvons ruser ici car maintenant que nous avons déjà implémenté la mé-

thode `__iter__`, nous pouvons la réutiliser pour `__reversed__` :

```
    def __repr__(self):
        if self.step == 1:
            return f'Range({self.start}, {self.stop})'
        return f'Range({self.start}, {self.stop}, {self.step})'

    def __reversed__(self):
        return (self[len(self)-i-1] for i in Range(len(self)))

    def __str__(self):
        return repr(self)
```

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 $\Theta(1)$.

Résolution. Si nous conservons un premier stack contenant les élément dans l'ordre d'insertion, nous pouvons garantir que `push` et `pop` s'exécutent en $\Theta(1)$ dans tous les cas. Cependant, trouver l'élément minimum de ce stack prend $\Theta(n)$ opérations dans le pire des cas (et en moyenne) puisqu'il faut le parcourir en entier.

Nous pouvons cependant conserver un deuxième stack contenant les éléments minimaux : lors de l'insertion d'un élément `x`, si cet élément est inférieur au *top of stack* de ce second stack, alors nous insérons également `x` dans ce stack. De manière similaire, lors de la suppression d'un élément, si cet élément correspond au *top of stack* de ce second stack, alors nous devons également le supprimer. Cela nous permet d'assurer que les éléments de ce stack sont en ordre décroissant et que le *top of stack* correspond toujours au plus petit élément actuellement contenu dans le stack.

```
class MinStack:
    def __init__(self):
        self.elements = Stack()
        self.mins = Stack()

    def push(self, x):
        self.elements.push(x)
        if self.mins.is_empty() or x < self.mins.top():
            self.mins.push(x)

    def pop(self):
        x = self.elements.pop()
        if x == self.mins.top():
            self.mins.pop()

    def min(self):
        return self.mins.top()

    def is_empty(self):
```

```
20          return self.elements.is_empty()
```

Il est clair que `push` et `pop` s'effectuent toujours en $\Theta(1)$ puisque le traitement ajouté correspond au pire des cas à une comparaison et une insertion dans un stack. La consultation du minimum est également en $\Theta(1)$ puisqu'elle correspond à une requête du *top of stack*.

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.

Résolution. Afin de pouvoir utiliser une approche récursive, il faut réussir à exprimer le problème de recherche de l'élément maximal d'une séquence sur base de l'élément maximal d'une sous-séquence. Pour cela, partons du lemme suivant :

Lemme 2.7. *Soit $x = (x_0, \ldots, x_{n-1})$ une séquence de nombre réels. Notons x^k la sous-séquence $(x_k, \ldots, x_{n-1})$ est notons m_k l'élément maximal de la sous-séquence x^k. Alors la relation suivante est vérifiée pour tout $k \in [\![0, n[\![$:*

$$m_k = \begin{cases} x_k & \text{si } k = n-1 \\ \max\{x_k, m_{k+1}\} & \text{sinon.} \end{cases}$$

De plus, m_0 est l'élément maximal de la séquence x.

Démonstration. Ce résultat se vérifie facilement puisque l'élément maximal de $x^{n-1} = (x_{n-1})$ est obligatoirement x_{n-1} (car c'est le seul élément), et puisque l'élément maximal de x^k (pour $k < n-1$) est soit x_k soit un élément de x^{k+1} ; et s'il est dans x^{k+1}, alors c'est forcément m_{k+1}. □

Ce résultat nous permet donc de définir le problème de recherche de l'élément maximal d'un vecteur de taille n sur base du problème de recherche de l'élément maximal d'un vecteur de taille $n-1$. Pour permettre une forme de récursion, il nous faut également un cas de base, qui est lui aussi donné dans le lemme ci-dessus : $m_{n-1} = x_{n-1}$. Nous pouvons alors écrire notre solution :

```
1 def maximum(vect, idx=0):
2     n = len(vect)
3     if idx == n-1:
4         ret = vect[idx]
5     else:
6         ret = maximum(vect, idx+1)
7         if ret <= vect[idx]:
8             ret = vect[idx]
9     return ret
```

Remarquez bien que les appels (récursifs) à la fonction `maximum` se font bien dans l'ordre croissant du paramètre `idx` (représentant donc la position dans le vecteur), mais que la valeur du maximum est bien calculée *de droite à gauche dans le tableau*.

Remarque. *Rien ne nous oblige à procéder de cette manière-ci, et il est tout à fait possible de travailler de* gauche à droite *car le lemme 2.7 peut se formuler* de manière équivalente

sur base de sous-suites $x^k = (x_0, \ldots, x_k)$ avec la relation suivante :

$$m_k = \begin{cases} x_k & \text{si } k = 0 \\ \max\{x_k, m_{k-1}\} & \text{sinon.} \end{cases}$$

Il faut alors calculer m_{n-1} pour avoir l'élément maximal du vecteur.

Proposition 2.8. *La fonction* `maximum` *trouve l'élément maximum d'un vecteur en $n-1 = \Theta(n)$ comparaisons.*

Démonstration. il est clair, par le lemme 2.7, que déterminer m_k nécessite une comparaison de plus que de déterminer m_{k+1}. Or $m_{n-1} = x_{n-1}$ ne nécessite aucune comparaison. Cela donne bien $n-1$ comparaisons. □

Remarque. *La fonction* `max` *de Python effectue également $\Theta(n)$ comparaisons, et de même pour la fonction* `maximum_for` *suivante :*

```
1 def maximum_for(vect):
2     ret = -float('inf')
3     for value in vect:
4         if value > ret:
5             ret = value
6     return ret
```

Cependant, regardons maintenant le temps d'exécution de ces 3 fonctions (sur le même vecteur en entrée) :

```
[        max] Trouvé : 9997  (0.195ms, moyenne sur 100 appels)
[maximum_for] Trouvé : 9997  (0.301ms, moyenne sur 100 appels)
[    maximum] Trouvé : 9997  (7.143ms, moyenne sur 100 appels)
```

Nous y constatons que la fonction `maximum_for` *est plus lente que la fonction* builtin `max`*, ce qui s'explique par le fait que cette fonction* `max` *est codée directement dans l'interpréteur Python alors que la fonction* `maximum_for` *est quant à elle entièrement codée en Python et doit donc être interprétée. Cette dernière a donc l'air de prendre approximativement* 1.5 *fois le temps d'exécution de* `max`*.*

Remarquons également que la fonction `maximum` *est, elle, plus de* 30 *fois plus lente que* `max`*! Ici, la différence entre* compilé *et* interprété *n'est pas suffisante pour expliquer la différence de temps d'exécution. Cette différence est bien causée par l'implémentation récursive qui crée un* overhead *par rapport à une approche itérative. En effet, les appels successifs s'ajoutent l'un après l'autre sur le stack d'exécution (c.f. INFO-F105) ce qui nécessite beaucoup de temps.*

Parfois une implémentation récursive peut être une solution très élégante *à un problème que vous cherchez à résoudre. Ce n'est pas pour autant que cette solution est efficace. De plus, il est important de chercher un algorithme avec une complexité* la plus basse possible *(e.g. si une solution en $\Theta(n^3)$ existe, il n'y a aucun intérêt à implémenter une solution en $\Theta(n!)$) mais il est également important d'implémenter cette solution de manière efficace (mais cela sort du cadre purement algorithmique qui ne s'intéresse pas aux détails d'implémentation).*

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

Résolution. La solution de cet exercice est assez courte : pour inverser les entrées du vecteur, il suffit d'aller du début jusqu'au milieu en inversant l'élément `i` et l'élément `n-1-i`, pour `n`, la taille du vecteur. Il faut cependant faire attention à ne pas dépasser la moitié sinon on échangerait deux fois les mêmes entrées. Il faut donc uniquement s'assurer que l'indice actuel est inférieur à la moitié tout en gérant les cas où la longueur de `vect` est paire ou impaire.

```
def image_miroir(vect, idx=0):
    n = len(vect)
    if idx < n/2-1:
        image_miroir(vect, idx+1)
    vect[n-1-idx], vect[idx] = vect[idx], vect[n-1-idx]
```

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
```

Résolution. Le problème des tours de Hanoï est un exemple classique de récursivité. En effet, pour pouvoir déplacer n disques de la tour de départ à la tour d'arrivée sans enfreindre la règle sur l'ordre des disques, il faut d'abord (i) pouvoir déplacer les $n-1$ disques les plus petits sur la tour auxiliaire, (ii) déplacer le $n$ème disque sur la tour d'arrivée, et enfin (iii) déplacer les $n-1$ disques depuis la tour auxiliaire vers la tour d'arrivée. Cela permet d'exprimer simplement une solution au problème de Hanoï à n disques en fonction du problème de Hanoï à $n-1$ disques. Il faut également un cas de base qui est trivial : s'il y a une unique disque à déplacer, alors il suffit de le déplacer...

Voici une implémentation possible :

```
def hanoi_move_one(start, arrival):
     print(f"Déplacer un disque de {start} à {arrival}")

def hanoi(n=3, start="A", arrival="C", auxiliary="B"):
```

```
    if n == 1:
        hanoi_move_one(start, arrival)
    else:
        hanoi(n-1, start, auxiliary, arrival)
        hanoi_move_one(start, arrival)
        hanoi(n-1, auxiliary, arrival, start)
```

Notez qu'en prenant le cas $n = 0$ comme cas de base, nous pouvons écrire une fonction équivalente :

```
def hanoi_alternative(n=3, start='A', arrival='C', auxiliary='B'):
    if n > 0:
        hanoi_alternative(n-1, start, auxiliary, arrival)
        hanoi_move_one(start, arrival)
        hanoi_alternative(n-1, auxiliary, arrival, start)
```

Proposition 2.9. *Cet algorithme permet de résoudre le problème des tours de Hanoï à trois tours et n disques en $2^n - 1$ déplacements.*

Démonstration. Montrons cela par récurrence. Si $n = 1$, alors il y a une unique disque, et le mouvement `start` $\to$ `arrival` permet bien de résoudre le problème et le nombre de déplacements est bien $1 = 2^1 - 1$.

Maintenant supposons que l'algorithme permet de résoudre le problème des tours de Hanoï à $n - 1$ disques sans enfreindre la règle et montrons que l'on peut le résoudre pour n disques. Déplacer les $n - 1$ premiers disques de la tour de départ vers la tour auxiliaire n'enfreint pas la règle sur l'ordre par hypothèse de récurrence. Déplacer le $n$ème disque sur la tour d'arrivée est un unique déplacement et ne peut enfreindre la règle puisque tous les disques plus petits sont sur une autre tour. Pour finir, déplacer les $n-1$ premiers disques de la tour auxiliaire vers la tour d'arrivée n'enfreint toujours pas la règle par hypothèse de récurrence et puisque seuls des disques plus grands se situent sur cette dernière tour.

Dès lors, cet algorithme permet en effet de déplacer n disques de la tour de départ vers la tour d'arrivée en utilisant uniquement la tour auxiliaire. Le nombre de déplacements est alors :

$$(2^{n-1} - 1) + 1 + (2^{n-1} - 1) = 2 \cdot 2^{n-1} - 1 = 2^n - 1.$$

□

Remarque. *Nous pouvons aussi simplement observer que le nombre d'opérations H_n pour résoudre le problème des tours de Hanoï à n disques suit la relation de récurrence suivante :*

$$H_n = 2H_n + 1,$$

ce qui est un cas particulier de la proposition 1.25 pour $\alpha = 2$ et $\beta = 1$. Dès lors la solution est donnée par :

$$H_n = 2^n H_0 + 1\frac{2^n - 1}{2 - 1}.$$

Or $H_0 = 0$ car résoudre le problème des tours de Hanoï à 0 disques se fait en 0 opérations, d'où nous pouvons conclure :

$$H_n = 2^n - 1.$$

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.
```

Résolution. Pour afficher les nombres dans l'ordre inverse, il faut commencer à les afficher après les avoir tous récupérés sur l'input. De manière récursive, on peut dire plus précisément qu'on récupère un nombre, ensuite on récupère les $n - 1$ nombres restant et on les affiche dans l'ordre inverse et puis on affiche le nombre récupéré. Le cas de base est donc $n = 1$: dans ce cas, il suffit de récupérer un nombre et de l'afficher directement.

```
1 def inverse(n):
2     if n != 0:
3         valeur = input()
4         inverse(n-1)
5         print(valeur)
```

La partie importante de cet exercice est de réaliser que le code mis *avant* l'appel récursif sera exécuté dans l'ordre des appels récursifs alors que le code mis *après* l'appel récursif sera exécuté dans l'ordre inverse des appels.

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.
```

Résolution. Sur base de la remarque à la fin de l'exercice précédent, on peut se rendre compte qu'il est question ici de jouer sur l'ordre des opérations (i.e. avant ou après l'appel récursif) pour traiter les nombres soit de manière croissante (pour les nombres pairs), soit de manière décroissante (pour les nombres impairs). La solution est donc :

```
1 def pair_impair(n):
2     if n != 0:
3         valeur = int(input())
4         if valeur%2 == 0: # si pair
5             print(" ", valeur)
6             pair_impair(n-1)
7         else:
8             pair_impair(n-1) # si impair
```

```
9           print(" ", valeur)
```

Notons que puisque les nombres pairs sont affichés avant l'appel récursif associé, les nombres sont écrits sur l'output standard pendant que des nombres sont encore entrés sur l'entrée standard. Dans le cas usuel du terminal, cela peut nuire à la lisibilité. Afin de bien distinguer les nombres entrés par l'utilisateur et ceux affichés par le programme, nous ajoutons des espaces avant les `print`.

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.

Résolution. Fixons $j = 0$, et calculons donc :

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

Une solution *naïve* serait de créer explicitement les sous-matrices A^{0k} et de calculer récursivement le déterminant en suivant la formule ci-dessus :

```
1  def det_naive(a):
2      n = len(a)
3      if n == 1:
4          return a[0][0]
5      factor = -1
6      ret = 0
7      for k in range(n):
8          factor *= -1
9          sub_matrix = [[a[row][col] for col in range(n) if col != k] \
10                        for row in range(1, n)]
```

```
        minor = det_naive(sub_matrix)
        ret += factor*a[0][k]*minor
    return ret
```

Cette approche a cependant un gros défaut : une nouvelle matrice est recréée avant chaque appel récursif. Or le nombre d'appels récursifs est $\sim n!$. Donc non seulement l'espace de stockage nécessaire pour réaliser tous les appels est un $\Theta(n^3)$, mais en plus, une grande partie du temps d'exécution est dédié à créer ces sous-matrices. Il faut donc trouver un moyen de calculer le déterminant d'une matrice sans passer par des nouvelles (sous-)matrices.

Nous allons définir une fonction `det(a, j, ignored)` où `a` est la matrice (liste de listes), `j` est la ligne à partir de laquelle on considère la sous-matrice et `ignored` est une liste de booléens déterminant les colonnes considérées dans la sous-matrice. Le cas de base est `j == len(a)-1`, en quel cas on s'intéresse à une sous-matrice de taille 1×1 dont le déterminant est l'unique élément. Pour le cas récursif, nous avons la variable `factor` qui correspond au facteur $(-1)^k$, et la variable `minor` qui est le déterminant de la sous-matrice A^{0k}.

```
def det(a, j=0, ignored=None):
    n = len(a)
    # Création d'un vecteur pour ignorer les colonnes lors du 1 er appel
    if ignored is None:
        ignored = [False] * n
    ret = 0
    factor = -1 # (-1)^k
    if j == n-1: # La matrice a été réduite n-1 fois => on cherche
        l'unique élément
        k = 0
        while ignored[k]:
            k += 1
        ret = a[j][k]
    else: # La matrice n'a pas encore été réduite n-1 fois
        # Pour chaque colonne non ignorée
        for k in range(n):
            if not ignored[k]:
                factor *= -1
                ignored[k] = True
                # on calcule le déterminant de la sous-matrice
                # a[j+1:][ignored]
                minor = det(a, j+1, ignored)
                ignored[k] = False
                ret += factor*a[j][k]*minor
    return ret
```

Notez bien que la liste `ignored` est modifiée juste avant l'appel récursif à `det` (en passant la `k`ème entrée à `True`) et est remise à `False` juste après (cette forme de récursivité particulière s'appelle *backtracking*, voir les deux séances suivantes). C'est en effet comme cela que l'on *extrait* virtuellement les sous-matrices desquelles on prend le déterminant.

Remarque. *Calculer le déterminant d'une matrice est une opération assez courante en informatique, et un algorithme de complexité $\sim n!$ n'est pas envisageable. Il existe d'autres algorithmes (beaucoup plus efficaces) pour calculer le déterminant. Vous verrez le plus commun en bloc 2 (au cours INFO-F205) : la décomposition LU. Pour faire (très) résumé, le principe est de décomposer votre matrice A en deux matrices triangulaires L et U telles que $A = LU$ (de telles matrices existent toujours). Calculer le déterminant d'une matrice triangulaire est beaucoup plus simple : c'est simplement le produit des éléments de la diagonale. De plus, par construction de ces matrices, la matrice L a un déterminant égal à 1. Donc $\det A = \det(LU) = \det L \det U = \det U$. L'algorithme permettant le calcul de ces matrices L et U requiert $\mathcal{O}(n^3)$ opérations, ce qui est très largement plus intéressant (même si cela reste assez coûteux). Nous laissons les détails de cette approche au cours de* calcul formel et numérique *car ceci ne fait pas partie de la matière du cours.*

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`.

Résolution. Commençons par la version récursive dont le cass de base est simplement le cas où x et y sont à une distance inférieure à `eps` :

```
def recursive_agm(x, y, eps=1e-10):
    return x \
        if abs(x-y) < eps
        else recursive_agm((x+y)/2, sqrt(x*y), eps)
```

Une possibilité non-récursive est donnée ci-dessous :

```
def agm(x, y, eps=1e-10):
    a_n = a_nm1 = x
    g_n = g_nm1 = y
    while abs(a_n - g_n) >= eps:
        a_n = (a_nm1 + g_nm1) / 2
        g_n = sqrt(a_nm1 * g_nm1)
        a_nm1 = a_n
        g_nm1 = g_n
    return a_n # ou g_n
```

Il suffit uniquement de maintenir les valeurs actuelles de a_n et g_n ainsi que les valeurs

précédentes a_{n-1} et g_{n-1} afin de calculer les valeurs consécutives.

Proposition 2.10. *Ces deux fonctions requièrent moins de* $\log_2 \frac{|x-y|}{\varepsilon}$ *itérations pour calculer* $\mathrm{agm}(x, y)$ *à précision* ε.

Démonstration. Si nous notons $\delta_n = a_n - g_n$, l'erreur d'estimation après la $n$ème itération, nous avons l'inégalité suivante :

$$\begin{aligned}\delta_{n-1} = a_{n-1} - g_{n-1} &= (a_{n-1} - a_n) + (a_n - g_n) + (g_n - g_{n-1}) \\ &= \frac{1}{2}(a_{n-1} - g_{n-1}) + (a_n - g_n) + (g_n - g_{n-1}) \\ &\geq \frac{1}{2}\delta_{n-1} + \delta_n.\end{aligned}$$

En particulier, on peut en déduire :

$$\delta_n \leq \frac{1}{2}\delta_{n-1},$$

ou encore :

$$\delta_n \leq 2^{-n}\delta_0 = 2^{-n}\,|x - y|\,.$$

Dès lors, si $n > \log_2 \frac{|x-y|}{\varepsilon}$, nous savons que :

$$\delta_n \leq 2^{-\log_2 \frac{|x-y|}{\varepsilon}}\,|x - y| = \varepsilon.$$

□

Remarque. *Bien que ce ne soit pas le sujet ici, on peut déterminer des bornes nettement plus restrictives sur la vitesse de convergence de cet algorithme. En particulier, on peut garantir que l'algorithme nécessite au plus :*

$$\log_2 \log_2 \frac{1}{\varepsilon} + \log_2 |x - y| + 2$$

itérations (car la vitesse de convergence de δ_n est quadratique), autrement dit chaque itération supplémentaire va, en moyenne, au moins doubler le nombre de décimales valides (ce qui, soyons honnêtes, est assez costaud).

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.

Résolution. Il nous suffit d'appeler récursivement la fonction `linear_rec_rel` sur les indices `N-j` pour `j` entre `1` et `k` et puis d'effectuer la somme pondérée :

```
def linear_rec_rel(N, alpha, x):
    k = len(alpha)
    assert k == len(x)
    return x[N-1] if N <= k \
      else sum(alpha[j-1]*linear_rec_rel(N-j, alpha, x) for j in range(1,
          k+1))
```

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 $\mathcal{O}(n^{2.807\cdots})$ 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.

Résolution.

Proposition 2.11. *L'algorithme de Strassen produit le résultat de la multiplication de deux matrices.*

Démonstration. Par la décomposition de A et B en blocs, nous pouvons exprimer :

$$C = \begin{bmatrix} C^1 & C^2 \\ C^3 & C^4 \end{bmatrix} = AB = \begin{bmatrix} A^1 & A^2 \\ A^3 & A^4 \end{bmatrix} \begin{bmatrix} B^1 & B^2 \\ B^3 & B^4 \end{bmatrix} = \begin{bmatrix} A^1B^1 + A^2B^3 & A^1B^2 + A^2B^4 \\ A^3B^1 + A^4B^3 & A^3B^2 + A^4B^4 \end{bmatrix}.$$

Il faut alors vérifier le théorème de Strassen pour les 4 blocs de C indépendamment :

$$\begin{aligned} M^1 + M^4 - M^5 + M^7 = &\left(A^1B^1 + A^4B^1 + A^1B^4 + A^4B^4\right) + \left(A^4B^3 - A^4B^1\right) + \\ &\left(-A^1B^4 - A^2B^4\right) + \left(A^2B^3 - A^4B^3 + A^2B^4 - A^4B^4\right) \end{aligned}$$

$$
\begin{aligned}
&= A^1B^1 + A^2B^3 = C^1 \\
M^3 + M^5 &= \left(A^1B^2 - A^1B^4\right) + \left(A^1B^4 + A^2B^4\right) \\
&= A^1B^2 + A^2B^4 = C^2 \\
M^2 + M^4 &= \left(A^3B^1 + A^4B^1\right) + \left(A^4B^3 - A^4B^1\right) \\
&= A^3B^1 + A^4B^3 = C^3 \\
M^1 - M^2 + M^3 + M^6 &= \left(A^1B^1 + A^4B^1 + A^1B^4 + A^4B^4\right) + \left(-A^3B^1 - A^4B^1\right) \\
&\quad \left(A^1B^2 - A^1B^4\right) + \left(A^3B^1 - A^1B^1 + A^3B^2 - A^1B^2\right) \\
&= A^4B^4 + A^3B^2 = C^4.
\end{aligned}
$$

□

Le code proposé ici utilise le package `numpy` en Python qui propose une classe `np.ndarray` qui peut représenter un tableau n-dimensionnel pour n arbitraire. Ce package est **très** utilisé dans le monde scientifique et il vous est conseillé de vous y familiariser sans trop tarder.

La classe `np.ndarray` (qui s'instancie par l'intermédiaire de la fonction `np.array`) est un véritable ADT qui propose tout un tas de méthodes ainsi que les opérateurs classiques, c'est pour cette raison que nous l'utilisons ici (comme ça nous ne devons pas réimplémenter l'addition ou la soustraction). De plus, `numpy` gère un parallélisme automatique des opérations sur les `np.ndarray` si ceux-ci sont suffisamment grands afin de minimiser le temps passé en calculs.

Notre code va donc commencer par l'import de ce package (qui par convention est **toujours** importé comme `np`; bien qu'il ne soit pas impossible de donner un autre nom à ce package ou encore de le conserver sous son nom complet, ce serait une très mauvaise pratique et cela nuirait à la lisibilité du code). Il nous faut également la fonction `submatrix(M, i0, j0, n)` qui nous permet de récupérer la sous-matrice de `M` dont la première ligne correspond à la ligne `i0` de `M`, la première colonne correspond à la colonne `j0` de `M` et de `n` lignes et colonnes :

```
import numpy as np

def submatrix(M, i0, j0, n):
    return M[i0:i0+n,j0:j0+n]
```

Nous utilisons les *slices* Python pour cela qui sont gérées par `numpy`.

La fonction `strassen` peut alors s'implémenter comme ceci :

```
def strassen(A, B):
    n = len(A)
    if i == 1:
        return A*B
    As = [
        submatrix(A, i0,    j0,      n//2), # A^1
        submatrix(A, i0,    j0+n//2, n//2), # A^2
        submatrix(A, i0+n//2, j0,    n//2), # A^3
        submatrix(A, i0+n//2, j0+n//2, n//2) # A^4
```

```
    ]
    Bs = [
        submatrix(B, i0,    j0,     n//2), # B^1
        submatrix(B, i0,    j0+n//2, n//2), # B^2
        submatrix(B, i0+n//2, j0,   n//2), # B^3
        submatrix(B, i0+n//2, j0+n//2, n//2) # B^4
    ]
    Ms = [
        strassen(As[0]+As[3], Bs[0]+Bs[3]), # M^1
        strassen(As[2]+As[3], Bs[0]),     # M^2
        strassen(As[0],     Bs[1]-Bs[3]), # M^3
        strassen(As[3],     Bs[2]-Bs[0]), # M^4
        strassen(As[0]+As[1], Bs[3]),     # M^5
        strassen(As[2]-As[0], Bs[0]+Bs[1]), # M^6
        strassen(As[1]-As[3], Bs[2]+Bs[3]) # M^7
    ]
    Cs = [
        M1+M4-M5+M7, # C^1
        M3+M5,       # C^2
        M2+M4,       # C^3
        M1-M2+M3+M6  # C^4
    ]
    C = np.empty_like(A)
    for i in range(n//2):
        for j in range(n//2):
            C[i,j]          = Cs[0][i,j]
            C[i,j+n//2]     = Cs[1][i,j]
            C[i+n//2,j]     = Cs[2][i,j]
            C[i+n//2,j+n//2] = Cs[3][i,j]
    return C
```

Remarque. *L'algorithme de Strassen est un exemple classique d'application du* Master Theorem *pour la détermination de sa complexité. Ce théorème (que vous verrez plus en détail en bloc 2 en mathématiques discrètes,* MATH-F307*) dit (entre moult autres choses) que pour une relation de récurrence de la forme :*

$$x_n = \alpha x_{\frac{n}{\beta}} + \varphi(n),$$

pour $\alpha \geq 1, \beta \gneqq 1$, s'il existe un $\varepsilon > 0$ tel que $\varphi(n) = \mathcal{O}(n^{\log_\beta \alpha - \varepsilon})$, alors toute solution satisfait :

$$x_n = \mathcal{O}(n^{\log_\beta \alpha}).$$

Notons maintenant C_n le nombre d'opérations nécessaires pour l'exécution de l'algorithme de Strassen sur une matrice de taille $n \times n$. Nous pouvons écrire :

$$C_{2^n} = 7C_{2^{n-1}} + 18(2^n)^2.$$

En effet il y a les 7 multiplications pour calculer les matrices $M^1, \dots, M^7$ et 18 additions/soustractions de matrices $2^n \times 2^n$ (et chacune nécessite n^2 additions/soustractions). Si nous notons $N \coloneqq 2^n$, nous avons bien une relation de récurrence sous la forme :

$$C_N = \alpha C_{\frac{N}{\beta}} + \varphi(N),$$

pour $\alpha = 7, \beta = 2$ et $\varphi : \mathbb{N} \to \mathbb{N} : N \mapsto 18N^2 = \Theta(N^2) = \mathcal{O}(N^2)$.

En particulier, pour $\varepsilon \in (0, \log_2 7 - 2)$, nous avons bien $\varphi(N) = \mathcal{O}(N^{\log_2 7 - \varepsilon})$; dès lors nous savons que $C_N = \mathcal{O}(N^{\log_2 7})$, or $\log_2 7 \approx 2.807\dots$.

En un sens, cet algorithme *troque* des multiplications par des additions/soustractions. Or l'addition de matrices carrées $n \times n$ se fait en temps $\Theta(n^2)$ alors que le produit de deux matrices $n \times n$ se fait en $\Omega(n^2)$. La complexité de l'algorithme de Strassen est donc *meilleure* (au sens de la définition 10) que la complexité de l'approche naïve mais ce n'est pas pour autant que cet algorithme est préférable en pratique, surtout sur de petites matrices. Les notations O, Ω, Θ, etc. sont des concepts *asymptotiques* et n'ont de sens que pour n *très grand* (plus précisément pour n qui tend vers l'infini). En particulier, l'algorithme de Strassen souffre de ses 18 additions de sous-matrices qui sont très coûteuses pour n relativement petit.

L'*exposant de multiplication de matrices carrées*, noté ω, est la plus petite valeur réelle telle que le produit de deux matrices carrées $n \times n$ peut se réaliser en $\mathcal{O}(n^{\omega+\varepsilon})$ pour tout $\varepsilon > 0$. Nous savons que $\omega \geq 2$ de manière triviale et que $\omega \leq \log_2 7$ par l'algorithme de Strassen. Il existe même d'autres algorithmes *encore plus efficaces* que celui de Strassen (tout du moins asymptotiquement) qui permettent d'affiner la borne supérieure sur ω. La dernière en date a été publiée en 2021 (c.f. https://doi.org/10.1137/1.9781611976465.32) et annonce $\omega < 2.37286$, ce qui montre que ce sujet est encore très actif dans la recherche. Notons que certains (dont Coppersmith et Winograd qui ont proposé l'approche encore utilisée aujourd'hui pour affiner cette borne) ont conjecturé que $\omega = 2$, mais ceci n'a pas encore été prouvé.

Remarque. *Vous reprendrez bien une petite digression ? Ces algorithmes, depuis Coppersmith & Winograd, sont des algorithmes dits* galactiques*, c'est-à-dire qu'ils sont* asymptotiquement *efficaces mais inutilisables en pratique à cause de* gigantesques *constantes cachées par le grand O. Par exemple un algorithme en $\mathcal{O}(n^2)$ peut paraître efficace mais peut être bien moins efficace en pratique qu'un algorithme en $\mathcal{O}(2^n)$ si, par exemple le premier nécessite $\sim \exp(10^{100})n^2$ opérations alors que le second en nécessite 2^n.*

Ces algorithmes sont très rares en pratique, mais ils montrent qu'il est important de s'intéresser aux constantes de proportionnalité dès que possible.

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.

Résolution. L'idée derrière la réalisation de cet exercice est d'utiliser une liste de booléens dont l'entrée `i` vaut `True` si le `i`ème de l'ensemble est à prendre et vaut `False` sinon. La *solution partielle* est donc construite en mettant d'abord la `i`ème entrée à `True` (et puis en rappelant la fonction `generate_subsets`) et ensuite en la passant à `False` (et en rappelant à nouveau la fonction).

```
1  def generate_subsets(values, current=0, choices=None):
2      n = len(values)
3      if choices is None:
4          choices = [False] * n
5      if current == n:
6          print_subset_mask(values, choices)
7      else: # if current < n
8          choices[current] = True
9          generate_subsets(values, current+1, choices)
10         choices[current] = False
11         generate_subsets(values, current+1, choices)
```

La fonction `print_subset_mask` affiche simplement les éléments de `values` pour lesquels l'entrée de `choice` est à `True` :

```
1  def print_subset_mask(values, choices):
2      print('{ ', end='')
3      for mask, value in zip(choices, values):
4          if mask:
5              print(value, end=' ')
6      print('}')
```

Proposition 2.12. *Le nombre d'appels à* `print_subset_mask` *de cet algorithme est* 2^n.

Démonstration. Notons $C_{n\,j}$ le nombre d'appels récursifs lorsque `generate_subsets` est appelée avec `len(values)` $== n$ et `current` $== j$. Établissons la relation de récurrence suivante (pour $n \in \mathbb{N}$ et $j \in [\![0, n]\!]$) :

$$C_{n,j} = \begin{cases} 1 & \text{si } j = n \\ 2C_{n\,j+1} & \text{sinon.} \end{cases}$$

En effet, si $j = n$, i.e. si `current == len(values)` (l. 5-6), alors le programme exécute uniquement un appel à `print_subset_mask` (qui s'exécute en $\Theta(n)$ puisque cela nécessite de parcourir une unique fois l'entièreté des vecteurs `values` et `choices` et que le traitement de chaque élément se fait en temps constant) mais aucun appel récursif n'est exécuté ; et si $j \in]\!]1, n[\![$, alors la fonction `generate_subsets` fait deux appels récursifs (l. 9 et 11) avec le paramètre `current` augmenté de 1.

Le nombre total d'appels récursifs est donc $C_n \coloneqq C_{n\,0}$ que l'on peut calculer comme suit :

$$C_n = C_{n\,0} = 2C_{n\,1} = 2^2 C_{n\,2} = \ldots = 2^k C_{n\,k}$$

pour tout $k \in [\![1, n]\!]$. En particulier pour $n = k$:

$$C_n = 2^n C_{n\,n} = 2^n.$$

Remarque. *Intuitivement, on se rend vite compte que le nombre d'opérations est nécessairement $\Omega(n2^n)$ puisqu'un ensemble fini E de taille n admet exactement 2^n sous-ensembles distincts, et qu'afficher chacun de ces sous-ensembles doit au moins itérer sur tous les éléments du sous-ensemble en question (de taille $\sim \frac{n}{2}$ en moyenne). Le nombre minimum d'opérations est donc $\frac{n}{2}2^n$. Il est même en réalité $\Theta(n2^n)$.*

□

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}
```

Résolution. Gardons le principe de la liste de booléens `choices` et ajoutons un paramètre `k` qui représente le nombre d'éléments de `choices` *qu'il faut encore* passer à `True` :

```
1  def generate_subsets_k(values, k, current=0, choices=None):
2      n = len(values)
3      if choices is None:
4          choices = [False] * n
5      if k < 0:
6          return
7      elif k == 0:
8          print_subset_mask(values, choices)
9      elif current <= n-k:
10         choices[current] = True
11         generate_subsets_k(values, k-1, current+1, choices)
12         choices[current] = False
13         generate_subsets_k(values, k, current+1, choices)
```

Notez bien que lors des appels récursifs (l. 11 et 13), le paramètre `k` vaut soit `k-1` si `choices[current]` est à `True` puisqu'on nouvel élément a été sélectionné, soit `k` sinon puisque le nombre d'éléments restants est toujours le même.

Le test `if k < 0: return` (l. 5-6) n'est pas strictement nécessaire pour le bon fonctionnement de l'algorithme puisque la fonction `print_subsets_mask` n'est appelée que si `k == 0`.

Cependant, cela permet d'éviter un grand nombre d'appels récursifs non-nécessaires. En effet, si `k < 0`, peu importe l'ordre des appels récursifs suivants, `print_subsets_mask` ne sera jamais appelée puisque `k` ne vaudra jamais 0.

Proposition 2.13. *Le nombre d'appels à* `print_subsets_mask` *de cet algorithme est* $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Démonstration. Notons $C_{n\,k\,j}$ le nombre d'appels à `print_subsets_mask` lorsque la fonction `generate_subsets_k` est appelée avec `len(values) ==` n, `k ==` k, `current ==` j. $C_{n\,k\,j}$ satisfait la relation de récurrence suivante (pour $n \in \mathbb{N}$, $k \in [\![0, n]\!]$ et $j \in [\![0, n-k]\!]$) :

$$C_{n\,k\,j} = \begin{cases} 1 & \text{si } k = 0 \\ C_{n\,k\,j+1} + C_{n\,k-1\,j+1} & \text{sinon.} \end{cases}$$

On y reconnait les coefficients binomiaux qui satisfont cette relation, donc $C_{n\,k\,j} = \binom{n-j}{k}$. En effet :

$$\begin{aligned} C_{n\,k\,j+1} + C_{n\,k-1\,j+1} &= \binom{n-j-1}{k} + \binom{n-j-1}{k-1} \\ &= \frac{(n-j-1)!}{k!(n-j-k-1)!} + \frac{(n-j-1)!}{(k-1)!(n-j-k)!} \\ &= \frac{(n-j-1)!}{(k-1)!(n-j-k-1)!}\left(\frac{1}{n-j-k} + \frac{1}{k}\right) \\ &= \frac{(n-j-1)!}{(k-1)!(n-j-k-1)!}\frac{k+n-j-k}{(n-j-k)k} \\ &= \frac{(n-j-1)!(n-j)}{(k-1)!k(n-j-k-1)!(n-j-k)} = \frac{(n-j)!}{k!(n-j-k)!} \\ &= \binom{n-j}{k} = C_{n\,k\,j}. \end{aligned}$$

Le nombre total d'appels à `print_subsets_mask` est donc $C_{n\,k} := C_{n\,k\,0} = \binom{n-0}{k} = \binom{n}{k}$. □

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 $k = 25$.

Résolution. Au lieu d'avoir une liste `choices` de taille n contenant des booléens, nous allons opter pour une liste `indices` de taille k contenant les indices (dans l'ordre croissant) entre 0 et $n-1$ des nombres à prendre :

```
1 def combinaisons(n, k, current=0, indices=None):
2     if indices is None:
3         indices = [0] * k
4     if current == k:
5         print_indices(indices)
6     else:
7         if current > 0:
8             indices[current] = indices[current-1] + 1
```

```
9           while indices[current] <= n+1 - (k-current):
10              combinaisons(n, k, current+1, indices)
11              indices[current] += 1
```

`indices[current]` est initialisé à `indices[current-1]+1`, i.e. le premier nombre pas encore sélectionné (puisque `indices[current-1]` est le dernier nombre sélectionné), et ensuite cette entrée est incrémentée jusqu'à atteindre `n+1-(k-current)`. Cette boucle pourrait être remplacée par `while indices[current] <= n`, ce qui serait également fonctionnel, mais cela forcerait à itérer sur certains états qui de toute façon ne pourraient pas mener à une sélection de `k` nombres parmi les `n`.

En effet, si `indices[current] > n+1 - (k-current)`, alors le nombre de nombres qui peuvent encore être sélectionnés est `n-indices[current] < k-current-1`, i.e. il reste plus de nombres à choisir pour arriver à k que de nombres disponibles.

La fonction `print_indices` affiche simplement toutes les valeurs entrées de la liste donnée en paramètre :

```
1  def print_indices(indices):
2      print('{', end=' ')
3      for idx in indices:
4          print(idx, end=' ')
5      print('}')
```

Proposition 2.14. *Le nombre d'appels à* `print_indices` *de cet algorithme est* $\binom{n}{k}$.

Démonstration. Il est assez intuitif de se dire que le nombre d'appels à la fonction d'affichage doit être le même que précédemment puisque le nombre de sous-ensembles générés est le même ; seule la complexité en espace a changé.

Établissons maintenant la relation de récurrence suivante correspondant au nombre d'appels à `print_indices` lors d'un appel à `combinaisons(n, k, j)` tel que `indices[j] == v` :

$$C_{n\,j\,k\,v} = \begin{cases} 1 & \text{si } k-j=0, \\ \sum_{m=1}^{n-v+1-(k-j)} C_{n\,j+1\,k\,v+m} & \text{sinon.} \end{cases}$$

Nous voulons déterminer le nombre *total* d'appels à `print_indices` qui est donc $C_{n\,k} \coloneqq C_{n\,0\,k\,0}$.

Montrons que $C_{n\,j\,k\,v} = \binom{n-v}{k-j}$ satisfait la relation précédente. Pour cela, montrons que pour tout $n' \in \mathbb{N}^*$ et pour tout $k' \in [\![1, n']\!]$, l'égalité suivante est satisfaite :

$$\binom{n'}{k'} = \sum_{m=1}^{n'+1-k'} \binom{n'-m}{k'-1}.$$

En effet, procédons par récurrence. Notre cas de base est le cas $k' = n'$, où en effet $\binom{n'}{k'} = 1$ et :

$$\sum_{m=1}^{n'+1-k'} \binom{n'-m}{k'-1} = \sum_{m=1}^{1} \binom{n'-m}{k'-1} = \binom{n'-1}{k'-1} = 1.$$

Supposons maintenant que la propriété est satisfaite pour n' et k' et montrons qu'elle l'est également pour $n'+1$ et k' :

$$\begin{aligned}\sum_{m=1}^{n'+2-k'} \binom{n'+1-m}{k'-1} &= \sum_{m=2}^{n'+2-k'} \binom{n'+1-m}{k'-1} + \binom{n'+1-1}{k'-1} \\ &= \sum_{m=1}^{n'+1-k'} \binom{n'-m}{k'-1} + \binom{n'}{k'-1}.\end{aligned}$$

Ici, nous utilisons l'hypothèse de récurrence afin d'écrire :

$$\sum_{m=1}^{n'+1-k'} \binom{n'-m}{k'-1} + \binom{n'}{k'-1} = \binom{n'}{k'} + \binom{n'}{k'-1} = \binom{n'+1}{k'}.$$

Notons qu'une autre manière d'arriver à cette égalité est de répéter itérativement la propriété :

$$\binom{\nu}{\kappa} = \binom{\nu-1}{\kappa} + \binom{\nu-1}{\kappa-1}.$$

Ainsi nous pouvons écrire :

$$\begin{aligned}\binom{n'}{k'} &= \binom{n'-1}{k'-1} + \binom{n'-1}{k'} = \binom{n'-1}{k'-1} + \binom{n'-2}{k'-1} + \binom{n'-2}{k'} = \dots \\ &= \binom{n'-1}{k'-1} + \binom{n'-2}{k'-1} + \dots + \binom{k'}{k'-1} + \binom{k'-1}{k'-1}.\end{aligned}$$

Posons maintenant $n' = n - v$ et $k' = k - j$. Nous avons bien :

$$\begin{aligned}\sum_{m=1}^{n-v+1-(k-j)} C_{n\,j+1\,k\,v+m} &= \sum_{m=1}^{n-v+1-(k-j)} \binom{n-v-m}{k-j-1} = \sum_{m=1}^{n'+1-k'} \binom{n'-m}{k'-1} = \binom{n'}{k'} \\ &= \binom{n-v}{k-j} = C_{n\,j\,k\,v}.\end{aligned}$$

Nous pouvons alors conclure avec $C_{n\,k} = C_{n\,0\,k\,0} = \binom{n-0}{k-0} = \binom{n}{k}$. □

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 :

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

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 $[\![0, b]\!]$?

Résolution. Afin de minimiser le nombre de paramètres à passer lors des appels récursifs, écrivons une classe `IntegerLinearEquation` qui servira à résoudre ce problème :

```python
class IntegerLinearEquation:
    def __init__(self, a, b, y):
        self.a = a
        self.b = b
        self.y = y
        self.x = [0] * self.n
        self.current_sum = 0

    @property
    def n(self):
        return len(self.a)
```

Ajoutons une méthode `solve_rec` pour résoudre le problème de manière récursive. Cette méthode prend un paramètre `i` qui désigne l'indice des x_i qui est actuellement traité. Lorsque cet indice vaut n, c'est que toutes les variables x_i ont été traitées. Il faut alors vérifier si la solution actuelle est une solution à l'équation, et si oui, l'afficher. Si `i` $< n$, alors il faut itérativement construire toutes les solutions partielles pour x_i est continuer la construction partielle en appelant `solve_rec` avec `i+1`.

Afin de déterminer toutes les solutions partielles pour x_i, on peut itérer sur toutes les valeurs entre 0 et b (bornes comprises), mais en faisant cela, on génèrera des solutions que l'on sait impossibles (e.g. si $n = 10, b = 100$ et $y = 5$, il est clair qu'aucune entrée x_i ne peut être supérieur à $y = 5 < b$). Pour faire mieux, nous pouvons conserver une variable `current_sum` qui contient la valeur $a_0x_0 + a_1x_1 + \ldots + a_{i-1}x_{i-1}$ (afin de ne pas devoir systématiquement recalculer cette valeur) et l'utiliser pour trouver une borne supérieure sur la valeur x_i.

Lemme 2.15. *Soit x une solution de l'équation 2.1. Pour $i \in]\!]0, n[\![$, si $s = a_0x_0 + \ldots + a_{i-1}x_{i-1}$ on sait que :*

$$x_i \leq \min\left\{b, \left\lfloor \frac{y-s}{a_i} \right\rfloor\right\}.$$

Démonstration. Il est évident que si $x_i > b$, alors $x_0x_0 + a_1x_1 + \ldots + a_{n-1}x_{n-1}$ ne peut être égal à b (puisque tous les coefficients a_i sont strictement positifs). Supposons maintenant que $x_i > \left\lfloor \frac{y-s}{a_i} \right\rfloor$ et montrons que le vecteur $(x_0, \ldots, x_{n-1})$ ne peut pas être une solution de l'équation 2.1 :

$$\begin{aligned} a_0x_0 + \ldots + a_{n-1}x_{n-1} &= a_0x_0 + \ldots + a_{i-1}x_{i-1} + a_ix_i + \ldots + a_{n-1}x_{n-1} \\ &\geq a_0x_0 + \ldots + a_{i-1}x_{i-1} + a_ix_i = s + a_ix_i \\ &\gneq s + a_i \left\lfloor \frac{y-s}{a_i} \right\rfloor \geq s + a_i \frac{y-s}{a_i} = s + y - s = y, \end{aligned}$$

i.e. :

$$a_0x_0 + \ldots + a_{n-1}x_{n-1} \gneq y.$$

Dès lors on sait que $(x_0, \ldots, x_{n-1})$ ne peut pas être une solution de l'équation 2.1. □

Nous pouvons maintenant utiliser ce lemme pour déterminer les bornes sur l'itération des x_i, ce qui donne le code suivant :

```
1     def solve_rec(self, i=0):
2         if i == self.n:
3             if self.current_sum == self.y:
4                 print(self.x)
5         else:
6             max_val = min(self.b, (self.y-self.current_sum)//self.a[i])
7             for j in range(max_val+1):
8                 self.x[i] = j
9                 self.current_sum += self.a[i]*self.x[i]
10                self.solve_rec(i+1)
11                self.current_sum -= self.a[i]*self.x[i]
```

Notez bien que la construction de la solution partielle comprend les modifications de `current_sum` et l'assignation `x[i] = j`. La destruction de la solution partielle ne contient par contre que la modification de `current_sum` (l. 11) mais pas de réinitialisation de `x[i]`. En effet il n'est pas nécessaire de remettre `x[i]` à 0 puisque `x[i]` est de toute façon réassigné à l'entrée de la boucle. Et lors de la sortie de la boucle, l'algorithme retourne à l'appel récursif précédent où on suppose que les entrées `x` suivantes sont à 0.

Le flot d'exécution de la version non-récursive est le même que pour la version récursive, mais les appels récursifs doivent être remplacés par une boucle. En particulier, nous allons utiliser une variable locale `i` qui servira le même objectif que le paramètre de `solve_rec` et une boucle `while i >= 0` dans laquelle nous traiterons les cas `i == n` et `i < n`. L'équivalent d'un appel récursif est dès lors simplement un incrément de `i`, et le retour d'un appel récursif est un décrément de `i`.

Quelques remarques sur la solution :

— la liste `x` est initialisée à -1 au lieu de 0 car `x[i]` est incrémenté avant de modifier `current_sum` (l. 15-16), et donc la valeur initiale doit être -1 pour pouvoir gérer le cas `x[i] = 0`;

— la modification de `i`, `x[n-1]` et `current_sum` (l. 8-10) n'est pas nécessaire mais sert d'optimisation car si `i == `n, et qu'une solution est trouvée, alors x_{n-1} ne peut pas continuer à être augmenté, et il faut retourner à x_{n-2} ;

— lors de l'équivalent des retours des appels récursifs, `x[i]` est réinitialisé, ce qui n'était pas le cas dans `solve_rec` car une boucle `for` était utilisée pour assigner la valeur de `x[i]` alors que dans le cas non-récursif, `x[i]` est incrémenté ; donc sans réinitialisation, les x_i dépasseraient la borne du lemme 2.15 et ne reviendraient jamais en arrière.

```
1     def solve_non_rec(self):
2         self.x = [-1] * self.n
3         i = 0
4         while i >= 0:
5             if i == self.n:
6                 if self.current_sum == self.y:
7                     print(self.x)
8                     i -= 1
9                     self.current_sum -= self.a[i]*self.x[i]
10                    self.x[i] = -1
11                i -= 1
```

```
12                self.current_sum -= self.a[i]*self.x[i]
13            else:
14                if self.x[i] < min(b, (y-s)//self.a[i]):
15                    self.x[i] += 1
16                    self.current_sum += self.a[i]*self.x[i]
17                    i += 1
18                else:
19                    self.x[i] = -1
20                    i -= 1
21                    if i >= 0:
22                        self.current_sum -= self.a[i]*self.x[i]
```

Pour le bonus, dans le cas récursif il suffit de changer la boucle `for j in range(max_val+1)` (l. 7) en
`for j in range(b_prime, max_val+1)`, et dans le cas non-récursif, il faut remplacer les valeurs -1 par `b_prime-1`.

Proposition 2.16. *Le nombre d'appels récursifs de cet algorithme est $\mathcal{O}(b^n)$.*

Démonstration. Il est clair que tous les x_i sont contenus dans $[\![0, b]\!]$, il ne peut y avoir plus de $(b+1)^n = \mathcal{O}(b^n)$ appels récursifs représentant des solutions différentes. □

Remarque. *Bien que nous aimerions en dire plus concernant la complexité de cette approche, utiliser la borne $x_i \leq \left\lfloor \frac{y-s}{a_i} \right\rfloor$ n'est vraiment pas évident et nous n'obtiendrons pas de forme fermée intéressante. Il est alors important de savoir que b^n donne l'ordre de grandeur d'une borne supérieure, mais que cette borne n'est pas une borne inférieure pour autant. Nous avons donc ici bien un grand O et non un grand Theta.*

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`.

Résolution. Pour simplifier l'écriture (et la lecture) du code, écrivons une classe que nous appellerons `KnightWalker` pour résoudre ce problème. Son constructeur doit prendre les paramètres suivants :

- `width` (`int`) ;
- `height` (`int`) ;
- `end_x` (`int`) ;
- `end_y` (`int`).

Cette classe dispose des attributs suivants :

- `chess_board` (matrice) représentant l'échiquier ;
- `x` et `y` (`int`) représentant la position actuelle du cavalier ;
- `count` (`int`) contenant le nombre de déplacements déjà faits par le cavalier,

ainsi que des propriétés suivantes :

- `height` et `width` donnant respectivement m et n, les dimensions de l'échiquier ;
- `current_cell`, une propriété en lecture/écriture qui permet d'accéder à la case actuelle de l'échiquier (i.e. celle de coordonnée `(self.x, self.y)`).

Nous ajoutons également des attributs statiques (i.e. des attributs de la *classe* `KnightWalker` en elle-même et non des attributs des instances de cette classe) qui nous permettent de stoker les constantes de ce problème : `EMPTY = 0` représente une case vide dans `chess_board`, et `DELTA_X` et `DELTA_Y` sont des listes contenant les 8 mouvements possibles.

```
1  class KnightWalker:
2      EMPTY = 0
3      DELTA_X = [1, -1, 1, -1, 2, 2, -2, -2]
4      DELTA_Y = [2, 2, -2, -2, 1, -1, 1, -1]
5      def __init__(self, width, height, end_x, end_y):
6          self.chess_board = [[KnightWalker.EMPTY] * width for y in
               range(height)]
7
8          # Coordonnée de départ
9          self.x = 0
10         self.y = 0
11         # Coordonnée d'arrivée
12         self.end_x = end_x
13         self.end_y = end_y
14         # On place la première position du tour
15         self.current_cell = 1
16         # Compte combien de mouvements ont déjà  été faits
17         self.count = 1
18
19     @property
20     def height(self):
21         return len(self.chess_board)
22
```

```
23      @property
24      def width(self):
25          return len(self.chess_board[0])
26
27      def __len__(self):
28          return self.width * self.height
29
30      @property
31      def current_cell(self):
32          return self.chess_board[self.y][self.x]
33
34      @current_cell.setter
35      def current_cell(self, value):
36          self.chess_board[self.y][self.x] = value
```

La méthode `walk` va nous permettre de trouver une solution au problème (pour autant qu'il en existe une) par un backtracking :

```
1       def walk(self):
2           if self.found():
3               return True
4           else:
5               for i in range(0, len(KnightWalker.DELTA_X)):
6                   # On passe à la case suivante du tour
7                   self.x += KnightWalker.DELTA_X[i]
8                   self.y += KnightWalker.DELTA_Y[i]
9
10                  if self.is_position_valid() and self.current_cell ==
                        KnightWalker.EMPTY:
11                      # On ajoute la case au tour
12                      self.count += 1
13                      self.current_cell = self.count
14
15                      if self.walk():
16                          return True
17
18                      # On retire la case du tour
19                      self.current_cell = KnightWalker.EMPTY
20                      self.count -= 1
21
22                  # On revient à la case précédente
23                  self.y -= KnightWalker.DELTA_Y[i]
24                  self.x -= KnightWalker.DELTA_X[i]
25          return False
```

La construction e la solution partielle se fait en remplissant la matrice `chess_board` : on assigne la valeur `k` à `chess_board[y][x]` si la case `(x, y)` est la `k`ème case visitée. En procédant ainsi, il est facile de déterminer si une case n'a pas encore été visitée : il suffit de regarder si l'entrée de `chess_board` associée contient `KnightWalker.EMPTY` (i.e. 0).

La détection d'une solution se fait par la méthode `found` ci-dessous :

```
1     def found(self):
2         return self.count == len(self) \
3            and self.x == self.end_x \
4            and self.y == self.end_y
```

qui vérifie que la position actuelle est bien la position `(end_x, end_y)` demandée, et qui vérifie également que toutes les cases ont bien été visitées (par l'attribut `count`). Le backtracking en lui-même suit la même forme que les exercices de la séance précédente : pour toutes les solutions partielles possibles (i.e. tous les mouvements du cavalier), on regarde si la solution est acceptable ou non (via la méthode `is_position_valid` ci-dessous et en vérifiant que la case d'arrivée est bien vide) ; et si elle l'est, on continue de construire la solution.

```
1     def is_position_valid(self):
2         return 0 <= self.x < self.width \
3            and 0 <= self.y < self.height
```

Notez qu'il faut bien faire attention à détruire la solution partielle (l. 19-20) pendant le backtracking sinon `found` pourrait renvoyer `True` alors que la *solution partielle* actuelle pourrait ne correspondre à aucun chemin satisfaisant les déplacements d'un cavalier.

Proposition 2.17. *La méthode* `walk` *détermine s'il existe un parcours de cavalier (et le trouve) entre les cases* `(0, 0)` *et* `(end_x, end_y)` *en* $\mathcal{O}(7^{nm})$ *appels récursifs.*

Démonstration. Le stack des appels récursifs ne peut pas contenir plus de nm appels à `KnightWalker.walk` puisqu'un appel n'est fait que sur une case vide. Or il n'y a que nm cases sur l'échiquier. De plus, chaque appel à `walk` ne peut faire plus de 7 appels récursifs puisqu'il n'y a qu'au plus 7 cases accessibles libres depuis chaque case (à savoir les 8 mouvements possibles, moins la case de laquelle le cavalier vient qui n'est pas libre). Dès lors si nous notons C_k le nombre d'appels récursifs à `walk` lorsque `count ==` k, alors nous avons $C_{nm} = 0$ et $C_k \leq 7(C_{k+1} + 1)$ pour $k \lneq nm$. Nous en déduisons donc :

$$C_0 \leq 7(C_1 + 1) \leq 7^2(C_2 + 1) + 7^1 \leq 7^3 C_3 + 7^2 + 7^1 \leq \ldots \leq 7^k C_k + \sum_{\ell=1}^{k-1} 7^\ell.$$

En particulier, pour $k = mn$:

$$C_0 \leq \sum_{\ell=1}^{mn-1} 7^\ell = \frac{7^{mn} - 7}{7 - 1} = \mathcal{O}(7^{mn}).$$

□

Remarque. *Il est à nouveau difficile d'obtenir une meilleure borne sur le nombre d'appels récursifs.*

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)`.

Résolution. Écrivons une classe `Labyrinth` qui reprend globalement la structure de la classe `KnightWalker` :

```
class Labyrinth:
    EMPTY = '_'
    DELTA_X = [0, 1, 0, -1]
    DELTA_Y = [1, 0, -1, 0]
    def __init__(self, lab):
        self.lab = lab
        self.x = self.y = 0

    @property
    def width(self):
        return len(self.lab[0])

    @property
    def height(self):
        return len(self.lab)

    def __len__(self):
        return self.width * sef.height

    @property
    def current_cell(self):
        return self.lab[self.y][self.x]

    @current_cell.setter
    def current_cell(self, value):
        self.lab[self.y][self.x] = value

    def is_valid_position(self):
        return 0 <= self.x < self.width and 0 <= self.y < self.height
```

Ajoutons-y la méthode `solve` qui va s'occuper de trouver un chemin allant de `(0, 0)` à `(n-1, m-1)`. Pour cela, trouvons d'abord la condition de fin de récurrence, i.e. la fin de l'exécution de l'algorithme : lorsque `self.x == self.width-1` et `self.y == self.height-1`, alors c'est que l'on est arrivé à la fin du labyrinthe. Le backtracking en lui-même suit de très près la solution de l'exercice précédent : il faut itérer sur tous les mouvements possibles (donc sur toutes les valeurs de `Labyrinth.DELTA_X` et `Labyrinth.DELTA_Y`) et à chaque étape rappeler la méthode `solve` en ayant modifié `self.x` et `self.y` au préalable :

```
    def solve(self):
        if self.x == self.width-1 and self.y == self.height-1:
            self.current_cell = '.'
            return True # found a solution
        self.current_cell = '.'
```

```
        for i in range(len(Labyrinth.DELTA_X)):
            self.x += Labyrinth.DELTA_X[i]
            self.y += Labyrinth.DELTA_Y[i]
            if self.is_valid_position() and self.current_cell ==
                Labyrinth.EMPTY:
                if self.solve():
                    return True
            self.y -= Labyrinth.DELTA_Y[i]
            self.x -= Labyrinth.DELTA_X[i]
        self.current_cell = Labyrinth.EMPTY
        return False
```

Proposition 2.18. *La méthode* `solve` *trouve un chemin entre* `(0, 0)` *et* `(n-1, m-1)` *en* $\mathcal{O}(3^{mn})$ *appels récursifs.*

Démonstration. L'argument exactement le même que pour l'exercice précédent, mais ici $C_k \leq 3(1 + C_{k+1})$ pour $k \lneq mn$ car il n'y a plus que 4 mouvements possibles à chaque étape, contre 8 dans l'exercice précédent. □

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)`.

Résolution. Modifions la classe `Labyrinthe` afin de ne récupérer que le chemin le plus court. Pour cela, il nous faut ces attributs supplémentaires : `current_length` un entier représentant la longueur de la solution partielle actuelle, et `shortest_length`, un entier représentant la longueur de la meilleure solution trouvée jusqu'à présent ; ainsi que `optimal_solution`, une copie de la matrice `lab` correspondant à la meilleure solution trouvée. Le constructeur devient alors :

```
    def __init__(self, lab):
        self.lab = lab
        self.current_length = 0
        self.shortest_length = +float('inf')
        self.optimal_solution = deepcopy(self.lab)
        self.x = self.y = 0
```

La méthode `solve` doit être modifiée également pour y ajouter les deux éléments suivants :

— lorsqu'une solution est trouvée, il faut s'assurer qu'elle soit en effet meilleure que la meilleure solution trouvée jusqu'à présent avant de la sauvegarder ;
— lorsque la longueur de la solution partielle actuelle est au moins aussi grande que la longueur de la solution optimale trouvée jusqu'à présent, alors il n'est pas nécessaire de continuer la recherche.

```
    def solve(self):
        if self.x == self.width-1 and self.y == self.height-1:
            if self.current_length < self.shortest_length:
                self.shortest_length = self.current_length
                self.optimal_solution = deepcopy(self.lab)
```

```
6                 self.optimal_solution[-1][-1] = '.'
7             return
8         if self.current_length >= self.shortest_length:
9             return
10        self.current_cell = '.'
11        self.current_length += 1
12        for i in range(len(Labyrinth.DELTA_X)):
13            self.x += Labyrinth.DELTA_X[i]
14            self.y += Labyrinth.DELTA_Y[i]
15            if self.is_valid_position() and self.current_cell ==
                  Labyrinth.EMPTY:
16                self.solve()
17            self.y -= Labyrinth.DELTA_Y[i]
18            self.x -= Labyrinth.DELTA_X[i]
19        self.current_length -= 1
20        self.current_cell = Labyrinth.EMPTY
```

Proposition 2.19. *La méthode* `solve` *trouve un chemin de taille minimale entre* `(0, 0)` *et* `(n-1, m-1)` *en* $\mathcal{O}(3^{mn})$ *appels récursifs.*

Démonstration. L'argument est à nouveau le même que pour l'exercice précédent. □

Remarque. *Dans les trois exercices de cette séance, la borne supérieure sur le nombre d'appels récursifs que nous avons pu déterminer est à chaque fois exponentiel (ce qui, il faut l'admettre, est loin d'être exceptionnel). Au delà des considérations d'efficacité, remarquons que* $\mathcal{O}(7^n)$ *et* $\mathcal{O}(3^n)$ *se ressemblent fortement. Malheureusement, l'égalité* $7^n = \mathcal{O}(3^n)$ *est fausse.*

Nous ne pouvons dès lors pas agréger toutes les exponentielles à l'aide de la notation grand O *de cette manière. Cependant, remarquons que* $7 = 3^{\log_3 7} = 2^{\log_2 7}$. *Dès lors :* $7^n = 3^{\log_3(7)n} = 2^{\log_2(7)n}$. *Or pour tout* $k > 0$, *nous savons que* $kn = \mathcal{O}(n)$. *Nous pouvons alors exprimer* $7^n = 2^{\mathcal{O}(n)}$. *De manière générale, nous pouvons regrouper toutes les fonctions* $n \mapsto b^n$ *(à* $b > 1$ *fixé) par la notation* $2^{\mathcal{O}(n)}$.

Gardez alors bien en tête que $\mathcal{O}(2^n)$ *et* $2^{\mathcal{O}(n)}$ *ne représentent pas la même chose ! Le premier représente les fonctions bornées (en ordre de grandeur) par* la *fonction* $n \mapsto 2^n$ *alors que le second représente les fonctions bornées (en ordre de grandeur) par* une *fonction de la forme* $n \mapsto b^n$ *pour un certain* $b > 1$.

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`.

Résolution. Afin de résoudre cet exercice, nous pouvons nous inspirer de l'exercice 5.2 : si nous générons tous les sous-ensembles de taille 3, il nous suffit de vérifier si ces sous-ensembles somment à 0. Pour cela, nous procédons de manière récursive avec comme cas de base le cas `idx == n` (si on a visité tout le vecteur). Si on n'est pas dans ce cas de base, il faut s'assurer qu'il reste encore des éléments à ajouter à notre sous-ensemble (attribut `choices`), et s'il en reste, il y a deux appels récursifs : soit on cherche un sous-ensemble de même taille parmi les éléments suivants l'indice `idx`, soit on cherche un sous-ensemble

de taille moins un parmi les éléments suivants :

```python
class ThreeSum:
    def __init__(self, vect):
        self.vect = vect
        self.choices = []
        self.n = len(self.vect)

    def solve(self):
        if self._solve():
            return self.choices
        else:
            return None

    def _solve(self, idx=0, current_sum=0):
        if idx == self.n:
            return False
        if len(self.choices) == 3:
            return current_sum == 0
        if self._solve(idx+1, current_sum):
            return True
        self.choices.append(self.vect[idx])
        if self._solve(idx+1, current_sum+self.vect[idx]):
            return True
        self.choices.pop()
        return False
```

Proposition 2.20. *La méthode* `_solve` *exécute* $\binom{n}{3} \sim \frac{n^3}{6} = \Theta(n^3)$ *appels récursifs dans le pire des cas.*

Démonstration. Nous avons vu qu'il existe $\binom{n}{k}$ sous-ensembles de taille k, et donc en particulier $\binom{n}{3} = \frac{1}{6}n(n-1)(n-2)$. Dans le pire des cas, il faut tous les générer. □

Remarque (à titre indicatif)**.** *Ce problème est bien connu en algorithmique et s'appelle 3-SUM. Il peut être résolu en temps* $\Theta(n^2)$ *de manière non récursive de la manière suivante :*

```python
def optimized_three_sum(vect):
    vect = sorted(vect) # n log n
    n = len(vect)
    for i in range(n-2): #O(n)
        j = i+1
        k = n-1
        while j < k: # O(n)
            sol = [vect[i], vect[j], vect[k]]
            total = sum(sol)
            if total == 0:
                return sol
            elif total > 0:
```

```
13                k -= 1
14            else:
15                j += 1
```

En effet, le vecteur `vect` *est d'abord trié (donc en* $\mathcal{O}(n \log n)$*) et puis la boucle* `for` *s'exécute* $\Theta(n)$ *fois dans le pire des cas, et la boucle* `while` *s'exécute* $\mathcal{O}(n)$ *fois. En particulier, le nombre d'exécutions de la boucle* `while` *(dans le pire des cas) est* $n-2+n-3+\ldots+1 = \frac{1}{2}(n-2)(n-1) = \Theta(n^2)$.

Nous savons donc que n^2 *est une borne supérieure, mais nous ne savons pas si c'est également une borne inférieure. Autrement dit, il reste une question ouverte : est-ce que 3-SUM peut être résolu en temps* $\mathcal{O}(n^{2-\varepsilon})$ *pour un certain* $\varepsilon > 0$ *?*

Il est conjecturé que la réponse à cette question est non *mais aucune preuve n'a été trouvée à ce jour.*

Cependant, à ce stade-ci, il ne doit pas être clair pour vous ce à quoi peut correspondre un algorithme s'exécutant en $\Theta(n^m)$ *pour un certain* m *non-entier. Une digression sur le sujet peut être trouvée dans la discussion sur l'algorithme de Strassen (exercice supplémentaire 4.6).*

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.

Résolution. Écrivons une classe `KPeacefulQueens` contenant comme attributs l'état du plateau et le nombre de reines blanches et noires qu'il faut encore y ajouter :

```
class KPeacefulQueens:
    EMPTY = 0
    BLACK = 1
    WHITE = 2
    COLOURS = (BLACK, WHITE)
    #         LL   D  LR  R   UR   U  UL   L
    DELTA_X = [-1, 0, +1, +1, +1, 0, -1, -1]
    DELTA_Y = [+1, +1, +1, 0, -1, -1, -1, 0]
    DELTAS = list(zip(DELTA_X, DELTA_Y))
    def __init__(self, n, k):
        self.n = n
        self.board = [[KPeacefulQueens.EMPTY] * n for _ in range(n)]
        self.ks = [k, k] # [k_black, k_white]
        self.y = self.x = 0
```

Nous avons également les attributs `x` et `y` qui déterminent la position actuelle qui doit être analysée. Il nous faut alors écrire une méthode `solve` résolvant ce problème par backtracking. En particulier, il faut, pour chaque position `(x, y)` tenter d'y mettre soit une reine noire, soit une reine blanche, soit aucune reine et ensuite poursuivre la construction en partant de là. Notre cas de base est identifié comme le cas où les k reines noires et blanches ont été placées et que l'état est *valide* (dans le sens où aucune reine n'attaque une reine de l'autre couleur). Si ce cas est atteint, alors une solution a été trouvée et il n'est plus nécessaire de chercher. Notons qu'un autre cas de base existe : si toutes les cases ont reçu une assignation et que l'état n'est toujours pas valide ou qu'il faut encore ajouter des reines. Dans ce cas, il ne faut pas continuer à construire une solution (car la solution est entièrement construite) :

```
    def solve(self):
        if self.ks == [0, 0]:
            return True
        if self.out_of_board(self.x, self.y):
            return False
        for colour in KPeacefulQueens.COLOURS:
            if self.ks[colour-1] > 0 and self.step(colour):
                return True
        self.next_pos()
        if self.solve():
            return True
        self.prev_pos()
        return False
```

Notons l'existence des méthodes `next_pos` et `prev_pos` qui servent juste à passer à la position `(x, y)` suivante (ou précédente) :

```
    def next_pos(self):
        self.x = (self.x+1) % self.n
        if self.x == 0:
            self.y += 1

    def prev_pos(self):
        if self.x == 0:
            self.y -= 1
        self.x = (self.x-1) % self.n
```

La méthode `out_of_board` est une méthode booléeenne qui détermine si la position actuelle est en dehors des bornes :

```
    def out_of_board(self, x, y):
        return not (0 <= x < self.n and 0 <= y < self.n)
```

La boucle `for` dans `solve` teste uniquement la possibilité de mettre une reine sur la position actuelle, et le cas où la case est laissée vide est gérée par l'appel récursif à la fin. La tentative de placement d'une reine d'une couleur donnée à la position actuelle est gérée par la méthode `step` qui tente de placer la reine, vérifie que l'état est toujours valide et puis continue la récursion en appelant à nouveau la méthode `solve` :

```
    def step(self, colour):
        self.board[self.y][self.x] = colour
        self.ks[colour-1] -= 1
        if self.is_ok():
            self.next_pos()
            if self.solve():
                return True
            self.prev_pos()
        self.ks[colour-1] += 1
        self.board[self.y][self.x] = KPeacefulQueens.EMPTY
        return False
```

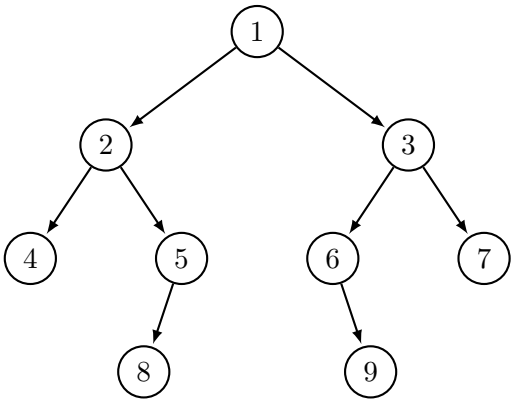
Déterminer la validité de l'état se fait via la méthode `is_ok` qui regarde dans chacune des 8 directions (inter)cardinales si une reine de l'autre couleur est présente (via la méthode `first_non_empty`) :

```
    def first_non_empty(self, x, y, dx, dy):
        while True:
            x += dx
            y += dy
            if self.out_of_board(x, y):
                return -1
            if self.board[y][x] != KPeacefulQueens.EMPTY:
                return self.board[y][x]
```

```
9
10      def is_ok(self):
11          other_colour = KPeacefulQueens.WHITE if self.board[self.y][self.x]
                == KPeacefulQueens.BLACK else KPeacefulQueens.BLACK
12          for dx, dy in KPeacefulQueens.DELTAS:
13              if self.first_non_empty(self.x, self.y, dx, dy) == other_colour:
14                  return False
15          return True
```

Séance 7 — Arbres (partie 1)

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



Résolution.

Définition 12. soit T un arbre binaire de sommets V. Un parcours de T est dit :

— *préfixé* si pour tout sommet v :

1. v est traité avant tous ses descendants ;
2. les descendants gauches de v sont traités avant ses descendants droits.

— *infixé* si pour tout sommet v :

1. les descendants gauches de v sont traités avant v ;
2. les descendants droits de v sont traités après v.

— *suffixé* si pour tout sommet v :

1. v est traité après tous ses descendants ;
2. les descendants gauches de v sont traités avant ses descendants droits.

Remarque. *Un parcours peut ne correspondre à aucune de ces trois définitions. Pouvez-vous en trouver un exemple ?*

Ces trois types de parcours peuvent tous être implémentés à l'aide d'un stack :

Voici les parcours :

préfixé

sommet actuel : 1 on traite ce sommet (1) que l'on traite, et puis on passe au fils gauche ;

sommet actuel : 2 on traite ce sommet (2) et on passe au fils gauche ;

sommet actuel : 4 on traite ce sommet (4), mais il n'a pas de fils gauche ou droit donc on remonte au parent ;

sommet actuel : 2 le fils gauche a déjà été traité donc on passe au fils droit ;

sommet actuel : 5 on traite ce sommet (5) et on passe au fils gauche ;

sommet actuel : 8 on traite ce sommet (8) mais il n'a pas de fils gauche ou droit donc on remonte au parent ;

sommet actuel : 5 le fils gauche a déjà été traité mais il n'y a pas de fils droit donc on remonte au parent ;

sommet actuel : 2 le fils gauche a déjà été traité et le fils droit aussi donc on remonte au parent ;

sommet actuel : 1 le fils gauche a déjà été traité donc on passe au fils droit ;

sommet actuel : 3 on traite ce sommet (3) et on passe au fils gauche ;

sommet actuel : 6 on traite ce sommet (6) mais il n'a pas de fils gauche donc on passe au fils droit ;

sommet actuel : 9 on traite ce sommet (9) mais il n'a pas de fils gauche ou droit donc on remonte au parent ;

sommet actuel : 6 le fils droit a déjà été traité donc remonte au parent ;

sommet actuel : 3 le fils gauche a déjà été traité donc on passe au fils droit ;

sommet actuel : 7 on traite ce sommet (7) mais il n'a pas de fils gauche ou droit donc on remonte au parent ;

sommet actuel : 3 le fils droit a déjà été traité donc on remonte au parent ;

sommet actuel : 1 le fils droit a déjà été traité donc on remonte au parent ;

sommet actuel : $\perp$ on a quitté l'arbre donc le parcours est terminé.

Nous pouvons également voir le parcours préfixé par le pseudo-code suivant :

```
PREFIX(node):
    TRAITEMENT(node)
    PREFIX(node.left)
    PREFIX(node.right)
```

En commençant à la racine, cela nous donne donc :

```
PREFIX(1):
    TRAITEMENT(1)                # 1
    PREFIX(2):
        TRAITEMENT(2)            # 2
        PREFIX(4):
            TRAITEMENT(4)        # 4
            PREFIX(None)
            PREFIX(None)
        PREFIX(5):
            TRAITEMENT(5)        # 5
            PREFIX(8):
                TRAITEMENT(8)    # 8
                PREFIX(None)
                PREFIX(None)
            PREFIX(None)
    PREFIX(3):
        TRAITEMENT(3)            # 3
        PREFIX(6):
            TRAITEMENT(6)        # 6
            PREFIX(None)
            PREFIX(9):
                TRAITEMENT(9)    # 9
                PREFIX(None)
                PREFIX(None)
        PREFIX(7):
```

```
            TRAITEMENT(7)         # 7
            PREFIX(None)
            PREFIX(None)
```

L'ordre de parcours est donc :

```
1 2 4 5 8 3 6 9 7
```

infixé

sommet actuel : 1 on passe au fils gauche

sommet actuel : 2 on passe au fils gauche ;

sommet actuel : 4 ce sommet n'a pas de fils gauche donc on le traite (4), mais il n'a pas de fils droit donc on remonte au parent ;

sommet actuel : 2 le fils gauche a été traité donc on traite ce sommet (2) et on passe au fils droit ;

sommet actuel : 5 on passe au fils gauche ;

sommet actuel : 8 ce sommet n'a pas de fils gauche donc on le traite (8), mais il n'a pas de fils droit donc on remonte au parent ;

sommet actuel : 5 le fils gauche a été traité donc traite ce sommet (5), mais il n'a pas de fils droit donc on remonte au parent ;

sommet actuel : 2 le fils droit a été traité donc on remonte au parent ;

sommet actuel : 1 le fils gauche a été traité donc on traite ce sommet (1) et on passe au fils droit ;

sommet actuel : 3 on passe au fils gauche ;

sommet actuel : 6 ce sommet n'a pas de fils gauche donc on le traite (6) et on passe au fils droit ;

sommet actuel : 9 ce sommet n'a pas de fils gauche donc on le traite (9), mais il n'a pas de fils droit donc on remonte au parent ;

sommet actuel : 6 le fils droit a été traité donc on remonte au parent ;

sommet actuel : 3 le fils gauche a été traité donc on traite ce sommet (3) et on passe au fils droit ;

sommet actuel : 7 ce sommet n'a pas de fils gauche donc on le traite (7), mais il n'a pas de fils droit donc on remonte au parent ;

sommet actuel : 3 le fils droit a été traité donc on remonte au parent ;

sommet actuel : 1 le fils droit a été traité donc on remonte au parent ;

sommet actuel : $\perp$ on a quitté l'arbre donc le parcours est terminé.

le parcours infixé peut s'écrire avec le pseudo-code suivant :

```
INFIX(node):
    INFIX(node.left)
    TRAITEMENT(node)
    INFIX(node.right)
```

En commençant à la racine, cela nous donne donc le parcours suivant :

```
INFIX(1):
    INFIX(2):
        INFIX(4):
            INFIX(None)
            TRAITEMENT(4)         # 4
```

```
            INFIX(None)
        TRAITEMENT(2)             # 2
        INFIX(5):
            INFIX(8):
                INFIX(None)
                TRAITEMENT(8)     # 8
                INFIX(None)
            TRAITEMENT(5)         # 5
            INFIX(None)
    TRAITEMENT(1)                 # 1
    INFIX(3):
        INFIX(6):
            INFIX(None)
            TRAITEMENT(6)         # 6
            INFIX(9):
                INFIX(None)
                TRAITEMENT(9)     # 9
                INFIX(None)
        TRAITEMENT(3)             # 3
        INFIX(7):
            INFIX(None)
            TRAITEMENT(7)         # 7
            INFIX(None)
```

L'ordre de parcours est donc :

```
4 2 8 5 1 6 9 3 7
```

suffixé

sommet actuel : 1 on passe au fils gauche

sommet actuel : 2 on passe au fils gauche ;

sommet actuel : 4 ce sommet n'a pas de fils gauche ni de fils droit donc on le traite (4) et on remonte au parent ;

sommet actuel : 2 le fils gauche a été traité donc on passe au fils droit ;

sommet actuel : 5 on passe au fils gauche :

sommet actuel : 8 ce sommet n'a pas de fils gauche ni de fils droit donc on le traite (8) et on remonte au parent ;

sommet actuel : 5 le fils gauche a été traité et ce sommet n'a pas de fils droit donc on le traite (5) et on remonte au parent ;

sommet actuel : 2 le fils droit a été traité donc on traite ce sommet (2) et on remonte au parent ;

sommet actuel : 1 le ils gauche a été traité donc on passe au fils droit ;

sommet actuel : 3 on passe au fils gauche ;

sommet actuel : 6 ce sommet n'a pas de fils gauche donc on passe au fils droit ;

sommet actuel : 9 ce sommet n'a pas de fils gauche ni de fils droit donc on le traite (9) et on remonte au parent ;

sommet actuel : 6 le fils droit a été traité donc on traite ce sommet (6) et on remonte au parent ;

sommet actuel : 3 le fils gauche a été traité donc on passe au fils droit ;

sommet actuel : 7 ce sommet n'a pas de fils gauche ni de fils droit donc on le traite (7) et on remonte au parent ;

sommet actuel : 3 le fils droit a été traité donc on traite ce sommet (3) et on remonte au parent ;

sommet actuel : 1 le fils droit a été traité donc on traite ce sommet (1) et on remonte au parent ;

sommet actuel : $\perp$ on a quitté l'arbre donc le parcours est terminé.

Le pseudo-code pour un parcours suffixé est :

```
SUFFIX(node):
    SUFFIX(node.left)
    SUFFIX(node.right)
    TRAITEMENT(node)
```

En commençant à la racine, cela nous donne donc :

```
SUFFIX(1):
    SUFFIX(2):
        SUFFIX(4):
            SUFFIX(None)
            SUFFIX(None)
            TRAITEMENT(4)          # 4
        SUFFIX(5):
            SUFFIX(8):
                SUFFIX(None)
                SUFFIX(None)
                TRAITEMENT(8)      # 8
            SUFFIX(None)
            TRAITEMENT(5)          # 5
        TRAITEMENT(2)              # 2
    SUFFIX(3):
        SUFFIX(6):
            SUFFIX(None)
            SUFFIX(9):
                SUFFIX(None)
                SUFFIX(None)
                TRAITEMENT(9)      # 9
            TRAITEMENT(6)          # 6
        SUFFIX(7):
            SUFFIX(None)
            SUFFIX(None)
            TRAITEMENT(7)          # 7
        TRAITEMENT(3)              # 3
    TRAITEMENT(1)                  # 1
```

L'ordre de parcours est donc :

```
4 8 5 2 9 6 7 3 1
```

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
```

Résolution. Le code suivant permet de générer l'arbre donné ci-dessus :

```
BinaryTree(1,
    BinaryTree(2,
        BinaryTree(4),
        BinaryTree(5,
            BinaryTree(8)
        )
    ),
    BinaryTree(3,
        BinaryTree(6,
            None,
            BinaryTree(9)
        ),
        BinaryTree(7)
    )
)
```

Notez bien que 9 est le fils *droit* de 6, et que 6 n'a pas de fils gauche. Il faut donc bien penser à donner `None` en premier paramètre au constructeur de `BinaryTree` à cet endroit.

Nous pouvons également décider de stocker les sommets dans des variables intermédiaires et utiliser ces dernières pour la création de l'arbre en entier :

```
node7 = BinaryTree(7)
node9 = BinaryTree(9)
node6 = BinaryTree(6, None, node9)
node3 = BinaryTree(3, node6, node7)
node8 = BinaryTree(8)
```

```
node5 = BinaryTree(5, node8)
node4 = BinaryTree(4)
node2 = BinaryTree(2, node4, node5)
node1 = BinaryTree(1, node2, node3)
```

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.

Résolution. Puisqu'une série d'appels récursifs crée virtuellement un stack des appels en mémoire, afin de dérécursifier la fonction de parcours préfixé, il nous suffit de maintenir un stack (que nous allons représenter par une liste par simplicité) sur lequel nous allons progressivement `push` les fils gauche et droit des sommets traités.

```
def preorder_non_rec(tree):
    if tree is None:
        return
    stack = [tree]
    while len(stack) > 0:
        node = stack.pop()
        print(node.root_value)
        if node.right_child is not None:
            stack.append(node.right_child)
        if node.left_child is not None:
            stack.append(node.left_child)
```

Ici, chaque `pop` sur le stack correspond à l'exécution d'un des appels récursifs.

Remarque. *Puisqu'un stack est un conteneur LIFO (*last in, first out*), si on veut que le fils gauche d'un nœud soit traité avant son fils droit, il faut d'abord* `push` *le fils droit et puis seulement* `push` *le fils gauche.*

Proposition 2.21. *Cette fonction nécessite $\Theta(n)$ opérations pour parcourir un arbre binaire à n sommets et nécessite un espace $\mathcal{O}(n)$.*

Démonstration. Chacun des sommets n'est traité qu'une unique fois et est placé sur le stack une unique fois. Le traitement de chacun de ces sommets est fait en temps constant ; le nombre d'opérations est donc $\Theta(n)$.

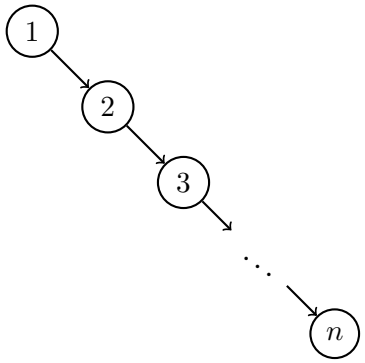
À chaque étape, le stack perd un élément et s'en voit ajouter maximum deux. Dès lors, si T_k désigne la taille du stack après le traitement du $k$ème sommet, alors nous savons que $T_{k+1} \leq T_k + 1$. En particulier :

$$T_n \leq 1 + T_{n-1} \leq 2 + T_{n-2} \leq \ldots \leq k + T_{n-k}.$$

Pour $k = n - 1$, nous avons donc $T_n \leq n - 1 + T_1 = n$, i.e. T_n est borné par n et donc $T_n = \mathcal{O}(n)$. □

Remarque. *Le nombre d'opérations a pu être exprimé à l'aide d'un grand Θ car nous savons que tous les sommets sont visités. Cependant l'espace nécessaire n'a pu être borné*

qu'en grand O. Nous pouvons aisément trouver des exemples d'arbres qui ne nécessiteront qu'un espace $\mathcal{O}(1)$ ou $\Theta(\log n)$. En effet, considérons l'arbre suivant :

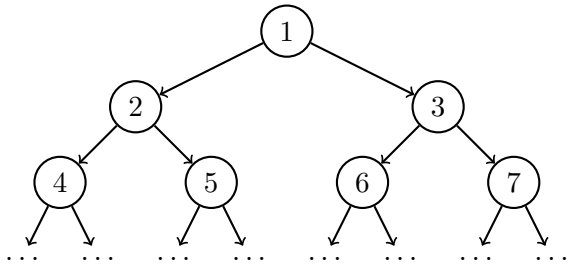


Lorsqu'un sommet est sorti du stack, ses enfants y sont ajoutés. Or chacun des sommets ici n'a qu'un seul enfant (son fils droit). La taille du stack T_k respecte donc les égalités suivantes :

$$T_k = \begin{cases} 1 & \text{si } k = 1, \\ T_{k-1} + 1 - 1 = T_{k-1} & \text{si } 1 < k < n, \\ T_{k-1} - 1 & \text{si } k = n \end{cases}$$

Dès lors $T_k = 1$ pour tout $k < n$ et $T_n = 0$. L'espace nécessaire est donc $\mathcal{O}(1)$.

Considérons maintenant l'arbre complet *suivant :*



Tous les sommets (excepté les feuilles) ont deux enfants (le fils gauche et le fils droit). Remarquons maintenant que lors du traitement d'un sommet quelconque v, le stack contient tous les fils droits des sommets qui ont mené jusque v (si ces derniers ne font pas partie du chemin entre la racine et v). Or puisque l'arbre est complet, tout chemin menant de la racine à un sommet v ne peut contenir qu'au plus $\lceil \log_2 n \rceil$ sommets, et donc le stack contient au plus $\sim \log_2 n$ éléments à chaque instant.

Sur base de cette remarque, nous pouvons améliorer notre analyse des besoins en espace de notre approche :

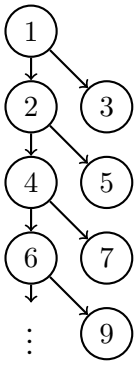
Proposition 2.22. *La fonction* `preorder_non_rec` *nécessite un espace $\mathcal{O}(p)$ pour parcourir un arbre T de profondeur p.*

Démonstration. Comme nous l'avons remarqué ci-dessus, lors du traitement d'un sommet v, le stack ne contient que les fils droits des sommets qui joignent la racine à v, si ces derniers ne sont pas dans le chemin. Le stack ne peut alors contenir plus de p éléments. □

Remarque. *Cette formulation plus précise nous permet alors d'obtenir le fait que l'espace nécessaire est $\Theta(\log n)$ pour un arbre binaire complet puisque la profondeur d'un tel arbre*

est $\log_2 n$.

Remarque. *Malheureusement, nous ne pouvons pas aller plus loin sur l'analyse des besoin en espace. En effet, voici un exemple d'arbre nécessitant* $\Theta(n)$ *en espace pour son parcours :*



Un tel arbre contient $n = 2k+1$ *sommets où pour tout* $j \in [\![1, k-1]\!]$*, le sommet* $2j$ *a pour fils gauche le sommet* $2j+2$ *et pour fils droit le sommet* $2j+3$*, et la racine* 1 *a pour fils gauche et droit les sommets* 2 *et* 3*. La profondeur d'un tel arbre est* $p = k+1 = \frac{n}{2}+1 = \Theta(n)$*. Et comme de plus tout les sommets (excepté les feuilles) ont un fils droit, lorsque le descendant le plus à gauche de la racine est traité, le stack contient bien* $\Theta(p) = \Theta(n)$ *éléments.*

Nous pouvons cependant faire mieux que l'implémentation précédente puisqu'on sait que lorsqu'un sommet quelconque v vient d'être traité, la suite de sommets suivants est donnée par la suite des descendants gauches. Nous pouvons alors introduire une boucle nous permettant d'aller chercher le dernier de ces descendants à chaque étape :

```
1  def preorder_non_rec_opti(tree):
2      if tree is None:
3          return
4      stack = [tree]
5      while len(stack) > 0:
6          node = stack.pop()
7          while node is not None:
8              print(node.root_value)
9              if node.right_child is not None:
10                 stack.append(node.right_child)
11             node = node.left_child
```

Ainsi, puisque le fils gauche de tout sommet est directement traité, il n'est plus nécessaire de les mettre sur le stack ; et le stack contient donc uniquement les fils droits dans l'ordre dans lequel ils doivent être traités.

Proposition 2.23. *La complexité en temps et en espace de* `preorder_non_rec_opti` *est identique à celle de* `preorder_non_rec`.

Démonstration. Bien que le *nombre d'instructions exécutées* n'est pas identique (seuls les fils droits sont ajoutés sur le stack dans cette implémentation-ci alors que les fils gauches le sont également dans `preorder_non_rec`), les sommets sont toujours tous traités une unique fois, et le stack contient toujours au plus $\mathcal{O}(p)$ sommets (les fils droits des sommets

joignant la racine au sommet actuel). □

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.

Résolution. À nouveau, de manière naïve, nous pourrions utiliser un stack simulant directement les appels récursifs :

```
1  def inorder_non_rec_naive(tree):
2      if tree is None:
3          return
4      stack = [(0, tree)]
5      while len(stack) > 0:
6          i, node = stack.pop()
7          if i == 0: # pas traitement
8              if node.right_child is not None:
9                  stack.append((0, node.right_child))
10             stack.append((1, node))
11             if node.left_child is not None:
12                 stack.append((0, node.left_child))
13         else:
14             print(node.root_value) # traitement
```

Nous n'ajoutons pas uniquement le sommet sur le stack mais nous ajoutons une paire `(i, node)` où `node` est le sommet et `i` est un entier qui vaut 0 si le `pop` correspond à un appel récursif et qui vaut 1 si le `pop` correspond au moment de traiter le sommet `node` (donc si les descendants gauches ont déjà été traités).

Proposition 2.24. *La fonction* `inorder_non_rec_naive` *effectue un parcours infixé d'un arbre à n sommets en $\Theta(n)$ opérations et nécessite un espace $\Theta(p)$.*

Démonstration. À nouveau, puisque tous les sommets sont visités une unique fois et puisque le traitement de chaque sommet est en $\mathcal{O}(1)$, le nombre d'opérations est $\Theta(n)$.

La proposition 2.22 est également valide pour la fonction `inorder_non_rec_naive` : le stack a une taille maximale qui est $\mathcal{O}(p)$ où p est la profondeur de l'arbre `tree`. Il y a cependant une différence ici : dans la fonction `preorder_non_rec`, un élément est ajouté au stack seulement si le sommet traité a un fils droit, alors que dans `inorder_non_rec_naive`, le sommet traité est toujours ajouté sur le stack (sous la forme `(1, node)`) et donc lors du traitement d'un sommet v, le stack a une taille comprise entre $\sim p(v)$ et $\sim 2p(v)$ (pour $p(v)$ la profondeur du sommet v). En particulier, lors du traitement des feuilles, le stack est de taille $\Theta(p)$. □

Mais nous pouvons bien entendu faire mieux. En effet, en nous basant sur l'amélioration de l'exercice précédent et en utilisant le fait que dans un parcours infixé, lorsqu'un sommet quelconque est traité, le sommet suivant à être traité est son descendant gauche le plus lointain, nous pouvons à présent modifier le parcours préfixé en allant chercher le descendant gauche et en ajoutant sur le stack tous les sommets intermédiaires :

```python
def inorder_non_rec(tree):
    if tree is None:
        return
    stack = []
    node = tree
    while len(stack) > 0 or node is not None:
        while node is not None:
            stack.append(node)
            node = node.left_child
        node = stack.pop()
        print(node.root_value)
        node = node.right_child
```

Proposition 2.25. *La fonction* `inorder_non_rec` *effectue un parcours infixé d'un arbre à* n *sommets en* $\Theta(n)$ *opérations et nécessite un espace* $\Theta(p)$.

Démonstration. Le fonctionnement est similaire à la fonction `inorder_non_rec_naive` à la différence que le stack contient $\sim p(v)$ éléments lors du traitement d'un sommet v quelconque au lieu d'une taille entre $p(v)$ et $2p(v)$. □

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
```

Résolution. Le premier sommet à être traité lors d'un parcours préfixé est la racine de l'arbre. Donc la fonction `first` est triviale :

```
def first(tree):
    return tree
```

La fonction `next` quant à elle demande un peu plus de réflexion et nous demande de considérer trois cas différents :

— soit `node` a un fils gauche ;

— soit `node` n'a pas de fils gauche mais a un fils droit ;

— soit `node` n'a pas de fils.

Dans le premier cas, le sommet suivant est tout simplement le fils gauche, et dans le second cas, le sommet suivant est le fils droit. Supposons maintenant que `node` n'a pas de fils. Ce n'est pas pour autant que le parcours est terminé (e.g. voir le sommet 8 dans l'exercice 7.1) ! En effet il est possible que le nœud actuel fasse partie d'une branche gauche d'un sommet et que le sommet suivant soit dans la branche droite de ce même sommet. Il faut alors *remonter* l'arbre à l'aide de la propriété `father` afin de trouver à un moment un fils droit qui n'a pas encore été visité.

```
def next(node):
    if node.left_child is not None:
        return node.left_child
    if node.right_child is not None:
        return node.right_child
    found = False
    while not found:
        child = node
        node = node.father
        if node is None: # on a atteint la racine -> fin du parcours
            found = True
        elif child is node.left_child and node.right_child is not None:
            node = node.right_child
```

```
14              found = True
15      return node
```

Notez bien que la seule possibilité pour que le parcours soit fini après le traitement de `node` est celle où `node` est le dernier nœud, i.e. remonter l'arbre depuis `node` ne nous fera remonter que des branches droites, ou des branches gauches menant à des nœuds n'ayant pas de fils droit.

Proposition 2.26. *La fonction* `next` *trouve le sommet suivant en* $\mathcal{O}(p)$ *opérations.*

Démonstration. Si `node` a au moins un fils, alors il est clair que la fonction s'exécute en $\mathcal{O}(1)$. Si par contre `node` n'a aucun fils, alors on remonte de père en père. Comme on ne peut pas remonter plus loin que la racine, le nombre de tours de boucle est donc maximum p (et chaque tour s'exécute en $\mathcal{O}(1)$). □

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

Résolution. Partons de l'observation suivante : l'élément `value` est dans l'arbre binaire `tree` si et seulement si au moins une des trois propriétés suivantes est satisfaite :

- `tree.root_val == value` ;
- `tree` a un fils gauche et `value` est dans le sous-arbre engendré par le fils gauche ;
- `tree` a un fils droit et `value` est dans le sous-arbre engendré par le fils droit.

Nous avons ici la formulation récursive du problème qui nous permet alors d'écrire :

```
1  def contains(tree, value):
2      if tree is None:
3          return False
4      if value == tree.root_value:
5          return True
6      return contains(tree.left_child, value) or contains(tree.right_child,
           value)
```

Remarque. *Puisque nous vérifions que le paramètre* `tree` *n'est pas* `None`*, il n'est pas nécessaire de s'assurer de l'existence des fils gauche et droit avant les appels récursifs à* `contains`*.*

Proposition 2.27. *La fonction* `contains` *effectue au plus* n *comparaisons si* `tree` *est un arbre binaire à* n *sommets.*

Démonstration. Pour tout arbre T à n éléments, notons C_n le nombre de comparaisons maximum effectuées lors d'un appel à `contains`. Il est clair que $C_0 = 0$ puisque si T ne contient aucun sommet, c'est que `tree` vaut `None` et donc la ligne 4 n'est jamais atteinte. Notons maintenant T_1 et T_2 les sous-arbres gauche et droit de T, et notons n_1 et n_2 le nombre de sommets de T_1 et T_2. Nous avons donc l'égalité suivante : $C_n = 1 + C_{n_1} + C_{n_2}$. En effet, le nombre total de comparaisons est au plus celle de la ligne 4 (donc 1) plus celles lors de l'appel récursif sur le fils gauche (donc C_{n_1}) et plus celles lors de l'appel récursif sur le fils droit (donc C_{n_2}).

Montrons que $C_n = n$ satisfait cette relation de récurrence. Il est clair que la relation est vérifiée pour $n = 0$. Maintenant supposons que la relation soit vérifiée pour tous $k \lneqq n$ et montrons qu'elle est vraie pour n. Par hypothèse de récurrence (puisque $n_1 < n$ et $n_2 < n$) :

$$C_{n_1} + C_{n_2} + 1 = n_1 + n_2 + 1 = n.$$

□

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

Résolution.

Lemme 2.28. *Deux arbres binaires T_1 et T_2 sont égaux si et seulement si (i) les racines sont égales et (ii) les sous-arbres gauche et droit de T_1 et T_2 sont égaux.*

À nouveau, sur base de cette formulation récursive, il est aisé d'écrire le code permettant de vérifier l'égalité de deux arbres binaires :

```
1 def egalite(tree1, tree2):
2     if tree1 is None and tree2 is None:
3         return True
4     elif tree1 is None or tree2 is None:
5         return False
6     if tree1.root_value != tree2.root_value:
7         return False
8     return egalite(tree1.left_child, tree2.left_child) \
9        and egalite(tree1.right_child, tree2.right_child)
```

Proposition 2.29. *La fonction* `egalite` *détermine si deux arbres T_1 et T_2 (de taille respective n_1 et n_2) sont égaux en $\mathcal{O}(\min\{n_1, n_2\})$ comparaisons.*

Démonstration. Si T_1 et T_2 sont égaux, alors en particulier $n_1 = n_2$. De plus, les paires de sommets correspondants sont comparées une unique fois. La fonction détermine donc leur égalité en $n_1 = n_2 = \mathcal{O}(\min\{n_1, n_2\})$ comparaisons.

Si maintenant T_1 et T_2 ne sont pas égaux, alors il faut au plus $\min\{n_1, n_2\}$ comparaisons avant que l'un des deux sommets comparés ne soit `None`, ce qui amène la fonction à retourner `False`. Notons qu'il est possible que la fonction renvoie `False` en moins de $\min\{n_1, n_2\}$ comparaisons si une des comparaisons sur les valeurs des sommets (l. 6) n'est pas vérifiée. □

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.

Résolution. Afin qu'un arbre T' soit l'image miroir d'un arbre binaire T, il faut que pour tout sommet v de T, le fils gauche de v dans T soit l'image miroir du fils droit de l'équivalent du sommet v dans T' et que le fils droit de v dans T soit l'image miroir du fils gauche de l'équivalent de v dans T'.

L'approche récursive est donc la suivante : pour un sommet v de T, notons v_L et v_R les fils gauche et droit de v. Prenons l'image miroir de v_L (notons-la v'_L) et prenons l'image miroir de v_R (notons-la v'_R). Créons alors un nouveau sommet v' ayant la même valeur que v mais dont le fils gauche est v'_R et dont le fils droit est v'_L :

```
1 def mirror(tree):
2     if tree is None:
3         return None
4     reversed_left = mirror(tree.left_child)
5     reversed_right = mirror(tree.right_child)
6     return BinaryTree(tree.root_value, left=reversed_right,
          right=reversed_left)
```

Proposition 2.30. *La fonction* `mirror` *crée l'image miroir d'un arbre à n sommets en $\Theta(n)$ opérations.*

Démonstration. Il est clair que tous les sommets doivent être visités au moins une fois pour pouvoir créer l'image miroir d'un arbre. Le nombre d'opérations est donc $\Omega(n)$.

De manière similaire à la proposition 2.27, notons C_n le nombre d'opérations effectuées par la fonction `mirror` sur un arbre T à n sommets. La relation suivante est donc vérifiée si n_1 et n_2 désignent le nombre de sommets des fils gauche et droit de T :

$$C_n = \begin{cases} \mathcal{O}(1) & \text{si } n = 0, \\ 1 + C_{n_1} + C_{n_2} & \text{sinon.} \end{cases}$$

Comme montré dans la proposition 2.27, $C_n = \mathcal{O}(n)$ est une solution de cette relation.

Puisque $C_n = \Omega(n)$ et $C_n = \mathcal{O}(n)$, par définition nous avons que $C_n = \Theta(n)$. □

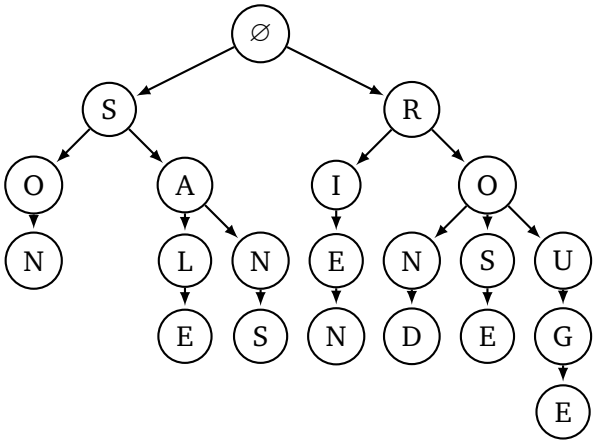
Exercices supplémentaires

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.



Résolution. Puisqu'un *trie* est un arbre d-aire, écrivons une classe `TrieNode` pouvant stocker un nombre arbitraire de nœuds fils ainsi qu'un caractère :

```
1  class TrieNode:
2      def __init__(self, char, *children):
3          self.char = char
4          self.children = list(children)
5
6      def add_child(self, child):
7          self.children.append(child)
8
9      def remove_child(self, child):
10         for i in range(len(self.children)):
11             if self.children[i] is child:
12                 idx = i
13                 break
14         self.children.pop(idx)
15
```

```
16      def __iter__(self):
17          return iter(self.children)
18
19      def is_leaf(self):
20          return len(self.children) == 0
```

Remarque. *Pour rappel, la notation :*

```
def f(*args, **kwargs):
   # ...
```

permet de récupérer tous les paramètres donnés à la fonction `f`*, sans en préciser le nombre ou la nature. Pour cela, tous les paramètres non-*keyword *(qui ne sont pas passés sous la forme* `nom=valeur`*) sont stockés dans un tuple nommé* `args` *et tous les paramètres* keyword *sont stockés dans un dictionnaire dont les clefs sont les noms et les valeurs sont les valeurs associées. Par exemple :*

```
def f(*args, **kwargs):
   print(args)
   for k, v in kwargs.items():
      print(f'{k} -> {v}')
f(1, 2, 3, 'param4', size=4, colour='red', ignore=False)
```

affichera :

```
(1, 2, 3, 'param4')
size -> 4
colour -> red
ignore -> False
```

La méthode `remove_child` utilise l'opérateur `is` afin de déterminer, pour chaque `i`, si le `i`ème fils `self.children[i]` et le paramètre `node` sont bien le même objet (et pas s'ils sont *juste* égaux). La méthode spéciale `__iter__` permet d'utiliser la notation `for child in node` sans devoir faire un *getter* ou une propriété pour l'attribut `children`. L'attribut `char` est atteint (en lecture) depuis l'extérieur de la classe. Bien que ça ne soit pas très propre au niveau de l'encapsulation, ce choix a été fait ici pour la lisibilité.

La classe `Trie`, quant à elle, représente un arbre d-aire avec pour unique attribut une référence vers la racine de l'arbre (qui est donc un nœud vide) :

```
1  class Trie:
2      def __init__(self):
3          self.root = TrieNode('')
```

La méthode d'insertion peut s'écrire ainsi :

```
1      def insert(self, key):
2          node = self.root
3          i = 0
4          found = True
```

```
        while i < len(key) and found:
            for child in node:
                if key[i:].startswith(child.char):
                    node = child
                    i += len(node.char)
                    break
            else:
                found = False
        # On a trouvé le plus grand préfixe possible
        # Maintenant ajoutons le reste, caractère par caractère
        new_node = TrieNode(key[-1])
        j = len(key)-2
        while j >= i:
            new_node = TrieNode(key[j], new_node)
            j -= 1
        node.add_child(new_node)
```

La première partie de cette insertion est un parcours en profondeur de l'arbre jusqu'à trouver le plus grand préfixe possible de `key` dans l'arbre ; et la seconde partie est l'insertion de tous les caractères qui viennent après ce préfixe sus forme d'une chaîne.

La méthode `contains` est un peu plus courte car il ne faut que la première partie (le parcours en profondeur de l'arbre) :

```
    def contains(self, key):
        node = self.root
        i = 0
        while i < len(key) and not node.is_leaf():
            for child in node:
                if key[i:].startswith(child.char):
                    node = child
                    i += len(node.char)
                    break
            else:
                break
        return node.is_leaf() and i == len(key)
```

Il faut cependant faire bien attention à ne renvoyer `True` que si le dernier sommet trouvé (correspondant au dernier caractère de `key`) est bien une feuille (on considère que `'ROU'` n'est pas une clef stockée dans le trie donné en exemple).

La suppression est implémentée ici de manière récursive : afin de supprimer la clef `key` du trie, il faut identifier le nœud correspondant au caractère `key[0]`, supprimer la clef `key[1:]` du trie engendré par ce nœud et supprimer ce nœud uniquement si il est devenu une feuille après suppression de `key[1:]` :

```
    def __delete_rec(self, node, key):
        ''' Retourne True si on revient d'une feuille '''
        if len(key) == 0:
            return True
```

```
        for child in node:
            if key.startswith(child.char):
                # si on revient d'une feuille, on supprime cette feuille
                if self.__delete_rec(child, key[len(child.char):]):
                    node.remove_child(child)
                return node.is_leaf()

    def delete(self, key):
        self.__delete_rec(self.root, key)
```

Il doit apparaître de manière assez claire que ces trois méthodes s'exécutent bien en $\Theta(\ell)$, pour ℓ la longueur de `key`, dans le pire des cas. En effet, le trie est parcouru (en profondeur) caractère par caractère jusqu'à ce que la clef soit soit trouvée, soit insérée, soit supprimée. Dans tous les cas, la profondeur maximale visitée est ℓ et le traitement de chaque niveau se fait en $\Theta(1)$.

> **Remarque.** *Notre implémentation a un problème : si chaque clef contenue dans le trie doit partir de la racine et terminer sur une feuille, comment gère-t-on les préfixes de clefs qui sont eux-mêmes des clefs ? Par exemple comment pouvons-nous stocker les clefs* `ALGO` *et* `ALGORITHME` *dans notre trie ?*
>
> *En effet l'implémentation ci-dessus ne permet pas une telle chose puisque l'insertion de la clef* `ALGO` *va créer une chaîne de 4 nœuds en partant de la racine, et puis l'insertion de la clef* `ALGORITHME` *va suivre cette chaîne et puis insérer le suffixe* `RITHME` *en partant de là. Ainsi, le* `O` *n'est plus une feuille et donc* `ALGO` *n'est plus reconnu comme étant une clef du trie.*
>
> *Si par contre nous ajoutons un nœud vide (comme la racine) à la fin de chaque clef, nous aurons toujours une feuille correspondant à chaque clef. Il est également possible d'ajouter un* flag *à chaque nœud déterminant s'il est la fin d'un préfixe ou non (les clefs stockées dans le trie ne correspondent donc plus aux chemins menant de la racine vers une feuille mais bien aux chemins menant de la racine vers un sommet dont le* flag *est à* `True`*).*

Implémentons cette deuxième approche. La classe `TrieNode` doit donc se voir ajouter un flag booléen :

```
class TrieNode:
    def __init__(self, char, is_prefix, *children):
        self.char = char
        self.flag = is_prefix
        self.children = list(children)

    def add_child(self, child):
        self.children.append(child)

    def remove_child(self, child):
        for i in range(len(self.children)):
            if self.children[i] is child:
                idx = i
                break
        self.children.pop(idx)
```

```
16
17      def __iter__(self):
18          return iter(self.children)
19
20      def is_leaf(self):
21          return len(self.children) == 0
22
23      def is_prefix(self):
24          return self.flag
```

La classe `Trie`, quant à elle, doit voir une légère modification à sa méthode d'insertion (les lignes 15 et 18 ci-dessous doivent définir l'état du flag qui ne vaut `True` que pour le dernier caractère) :

```
1       def insert(self, key):
2           node = self.root
3           i = 0
4           found = True
5           while i < len(key) and found:
6               for child in node:
7                   if key[i:].startswith(child.char):
8                       node = child
9                       i += len(node.char)
10                      break
11              else:
12                  found = False
13          # On a trouvé le plus grand préfixe possible
14          # Maintenant ajoutons le reste, caractère par caractère
15          new_node = TrieNode(key[-1], True)
16          j = len(key)-2
17          while j >= i:
18              new_node = TrieNode(key[j], False, new_node)
19              j -= 1
20          node.add_child(new_node)
```

et à sa méthode `contains` qui doit maintenant vérifier que lors de la fin d'une recherche, le sommet sur lequel on arrive est bien un préfixe valide :

```
1       def contains(self, key):
2           node = self.root
3           i = 0
4           while i < len(key) and not node.is_leaf():
5               for child in node:
6                   if key[i:].startswith(child.char):
7                       node = child
8                       i += len(node.char)
9                       break
10              else:
11                  break
```

```
12          return node.is_prefix() and i == len(key)
```

Bien entendu, la complexité, tant en temps qu'en espace, des trois opérations implémentées n'a pas été modifiée (elles sont toujours linéaires en la taille de la clef manipulée).

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.

Résolution.

Définition 13. Un arbre binaire T est une arbre binaire de recherche (BST) si pour tout sommet v de T, la valeur maximale du sous-arbre gauche de v est strictement inférieure à la valeur de v ; et la valeur minimale du sous-arbre droit de v est supérieure ou égale à la valeur de v.

Un BST T est dit *dans* un ensemble $X \subseteq \mathbb{R}$ si pour tout sommet v de T, on a $v \in X$.

Remarque. *Pour $X \subseteq Y \subseteq \mathbb{R}$, si T est un arbre binaire dans X, alors T est un arbre binaire dans Y.*

Un BST T est toujours un BST dans $(-\infty, +\infty)$.

Lemme 2.31. *Soit T, un arbre binaire de racine v et soient $a < b \in \mathbb{R}$. Pour $X \subseteq \mathbb{R}$, si les conditions suivantes sont respectées, alors T est un BST dans X :*

1. *soit v n'a pas de fils gauche, soit le fils gauche de v, noté v_L engendre un BST dans $X \cap (-\infty, v)$;*
2. *soit v n'a pas de fils droit, soit le fils droit de v, noté v_R engendre un BST dans $X \cap [v, +\infty)$.*

Sur base de ce lemme, nous pouvons écrire une fonction récursive qui vérifie explicitement ces conditions :

```
1  def is_bst_rec(node, lower=-float('inf'), upper=+float('inf')):
2      if node is None:
3          return False
4      if not (lower <= node.root_value < upper):
5          return False
6      left = node.left_child
7      right = node.right_child
8      has_left = left is not None
9      has_right = right is not None
10     return (not has_left or is_bst_rec(left, lower, node.root_value)) \
11        and (not has_right or is_bst_rec(right, node.root_value, upper))
```

Cette solution utilise la valeur de chaque sommet ainsi que la valeur des fils gauche et droit de chaque sommet. Cela va s'avérer problématique lors de la dérécursification de l'algorithme puisque nous nous basons sur la structure d'un arbre binaire sans père.

Remarquons alors que, toujours de manière récursive, il nous suffit de faire un parcours préfixé de l'arbre tout en maintenant les variables `lower` et `upper` nous donnant respectivement une borne inférieure et supérieure sur les valeurs du sous-arbre :

```
def is_bst(node, lower=-float('inf'), upper=+float('inf')):
    if node is None:
        return True
    value = node.root_value
    if not (lower <= value < upper):
        return False
    return is_bst(node.left_child, lower, value) \
       and is_bst(node.right_child, value, upper)
```

En effet puisque pour tout sommet v, un BST doit satisfaire $v_L < v \leq v_R$, alors nous savons que si L et U sont des bornes inférieure et supérieure sur *tous* les sommets de T, alors $v_L \in [L, v)$ et $v_R \in [v, U]$. En initialisant L à $-\infty$ et U à $+\infty$, alors ces conditions sont suffisantes pour déterminer si T est un BST.

Pour la version non-récursive, il y a une subtilité : nous voulons garder un minimum de variables en mémoire pour des raisons d'efficacité. Dès lors, même s'il est possible de stocker (par exemple dans deux stacks dédiés) la valeur de `lower` et `upper` pour chaque traitement de sommet (ce qui recrée donc *virtuellement* le déroulement de l'approche récursive), ce n'est pas optimal.

En effet, si nous faisons un parcours infixé (pour lequel nous pouvons récupérer la solution de l'exercice 7.4) d'un BST, alors les sommets sont traités de manière croissante. Grâce à cette propriété, remarquons que nous devons uniquement nous assurer que la borne inférieure n'est pas dépassée, ce qui est donc le traitement du sommet :

```
def is_bst_non_rec(node, lower=-float('inf')):
    if node is None:
        return True
    stack = []
    while len(stack) > 0 or node is not None:
        while node is not None:
            stack.append(node)
            node = node.left_child
        node = stack.pop()
        if node.root_value < lower:
            return False
        lower = node.root_value
        node = node.right_child
    return True

if __name__ == "__main__":
```

Proposition 2.32. *Ces trois fonctions déterminent si T est un BST en $\sim n$ comparaisons dans le pire des cas.*

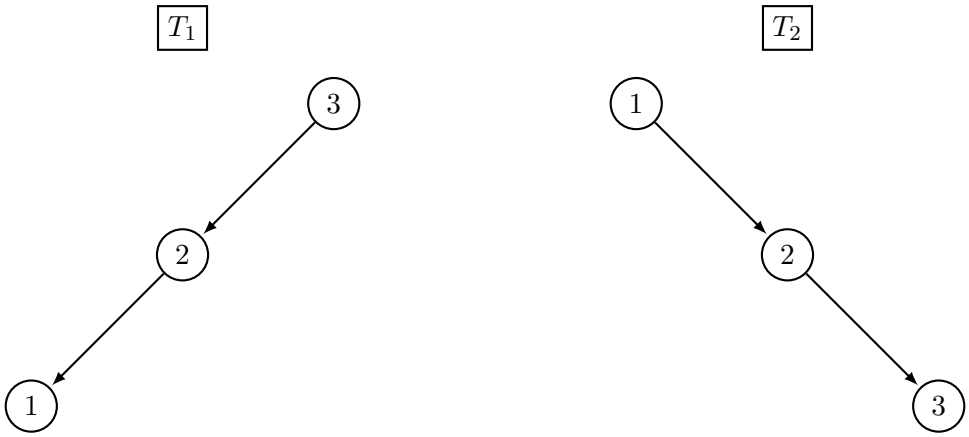
Démonstration. Nous avons déjà vu qu'un parcours d'arbre visite chaque sommet une

unique fois. Dans le pire des cas, tous les sommets sont visités et sont comparés avec leur sommet père (sauf pour la racine), donnant donc $n - 1 \sim n$ comparaisons. □

Bonus : pouvez-vous adapter cette solution en n'utilisant que la borne supérieure (et non la borne inférieure) ?

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.

Résolution. Dans l'exercice 8.3, nous avions déjà écrit une fonction récursive qui déterminait si deux arbres binaires étaient égaux. Les BST sont en particulier des arbres binaires, donc cette approche fonctionnerait ici également pour tester l'égalité. Cependant, nous cherchons à établir l'*équivalence* entre deux BSTs et non leur égalité ! Les deux BSTs suivants ne sont pas égaux mais sont pourtant équivalents (ils contiennent bien les mêmes valeurs) :



Lemme 2.33. *Deux BSTs T_1 et T_2 sont équivalents si et seulement si leur parcours infixé donne la même séquence de sommets.*

Démonstration. Puisqu'un parcours infixé d'un BST traite les sommets dans l'ordre croissant, si les parcours donnent la même séquence, c'est obligatoirement que les sommets sont les mêmes. □

Sur base du lemme précédent, la solution peut donc s'écrire comme suit :

```
1  def equivalent(node1, node2):
2      stack1 = []
3      stack2 = []
4
5      while (node1 is not None or len(stack1) > 0) \
6        and (node2 is not None or len(stack2) > 0):
7          while node1 is not None:
8              stack1.append(node1)
9              node1 = node1.left_child
10         while node2 is not None:
11             stack2.append(node2)
12             node2 = node2.left_child
```

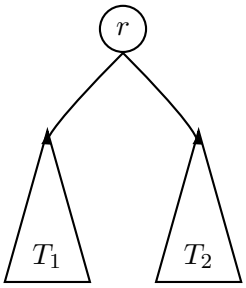
```
13          node1 = stack1.pop()
14          node2 = stack2.pop()
15          if node1.root_value != node2.root_value:
16              return False
17          node1 = node1.right_child
18          node2 = node2.right_child
19      return node1 is None \
20         and node2 is None \
21         and len(stack1) == len(stack2) == 0
```

Proposition 2.34. *La fonction* `equivalent` *détermine si deux arbres T_1 et T_2 de taille respective n_1 et n_2 sont équivalents en $\mathcal{O}(\min\{n_1, n_2\})$ comparaisons. Si T_1 et T_2 sont équivalents, alors la fonction effectue $n_1 = n_2$ comparaisons.*

Démonstration. L'argument est similaire à celui de la proposition 2.29 : si T_1 et T_2 ne sont pas équivalents, alors il faudra au plus $\sim \min\{n_1, n_2\}$ comparaisons avant que l'un des deux sommets testés soit `None`. Si par contre T_1 et T_2 sont équivalents, alors comme dans la proposition 2.32, tous les sommets sont visités et comparés deux à deux, donnant donc $n_1 = n_2$ comparaisons. □

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.

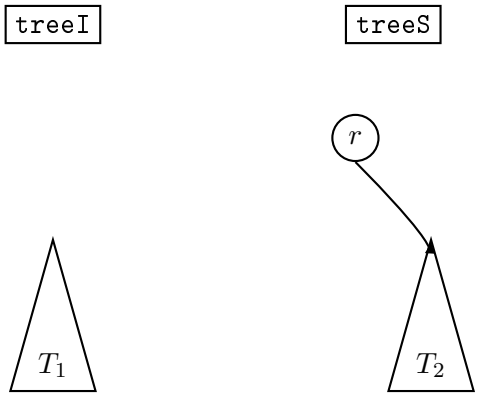
Résolution. Un BST T a systématiquement la forme suivante :



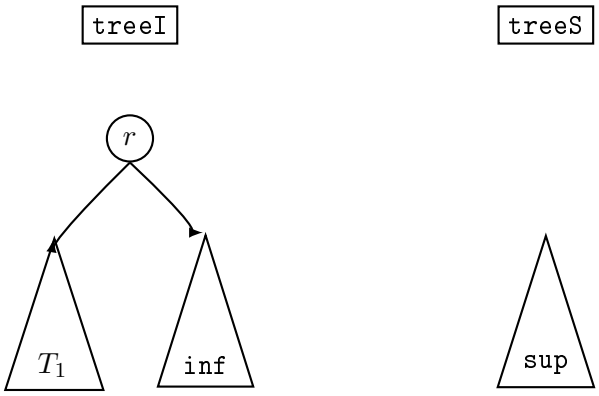
où r est la racine de T, T_1 est un sous-arbre (potentiellement vide) dont tous les éléments sont strictement inférieurs à r, et T_2 est un sous-arbre (potentiellement vide) dont tous les éléments sont supérieurs ou égaux à r.

Il nous faut alors distinguer les trois cas suivants :

Cas 1 : $r = x$. Ici, la scission est assez simple car elle se fait simplement sur la racine, donnant la solution suivante :

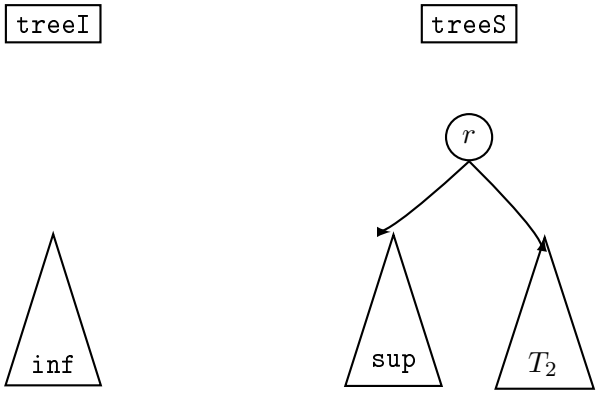


Cas 2 : $r < x$**.** Ici, la scission ne peut probablement se faire sur la racine car il peut y avoir un élément de T_2 dans $(r, x]$. Il faut alors appliquer la séparation récursivement sur T_2 de manière à obtenir les deux sous-arbres `inf` et `sup` de T_2 afin d'avoir la solution suivante :



En effet, puisque `inf` est un sous-arbre de T_2, c'est en particulier un BST, et tous ses éléments sont $\geq r$. `treeI` est donc bien un BST.

Cas 3 : $r > x$**.** Ici, nous avons le cas réciproque du cas 2, dans lequel c'est T_1 qui doit être séparé récursivement en `inf` et `sup` afin de pouvoir exprimer la solution :



Par le même argument que pour le cas précédent, puisque `inf` et `sup` sont des sous-arbres de T_1, `sup` a bien une structure de BST et dont les éléments sont $\leq r$, ce qui assure que `treeS` est bien un BST.

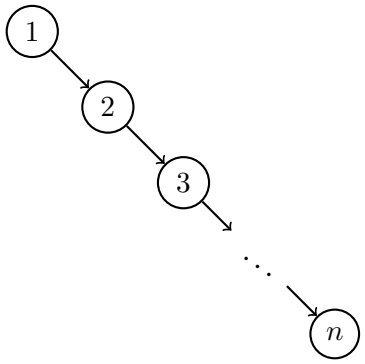
Sur base de ces trois cas, nous pouvons alors écrire le code suivant :

```python
def scission(tree, x, treeI, treeS):
    if tree is None:
        treeI = BinaryTree(None)
        treeS = BinaryTree(None)
    elif tree.root_value == x:
        treeI = tree.left_child
        treeS = tree
        treeS.left_child = None
    elif tree.root_value < x:
        inf, sup = scission(tree.right_child, x, tree.right_child, treeS)
        treeI = tree
        treeI.right_child = inf
        treeS = sup
    elif tree.root_value > x:
        inf, sup = scission(tree.left_child, x, treeI, tree.left_child)
        treeI = inf
        treeS = tree
        treeS.left_child = sup
    return treeI, treeS
```

Proposition 2.35. *Soit T un BST à n éléments et de profondeur p. Si la valeur* `x` *est dans T, alors la fonction* `scission` *sépare T en deux sous-arbres respectivement inférieur et supérieur à* `x` *en $p(\texttt{x})$ appels récursifs. Si la valeur* `x` *n'est pas dans T, alors la fonction* `scission` *exécute $\sim p$ appels récursifs dans le pire des cas.*

Démonstration. La fonction `scission` effectue une recherche dans un BST afin de trouver `x`. En particulier une telle recherche ne fait que *descendre* dans T jusqu'à ce que `x` soit trouvé s'il existe dans T et jusqu'à arriver à une feuille sinon. Dans le premier cas, cela correspond à $p(\texttt{x})$ appels récursifs (on descend un sommet à la fois jusqu'à trouver `x`), et dans le second cas, cela correspond bien à p appels récursifs tout au plus. □

Remarque. *Dans le cas où* `x` *n'apparait pas dans l'arbre T, il est possible que* `scission` *nécessite beaucoup moins que p appels récursifs. Prenons par exemple le BST suivant :*



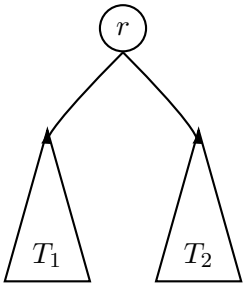
Il est clair que si `x` *vaut 1, alors il ne faut qu'un seul appel récursif afin de se rendre compte que l'arbre correspond déjà à* `treeS` *et que* `treeI` *est vide. On ne peut par contre pas faire mieux que $\sim p$ dans le pire des cas puisque si* `x` *est $> n$, alors il faut parcourir tous les fils*

droits afin de se rendre compte que l'arbre correspond à `treeI` *et que* `treeS` *est vide, i.e.* $\sim p = n$ *appels.*

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.

Résolution. En raisonnant de manière similaire à l'exercice précédent, il va falloir distinguer les trois mêmes cas et appeler la fonction d'enracinement de manière récursive tout recréant un BST en jouant sur les sous-arbres gauche et droit.

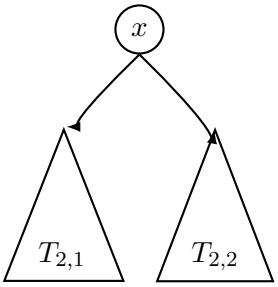
Puisqu'un BST T a la forme suivante :



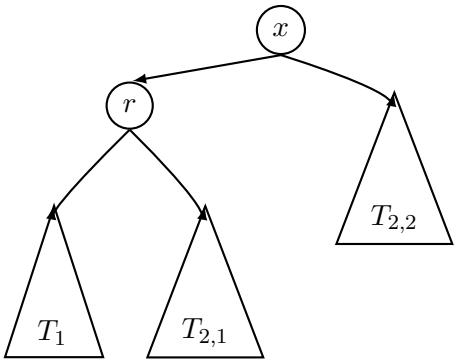
nous distinguons les cas :

Cas 1 : $r = x$**.** Il n'y a rien à faire dans ce cas : x est déjà la racine.

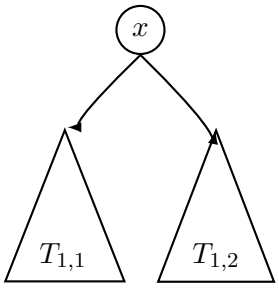
Cas 2 : $r < x$**.** Ici, il faut appliquer récursivement la procédure d'enracinement afin de transformer le sous-arbre droit T_2 en :



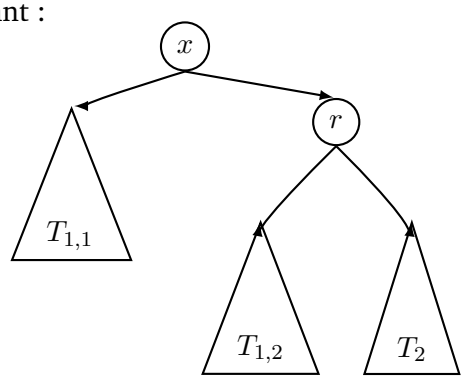
Or tous les éléments de $T_{2,1}$ et $T_{1,1}$ sont plus grands que r. Dès lors T_1 et r doivent se trouver à gauche de x. De plus, r sépare bien T_1 et $T_{2,1}$. Nous pouvons alors construire l'arbre suivant :



Cas 3 : $r > x$**.** De manière symétrique, il faut appliquer récursivement la procédure d'enracinement afin de transformer le sous-arbre gauche T_1 en :



Or tous les éléments de $T_{1,1}$ et $T_{1,2}$ sont plus petits que r. Dès lors T_2 et r doivent se trouver à droite de x. De plus, r sépare bien $T_{1,2}$ et T_2. Nous pouvons alors construire l'arbre suivant :



Ces trois cas peuvent donc être implémentés comme suit :

```
def root(tree, x):
    if tree is None:
        return BinaryTree(None)
    if x == tree.root_value:
        return tree
    elif x < tree.root_value:
        tmp = root(tree.left_child, x)
        return BinaryTree(x,
            tmp.left_child,
            BinaryTree(tree.root_value,
                tmp.right_child,
                tree.right_child
            )
        )
    else:
        tmp = root(tree.right_child, x)
        return BinaryTree(x,
            BinaryTree(tree.root_value,
                tree.left_child,
                tmp.left_child
            ),
            tmp.right_child
        )
```

Proposition 2.36. *Soit T un BST à n éléments et de profondeur p. Si la valeur* `x` *est dans T, alors la fonction* `root` *enracine T en fonction de* `x` *en $p(\mathtt{x})$ appels récursifs. Si la valeur*

`x` n'est pas dans T, alors la fonction `root` exécute $\sim p$ appels récursifs dans le pire des cas.

Démonstration. Tout comme la fonction `scission` dans la proposition 2.35, la fonction `root` effectue une recherche dans un BST. Le nombre d'appels est donc $p(\mathtt{x})$ si `x` apparait dans T et borné par p si `x` n'est pas dans T. □

Remarque. *Tant dans cette séance que dans les séances 7 et 8, certains des algorithmes présentés effectuent un nombre d'opérations borné par la* profondeur *de l'arbre. Nous savons que la profondeur d'un arbre binaire est un $\Omega(\log n)$. Plus précisément, lorsque la profondeur d'un arbre binaire T est $\sim p(T)$, on dit que T est* balancé *(ou devrait-on plutôt dire* équilibré *car* balancé *est une francisation de* balanced *en anglais qui veut bien dire* équilibré*, mais tout comme* library *qui est devenu* librairie *au lieu de* bibliothèque*, les informaticiens aiment les faux-amis).*

De manière plus générale, la plupart des opérations sur un BST T peuvent être exécutées en $\sim p(T)$ dans le pire des cas. Dès lors si nous pouvions nous assurer qu'un certain BST est toujours balancé, peu importe les insertions et suppressions que l'on peut y faire, nous nous assurerions que toutes les manipulations de T seraient en $\sim \log n$ dans le pire des cas.

*Il est possible de faire cela précisément : les arbres 2-3 (et en particulier leur implémentation sous forme d'arbres rouges-noirs) permettent de forcer un BST à rester équilibré. Pour cela, une opération (qui s'exécute en temps constant) est cruciale : la rotation. Les détails de cette structure seront vus en bloc 2 au cours d'*Algorithmique 2 *(INFO-F203).*

Séance 10 — Files à priorité

Définition 14. 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`.

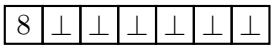
Résolution. L'insertion dans un max-heap se fait comme suit :

1. l'élément est ajouté à la première place disponible dans l'arbre ;
2. tant qu'elle est plus grande que son père, on l'échange avec son père.

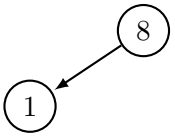
Remarque. *Grâce à cette construction, nous sommes assurés qu'un tas à n éléments a une hauteur au plus $\lfloor \log_2 n \rfloor + 1$, ce qui nous assure que les opérations sur un heap sont en $\Theta(\log n)$ dans le pire des cas.*

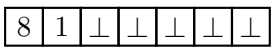
Ajoutons donc les éléments l'un après l'autre et regardons la représentation de T via un arbre et via un vecteur.

Ajout de 8 Puisque le tas est vide, 8 devient la racine du tas :

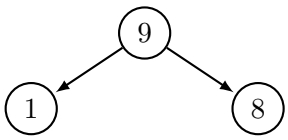


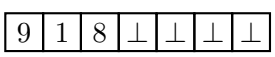
Ajout de 1 L'insertion de 1 se fait à la première place disponible, i.e. le fils gauche de la racine. 1 n'est pas plus grand que 8, il n'est donc pas nécessaire d'échanger ces deux nœuds :



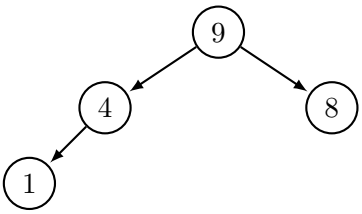


Ajout de 9 L'insertion de 9 se fait à la première place disponible, i.e. le fils droit de la racine. Comme $9 > 8$, il faut échanger ce nœud avec la racine :



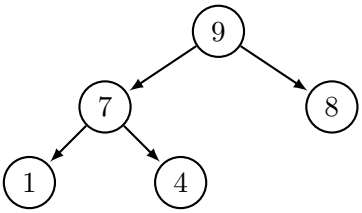


Ajout de 4 L'insertion de 4 se fait à la première place disponible, i.e. le fils gauche de 1. Comme $4 > 1$, il faut échanger ces deux nœuds. Par contre $4 < 9$, donc il ne faut pas échanger plus loin :



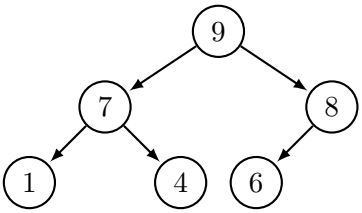
9	4	8	1	⊥	⊥	⊥

Ajout de 7 L'ajout de 7 se fait à la première place disponible, i.e. le fils droit de 4. Comme $7 > 4$, il faut échanger ces deux nœuds. Par contre $7 < 9$, donc il ne faut pas échanger plus loin :



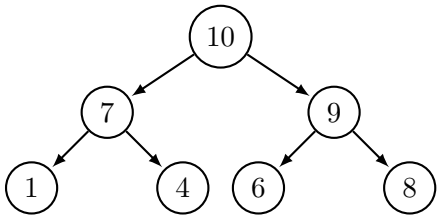
9	7	8	1	4	⊥	⊥

Ajout de 6 L'ajout de 6 se fait à la première place disponible, i.e. le fils gauche de 8. Comme $6 < 8$, il ne faut pas échanger ces deux nœuds :



9	7	8	1	4	6	⊥

Ajout de 10 Finalement, l'ajout de 10 se fait à la première place disponible, i.e. le fils droit de 8. Comme $10 > 9 > 8$, il faut échanger deux fois le nœud avec son père :



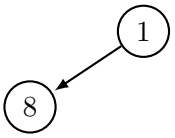
10	7	9	1	4	6	8

Les insertions dans un min-tas se font comme suit :

Ajout de 8 Puisque le tas est vide, 8 devient la racine du tas :

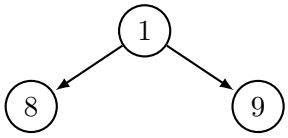
8	⊥	⊥	⊥	⊥	⊥	⊥

Ajout de 1 L'insertion de 1 se fait à la première place disponible, i.e. le fils gauche de la racine. Comme $1 < 8$, il faut échanger ce nœud avec la racine :



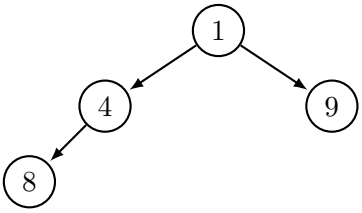
1	8	⊥	⊥	⊥	⊥	⊥

Ajout de 9 L'insertion de 9 se fait à la première place disponible, i.e. le fils droit de la racine. Comme $9 > 8$, il ne faut pas échanger ces deux nœuds :



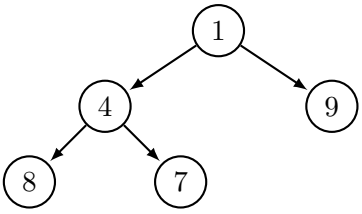
1	8	9	⊥	⊥	⊥	⊥

Ajout de 4 L'insertion de 4 se fait à la première place disponible, i.e. le fils gauche de 8. Comme $4 < 8$, il faut échanger ces deux nœuds. Par contre $4 > 1$, donc il ne faut pas échanger avec la racine :



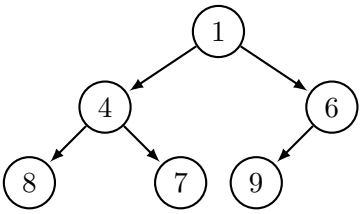
1	4	9	8	⊥	⊥	⊥

Ajout de 7 L'insertion de 7 se fait à la première place disponible, i.e. le fils droit de 4. Comme $7 > 4$, il ne faut pas échanger ces deux nœuds :



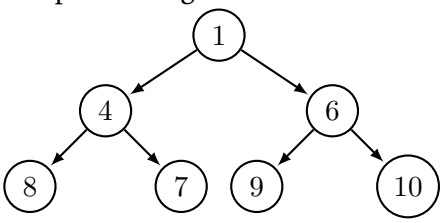
1	4	9	8	7	⊥	⊥

Ajout de 6 L'insertion de 6 se fait à la première place disponible, i.e. le fils gauche de 9. Comme $6 < 9$, il faut échanger ces deux nœuds. Par contre $6 > 1$, donc il faut pas échanger avec la racine :



1	4	6	8	7	9	⊥

Ajout de 10 L'insertion de 10 se fait à la première place disponible, i.e. le fils droit de 6. Comme $10 > 6$, il ne faut pas échanger ces deux nœuds :



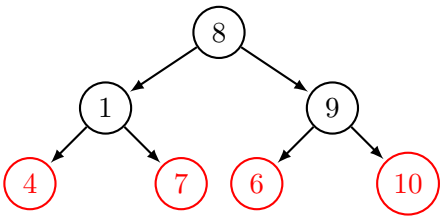
1	4	6	8	7	9	10

Appliquons maintenant l'algorithme `heapsort` sur le vecteur `v`.

> **Remarque.** *Nous pourrions partir directement du tas créé au point 1 qui a nécessité n insertions ; mais dans l'algorithme* `heapsort`*, le heap est créé* in place *qui nécessite uniquement $\sim \frac{n}{2}$ insertions. L'idée derrière ce choix est que les $n - \lfloor \frac{n}{2} \rfloor$ derniers éléments de* `v` *sont les feuilles du heap. Dès lors, faire descendre les pères itérativement permet de s'assurer que la propriété d'ordre des heaps soit vérifiée.*

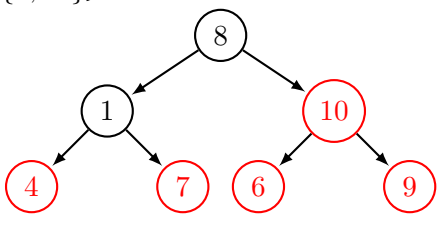
Créons donc le max-tas qui nous servira à trier `v`. Les sommets satisfaisant l'ordre de heap sont marqués en rouge.

État initial `v` n'est pas encore modifié :



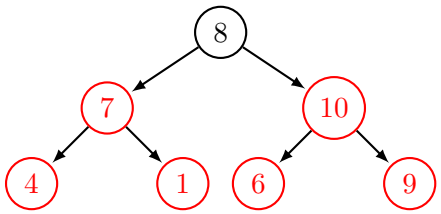
8	1	9	4	7	6	10

Descente de 9 On commence à descendre les pères à partir le l'indice $k = \lfloor \frac{n}{2} \rfloor - 1 = \lfloor \frac{7}{2} \rfloor - 1 = 3 - 1 = 2$ (pour des indices allant de 0 à $n - 1$), i.e. on descend 9 (qui est plus petit que $10 = \max\{6, 10\}$) :



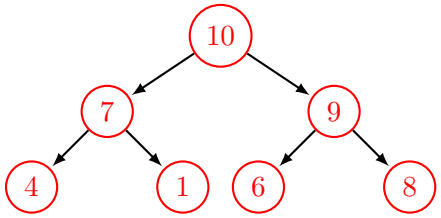
8	1	10	4	7	6	9

Descente de 1 On descend alors le père précédent à l'indice $k' = k - 1 = 2 - 1 = 1$, i.e. on descend 1 (qui est plus petit que $7 = \max\{4, 7\}$) :



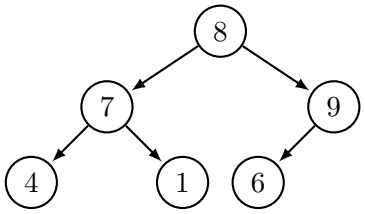
8	7	10	4	1	6	9

Descente de 8 On descend finalement le premier père (donc la racine) à l'indice $k'' = k' - 1 = 1 - 1 = 0$. Il faut ici faire deux échanges car $8 < 10 = \max\{7, 10\}$, mais après ce premier échange on a toujours $8 < 9 = \max\{6, 9\}$:

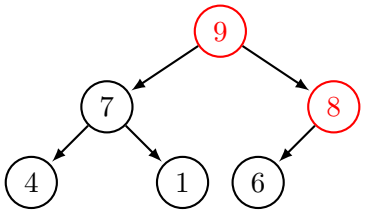


10	7	9	4	1	6	8

Extraction de 10 Puisque 10 est la racine, on la retire et on met le dernier sommet (i.e. 8) à sa place. Puisque $8 < 9 = \max\{7, 9\}$, il faut échanger ces deux sommets, et puisque $8 > 6$, il ne faut pas échanger plus loin :

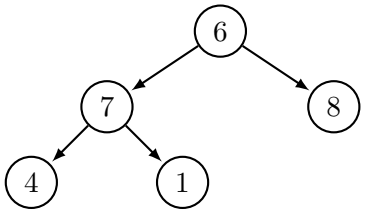


8	7	9	4	1	6	10

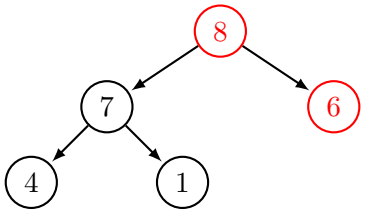


9	7	8	4	1	6	10

Extraction de 9 Puisque 9 est la racine, on la retire et on met le dernier sommet (i.e. 6) à sa place. Puisque $6 < 8 = \max\{7, 8\}$, il faut échanger ces deux sommets, et puisque 6 n'a alors plus de fils, on s'arrête là :

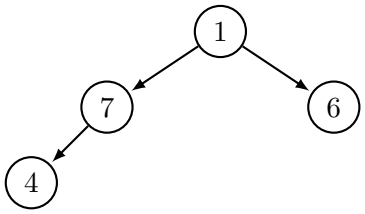


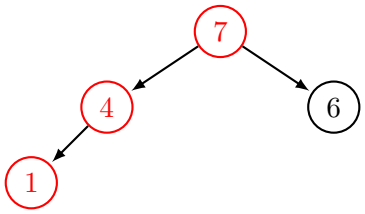
6	7	8	4	1	9	10



8	7	6	4	1	9	10

Extraction de 8 Puisque 8 est la racine, on la retire et on met le dernier sommet (i.e. 1) à sa place. Puisque $1 < 7 = \max\{7, 6\}$, il faut échanger ces deux sommets, et comme $1 < 4$, il faut encore échanger ces deux sommets :

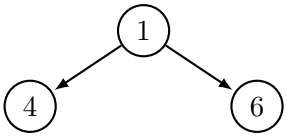


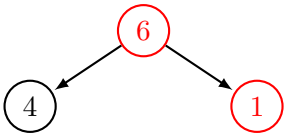


1	7	6	4	8	9	10

7	4	6	1	8	9	10

Extraction de 7 Puisque 7 est la racine, on la retire et on met le dernier sommet (i.e. 1) à sa place. Puisque $1 < 6 = \max\{4, 6\}$, il faut échanger ces deux sommets :

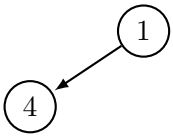


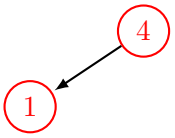


1	4	6	7	8	9	10

6	4	1	7	8	9	10

Extraction de 6 Puisque 6 est la racine, on la retire et on met le dernier sommet (i.e. 1) à sa place. Puisque $1 < 4$, il faut échanger ces deux sommets :





1	4	6	7	8	9	10

4	1	6	7	8	9	10

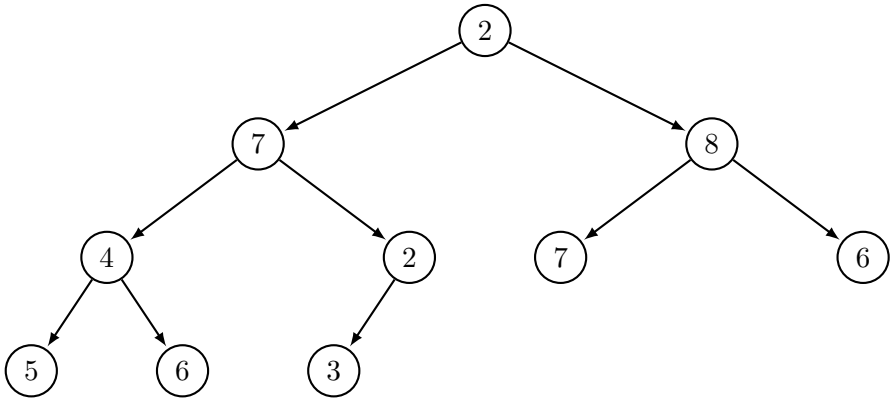
Extraction de 4 Puisque 4 est la racine, on la retire et on met le dernier sommet (i.e. 1) à sa place. 1 est le seul sommet dans le tas, donc il n'y a aucun échange à faire :

1	4	6	7	8	9	10

Extraction de 1 1 est le dernier sommet dans le tas, donc le tri s'arrête là et le vecteur trié est :

1	4	6	7	8	9	10

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.

Résolution. Conservons l'idée de base de l'insertion dans un tas : (i) insérer à première place disponible, et (ii) remonter l'élément tant que c'est nécessaire.

Ce deuxième point est celui qui va devoir être modifié afin de conserver la structure particulière de ce *tas*. Nous partons donc de la classe suivante :

```
class MinMaxHeap:
    def __init__(self):
        self.infos = []

    def __len__(self):
        return len(self.infos)

    def father(self, i):
        return (i-1) // 2

    def depth(self, idx):
        # return floor(log2(idx+1))
        ret = 0
        while idx > 0:
            idx = self.father(idx)
            ret += 1
        return ret

    def __str__(self):
        return str(self.infos)

    def swap(self, i, j):
        self.infos[i], self.infos[j] = self.infos[j], self.infos[i]
```

à laquelle nous allons ajouter la méthode `insert` qui est chargée d'insérer un élément tout en maintenant la structure. Il nous faut ici distinguer deux situations :

— si l'élément est inséré dans un niveau pair, alors il doit être strictement plus grand que ses ancêtres de niveau pair et son père – qui est de niveau impair – doit être supérieur ou égal à cet élément.
  — Si cette dernière condition *n'est pas* remplie, alors l'élément est échangé avec son père et puis le sommet (qui est donc maintenant dans un niveau impair)

doit potentiellement être remonté, tout en vérifiant la condition des sommets de niveau impair.
  - Si cette dernière condition *est* remplie, alors l'élément doit uniquement potentiellement être remonté tout en vérifiant la condition des sommets de niveau pair.
- si l'élément est inséré dans un niveau impair, alors il doit être strictement inférieur à tous ses ancêtres de niveau impair et son père – qui est de niveau pair – doit être inférieur ou égal à cet élément.
  - Si cette dernière condition *n'est pas* remplie, alors l'élément est échangé avec son père et puis le sommet (qui est donc maintenant dans un niveau pair) doit potentiellement être remonté, tout en vérifiant la condition des sommets de niveau pair.
  - Se cette dernière condition *est* remplie, alors l'élément doit uniquement potentiellement être remonté tout en vérifiant la condition des sommets de niveau impair.

```
1   def insert(self, e):
2       idx = len(self)
3       self.infos.append(e) # on ajoute à la fin
4       father_idx = self.father(idx)
5       if self.depth(idx)%2 == 0: # cas pair
6           if idx > 0 and self.infos[idx] > self.infos[father_idx]:
7               self.swap(idx, father_idx)
8               self.swim_max(father_idx)
9           else:
10              self.swim_min(idx)
11      else: # cas impair
12          if idx > 0 and self.infos[idx] < self.infos[father_idx]:
13              self.swap(idx, father_idx)
14              self.swim_min(father_idx)
15          else:
16              self.swim_max(idx)
```

Les méthodes `swim_max` et `swim_min` se chargent de faire remonter une entrée respectivement dans le cas max et le cas min. Elles sont définies comme suit :

```
1   def swim_max(self, idx):
2       father_idx = self.father(idx)
3       while father_idx > 0 and self.infos[idx] >
            self.infos[self.father(father_idx)]:
4           self.swap(idx, self.father(father_idx))
5           idx = self.father(father_idx)
6           father_idx = self.father(idx)
7
8   def swim_min(self, idx):
9       father_idx = self.father(idx)
10      while father_idx > 0 and self.infos[idx] <
            self.infos[self.father(father_idx)]:
11          self.swap(idx, self.father(father_idx))
12          idx = self.father(father_idx)
```

```
13            father_idx = self.father(idx)
```

Notez bien que les échanges ici se font entre un sommet et son grand-père et non entre un sommet et son père !

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
```

Résolution.

1. `h1` est une fonction de hachage triviale et donc `(x,)` et `(x-1, 1)` renvoient la même valeur, et ce pour tout `x`. En particulier `h1([x]) = x` pour tout `x` et donc `h1([177583]) = 177583`.
2. `h2` prend en compte la longueur du conteneur donc l'astuce du point précédent ne suffira pas ici. Par contre `(x1, x2)` et `(x1-1, x2+1)` auront le même hash. Ici `h2([x]) = x+1` pour tout `x` et donc `h2([177582]) = 177583`.
3. Cette fonction de hachage (qui n'est autre que la fonction `djb2`) est déjà un peu meilleure mais trouver une collision reste facile : `(x1, x2)` et `(x1+1, x2-33)` donnent le même résultat. De plus : $177583 = 33 \times 5381 + 10$, donc `x = (10,)` donne bien le hash demandé.

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)
```

```
8  assert p in S
9  p.x = -1
10 assert p not in S
```

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

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

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

Résolution. La deuxième assertion de ce code ne pourra pas passer puisque l'objet `p` de type `Pair` qui est créé est le même tout au long du code. Dès lors, même si la représentation interne de l'objet est modifiée (e.g. ligne 9), son adresse n'est pas modifiée, donc `id` et par extension `hash` renverront la même valeur aux lignes 8 et 10.

Pour éviter ce genre de problèmes, il faut définir une fonction de hachage pour le type `Pair` :

```
1     def __hash__(self):
2         return hash(self.x << 32 + (self.y & 31))
```

Maintenant, lorsque le hash est recalculé après que l'attribut `p.x` ait été modifié, la valeur retournée sera différente et la seconde assertion sera donc bien vérifiée.

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 :

$$\begin{aligned} M &= 11400714785074694791 \\ N &= 14029467366897019727 \\ P &= 2870177450012600261 \\ X &= 3527539 \end{aligned}$$

Implémentez cet algorithme et vérifiez le résultat.

Remarque : Cet algorithme suppose que les entiers sont encodés sur 64 bits (i.e. un `uint64_t` en C(++)). Toutes les opérations se font donc modulo 2^{64}.

Résolution. Commençons par définir les constantes nécessaires :

```
M = 11400714785074694791
N = 14029467366897019727
P = 2870177450012600261
X = 3527539
SIZE = 2 ** 64
```

La fonction `rotate_left(v, n)` effectue la rotation rotation de la valeur `v` (encodée sur 64 bits) de `n` bits vers la gauche :

```
def rotate_left_31(x):
    return (x << 31) | (x >> 33)
```

La fonction `xxhash` en elle-même doit donc itérer sur les éléments du tuple et calculer la valeur de hash du tuple en fonction des hash des éléments :

```
def xxhash(t):
    ret = P
    for item in t:
        ret += hash(item) * N
        ret &= SIZE-1
        ret = rotate_left_31(ret)
        ret *= M
        ret &= SIZE-1
    ret += (len(t) ^ P ^ X)
    return ret & (SIZE - 1)
```

Notons que les instructions `ret &= SIZE-1` correspondent à un 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 $\mathcal{O}(1)$.

Résolution. Voici une implémentation possible :

```
class BitVectorSet:
    def __init__(self, m):
        self.bit_vector = 0
        self.m = m

    def insert(self, x):
        self.bit_vector |= 1 << x

```

```
    def delete(self, x):
        self.bit_vector &= (1 << self.m) - 1 - (1 << x)

    def __contains__(self, x):
        return self.bit_vector & (1 << x)
```

Le vecteur de bits est ici représenté par l'attribut `bit_vector` (de type `int`). Cette implémentation est possible en Python car la taille des entiers n'est pas bornée. Dans un langage ne supportant que des entiers de taille fixe (e.g. C/C++), il faudrait utiliser un tableau de tels entiers. [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.

Résolution. Nous allons utiliser une liste `S` qui nous servira plus ou moins de *stack* qui stockera les entrées de l'ensemble. Plus précisément, lors de l'insertion d'un élément dans l'ensemble, une nouvelle entrée sera ajoutée dans cette liste et l'indice de cet élément sera placé dans la table.

```
class UninitializedDirectAddressingSet:
    def __init__(self, m):
        # pour simuler une table non initialisée
        self.T = [randint(0, 0xFFFFFFFF) for _ in range(m)]
        self.S = []

    def insert(self, x):
        self.T[x] = len(self.S)
        self.S.append(x)

    def __contains__(self, x):
        idx = self.T[x]
        return idx < len(self.S) and self.S[idx] == x

    def delete(self, x):
        idx = self.T[x]
        self.S[idx] = self.S.pop()
        self.T[self.S[idx]] = idx
```

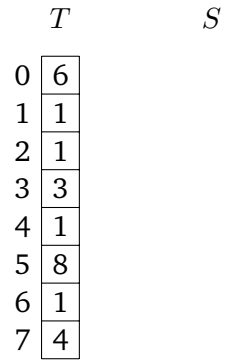
Il est aisé de déterminer si un élément est présent dans l'ensemble : il suffit de regarder si la valeur associée dans la table est bien un indice dans notre *stack* et si l'entrée associée

1. La spécialisation `std::vector<bool>` en C++ est potentiellement (bien que laissé libre à l'implémentation) déjà une implémentation efficace (en espace) d'un vecteur de bits

du *stack* est bien l'élément recherché.

Il y a tout de même une subtilité lors de la suppression d'un élément : on ne peut pas simplement retirer l'entrée du *stack*. En effet cela créerait un trou, ce qui n'est pas possible. Il faut donc remplir ce trou par le *top of stack* et mettre l'entrée de `T` associée à jour.

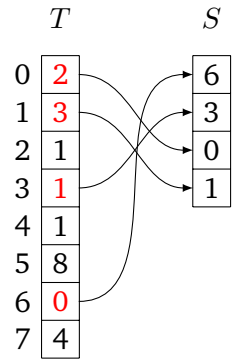
Prenons un exemple d'exécution. Voici une table T de taille 8 non-initialisée et un stack S vide :



Procédons aux insertions suivantes :

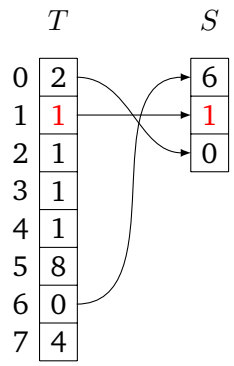
```
S.insert(6)
S.insert(3)
S.insert(0)
S.insert(1)
```

La table a donc été modifiée (les cases modifiées sont en rouge) et le stack n'est plus vide :



Nous pouvons voir que `1 in S` sera évalué à `True` puisque `T[1]` vaut 3, qui est un indice valable dans `S`, et l'entrée associée dans `S` contient bien 1. Par contre `2 in S` sera évalué à `False` puisque malgré le fait que `T[2] == 1`, l'entrée `S[1] == 3 != 2`, i.e. il y aurait pu y avoir n'importe quelle autre valeur dans `T[2]`, c'est juste un artefact de la non-initialisation.

Si depuis ces tableaux T et S, nous supprimons l'élément 3, nous obtenons la structure suivante (où les modifications sont à nouveau montrées en rouge) :



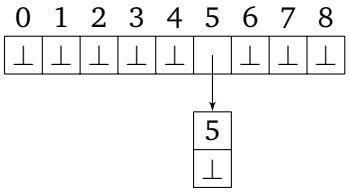
Exercice 11.6. Montrez comment on réalise l'insertion des éléments suivants dans une table de hachage où les collisions sont résolues par chaînage :
`5, 28, 19, 15, 20, 33, 12, 17, 10`

On suppose que la table contient 9 alvéoles et que la fonction de hachage est :

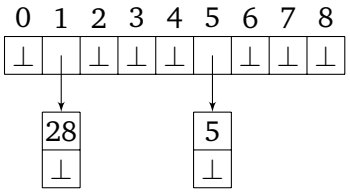
$$h(k) = k \mod 9.$$

Résolution.

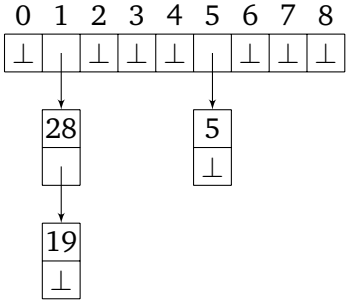
insertion de 5



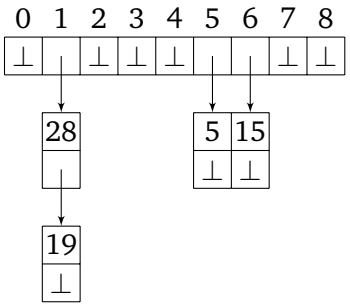
insertion de 28



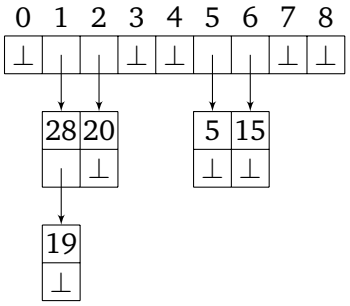
insertion de 19



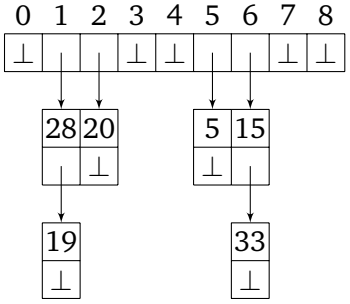
insertion de 15



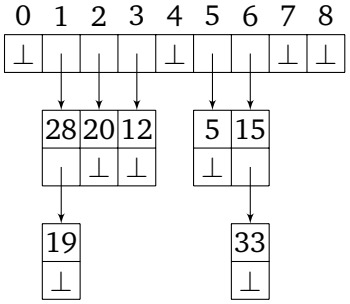
insertion de 20



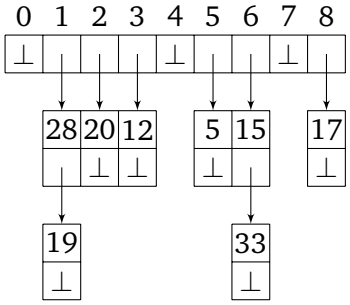
insertion de 33



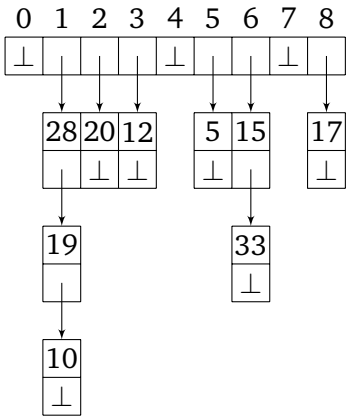
insertion de 12



insertion de 17



insertion de 10


0 1 2 3 4 5 6 7 8
28 20 12
5 15
17
19
33
10

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 ?

Résolution. Puisque les hashs sont précalculés (ou de manière équivalente que la fonction h se calcule en temps $\mathcal{O}(1)$), afin de comparer les strings s_1 et s_2 le plus efficacement, on compare d'abord $h(s_1)$ avec $h(s_2)$, et on ne compare s_1 avec s_2 que si les hashs sont égaux.

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. $d = \mathrm{GCD}(m, h(k)) \gneqq 1$) 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.

Résolution. Notons d le GCD entre $h_2(k)$ et m. Par définition, d est le plus grand nombre qui divise m et $h_2(k)$. Dès lors il existe un entier p tel que $dp = m$ et il existe un entier q tel que $dq = h_2(k)$ tels que p et q sont premiers entre eux. Pour $j > 0$:

$$h_1(k) + jh_2(k) \mod m = h_1(k) + jdq \mod m = h_1(k) + (\ell p + j)dq \mod m$$

car $\ell pd \mod m = 0$ pour tout $\ell > 0$. Dès lors en faisant varier j de 0 à $m-1$, seules $p = \frac{m}{d}$ valeurs distinctes seront visitées.

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) :

```python
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)
        # ...
```

Résolution. Chaque entrée de la table doit contenir un nœud d'une liste chaînée, et la classe `Set` doit également maintenir une liste chaînée des nœuds pas utilisés. Nous devons donc écrire la classe `Node` et la classe `LinkedList`. Chaque nœud doit doit contenir deux références vers d'autres sommets (le sommet précédent et le sommet suivant dans la chaînée puisqu'afin de permettre l'insertion et la suppression en $\mathcal{O}(1)$, la liste doit être doublement chaînée) ainsi que la valeur contenue (entrée `data`) et un flag binaire signifiant si le nœud est libre (donc dans la liste chaînée des entrées inutilisées) ou s'il est utilisé pour représenter une entrée de l'ensemble (entrée `flag`) :

```
class Node:
    def __init__(self, prev=None, next_=None, data=None):
        self.prev = prev
        self.next_ = next_
        self.data = data
        self.flag = False # True ssi le nœud est occupé

    def copy_from(self, other: Node) -> None: # O(1)
        self.prev = other.prev
        self.next_ = other.next_
        self.data = other.data
        self.flag = other.flag

    def insert_back(self, other: Node) -> None: # O(1)
        other.next_ = self.next_
        if self.next_ is not None:
            self.next_.prev = other
        self.next_ = other
        other.prev = self
```

La méthode `copy_from(self, other)` remplace tous les attributs de `self` par ceux de `other` et la méthode `insert_back(self, other)` place le nœud `other` après le nœud `self` en s'assurant que les références sont correctes.

La classe `LinkedList` est un wrapper pour la classe `Node` :

```
class LinkedList:
    def __init__(self, head: Node):
        self.head = head

    def insert(self, node: Node) -> None: # O(1)
        self.head.insert_back(node)

    def find(self, data: object) -> Node: # O(len(self))
```

```
        node = self.head
        while node is not None:
            if node.data == data:
                return node
            node = node.next_
        return None

    def delete(self, node: Node) -> None: # O(1)
        if node is self.head:
            self.head = self.head.next_
        prev_node = node.prev
        next_node = node.next_
        if prev_node is not None:
            prev_node.next_ = next_node
        if next_node is not None:
            next_node.prev = prev_node
```

On y ajoute une méthode `pop` qui permet de retirer un élément quelconque de la liste chaînée et de le récupérer :

```
    def pop(self) -> Node: # O(1)
        ret = self.head
        if ret is None:
            raise ValueError('Pop on an empty linked list')
        self.head = ret.next_
        self.head.prev = None
        ret.prev = ret.next_ = None
        return ret
```

Le constructeur de la classe `Set` doit créer la table et y initialiser la liste chaînée des cellules vides (qui contient initialement toutes les cases de la table) :

```
class Set:
    def __init__(self, m):
        self.T = [None] * m
        self.m = m
        self.T[0] = Node()
        for i in range(1, self.m): # Theta(m)
            self.T[i] = Node(self.T[i-1])
            self.T[i-1].next_ = self.T[i]
        self.free_nodes_ll = LinkedList(self.T[0])
```

Pour insérer un élément, on regarde si la cellule associée dans la table est libre ou non, i.e. si le flag est à `False` ou à `True`.

```
    def insert(self, x: object) -> None: # O(1)
        k = hash(x) % self.m
        # si nœud déjà  assigné
        if not self.T[k].flag: # Cas 1
```

```
5            self.insert_unused(k, x)
6        else: # Cas 2
7            self.insert_already_flagged(k, x)
```

Si elle l'est (cas 1), on peut déduire que cette cellule est dans la liste chaînée des cellules libres, et il faut donc la supprimer de la liste, y insérer la bonne valeur et adapter le flag :

```
1    def insert_unused(self, k: int, x: object) -> None: # O(1)
2        self.free_nodes_ll.delete(self.T[k])
3        self.T[k].data = x
4        self.T[k].flag = True
5        self.T[k].next_ = self.T[k].prev = None
```

Si elle ne l'est pas (cas 2), on peut déduire que la cellule n'est pas dans la liste chaînée des cellules libres, et que donc elle contient un nœud faisant partie d'une liste chaînée d'éléments ayant le même hash. Il faut encore y discerner deux cas : soit la case est utilisée par un autre objet ayant le même hash que la valeur que l'on cherche à insérer, i.e. une collision (cas 2.a) soit la case a été récupérée à un certain moment de la liste chaînée des cellules libres mais contient une entrée ayant un hash différent de `x`.

Dans le cas 2.a, il faut donc uniquement ajouter un nouveau nœud à cette liste chaînée en allant le récupérer dans la liste des nœuds libres alors que dans le cas 2.b, il faut déplacer le contenu de cette case dans une autre (récupérée dans la liste des cellules libres) et entamer un chaînage pour les collisions avec `x` :

```
1    def insert_already_flagged(self, k: int, x: object) -> None: # O(1)
2        # on récupère un nœud pas encore utilisé
3        new_node = self.free_nodes_ll.pop()
4        # on le marque
5        new_node.flag = True
6        # S'il est utilisé pcq un autre x de même hash a déjà  été inséré
7        if hash(self.T[k].data) == hash(x): # Cas 2.a
8            new_node.data = x
9            self.T[k].insert_back(new_node)
10       # S'il est utilisé dans un chaînage de collisions
11       else: # Cas 2.b
12           new_node.copy_from(self.T[k])
13           if new_node.prev is not None:
14               new_node.prev.next_ = new_node
15           if new_node.next_ is not None:
16               new_node.next_.prev = new_node
17           self.T[k].data = x
18           self.T[k].prev = self.T[k].next_ = None
```

Pour la recherche du nœud contenant un élément `x`, il faut regarder l'entrée d'indice `hash(x) % m` dans la table. Si le flag de cette entrée est à `False`, c'est que ce nœud est dans la liste des cellules libres et que donc aucun élément de même hash que `x` n'est dans l'ensemble (en particulier `x` lui-même n'y est pas). Si par contre le flag est à `True`, alors

`T[hash(x) % m]` est le premier élément de la liste chaînée correspondant aux collisions avec `x`. Il faut donc parcourir la liste chaînée jusqu'à trouver cet élément (via la méthode `LinkedList.find`) :

```
    def find(self, x: object) -> Node: # O(m)
        k = hash(x) % self.m
        if not self.T[k].flag:
            return None
        ll = LinkedList(self.T[k])
        return ll.find(x)
```

Pour la suppression d'un élément, nous avons accès au nœud (donc l'entrée de la table) qui contient l'élément à supprimer et nous voulons le remettre dans la liste chaînée des cellules libres. Il faut tout de même, pour cela, distinguer deux cas : soit ce nœud est la tête de sa propre liste (cas 1), soit il vient après la tête (cas 2). Le cas 1 se sépare encore en deux : soit la tête est le seul élément de la liste (cas 1.a) soit la liste contient au oins deux éléments (cas 1.b).

Dans le cas 1.a, retirer ce nœud ne posera pas de problème pour la recherche/suppression d'autres valeurs ayant le même hash que celui de l'élément qui est en cours de suppression car aucun tel élément n'est dans l'ensemble (sinon la liste aurait d'autres éléments que juste la tête). Dès lors son flag peut simplement être passé à `False` et il peut être ajouté à la liste des entrées libres (notons également que dans le code présenté ici, le champ `data` est remis à `None` lors d'une suppression pour éviter de maintenir des références qui empêcheraient la libération de mémoire). Dans le cas 1.b par contre, la tête de liste ne peut être retirée. Dès lors, c'est le nœud qui vient juste après la tête qui est supprimé, juste après avoir copié sa donnée dans le nœud de tête et mis à jour les pointeurs.

Le cas 2 ressemble très fort au cas 1.a avec la particularité qu'il faut d'abord aller mettre à jour les références dans la liste chaînée qui contient ce nœud, et seulement ensuite mettre ce nœud dans la liste chaînée des cases libres :

```python
    def delete(self, node: Node) -> None: # O(1)
        assert node.flag # seul un nœud assigné peut être supprimé
        # Si le nœud à supprimer est la tête d'une chaîne
        if node.prev is None: # Cas 1
            if node.next_ is None: # Cas 1.a
                node.flag = False
                node.data = False
                self.free_nodes_ll.insert(node)
            else:  # Cas 1.b
                next_node = node.next_
                node.data = next_node.data
                node.next_ = next_node.next_
                next_node.flag = False
                next_node.data = None
                if node.next_ is not None:
                    node.next_.prev = node
                LinkedList(node).delete(next_node)
                self.free_nodes_ll.insert(next_node)
        # Si le nœud à supprimer est dans une chaîne sans en être la tête
        else:  # Cas 2
            node.flag = False
            node.data = None
            LinkedList(node.prev).delete(node)
            self.free_nodes_ll.insert(node)
```

La méthode `remove` se distingue de la méthode `delete` par le fait qu'elle ne prend pas une référence vers un nœud en paramètre mais bien un élément quelconque de l'ensemble. Il faut donc d'abord trouver le nœud associé (via la méthode `find`) et puis appeler `delete` sur ce nœud :

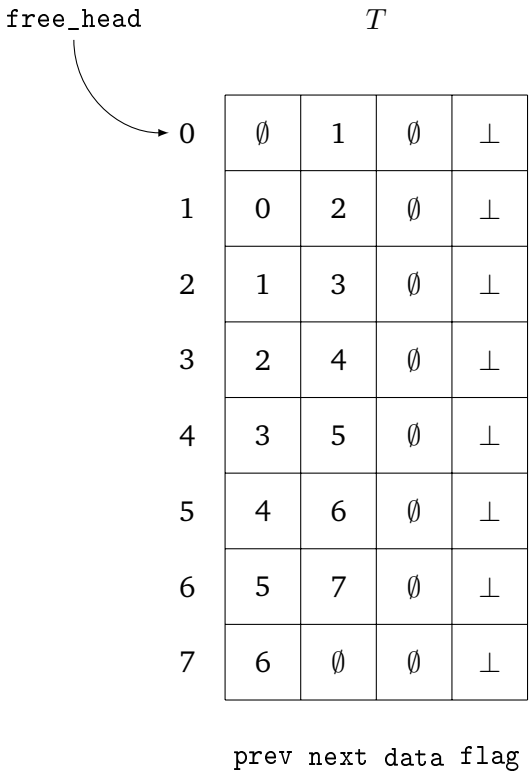
```python
    def remove(self, x: object) -> None: # O(m)
        node = self.find(x)
        if node is None:
            raise KeyError(f'{x} is not in the set')
        self.delete(node)
```

Prenons un exemple pour expliciter les différents chaînages. Dans le tableau suivant, le symbole $\emptyset$ est utilisé pour représenter `None` (pour ne pas le confondre avec $\perp$ qui représente `False`). De plus, les attributs `prev` et `next` sont montrés avec des indices et pas avec des références pour simplifier la lecture du schéma. Nous allons procéder à plusieurs insertions et suppressions afin de montrer les modifications appliquées à la table. Comme d'habitude, les entrées modifiées sont marquées en rouge.

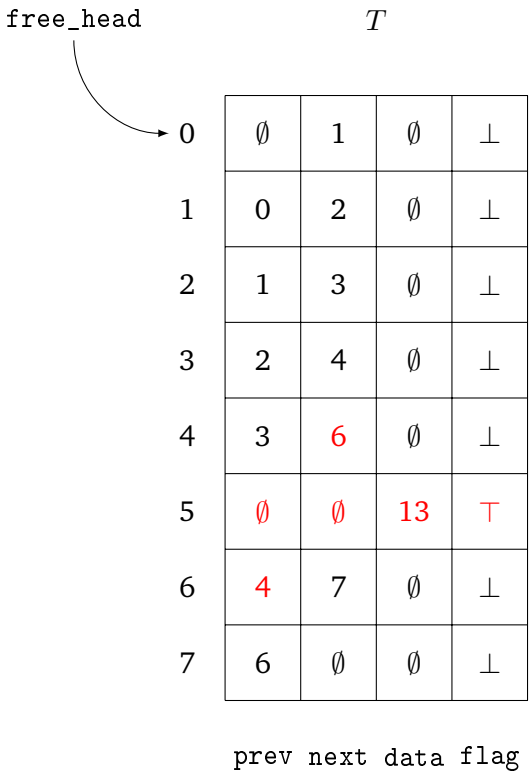
Notons que les schémas présentés montrent la table T comme étant une matrice à 4 colonnes. Cela nous sert ici à représenter séparément toutes les entrées du tableau qui sont de type `Node` et d'expliciter les attributs (c.f. les noms de colonnes en dessous de la table).

Ici supposons que l'ensemble contiendra uniquement des entiers positifs munis de la fonc-

tion de hachage identité (i.e. `hash(x) == x`) pour tout entier `x`. L'indice associé à une valeur `x` quelconque est donc `x % 8` (puisque la table est de taille 8).

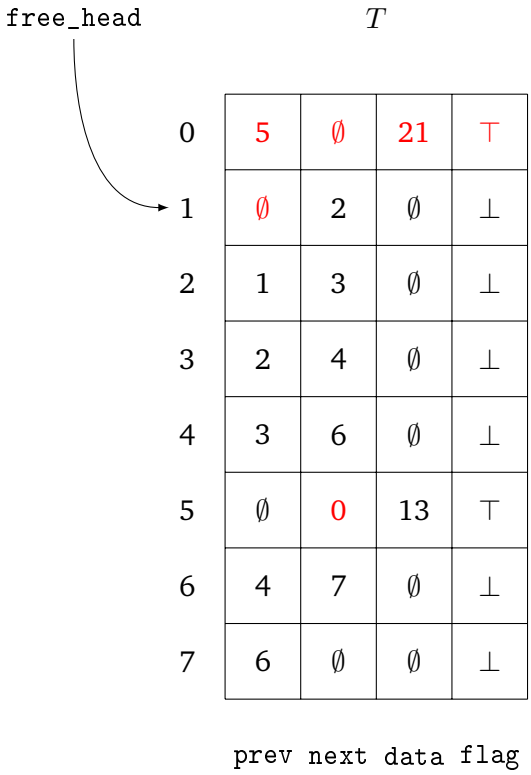


Insérons l'élément 13 à l'ensemble (cas 1) :

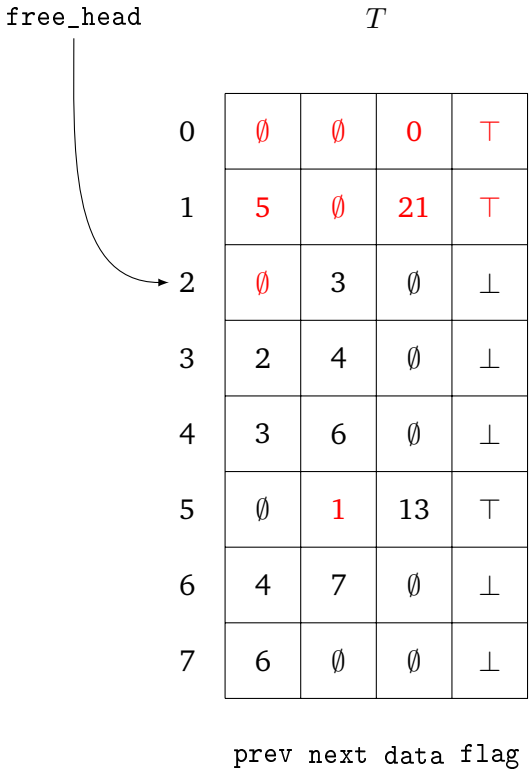


Insérons ensuite l'élément 21 (qui entre donc en collision avec l'élément 13 car tous deux

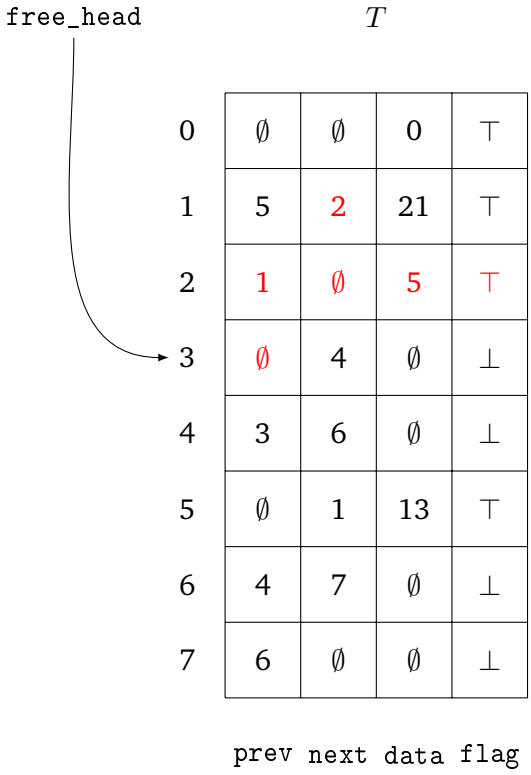
sont hachés vers l'indice 5). Il va donc falloir récupérer un nœud libre, i.e. la tête de la liste (donc l'entrée `T[0]`) et le chaîner avec l'entrée d'indice 5 (cas 2.a) :



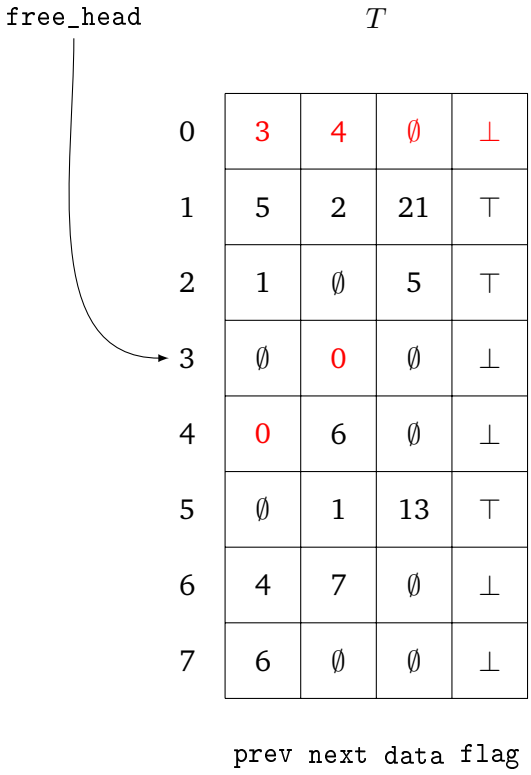
Insérons maintenant l'élément 0 qui devrait donc être à l'indice 0. Or puisque cet indice est déjà occupé par l'entrée 21 (ce qui se remarque au flag qui est à $\top$), il va falloir déplacer cette entrée pour pouvoir utiliser l'entrée d'indice 0 pour l'élément 0 (cas 2.b) :



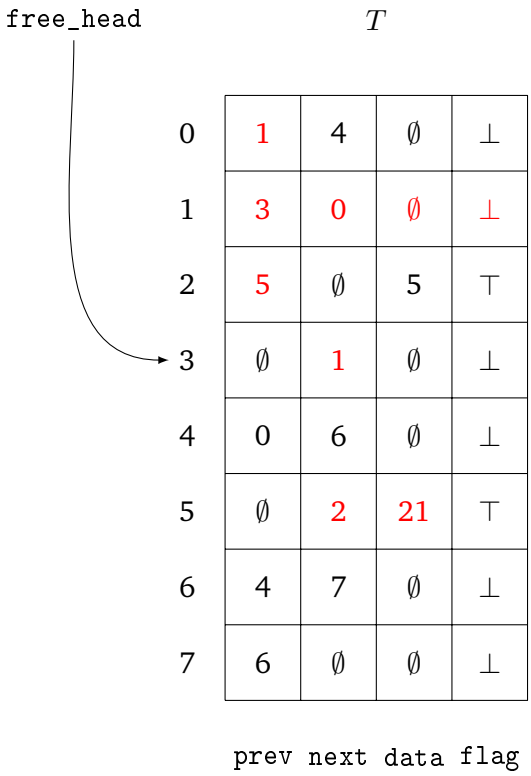
Afin de pouvoir explorer les différentes situations lors de la suppression, il nous faut encore ajouter un élément en collision avec 13 et 21. Insérons donc l'élément 5 :



Supprimons maintenant l'élément 0 qui est seul sur sa chaîne de collisions, ce qui se voit par le fait que `prev` et `next` sont tous deux à `None` (cas 1.a) :

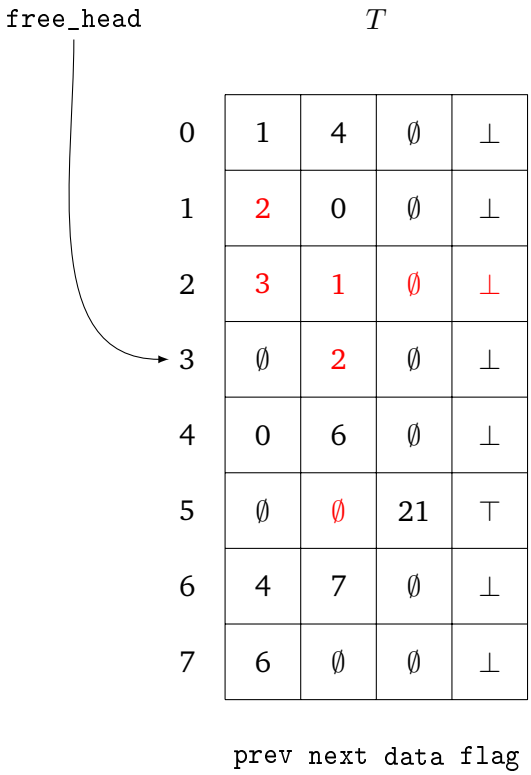


Supprimons ensuite l'élément 13 (à l'indice 5), qui est également une tête de liste puisque son attribut `prev` est à `None` mais qui n'est pas seul puisque `next` n'est pas à `None` (cas 1.b) :



Il nous reste alors à supprimer l'élément 5 (qui est actuellement à l'indice 2) qui est donc

une collision mais qui n'est pas la tête de liste (cas 2) :



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.

Résolution. Le constructeur de la classe `Dict` va juste créer une table pour les valeurs et une table pour les indices :

```
def log2(m):
    log2m = 0
    while m > 0:
        m >>= 1
        log2m += 1
    return log2m-1

def is_power_of_2(m):
    return (1 << log2(m)) == m

class Dict:
    def __init__(self, m):
        assert is_power_of_2(m)
        self.keys = [None] * m
        self.values = [None] * m
        self.size = 0
        self.m = m
```

Implémentons également quelques méthodes spéciales pour faciliter l'utilisation de la classe :

```
    def __len__(self):
        return self.size

    def __getitem__(self, key):
        return self.find(key)

    def __setitem__(self, key, value):
        self.insert(key, value)

    def __contains__(self, key):
        return self.keys[self._find_idx(key)] == key
```

Les méthodes `find` et `insert` ont besoin de lier une clef à un indice via les sondages. Cette partie est donnée à la méthode `_find_idx(self, key)` :

```
    def _find_idx(self, key):
        p = hash(key)
        j = p % self.m
        counter = 0
        while self.keys[j] is not None and self.keys[j] != key and counter
            < self.m:
            p //= 32
            j = (5*j + 1 + p) % self.m
            counter += 1
        if counter == self.m:
            return None
        return j
```

C'est bien cette méthode qui se charge de sonder les indices sur base des relations de récurrence définies et qui gère un potentiel overflow en renvoyant `None`. Les méthodes `find` et `insert` doivent maintenant uniquement déterminer l'indice adapté pour la clef demandée et soit renvoyer la valeur associée, soit modifier les entrées de `keys` et `values` à cet indice :

```
    def find(self, key):
        idx = self._find_idx(key)
        if idx is None or self.keys[idx] != key:
            raise KeyError(key)
        return self.values[idx]

    def insert(self, key, value):
        idx = self._find_idx(key)
        if idx is None:
            raise ValueError('Overflow in the Dict')
        if self.keys[idx] != key:
            self.size += 1
```

```
13          self.keys[idx] = key
14          self.values[idx] = value
```

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.

Résolution. La fonction de hachage h est définie par :

$$h(x) \coloneqq \sum_{k=1}^{n-1} x_k 2^{pk} \mod (2^p - 1).$$

Soient x et y tels qu'il existe $k_1, k_2 \in [\![0, n[\![$ tels que $x_{k_1} = y_{k_2}$, $x_{k_2} = y_{k_1}$ et $x_k = y_k$ pour tout $k \notin \{k_1, k_2\}$. Alors :

$$\begin{aligned}
h(x) - h(y) &= \left[\left(x_{k_1} 2^{k_1 p} + x_{k_2} 2^{k_2 p}\right) - \left(y_{k_1} 2^{k_1 p} + y_{k_2} 2^{k_2 p}\right)\right] \mod 2^p - 1 \\
&= \left[\left(x_{k_1} 2^{k_1 p} + x_{k_2} 2^{k_2 p}\right) - \left(x_{k_2} 2^{k_1 p} + x_{k_1} 2^{k_2 p}\right)\right] \mod 2^p - 1 \\
&= \left[x_{k_1}\left(2^{k_1 p} - 2^{k_2 p}\right) + x_{k_2}\left(2^{k_2 p} - 2^{k_1 p}\right)\right] \mod 2^p - 1 \\
&= \left[(x_{k_1} - x_{k_2})\left(2^{k_1 p} - 2^{k_2 p}\right)\right] \mod 2^p - 1.
\end{aligned}$$

Or $2^p - 1$ divise $2^{k_1 p} - 2^{k_2 p}$, donc $h(x) = h(y)$. En effet, par la formule de somme d'une suite géométrique :

$$\begin{aligned}
(2^p - 1)\left(\sum_{k=0}^{k_1 - 1} 2^{kp} - \sum_{k=0}^{k_2 - 1} 2^{kp}\right) &= (2^p - 1)\frac{2^{k_1 p - 1 + 1} - 1}{2^p - 1} - (2^p - 1)\frac{2^{k_2 p - 1 + 1} - 1}{2^p - 1} \\
&= (2^{k_1 p} - 1) - (2^{k_2 p} - 1) \\
&= 2^{k_1 p} - 2^{k_2 p}.
\end{aligned}$$

Cette propriété est indésirable dans le cas où la fonction de hash est utilisée en tant que *one way function* (i.e. une fonction qui est facile à calculer dans un sens mais dont la réciproque est virtuellement impossible). Par exemple si la fonction h est censée être utilisée pour signer numériquement un document, une telle propriété rendrait la fraude trop facile.

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 :

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

Résolution. Nous avons $\mathcal{H} = \{h_\alpha\}_{\alpha \in A} \subset B^U$, donc l'ensemble $\mathcal{H}$ est fini (et par extension, l'ensemble A des indices également). Supposons que $\mathcal{H}$ est ε-universelle, mais supposons par l'absurde que cette valeur :

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

Définissons la valeur $N_{h,k,\ell}$ qui vaut 0 si $h(k) \neq h(\ell)$ et qui vaut 1 si $h(k) = h(\ell)$. Regardons ensuite le nombre total de collisions existantes sur toutes les fonction $h \in \mathcal{H}$ et entre toutes les paires d'éléments disjoints de U :

$$N = \sum_{h \in \mathcal{H}} \sum_{k \neq \ell} N_{h,k,\ell}.$$

Par définition :

$$\mathbb{P}_{\mathcal{H}}[h(k) = h(\ell)] = \frac{|\{\alpha \in A : h_\alpha(k) = h_\alpha(\ell)\}|}{|A|},$$

et par hypothèse :

$$\mathbb{P}_{\mathcal{H}}[h(k) = h(\ell)] < \frac{1}{|B|} - \frac{1}{|U|}.$$

Dès lors, pour une paire (k, ℓ) (avec $k \neq \ell$) fixée, on sait que :

$$\sum_{h \in \mathcal{H}} N_{h,k,\ell} = |\{\alpha \in A : h_\alpha(k) = h_\alpha(\ell)\}| < |A| \left(\frac{1}{|B|} - \frac{1}{|U|} \right).$$

On en déduit que :

$$N < \binom{|U|}{2} |A| \left(\frac{1}{|B|} - \frac{1}{|U|} \right) = \frac{1}{2}(|U| - 1) |A| \left(\frac{|U|}{|B|} - 1 \right).$$

Remarquons ensuite qu'à $h \in \mathcal{H}$ fixé, le nombre de paires (k, ℓ) d'éléments distincts tels que $h(k) = h(\ell)$ est au moins

$$\rho \binom{\beta + 1}{2} + (|B| - \rho) \binom{\beta}{2} = |B| \binom{\beta}{2} + \rho\beta$$

pour $\beta \coloneqq \left\lfloor \frac{|U|}{|B|} \right\rfloor$ et $\rho \coloneqq |U| - |B|\,\beta$ (ou encore $|U| = \beta\,|B| + \rho$) puisque dans le meilleur cas, ρ des $|B|$ valeurs de hash possible sont associées à $\beta + 1$ valeurs distinctes de U, et les $|B| - \rho$ valeurs restantes sont associées à β valeurs distinctes de U.

Dès lors, on sait que :

$$N \geq \sum_{h \in \mathcal{H}} \left(|B| \binom{\beta}{2} + \rho\beta \right).$$

On en déduit finalement que :

$$|A| \left(|B| \binom{\beta}{2} + \rho\beta \right) \leq N < \frac{1}{2}(|U| - 1) |A| \left(\frac{|U|}{|B|} - 1 \right),$$

ou encore de manière équivalente :

$$|B|\,\beta(\beta - 1) + 2\rho\beta < (|U| - 1) \left(\frac{|U|}{|B|} - 1 \right) = (\rho + |B|\,\beta - 1) \left(\frac{\rho}{|B|} + \beta - 1 \right),$$

qui se simplifie en :

$$0 < \rho^2 - |B|\,\rho - (|U| - |B|).$$

Or puisque $\rho < |B|$, on sait que $\rho^2 < |B|\,\rho$. De plus, $|B| \leq |U|$. On a alors :

$$0 < \rho^2 - |B|\,\rho - (|U| - |B|) < -(|U| - |B|) < 0,$$

ce qui est une contradiction.

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 15. Soit V un ensemble de sommets.
- Un ensemble $E \subset \binom{V}{2}$ (i.e. un ensemble E de la forme $\{\{v_{i_1}, v_{j_1}\}, \dots, \{v_{i_m}, v_{j_m}\}\}$ où pour tout k : $i_k \neq j_k$) est appelé *ensemble d'arêtes*.
- Un ensemble $E \subset V \times V$ (i.e. un ensemble E de la forme $\{(v_{i_1}, v_{j_1}), \dots (v_{i_m}, v_{j_m})\}$) est appelé *ensemble d'arcs*.
- Si E est un ensemble d'arêtes, on appelle la paire $G = (V, E)$ un *graphe non-dirigé*, et si E est un ensemble d'arcs, on appelle la paire $G = (V, E)$ un *graphe dirigé* (ou *digraphe*).
- On appelle *graphe pondéré* tout triplet $G = (V, E, \gamma)$ tel que (V, E) est un (di)graphe et où $\gamma : E \to \mathbb{R}$ 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 16. Soit un graphe dirigé pondéré simple $G = (V, E, \gamma)$ tel que $V = \{v_1, \dots, v_n\}$ et $E = \{e_1, \dots e_m\}$.
- La matrice d'adjacence de G est une matrice $A \in \mathbb{R}^{n \times n}$ telle que :

$$A_{ij} = \begin{cases} \gamma(v_i, v_j) & \text{si } (v_i, v_j) \in E, \\ 0 & \text{sinon.} \end{cases}$$

- La matrice d'incidence de G est une matrice $I \in \mathbb{R}^{n \times m}$ telle que :

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

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 17. 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 2.37. *(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 2.38. *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 :

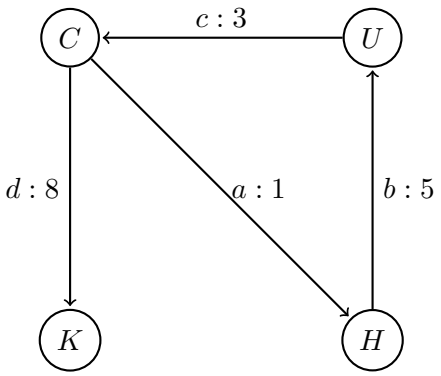
$$\sum_{v \in V \setminus U} \deg v = 0 \mod 2.$$

Dès lors nous déduisons que (par la proposition 2.37) :

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

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.

Résolution. Notons pour commencer que G_1 est un graphe dirigé pondéré simple puisqu'aucun lacet n'est présent. L'ensemble des sommets de G_1 est donné par $V(G_1) = \{C, H, K, U\}$ et l'ensemble des arêtes de G_1 est donné par $E(G_1) = \{a, b, c, d\}$ dont les poids sont :

— $\gamma(a) = 1$;
— $\gamma(b) = 5$;
— $\gamma(c) = 3$;
— $\gamma(d) = 8$.

Nous avons alors la matrice d'adjacence $A(G_1)$ suivante :

	C	H	K	U
C	0	1	8	0
H	0	0	0	5
K	0	0	0	0
U	3	0	0	0

ainsi que la matrice d'incidence $I(G_1)$ suivante :

	a	b	c	d
C	1	0	-3	8
H	-1	5	0	0
K	0	0	0	-8
U	0	-5	3	0

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

Résolution. Il nous suffit de replacer toutes les valeurs strictement positives par 1 et toutes les valeurs strictement négatives par -1 (nous laissons donc les 0 inchangés).

La matrice d'adjacence $A(G'_1)$ est donc :

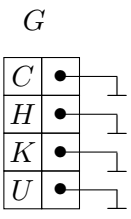
	C	H	K	U
C	0	1	1	0
H	0	0	0	1
K	0	0	0	0
U	1	0	0	0

et la matrice d'incidence $I(G'_1)$ est donc :

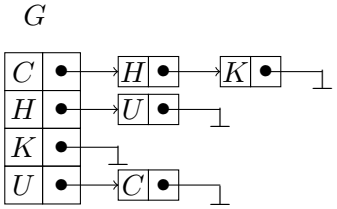
	a	b	c	d
C	1	0	-1	1
H	-1	1	0	0
K	0	0	0	-1
U	0	-1	1	0

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

Résolution. Nous partons donc d'une liste vide pour chaque sommet :

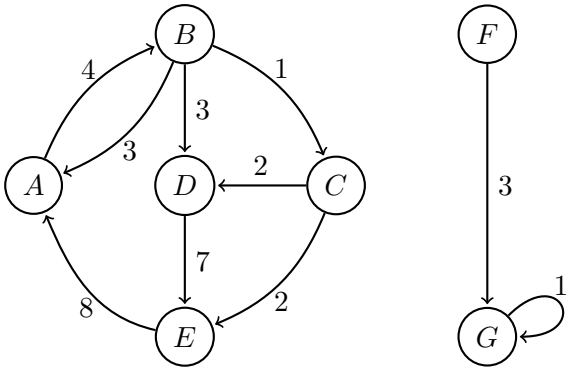


dans laquelle nous devons ajouter tous les arcs :



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

Résolution. Partons d'un graphe vide et ajoutons l'un après l'autre les sommets en lisant la matrice d'adjacence ligne par ligne :



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 ?

Résolution. Commençons par écrire des classes `Vertex`, `Edge` et `LinkedList` pour pouvoir implémenter une structure dynamique :

```
class Vertex:
    def __init__(self, vertex_idx):
        self.idx = vertex_idx
        self.edges = LinkedList()

    def add_edge(self, dest, weight):
        self.edges.insert(dest, weight)

    @property
    def out_edges(self):
        return self.edges

class Edge:
    def __init__(self, vertex, weight, previous=None, next=None):
        self.dest = vertex
        self.weight = weight
        self.previous = previous
        self.next = next

class LinkedList:
    def __init__(self):
```

```
        self.head = None

    def insert(self, vertex, weight):
        e = Edge(vertex, weight, previous=None, next=self.head)
        self.head.previous = e
        self.head = e

    def __iter__(self):
        e = self.head
        while e is not None:
            yield e
            e = e.next
```

La structure dynamique d'un graphe peut être implémentée comme suit :

```
class DynamicGraph:
    def __init__(self, n):
        self.neighbours = [Vertex(i) for i in range(n)]

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

    def link(self, i, j, weight):
        vertex = self.neighbours[i]
        vertex.add_edge(j, weight)

    def out_edges(self, i):
        return self.neighbours[i].out_edges
```

et la structure statique comme suit :

```
class StaticGraph:
    def __init__(self, n):
        self.adj = [[0]*n for i in range(n)]

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

    def link(self, i, j, weight=1):
        self.adj[i][j] = weight

    def weight(self, i, j):
        return self.adj[i][j]
```

Pour la conversion statique vers dynamique, il faut juste itérer sur toutes les entrées de la matrice d'adjacence et les ajouter progressivement dans la structure dynamique :

```
1 def static_to_dynamic(G):
2     ret = DynamicGraph(G.n)
3     for i in range(G.n):
4         for j in range(G.n):
5             weight = G.weight(i, j)
6             if weight > 0:
7                 ret.link(i, j, weight)
8     return ret
```

alors que pour la conversion statique vers dynamique, il faut itérer sur les listes chaînées de voisins pour chaque sommet et les entrer dans la matrice d'adjacence :

```
1 def dynamic_to_static(G):
2     ret = StaticGraph(G.n)
3     for i in range(G.n):
4         for edge in G.out_edges(i):
5             ret.link(i, edge.dest, edge.weight)
6     return ret
```

Proposition 2.39. *Les fonctions* `static_to_dynamic` *et* `dynamic_to_static` *effectuent* $\Theta(n^2)$ *opérations pour passer d'une représentation à l'autre sur un graphe* G *à* n *sommets et* m *arêtes.*

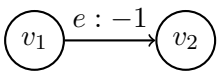
Démonstration. Dans `static_to_dynamic`, un graphe dynamique `ret` est créé (en $\Theta(n)$ opérations) et les variables `i` et `j` parcourent toutes les paires $(i, j) \in [\![0, n[\![^2$. Il y a n^2 telles paires, et le traitement de chacune de ces paires est faite en $\mathcal{O}(1)$ (l. 5-7). Il y a donc au total $\Theta(n) + \Theta(n^2) = \Theta(n^2)$ opérations.

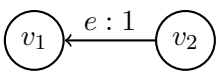
La fonction `dynamic_to_static` crée un graphe statique `ret` (en $\Theta(n^2)$ opérations pour initialiser la matrice d'adjacence) et puis tous les sommet sont visités, et chaque voisin de chacun de ces sommets également pour un total de $2m$ (voir proposition 2.37) visites. Chacun de ces traitements est fait en $\mathcal{O}(1)$ pour un total de $\Theta(n^2) + \Theta(m) = \Theta(n^2)$ opérations. □

Pour le bonus, remarquons que contrairement à une matrice d'adjacence, une matrice d'incidence n'est pas suffisante pour définir un digraphe de manière univoque. Par exemple, la matrice d'incidence suivante :

	e
v_1	-1
v_2	1

peut désigner les deux graphes suivants :

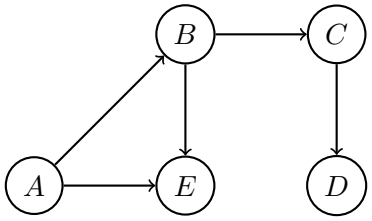




Si on impose que le graphe soit non-pondéré, alors il est clair qu'il est question du deuxième graphe. De manière plus générale, on peut imposer que les poids soient positifs afin d'assurer qu'une matrice d'incidence représente un unique graphe.

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.

Résolution. La matrice d'adjacence de G est :

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	1
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	0	0	0

Listons les sommets accessibles (et la longueur des chemins y menant) :

Depuis A

Longueur 0 A ;

Longueur 1 B et E ;

Longueur 2 C ;

Longueur 3 D.

Depuis B

Longueur 0 B ;

Longueur 1 C et E ;

Longueur 2 D.

Depuis C

Longueur 0 C ;

Longueur 1 D.

Depuis D D (longueur 0).

Depuis E E (longueur 0).

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

Résolution. L'idée derrière l'algorithme de Floyd-Warshall est la suivante : pour chaque sommet k du graphe G, on itère sur toutes les paires de sommets i, j, et on regarde si on peut joindre i à j en passant par k. Voici une implémentation possible de l'algorithme de Floyd-Warshall utilisant la classe `StaticGraph` de la séance précédente :

```
1  def floyd_warshall(G):
2      assert isinstance(G, StaticGraph)
3      M = [[bool(e) for e in row] \
4           for row in G.adj]
5      for i in range(G.n):
6          M[i][i] = True
7      for k in range(G.n):
8          for i in range(G.n):
9              if M[i][k]:
10                 for j in range(G.n):
11                     M[i][j] |= M[k][j]
```

La première boucle `for` (sur `k`) permet d'itérer sur les colonnes (c.f. l. 7). Observons la modification de la matrice M (qui vaut initialement la matrice d'adjacence) après chaque valeur de k :

Lorsque $k = 0$. Aucun arc n'arrive dans le sommet A, donc la matrice M n'est pas modifiée :

	A	B	C	D	E
A	1	1	0	0	1
B	0	1	1	0	1
C	0	0	1	1	0
D	0	0	0	1	0
E	0	0	0	0	1

Lorsque $k = 1$. Le seul arc arrivant dans le sommet B vient du sommet A et B est lié à C et E donc les entrées de M relatives aux paires (A, C) et (A, E) sont passées à `True` (même si la dernière l'était déjà) :

	A	B	C	D	E
A	1	1	1	0	1
B	0	1	1	0	1
C	0	0	1	1	0
D	0	0	0	1	0
E	0	0	0	0	1

Lorsque $k = 2$. Le seul arc arrivant dans le sommet C vient du sommet B et C ne mène qu'au sommet D. Les sommets depuis lesquels C est accessible sont A et B, dès lors les entrées de M relatives aux paires (A, D) et (B, D) sont passées à `True` :

	A	B	C	D	E
A	1	1	1	1	1
B	0	1	1	1	1
C	0	0	1	1	0
D	0	0	0	1	0
E	0	0	0	0	1

Lorsque $k = 3$. Le sommet D ne mène nulle part donc la matrice M n'est pas mise à jour.

Lorsque $k = 4$. Le sommet E ne mène nulle part donc la matrice M n'est pas mise à jour.

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

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

Résolution. Calculons les puissances de M :

— M^0 est la matrice identité I_5.

— $M^1 = M$ a été trouvée au point 1 :

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	1
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	0	0	0

— M^2 est donnée par :

	A	B	C	D	E
A	0	0	1	0	1
B	0	0	0	1	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

— M^3 est donnée par :

	A	B	C	D	E
A	0	0	0	1	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

— M^4 est la matrice nulle $\mathbf{0}_5$.

— $M^5 = M^4 M = \mathbf{0}_5 M = \mathbf{0}_5$.

La fermeture transitive M^* est donc :

	A	B	C	D	E
A	1	1	1	1	1
B	0	1	1	1	1
C	0	0	1	1	0
D	0	0	0	1	0
E	0	0	0	0	1

Exercice 14.4. Comparez les résultats des trois questions précédentes.

Résolution. La matrice M trouvée par l'algorithme de Floyd-Warshall correspond bien à la fermeture transitive du graphe G. De plus les entrées qui valent 1 dans ces matrices sont :

— $(A, A), (A, B), (A, C), (A, D), (A, E)$;

— $(B, B), (B, C), (B, D), (B, E)$;

— $(C, C), (C, D)$;

— (D, D) ;
— (E, E).

Ce sont bien les paires de sommets joignables trouvées dans l'exercice 14.1.

Nous pouvons même montrer rigoureusement que ce résultat reste vrai, peu importe le graphe de départ sur lequel on travaille (les preuves qui suivent sont à titre informatif).

Proposition 2.40. *Soit $A \in \mathbb{B}^{n\times n}$ la matrice d'adjacence d'un graphe G vue comme matrice booléenne. $A^k_{ij} = \top$ si et seulement si il existe un chemin P dans G de longueur k liant v_i et v_j et $A^k_{ij} = \bot$ sinon.*

Démonstration. Montrons cela par récurrence sur k. Comme cas de base, prenons $k = 0$: dans ce cas-là, $A^k = I$, la matrice identité. Dès lors $A^0_{ij} = \top$ ssi $i = j$ et en effet les seuls chemins de longueur 0 dans un graphe sont les chemins d'un sommet vers lui-même.

Par récurrence, montrons la double implication séparément : supposons qu'il existe un chemin

$$P = (v_{i_1}, v_{i_2}, \dots, v_{i_{k-1}}, v_{i_k})$$

de longueur k dans G tel que $i_1 = i$ et $i_k = j$. En particulier $\overline{P} = (v_{i_1}, \dots, v_{i_{k-1}})$ est un chemin dans G depuis v_i vers $v_{i_{k-1}}$ de longueur $k-1$. De plus, nous savons que $A_{i_{k-1}j} = \top$. Par hypothèse de récurrence, $A^{k-1}_{ii_{k-1}} = \top$. Dès lors :

$$\begin{aligned} A^k_{ij} = \bigvee_{\ell=1}^{n} \left(A^{k-1}_{i\ell} \wedge A_{\ell j}\right) &= \left(\bigvee_{\substack{\ell=1 \\ \ell\neq i_{k-1}}}^{n} A^{k-1}_{i\ell} \wedge A_{\ell j} \right) \vee \left(A^{k-1}_{ii_{k-1}} \wedge A_{i_{k-1}j}\right) \\ &= \left(\bigvee_{\substack{\ell=1 \\ \ell\neq i_{k-1}}}^{n} A^{k-1}_{i\ell} \wedge A_{\ell j} \right) \vee (\top \wedge \top) \\ &= \top. \end{aligned}$$

Supposons maintenant que $A^k_{ij} = \top$. Par l'égalité suivante :

$$A^k_{ij} = \bigvee_{\ell=1}^{n} \left(A^{k-1}_{i\ell} \wedge A_{\ell j}\right),$$

nous savons qu'il existe un $\ell \in [\![1, n]\!]$ tel que $A^{k-1}_{i\ell} = \top$ et tel que $A_{\ell j} = \top$. Par hypothèse de récurrence, il existe un chemin $\widetilde{P} = (v_{i_1}, \dots, v_{i_{k-1}})$ dans G, de longueur $k-1$ tel que $i_1 = i$ et tel que $i_{k-1} = \ell$. Or, puisque $A_{\ell j} = \top$, nous savons que $(v_\ell, v_j) \in E(G)$. Nous pouvons en déduire que $P = (v_{i_1}, \dots, v_{i_{k-1}}, v_j)$ est un chemin de G, de longueur k liant v_i à v_j. □

Lemme 2.41. *Soit $G = (V, E)$ un graphe fini. Si il existe deux sommets distincts $u, v \in V$ tels que v est accessible depuis u, alors il existe un chemin de longueur au plus n liant u à v, où $n = |V|$ est le nombre de sommets de G.*

Démonstration. Puisque v est accessible depuis u, nous savons qu'il existe un chemin :

$$P = (u, v_{i_1}, v_{i_2}, \dots, v_{i_{k-1}}, v_{i_k}, v)$$

de G de longueur minimale. Supposons par l'absurde que $K = k + 2 = |P|$ (la longueur du chemin) est strictement supérieure à n. Puisque le chemin est de longueur minimale, nous savons que $u \neq v_j$ pour tout $j \in [\![1, k]\!]$ et $v \neq v_j$ pour tout $j \in [\![1, n]\!]$.

Cependant, $K > n$ implique qu'il y a nécessairement deux indices distincts j et ℓ entre 1 et k tels que $v_{i_j} = v_{i_\ell}$. Supposons sans perte de généralité que $j < \ell$. Alors :

$$P' = (u, v_{i_1}, v_{i_2}, \dots, v_{i_{j-1}}, v_{i_j}, v_{i_{\ell+1}}, \dots, v)$$

est un chemin liant u et v de longueur $K - (\ell - j) \lneq K$, ce qui contredit l'hypothèse que P est un chemin de longueur minimale. Nous en déduisons que K doit obligatoirement être $\leq n$. □

Théorème 2.42 ((plus connu ici sous le nom de *Théorème de Maréchal*)). *Soit $G = (V, E)$ un graphe à n sommets et soit $A \in \mathbb{B}^{n \times n}$ sa matrice d'adjacence vue comme matrice booléenne. Si M est la matrice d'accessibilité du graphe G et si A^* désigne la fermeture transitive de A, alors $A^* = M$.*

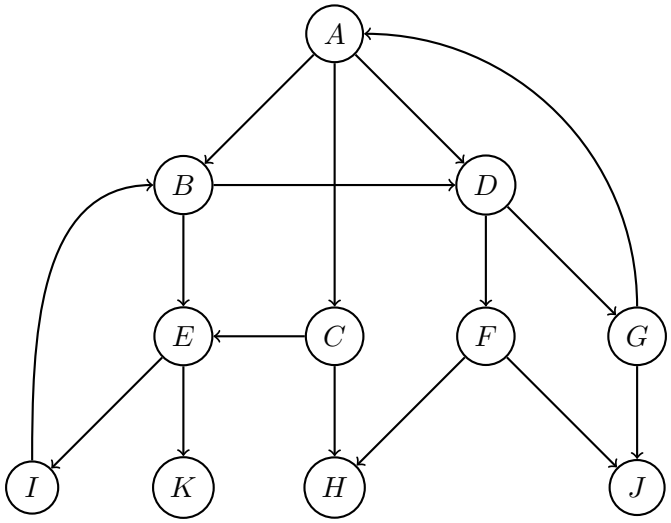
Démonstration. Montrons que pour toute paire $(i, j) \in [\![1, n]\!]^2$, nous avons bien $A^*_{ij} = \top \iff M_{ij} = \top$. Soit (i, j) une telle paire. Supposons que $A^*_{ij} = \top$. Par définition de la fermeture transitive, il existe un $k \in [\![0, n]\!]$ tel que $A^k_{ij} = \top$. Par la proposition 2.40, nous savons qu'il existe un chemin de longueur k liant v_i à v_j. Dès lors le sommet v_j est accessible depuis le sommet v_i et donc $M_{ij} = \top$.

Maintenant supposons que $M_{ij} = \top$. Par définition de la matrice d'accessibilité, nous savons qu'il existe un chemin liant v_i à v_j. Par le lemme 2.41, nous savons qu'il existe un chemin liant v_i à v_j de longueur $k \leq n$. Dès lors, par la proposition 2.40, nous savons que $A^k_{ij} = \top$. Par définition de la fermeture transitive, cela implique $A^*_{ij} = \top$.

Nous avons donc montré que $A^*_{ij} = \top \iff M_{ij} = \top$, i.e. $A^* = M$. □

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`.

Résolution. Un DFS se gère avec un stack dans lequel sont ajoutés tous les fils du nœud actuel. Les sommets sont traités dans l'ordre des `pop` du stack :

```python
def DFS(G, v):
    stack = [v]
    flags = [False] * G.n
    flags[v] = True
    while len(stack) > 0:
        v = stack.pop()
        # traitement de v
        for neighbour in G.neighbours(v):
            if not flags[neighbour]:
                flags[neighbour] = True
                stack.append(neighbour)
```

Partons donc d'un stack contenant uniquement A et regardons itérativement comment ce dernier est modifié pour un DFS partant de A :

- On pop `A` du stack. A est parent des sommets B, C, D, le stack vaut donc : `B C D`.
- On pop `D` du stack. D est parent des sommets F et G, le stack vaut donc : `B C F G`.
- On pop `G` du stack. G est parent des sommets A et J, mais A est déjà marqué donc n'est pas ajouté au stack qui devient donc : `B C F J`.
- On pop `J` du stack. J n'a pas de fils donc le stack est :

`B C F`.
— On pop `F` du stack. F est parent des sommets J et H, mais J est déjà marqué donc n'est pas ajouté au stack qui devient :
`B C H`.
— On pop `H` du stack. H n'a pas de fils donc le stack est :
`B C`.
— On pop `C` du stack. C est parent des sommets E et H, mais H est déjà marqué donc n'est pas ajouté au stack qui devient :
`B E`.
— On pop `E` du stack. E est parent des sommets I et K, le stack vaut donc :
`B I K`.
— On pop `K` du stack. K n'a pas de fils donc le stack est :
`B I`.
— On pop `I` du stack. I est parent du sommet B qui est déjà marqué donc le stack est :
`B`.
— On pop `B` du stack. B est parent des sommets D et E qui sont tous les deux déjà marqués. Le stack est maintenant vide et le parcours se termine.

La séquence des sommets parcourus est donc : `A D G J F H C E K I B`.

Proposition 2.43. *Un parcours en profondeur d'un graphe G à n sommets et m arêtes représenté de manière dynamique nécessite $\mathcal{O}(m+n) = \mathcal{O}(n^2)$ opérations.*

Démonstration. Lors du parcours, chaque sommet atteignable depuis le sommet initial est visité et traité au plus une fois, et chaque arête est prise au plus une fois (et n'est potentiellement revisitée qu'une deuxième fois). On peut alors en déduire que le parcours demande $\mathcal{O}(m+n)$ opérations. De plus, puisque $m = \mathcal{O}(n^2)$, nous savons que $\mathcal{O}(m+n) = \mathcal{O}(n^2)$.

Si par contre le graphe est représenté de manière statique par une matrice d'adjacence, alors toutes les *potentielles* arêtes partant de chaque sommet traité doivent être considérées, et il y a $\sim n$ telles arêtes. □

Remarque. *Cette borne n'est pas nécessairement atteinte. En effet, si nous faisons un parcours dans un graphe depuis un sommet isolé (i.e. un sommet $v \in V$ tel que $\deg v = 0$), alors le nombre d'opérations est $\mathcal{O}(1)$. Par contre si G est un graphe non dirigé connexe, alors tous les sommets seront visités et donc toutes les arêtes également.*

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`.

Résolution. Un BFS se gère avec une *queue* dans laquelle sont ajoutés tous les fils du nœud actuel. Les sommets sont traités dans l'ordre des `pop` de la queue.

```
1 def BFS(G, v):
2     queue = [v]
3     flags = [False] * G.n
4     flags[v] = True
5     while len(queue) > 0:
```

```
6           v = queue.pop(0)
7           # traitement de v
8           for neighbour in G.neighbours(v):
9               if not flags[neighbour]:
10                  flags[neighbour] = True
11                  queue.append(neighbour)
```

Partons donc d'une `queue` contenant uniquement A et regardons itérativement comment cette dernière est modifié pour un BFS partant de A :

- On pop `A` de la queue. A est parent des sommets B, C, D, la queue est donc : `B C D`.
- On pop `B` de la queue. B est parent des sommets D et E, mais D est déjà marqué donc n'est pas ajouté à la queue qui devient : `C D E`.
- On pop `C` de la queue. C est parent des sommets E et H, mais E est déjà marqué donc n'est pas ajouté à la queue qui devient : `D E H`.
- On pop `D` de la queue. D est parent des sommets F et G, la queue est donc : `E H F G`.
- On pop `E` de la queue. E est parent des sommets I et K, la queue est donc : `H F G I K`.
- On pop `H` de la queue. H n'a pas de fils donc la queue est : `F G I K`.
- On pop `F` de la queue. F est parent des sommets H et J, mais H est déjà marqué donc n'est pas ajouté à la queue qui devient : `G I K J`.
- On pop `G` de la queue. G est parent des sommets A et J qui sont tous les deux déjà marqués. La queue est donc : `I K J`.
- On pop `I` de la queue. I est parent de B qui est déjà marqué donc la queue est : `K J`.
- On pop `K` de la queue. K n'a pas de fils donc la queue est : `J`.
- On pop `J` de la queue. J n'a pas de fils. La queue est maintenant vide et le parcours se termine.

La séquence des sommets parcourus est donc : `A B C D E H F G I K J`.

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`.

Résolution. Procédons de la même manière mais partons de I au lieu de A :

- On pop `I` de la queue. I est parent de B donc la queue est : `B`.
- On pop `B` de la queue. B est parent des sommets D et E, la queue est donc : `D E`.
- On pop `D` de la queue. D est parent des sommets F et G, la queue est donc : `E F G`.
- On pop `E` de la queue. E est parent de I et K, mais I est déjà marqué donc n'est

pas ajouté à la queue qui devient :
`F G K`.

— On pop `F` de la queue. F est parent de H et J, la queue est donc :
`G K H J`.
— On pop `G` de la queue. G est parent de A et J, mais J est déjà marqué donc n'est pas ajouté à la queue qui devient :
`K H J A`.
— On pop `K` de la queue. K n'a pas de fils donc la queue est :
`H J A`.
— On pop `H` de la queue. H n'a pas de fils donc la queue est :
`J A`.
— On pop `J` de la queue. J n'a pas de fils donc la queue est :
`A`.
— On pop `A` de la queue. A est parent des sommets B, C, D, mais B et D sont déjà marqués donc ne sont pas ajoutés à la queue qui devient :
`C`.
— On pop `C` de la queue. C est parent des sommets E et H qui sont tous deux déjà marqués. La queue est maintenant vide et le parcours s'arrête.

La séquence des sommets parcourus est donc : `I B D E F G K H J A C`.

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 ?

Résolution. Nous pouvons généraliser la solution des exercices 6.2 et 6.3 en utilisant la matrice d'adjacence pour déterminer les mouvements possibles à la place d'un ensemble de mouvements prédéfinis (i.e. haut, gauche, bas, droite). Modifions la classe `StaticGraph` de l'exercice 13.5 pour représenter un graphe non-dirigé et non-pondéré :

```
class Graph: # graphe non-dirigé et non pondéré
    def __init__(self, n):
        self.adj = [[False]*n for i in range(n)]

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

    def link(self, i, j):
        self.adj[j][i] = self.adj[i][j] = True

    def are_linked(self, i, j):
        return self.adj[i][j]

    def __str__(self):
        ret = f'vertices: 0-{self.n-1}, edges: '
        ret += '{ '
        for i in range(self.n):
            for j in range(i+1, self.n):
                if self.are_linked(i, j):
                    ret += '{' + f'{i}, {j}' + '} '
        ret += ' }'
        return ret
```

Il faut maintenant écrire la méthode `shortest_path(self, u, v)` qui va appeler la méthode récursive (de backtracking) `_shortest_path`. Afin de trouver le chemin *le plus court* entre deux sommets, il nous faut maintenir une liste contenant le chemin actuel et une liste contenant le chemin le plus court trouvé jusqu'à présent (ainsi que sa longueur). Il faut également marquer les sommets déjà visités afin de ne pas finir dans une boucle infinie d'appels récursifs :

```
    def shortest_path(self, u, v):
        self.current_path = []
        self.best_path = []
```

```
4         self.marks = [False] * self.n
5         self.shortest_length = +float('inf')
6         self._shortest_path(u, v)
7         return len(self.best_path)-1, self.best_path
```

La méthode `_shortest_path` ressemble fort à la résolution de l'exercice 6.3 :

```
1     def _shortest_path(self, u, v):
2         self.current_path.append(u)
3         self.marks[u] = True
4         if len(self.current_path) < self.shortest_length:
5             if u == v:
6                 self.best_path = self.current_path.copy()
7                 self.shortest_length = len(self.best_path)
8             else:
9                 for k in range(self.n):
10                    if self.adj[u][k] and not self.marks[k]:
11                        self._shortest_path(k, v)
12        self.marks[u] = False
13        self.current_path.pop()
```

Si `u == v`, alors un chemin est trouvé et il faut vérifier s'il est plus court que le plus court chemin trouvé jusqu'à présent. Si par contre `u != v`, alors il faut continuer à chercher. Pour cela on regarde tous les sommets voisins de `u` qui ne sont pas encore marqués, et on appelle récursivement la fonction `_shortest_path` sur ces sommets. La construction/destruction de solution partielle est faite ici au début et à la fin de la fonction.

Notez bien que la première condition (l. 4) englobe tout le reste car si le chemin actuel est plus grand que le plus court chemin trouvé jusqu'à présent, alors il est inutile de continuer d'avancer dans la recherche de ce chemin actuel.

Un backtracking sur les sommets d'un graphe correspond à (mais n'est pas exactement) un parcours en profondeur puisque les enfants d'un sommet sont traités avant ses frères. Un parcours en largeur aurait été ici plus adapté car lors d'un BFS partant d'un sommet u, au moment de traiter un sommet v, tous les chemins partant de u et de longueur strictement inférieure à la longueur du chemin (u, v) ont déjà été parcourus. Dès lors, si un BFS trouve un chemin partant de u et allant en v qui a une longueur ℓ, alors ce chemin est de longueur minimale.

Une telle recherche par BFS peut être implémentée comme suit :

```
1     def shortest_path_length_bfs(self, u, v):
2         queue = deque()
3         queue.appendleft((0, u))
4         self.marks = [False] * self.n
5         self.marks[u] = True
6         while len(queue) > 0:
7             d, u = queue.pop()
8             if u == v:
9                 return d
10            for k in range(self.n):
```

```
                if self.adj[u][k] and not self.marks[k]:
                    queue.appendleft((d+1, k))
                    self.marks[k] = True
        return -1
```

Nous maintenons une `queue` pour faire ce BFS, mais au lieu d'y stocker uniquement les sommets enfants à visiter, il faut également y stocker la distance entre le sommet de départ et le sommet ajouté dans la queue. Dès que le BFS arrive sur le sommet `v`, c'est qu'un chemin (le plus court) a été trouvé, et le premier élément du tuple stocké dans la queue donne la distance entre `u` (le sommet de départ) et `v` (le sommet d'arrivée).

Notez qu'il est très important de bien marquer les sommets visités afin de ne pas finir dans une boucle infinie, tout comme pour la solution précédente. La liste `marks` est ici définie comme attribut pour ressembler à la solution précédente mais aurait pu être simplement une variable locale de la méthode `shortest_path_length_bfs`.

Cependant cette méthode ne donne que la longueur du chemin, et pas le chemin en lui-même. Afin de le retrouver, nous pouvons simplement maintenir une liste `parents` telle que `parents[j] == k` lorsque le sommet `k` est visité en tant que fils du sommet `j` :

```
    def shortest_path_bfs(self, u, v):
        start = u
        queue = deque()
        queue.appendleft((0, u))
        self.marks = [False] * self.n
        self.marks[u] = True
        self.parents = [None] * self.n
        while len(queue) > 0:
            d, u = queue.pop()
            if u == v:
                return d, self.extract_path(start, v)
            for k in range(self.n):
                if self.adj[u][k] and not self.marks[k]:
                    queue.appendleft((d+1, k))
                    self.marks[k] = True
                    self.parents[k] = u
        return -1, []
```

Finalement, retrouver le chemin se fait par la méthode `extract_path` qui *remonte* les liens de parenté jusqu'à tomber sur le sommet de départ `u` :

```
    def extract_path(self, u, v):
        path = [v]
        while v != u:
            v = self.parents[v]
            path.append(v)
        return path[::-1]
```

Il faut tout de même noter deux choses : premièrement dans la méthode `shortest_path_bfs`, la variable `u` est écrasée pendant le parcours en largeur. Il faut donc bien penser à la sauve-

garder dans une autre variable (ici `start`) pour pouvoir appeler la méthode `extract_path` sur les bons sommets (ou alors il faut utiliser une autre variable que `u` pour faire le parcours). Deuxièmement, la méthode `extract_path` visite les sommets depuis `v` jusqu'à `u`. Le chemin trouvé est donc inversé, il faut donc bien penser à retourner la liste `path` avant de la retourner, ce qui peut se faire soit via `path.reverse()` avant le `return`, soit via `path[::-1]` comme proposé dans la solution. Il est également possible de faire des insertions en début de liste avec `path.insert(0, v)` au lieu de `path.append(v)`.

Proposition 2.44. *La fonction* `_shortest_path` *trouve le plus court chemin entre u et v en $\mathcal{O}(n!)$ appels récursifs.*

Démonstration. Au pire, la fonction doit parcourir tous les chemins du graphe. Il y en a au plus $n!$. □

Nous pouvons tout de même faire mieux en terme de borne supérieure :

Proposition 2.45. *Notons Δ est le degré maximum du graphe. La fonction* `_shortest_path` *trouve le plus court chemin entre u et v en $\mathcal{O}(\Delta^n)$ appels récursifs.*

Démonstration. Puisque pour tout $v \in V$, nous savons que $\deg v \leq \Delta$, nous savons que dans la boucle `for` (l. 9), au plus Δ valeurs de `k` vont satisfaire la condition et mener à un appel récursif. De plus, à cause de la condition sur le vecteur `marks`, nous savons qu'une séquence d'appels récursifs ne peut être de longueur supérieure à n. Nous en déduisons que le nombre d'appels récursifs est borné par Δ^n. □

Remarque. *Dans tous les cas, la borne que l'on a sur le nombre d'appels récursifs de* `_shortest_path` *très grande, mais Δ^n peut être beaucoup plus petit que $n!$ si $\Delta \ll n$. En particulier si Δ ne dépend pas de n (prenons par exemple un* grid graph*, i.e. un graphe dont les sommets correspondent à des coordonnées de $\mathbb{N} \times \mathbb{N}$ et deux sommets (x_1, y_1) et (x_2, y_2) sont reliés par une arête lorsque $x_1 = x_2$ et $|y_1 - y_2| = 1$ ou lorsque $|x_1 - x_2| = 1$ et $y_1 = y_2$), alors lorsque n devient grand, $\Delta = \mathcal{O}(1)$ devient négligeable devant n. Δ^n est donc beaucoup plus petit que $n!$.*

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`.

Résolution. Commençons donc par écrire une classe `Vertex` représentant un sommet du graphe et contenant les éléments demandés :

```
class Vertex:
    def __init__(self, name, score, flag=False):
        self.name = name
        self.score = score
        self.flag = flag
        self.neighbours = []
```

Nous n'allons pas mettre de getters/setters ou propriétés dans cette classe afin de garder le code le plus concis possible.

Écrivons maintenant la classe `Graph` demandée (voir la classe `DynamicGraph` de l'exercice 13.5) :

```
class Graph:
    def __init__(self, adj_matrix, vertices):
        self.vertices = vertices
        for i in range(len(adj_matrix)):
            for j in range(i):
                if adj_matrix[i][j]:
                    self.link(i, j)

    def link(self, i, j):
        self.vertices[i].neighbours.append(self.vertices[j])
        self.vertices[j].neighbours.append(self.vertices[i])
```

La méthode `ranking` peut s'écrire comme ceci :

```
    def ranking(self, depth, node):
        candidates = self.k_distance_nodes(depth, node)
        candidates.sort(reverse=True, key=lambda x: x.score)
        print("======= RANKING =======")
        for i, player in enumerate(candidates):
            print("{}. {}: {}".format(i+1, player.name, player.score))
```

où la méthode `k_distance_nodes(k, v)` renvoie une liste contenant tous les sommets à distance `<= k` du sommet `v`. Cette liste est ensuite triée par ordre décroissant afin de faire un classement. Le paramètre `reverse` permet de choisir si le tri est fait par ordre croissant ou par ordre décroissant, et le paramètre `key` est une fonction qui permet de déterminer selon quel critère les éléments de la liste sont triés.

`k_distance_nodes` peut être implémentée via un DFS ou un BFS :

```
    #DFS
    def k_distance_nodes_dfs(self, k, v):
        ret = []
        if not v.flag:
            ret = [v]
```

```
            v.flag = True
        if k == 0:
            return ret
        for neighbour in v.neighbours:
            ret += self.k_distance_nodes_dfs(k-1, neighbour)
        return ret

    #BFS
    def k_distance_nodes_bfs(self, k, v):
        d = 0
        ret = [v]
        v.flag = True
        q = deque()
        while d < k:
            for neighbour in v.neighbours:
                if neighbour.flag:
                    continue
                neighbour.flag = True
                q.appendleft((d+1, neighbour))
                ret.append(neighbour)
            d, v = q.pop()
        return ret
```

où deque est une classe standard de Python permettant d'implémenter une queue telle que les opérations d'insertion et de suppression sont en $\mathcal{O}(1)$.

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.

Résolution.

1. Le n-chemin P_n (i.e. le graphe dont les sommets sont $\{1, \dots, n\}$ et dont les arêtes sont les paires non ordonnées $\{k, k+1\}$ pour $1 \leq k \lneqq n$) est un graphe connexe à $n - 1 = \Theta(n)$ sommets. Le n-cycle C_n (qui est le n-chemin auquel on a ajouté l'arête $\{1, n\}$) est également un graphe connexe à $n = \Theta(n)$ arêtes.
2. Considérons le graphe (notons-le G_n) sur $\{1, \dots, n\}$ dans lequel chaque sommet m est lié à toutes les valeurs $\left\lfloor \frac{m}{2^k} \right\rfloor$ pour $1 \leq k \leq \log_2 m$. Ce graphe est bien connexe car en particulier tous les sommets sont adjacents au sommet 1 (remarque : cela implique que le diamètre du graphe est ≤ 2) et le nombre d'arêtes de G_n est (par la proposition 1.26) :

$$|E(G_n)| \sim \sum_{m=1}^{n} \log_2 m = \log n! = \Theta(n \log n).$$

Nous pouvons également définir le graphe G_n de graphes $\{1, \dots, n\}$ et dont les arêtes sont les paires non ordonnées $\{i, j\}$ où i divise j. Il est clair que ce graphe est connexe puisque tous sommets v et w peuvent joints l'un à l'autre en passant par le sommet 1. L'ensemble des arêtes peut s'exprimer comme :

$$E(G_n) = \bigsqcup_{i=1}^{n} \{\{i, ki\} \text{ s.t. } k > 1 \text{ et } ki \leq n\} = \bigsqcup_{i=1}^{n} \left\{\{i, ki\} \text{ s.t. } 1 \leq k \leq \left\lfloor \frac{n}{i} \right\rfloor\right\},$$

et donc :

$$|E(G_n)| = \sum_{i=1}^{n} \left(\left\lfloor \frac{n}{i} \right\rfloor - 1 \right) = \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor - n = \sum_{i=2}^{n} \left\lfloor \frac{n}{i} \right\rfloor.$$

Or $\left\lfloor \frac{n}{i} \right\rfloor \sim \frac{n}{i}$ et donc par le corollaire 1.37 :

$$|E(G_n)| \sim \sum_{i=2}^{n} \frac{n}{i} = n(H_n - 1) \sim nH_n \sim n \log n.$$

3. Construisons le graphe G_n sur $\{1, \dots, n\}$ dont les arêtes sont les paires non ordonnées $\{i, j\}$ où $i > j$ et j est un carré parfait. Il est clair que le graphe est connexe puisque tous les sommets (hormis 1) sont liés à 1. Les arêtes sont donc données par :

$$E(G_n) = \bigsqcup_{i=1}^{n} \mathcal{N}(i),$$

où $\mathcal{N}(i)$ est défini par :

$$\mathcal{N}(i) = \left\{j < i \text{ s.t. } j = k^2 \text{ pour un certain } k \in \mathbb{N}_0\right\} = \left\{k^2 \text{ s.t. } 0 < k \leq \left\lceil \sqrt{i-1} \right\rceil \right\}.$$

Dès lors :

$$|E(G_n)| = \sum_{i=1}^{n} |\mathcal{N}(i)| = \sum_{i=1}^{n} \left\lceil \sqrt{i-1} \right\rceil = \sum_{i=0}^{n-1} \left\lceil \sqrt{i} \right\rceil \sim \frac{2}{3} n^{3/2},$$

par le corollaire 1.37 (puisque $\lceil \sqrt{x} \rceil \sim \sqrt{x}$).

4. Le graphe complet K_n sur $\{1, \ldots, n\}$ a exactement $\binom{n}{2} = \Theta(n^2)$ arêtes.

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.

Résolution. À $u, v \in V(T)$ fixés, par connexité, il existe bien un chemin P joignant u et v. Supposons par l'absurde qu'il existe un chemin P' joignant u et v tel que $P \neq P'$. WLOG supposons que P et P' sont minimaux. En particulier $|P| = |P'|$ et donc il existe deux sommets $w, w' \in V(T)$ tel que $v_i \in P$ mais $v_i \notin P'$ et $v_j \in P'$ mais $v_j \notin P$.

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

Résolution. Soit $C = \{v_1, \ldots, v_k\}$ un k-cycle de G (i.e. les v_i sont distincts deux à deux et $v_i \sim v_{i+1}$ dans G ainsi que $v_1 \sim v_k$). Fixons $e = \{v_1, v_k\} \in C$ une arête du cycle et prenons maintenant deux sommets $u, w \in V(G)$ quelconques. Distinguons les trois cas suivants :

(i) u et w sont tous les deux C ;
(ii) u est dans C mais w est dans $V(G) \setminus C$;
(iii) ni u ni w n'est dans C.

Remarquons que le cas $u \in V(G) \setminus C$ et $w \in C$ se déduit du cas (ii) par symétrie. Montrons que dans chacun des trois cas ci-dessus, il existe un chemin joignant u et w dans $G - e$, i.e. dans l'unique graphe G' tel que $V(G') = V(G)$ et $E(G') = E(G) \setminus \{e\}$.

(i) Nous savons que $u = v_i$ et $w = v_j$ pour un certain i et un certain j distincts et tous deux entre 1 et k. Si $i < j$, alors

$$P = \{v_i, v_{i+1}, \ldots, v_{j-1}, v_j\}$$

est un chemin entre u et w dans $G - e$; et si $i > j$, alors

$$P' = (v_j, v_{j+1}, \ldots, v_{i-1}, v_i)$$

en est un.

(ii) Prenons $P_{u,w}$ un chemin minimal entre u et w dans G. Si $e \in P_{u,w}$, alors il existe un certain $1 \leq i \leq k$ et un certain $v' \in V(G) \setminus C$ tel que $\{v', v_i\} \in P_{u,w}$ et tel que la partie $P_{u,v'}$ de $P_{u,w}$ ne passe pas par C. Par le point (i), nous savons qu'il existe un chemin P entre v_i et w ne passant pas par e ; nous pouvons donc joindre $P_{u,v'}$ et P' afin d'obtenir un chemin entre e et w dans $G - e$.

(iii) Prenons un chemin $P_{u,w}$ entre u et v dans G. Si $e \in P_{u,w}$, comme dans le point (ii), nous pouvons trouver $i \neq j$ qui découpent $P_{u,w}$ en P_{u,v_i}, P_{v_i,v_j} et $P_{v_j,w}$ tels que P_{u,v_i} et $P_{v_j,w}$ n'ont aucune arête du cycle C. Par le point (i), il existe un $P_{i,j}$ joignant v_i à v_j sans passer par e. Le chemin $P_{u,v_i} + P_{i,j} + P_{v_j,w}$ est donc un chemin entre u et w dans $G - e$.

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

Résolution. Soient $u, v \in V(G)$ deux sommets. Si u et v sont dans des composantes connexes différentes de G, alors en particulier $\{u, v\} \notin E(G)$, et donc $\{u, v\} \in E(G^{\complement})$. Si par contre u et v sont dans la même composante connexe de G, il est possible que $\{u, v\} \notin E(G^{\complement})$ (par exemple si G est une union de graphes complets). Cependant par hypothèse, il existe au moins une autre composante connexe non-vide. Prenons donc un sommet y qui n'est pas connexe à u ou v. Par le point précédent, nous savons que $\{u, y\}$ et $\{v, y\}$ sont des arêtes de $G^{\complement}$. Dès lors (u, y, v) est un chemin allant de u vers v dans $G^{\complement}$. En particulier pour toute paire de sommets u, v, nous savons que $d_{G^{\complement}}(u, v) \leq 2$.

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.

Résolution.

1. Nous cherchons un graphe G connexe à m arêtes mais tel que la suppression de toute arête sépare le graphe en composantes connexes. Il faut donc que G soit acyclique par l'exercice 16.5, en particulier G est un arbre (puisque non-dirigé connexe et acyclique). Montrons maintenant par récurrence (sur n) qu'un arbre à n sommets a exactement $n - 1$ arêtes.

   Prenons le cas $n = 1$ comme cas de base : le graphe $(\{v\}, \emptyset)$ est l'unique 1-arbre et a bien 0 arête pour 1 sommet.

   Par induction forte (donc supposons qu'à n fixé, la propriété est vraie pour tout $1 \leq k \lneq n$), prenons G un arbre à n sommets. Prenons e une de ses arêtes au hasard. Cette arête e sépare le graphe G en deux composantes connexes non vides (notons-les G_1 et G_2). En effet si deux sommets $u, v \in V(G)$ étaient joignables dans G sans emprunter e alors ils sont dans la même composante connexe de $G - e$; et si tout chemin (en réalité l'unique chemin puisque G est un arbre) entre u et w contient e, alors u et w ne sont plus joignables dans $G - e$. En notant v_1 et v_2 les sommets joints par e, $G - e$ est bien composé de G_1, la composante connexe de v_1 et de G_2, la composante connexe de v_2.

   Notons n_i et m_i le nombre de sommets et d'arêtes de G_i pour $i = 1, 2$. Par récurrence nous savons que $m_i = n_i - 1$, mais nous savons également que :

   $$V(G) = V(G_1) \sqcup V(G_2) \qquad \text{ainsi que} \qquad E(G - e) = E(G_1) \sqcup E(G_2).$$

   Dès lors, en prenant la cardinalité de part et d'autre, nous déterminons bien que $n = n_1 + n_2$ et $|E(G - e)| = m_1 + m_2 = n_1 + n_2 - 2 = n - 2$. En ajoutant l'arête e, on obtient bien G avec ses $n - 2 + 1 = n - 1$ arêtes.

2. Si G n'est pas connexe, par l'exercice 16.6 nous savons que $G^{\complement}$ est connexe. Dès lors :

$$|E(G)| = \frac{n(n-1)}{2} - \left|E(G^{\complement})\right| \leq \frac{n(n-1)}{2} - (n-1) = \frac{1}{2}(n-1)(n-2).$$

En particulier le seul graphe non connexe à n sommets et $\frac{1}{2}(n-1)(n-2)$ arêtes est l'union d'un K_{n-1} et d'un K_1.

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

Résolution. Prenons $u, v \in V(T)$ de distance maximale, i.e. tels que pour toute paire de sommets (x, y) : $d(u, v) \geq d(x, y)$ et prenons P l'unique chemin joignant u et v. Nous savons que $\deg u = 1$ car si on suppose par l'absurde qu'il existe au moins deux sommets $w_1, w_2 \in V(T)$ adjacents à u, alors au moins un de ces deux sommets n'est pas dans P. WLOG supposons que c'est w_1, dès lors $d(w_1, v) = d(u, v) + 1 > d(u, v)$, ce qui est impossible par choix de u et v. En raisonnant de la même manière sur v, nous savons que u et v sont des 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.

Résolution. Intuitivement, comme T est un arbre, il est acyclique donc chaque *branche* partant d'un sommet v de degré $\Delta(T)$ doit bien aboutir sur au moins une feuille, d'où la borne inférieure.

Plus formellement, pour v un sommet de degré $\Delta(T)$, si nous considérons le graphe (la forêt) $G = T - v$ (i.e. le graphe obtenu en partant de T et en retirant le sommet v et toutes les arêtes qui y sont incidentes), alors nous obtenons $\Delta(T)$ composantes connexes (puisqu'on peut obtenir T en retirant itérativement les arêtes de $E(v)$ ce qui augmente de 1 le nombre de composantes connexes à chaque fois, jusqu'à toutes les avoir retirées et donc à retirer v qui est sa propre composante connexe pour un total de $1 + \Delta(T) - 1 = \Delta(T)$ composantes). Puisque T est acyclique, chacune de ces composantes est également acyclique. Soit C_i une de ces composantes connexes. Si $|C_i| = 1$, alors C_i contient un unique sommet v_i, i.e. v_i est une feuille de T. Si par contre C_i contient au moins deux sommets, alors puisque C_i est un arbre, nous savons par l'exercice 16.8 que C_i admet au moins deux feuilles. En particulier il existe au moins une feuille qui n'est pas adjacente à v dans T.

Nous avons en particulier montré que pour chaque arête $\{v, v_i\}$ de T, il existe au moins une feuille $\ell_i \in V(T)$ telle que l'unique chemin entre ℓ_i et v passe par v_i. En particulier il doit y avoir au moins $\Delta(T)$ feuilles dans T.

Bonus : (i) l'étoile $S_n = (\{1, \ldots, n\}, \{\{1, i\} \text{ s.t. } 2 \leq i \leq n\})$ est un arbre à $n - 1$ feuilles tel que $\Delta(S_n) = \deg 1 = n - 1$. (ii) Prenons P_n le n-chemin auquel on ajoute un unique sommet v_i' pour chaque sommet v_i de degré $\deg v_i = 2$ et dans lequel on lie les v_i aux v_i'. Ce nouveau graphe est bien un arbre car aucun cycle n'a été introduit et tous les nouveaux sommets ont été ajoutés à la composante connexe initiale. Cet arbre a donc n feuilles pour $n + n - 2 = 2(n - 1) = \Theta(n)$ sommets.

Définition 18. Soient deux graphes dirigés simples finis G_1 et G_2 (les définitions qui suivent peuvent être étendues, mais concentrons-nous sur un cas simple). On définit :

- le *produit cartésien* entre G_1 et G_2 par le graphe $G_1 \square G_2$ tel que $V(G_1 \square G_2) = V(G_1) \times V(G_2)$ et $(u_1, u_2) \sim_\square (v_1, v_2)$ ssi soit (i) $u_1 = v_1$ et $u_2 \sim v_2$ dans G_2, soit (ii) $u_1 \sim v_1$ dans G_1 et $u_2 = v_2$;
- le *produit tensoriel* (ou *produit de Kronecker*) entre G_1 et G_2 par le graphe $G_1 \times G_2$ tel que
  $V(G_1 \times G_2) = V(G_1) \times V(G_2)$ et $(u_1, u_2) \sim_\times (v_1, v_2)$ ssi $u_1 \sim v_1$ dans G_1 et $u_2 \sim v_2$ dans G_2 ;
- le *produit fort* entre G_1 et G_2 par le graphe $G_1 \boxtimes G_2$ tel que $V(G_1 \boxtimes G_2) = V(G_1) \times V(G_2)$ et $(u_1, u_2) \sim_\boxtimes (v_1, v_2)$ ssi soit (i) $(u_1, u_2) \sim_\square (v_1, v_2)$, soit $(u_1, u_2) \sim_\times (v_1, v_2)$.

Exercice 16.10. 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.

Résolution.

Proposition 2.46. *Soient deux graphes G_1 et G_2. Si nous notons $n_i \coloneqq |V(G_i)|$ et $m_i \coloneqq |E(G_i)|$ pour $i \in \{1, 2\}$, alors les égalités suivantes sont toujours vérifiées :*

- $|V(G_1 \square G_2)| = |V(G_1 \times G_2)| = |V(G_1 \boxtimes G_2)| = n_1 n_2$;
- $|E(G_1 \square G_2)| = n_1 m_2 + n_2 m_1$;
- $|E(G_1 \times G_2)| = 2 m_1 m_2$;
- $|E(G_1 \boxtimes G_2)| = n_1 m_2 + 2 m_1 m_2 + n_2 m_1$.

Démonstration. Le premier point est immédiat par la propriété du produit cartésien entre deux ensembles : $|E \times F| = |E| \cdot |F|$.

Le second point peut se montrer par la décomposition suivante :

$$\begin{aligned} E(G_1 \square G_2) &= \{((u_1, v), (v_1, v)) \text{ s.t. } (u_1, v_1) \in E(G_1), v \in V(G_2)\} \\ &\quad \sqcup \{((v, u_2), (v, v_2)) \text{ s.t. } v \in V(G_1), (v_1, v_2) \in E(G_2)\} \\ &= \bigsqcup_{v \in V(G_2)} \{((u_1, v), (v_1, v)) \text{ s.t. } (u_1, v_1) \in E(G_1)\} \\ &\quad \sqcup \bigsqcup_{v \in V(G_1)} \{((v, u_2), (v, v_2)) \text{ s.t. } (v_1, v_2) \in E(G_2)\}. \end{aligned}$$

Or à v fixé dans $V(G_2)$, $\{((u_1, v), (v_1, v)) \text{ s.t. } (u_1, v_1) \in E(G_1)\}$ est en bijection avec $E(G_1)$, et à v fixé dans $V(G_1)$, $\{((v, u_2), (v, v_2)) \text{ s.t. } (v_1, v_2) \in E(G_2)\}$ est en bijection avec $E(G_2)$. Dès lors, en prenant la cardinalité de part et d'autre, nous obtenons :

$$|E(G_1 \square G_2)| = \sum_{v \in V(G_2)} |E(G_1)| + \sum_{v \in V(G_1)} |E(G_2)| = n_1 m_2 + n_2 m_1.$$

La troisième égalité découle directement du fait que :

$$E(G_1 \times G_2) = \bigsqcup_{(u_1, v_1) \in E(G_1)} \bigsqcup_{(u_2, v_2) \in E(G_2)} \{((u_1, u_2), (v_1, v_2)), ((v_1, u_2), (u_1, v_2))\}.$$

La dernière égalité vient tout simplement du fait que :

$$E(G_1 \boxtimes G_2) = E(G_1 \square G_2) \sqcup E(G_1 \times G_2).$$

□

Basons le code sur la classe `Graph` suivante (très simple) :

```
class Graph:
    def __init__(self, n):
        self.n = n
        self.m = 0
        self.edges = [[] for _ in range(n)]
        self.deg = [0 for _ in range(n)]

    def add_edge(self, u, v):
        if u > v: u, v = v, u
        self.edges[u].append(v)
        self.m += 1
        self.deg[u] += 1
        self.deg[v] += 1

    def are_connected(self, u, v):
        if u > v: u, v = v, u
        return v in self.edges[u]

    def degree(self, v):
        return self.deg[v]

    @property
    def V(self):
        return self.n

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

    def __iter__(self):
        for u, edges in enumerate(self.edges):
            for v in edges:
                yield (u, v)
```

Chacune de cecs trois fonctions va commencer par la création d'un graphe de taille $n_1 n_2$ et va contenir une fonction lambda permettant de mapper une paire `(u, v)` vers un identifiant de sommet dans le graphe produit. Les fonctions du produit cartésien et du produit de Kronecker peuvent donc s'écrire comme suit :

```
def cartesian_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:
        for u1 in range(G1.V):
            u1u2 = as_id(u1, u2)
            u1v2 = as_id(u1, v2)
            ret.add_edge(u1u2, u1v2)
    return ret

def kronecker_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, 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))
    return ret
```

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` :

```python
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
```

Résolution.

Lemme 2.47. *L'algorithme d'Euclide donne le* GCD *de deux nombres entiers.*

Démonstration. Si $b = 0$, alors il est clair que $a = \mathrm{GCD}(a,b)$ puisque a divise a et a divise 0 de manière triviale.

Fonctionnons maintenant par récurrence. Par division euclidienne, on sait qu'il existe $q, r \in \mathbb{N}$ tels que $a = qb + r$ (où $r < b$). Notons $d = \mathrm{GCD}(b,r)$. Observons que d divise a. En effet, puisqu'il il existe $q_b, q_r \in \mathbb{N}$ tels que $b = dq_b$ et $r = dq_r$, nous avons :

$$a = qb + r = qdq_b + dq_r = d\left(qq_b + q_r\right),$$

i.e. d divise a. Notons $d' = \mathrm{GCD}(a,b)$ et montrons alors que $d = d'$. Soient $p_a, p_b \in \mathbb{N}$ tels que $a = p_a d'$ et $b = p_b d'$. Dès lors :

$$r = a - qb = p_a d' - qp_b d' = d'\left(p_a - qp_b\right),$$

i.e. d' divise r. Or d' divise b par définition, donc $d' \leq d = \mathrm{GCD}(b,r)$. Cependant, puisque d divise a et b, on sait que $d \leq d' = \mathrm{GCD}(a,b)$. Nous pouvons alors en conclure que $d = d'$, i.e. $\mathrm{GCD}(a,b) = \mathrm{GCD}(b,r)$. □

Lemme 2.48. *L'algorithme d'Euclide étendu donne le* GCD *de deux nombres entiers ainsi que des entiers x, y tels que $xa + yb = \mathrm{GCD}(a,b)$.*

Remarque. *L'existence de ces nombres $x, y \in \mathbb{Z}$ est assurée par le théorème de Bachet-Bézout.*

Démonstration. Par le lemme précédent, nous savons que le l'algorithme d'Euclide étendu donne bien le GCD entre a et b. Montrons maintenant que les valeurs x, y calculées satisfont bien $xa + yb = \mathrm{GCD}(a, b)$. Procédons à nouveau par induction.

Cas de base : si $b = 0$, alors $1 \cdot a + 0 \cdot b = a = \mathrm{GCD}(a, b)$.

Pas de récurrence : soient $q, r \in \mathbb{N}$ tels que $a = qb + r$. Si nous avons x, y, g tels que $xb + yr = g = \mathrm{GCD}(b, r)$, alors pour $x' = y$ et $y' = x - qy$, alors $g = \mathrm{GCD}(a, b)$ par le lemme précédent et :

$$x'a + y'b = ya + (x - qy)b = ya + xb - yqb = xb + y(a - qb) = xb + yr = g.$$

□

Afin de dérécursifier l'algorithme d'Euclide, il faut se rendre compte que le paramètre a sert uniquement à calculer r, le reste de la division euclidienne de a par b. Donc une fois que r est calculé, a n'est plus nécessaire et b prend la place de b alors que r prend la place de b. Il suffit alors de répéter cet échange de variables tant que $b > 0$:

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

L'algorithme étendu doit en particulier calculer le GCD entre a et b donc il faut inclure le code de la fonction précédente. À cela il faut ajouter la relation qui permet de définir les valeurs x et y. Nous pouvons pour cela conserver un stack sur lequel nous ajoutons les valeurs de a et b correspondant aux différents appels récursifs virtuels :

```
def extended_gcd_naive(a, b):
    stack = []
    while b != 0:
        stack.append((a, b))
        a, b = b, a%b
    g = a
    x, y = 1, 0
    while len(stack) > 0:
        a, b = stack.pop()
        q = a//b
        x, y = y, x-q*y
    return x, y, g
```

Nous pouvons cependant faire mieux : en effet il est possible de ne pas maintenir de stack pour stocker les différentes valeurs de a et b et de calculer directement x et y. Cela nous permet d'avoir une complexité en espace de $\mathcal{O}(1)$ au lieu de $\mathcal{O}(\log b)$ (voir proposition 2.49).

Pour cela, définissons les suites suivantes :

$$\begin{cases} a_0 = a, a_{k+1} = b_k \\ b_0 = b, b_{k+1} = r_k \\ x_0 = 1, x_1 = 0, x_{k+1} = x_{k-1} - q_{k-1}x_k \\ y_0 = 0, y_1 = 1, y_{k+1} = y_{k-1} - q_{k-1}y_k, \end{cases}$$

où q_k et r_k satisfont $a_k = b_k q_k + r_k$ avec $0 \leq r_k \lneqq b_k$. Vérifions maintenant l'égalité suivante :

$$\forall k : x_k a_0 + y_k b_0 = a_k.$$

Par récurrence, si $k = 0$, alors il est trivial que :

$$x_0 a_0 + y_0 b_0 = 1 \cdot a_0 + 0 \cdot b_0 = a_0.$$

Pour $k = 1$ également :

$$x_1 a_0 + y_1 b_0 = b_0 = a_1.$$

Supposons maintenant que l'égalité est vraie pour tout $\ell \leq k$ et montrons-la pour k :

$$\begin{aligned} x_{k+1}a_0 + y_{k+1}b_0 &= (x_{k-1} - q_{k-1}x_k)a_0 + (y_{k-1} - q_{k-1}y_k)b_0 \\ &= x_{k-1}a_0 + y_{k-1}b_0 - q_{k-1}(x_k a_0 + y_k b_0) = a_{k-1} - q_{k-1}a_k \\ &= a_{k-1} - q_{k-1}b_{k-1} = r_{k-1} = b_k = a_{k+1} \end{aligned}$$

```python
def extended_gcd_opti(a, b):
    a0, b0 = a, b
    x = 0
    last_x = 1
    y = 1
    last_y = 0
    while b != 0:
        q, r = divmod(a, b) # a = q*b + r
        a, b = b, r
        x, last_x = last_x - q * x, x
        y, last_y = last_y - q * y, y
        assert last_x * a0 + last_y * b0 == a
    return last_x, last_y, a
```

La complexité de cet algorithme est donnée à titre *indicatif* : nous n'attendons pas de votre part que vous puissiez la déterminer à ce stade-ci.

Proposition 2.49. *Soient $a, b \in \mathbb{N}$ tels que $a \geq b$. L'algorithme d'Euclide détermine $\mathrm{GCD}(a, b)$ en temps $\mathcal{O}(\log b)$.*

La preuve est assez longue, donc nous ne la présenterons pas ici. Pour les intéressés, la preuve repose sur le fait que la séquence de nombres générés par l'algorithme d'Euclide arrive à b plus rapidement que la suite de Fibonacci. Utilisons l'équation suivante :

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \overline{\varphi}^n),$$

où $\varphi = \frac{1}{2}(1+\sqrt{5})$ est le nombre d'or et $\overline{\varphi} = 1 - \varphi$ et remarquons que :

$$|\overline{\varphi}^n| = |\overline{\varphi}|^n \leq |\overline{\varphi}|$$

car $|\overline{\varphi}| < 1$. Dès lors :

$$\left|\frac{\overline{\varphi}^n}{\sqrt{5}}\right| < \frac{1}{2}.$$

Nous pouvons en déduire que :

$$F_n > \frac{\varphi^n}{\sqrt{5}} - 1.$$

Dès lors, si $n > \log_\varphi\left(\sqrt{5}(b+1)\right)$, alors :

$$F_n > \frac{\varphi^{\log_\varphi(\sqrt{5}(b+1))}}{\sqrt{5}} - 1 = \frac{\sqrt{5}(b+1)}{\sqrt{5}} - 1 = b.$$

Nous en déduisons finalement qu'il y a au plus $\log_\varphi(\sqrt{5}(b+1)) = \mathcal{O}(\log b)$ nombres de la suite de Fibonacci qui sont inférieurs ou égaux à b, ce qui termine la preuve.

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

Résolution. Dans l'algorithme `Quicksort`, la fonction `partition` n'est pas récursive, donc nous pouvons la conserver :

```
def swap(vect, i, j):
    vect[i], vect[j] = vect[j], vect[i]

def partition(vect, beg, end):
    pivot = vect[end]
    i = beg
    for j in range(beg, end):
        if v[j] <= pivot:
            swap(v, i, j)
            i += 1
    swap(v, i, end)
    return i
```

La fonction `quicksort` doit quant à elle être dérécursifiée. Puisqu'il y a deux appels récursifs au sein de cette fonction, nous devons maintenir un stack qui va contenir les valeurs de `beg` et `end` des appels récursifs. Il faut alors progressivement récupérer les valeurs du stack et appeler `partition` sur base des valeurs récupérées. Après cela, il faut ajouter les bornes `(beg, pivot-1)` et `(pivot+1, end)` sur le stack (pour autant que ces bornes ne désignent pas une région vide) :

```
def quicksort_non_rec(vect):
    beg = 0
    end = len(vect) - 1
    stack = [(beg, end)]
    # Tant que le stack n'est pas vide, il reste des éléments à traiter
```

```
6       while len(stack) > 0:
7           beg, end = stack.pop()
8           pivot = partition(vect, beg, end)
9           # S'il y a des éléments à gauche du pivot, on les met dans le stack
10          if pivot - 1 > beg:
11              stack.append((beg, pivot - 1))
12          # S'il y a des éléments à droite du pivot, on les met dans le stack
13          if pivot + 1 < end:
14              stack.append((pivot + 1, end))
```

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

Résolution. Il est ici beaucoup plus compliqué d'avoir une forme *agréable* pour la version dérécursifiée de l'algorithme. Il nous faut alors utiliser le canevas général vu au cours. Puisque nous nous situons dans la situation avec deux appels récursifs (et avec un traitement intermédiaire) :

```
def P(x):
    if C(x):
        B(x)
    else:
        A0(x)
        P(f1(x))
        A1(x)
        P(f2(x))
        A2(x)
```

nous pouvons réécrire la fonction P comme suit :

```
def P(x):
    s = Stack()
    s.push((0, None)) # sentinelle
    while True:
        while not C(x):
            A0(x)
            s.push((1, x)) # 1er appel
            x = f1(x)
        B(x)
        (i, x) = s.pop()
        while i == 2:
            A2(x)
            (i, x) = s.pop()
        if i == 0:
            break
        A1(x)
        s.push((2,x)) # 2e appel
        x = f2(x)
```

Faisons le lien entre la version récursive et le canevas :

```
class State:
    def __init__(self, n, src, dest, aux):
        self.n = n
        self.src = src
        self.dest = dest
        self.aux = aux
def f1(x):
    return State(x.n-1, x.src, x.aux, x.dest)
def f2(x):
    return State(x.n-1, x.aux, x.dest, x.src)
def B(x):
    hanoi_moveOne(x.src, x.dest)
A1 = B

def hanoi_explicit(n=3, src='A', dest='C', aux='B'):
```

```
    x = State(n, src, dest, aux)
    stack = [(0, None)]
    while True:
        while x.n > 1:
            stack.append((1, x))
            x = f1(x)
        B(x)
        i, x = stack.pop()
        while i == 2:
            i, x = stack.pop()
        if i == 0:
            break
        A1(x)
        stack.append((2, x))
        x = f2(x)
```

On peut réécrire cela de manière moins explicite concernant la dérécursification mais de manière plus explicite concernant le problème à résoudre :

```
def hanoi_non_rec(n=3, src="A", dest="C", aux="B"):
    stack = [(0, None)]
    while True:
        while n > 1:                        # wile not C(x)
            x = (n, src, dest, aux)
            stack.append((1, x))
            n -= 1                          # x = f1(x)
            aux, dest = dest, aux           # ---
        hanoi_moveOne(src, dest)            # B(x)
        i, x = stack.pop()
        n, src, dest, aux = x
        while i == 2:
            i, x = stack.pop()
        if i == 0:
            break
        n, src, dest, aux = x
        hanoi_moveOne(src, dest)            # A1(x)
        x = (n, src, dest, aux)
        stack.append((2, x))
        n -= 1                              # x = f2(x)
        aux, src = src, aux                 # ---
```

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 ?

Résolution. L'implémentation récursive doit être triviale à ce stade-ci du cours :

```
1 def mccarthy_rec(n):
2     return n-10 if n > 100 else mccarthy_rec(mccarthy_rec(n+11))
```

Afin de dérécursifier cette approche, il est important de remarquer que la récursivité est binaire : un appel à la fonction `mccarthy_rec` peut soit s'exécuter en $\Theta(1)$ sans appel récursif (cas de base), soit faire deux appels récursifs. Il est également très important de remarquer que la valeur de retour des appels récursifs est prise en compte, ce qui n'apparaît pas dans le canevas. Puisque l'état à chaque instant de l'exécution de l'algorithme est un unique nombre entier, nous pouvons tout simplement conserver une unique variable qui ne doit même plus apparaître sur le stack. En effet cette dernière est constamment modifiée pendant l'exécution de l'algorithme et à chaque instant, c'est la dernière valeur prise par cette variable qui est considérée.

Le stack ne nous sert alors plus qu'à déterminer de quel endroit de la fonction nous *revenons* à chaque étape :

```
1
2  def mccarthy_nonrec(n):
3      stack = [0]
4      while True:
5          while n <= 100:
6              stack.append(1)
7              n += 11
8          n -= 10
9          i = stack.pop()
10         while i == 2:
11             i = stack.pop()
12         if i == 0:
13             break
14         stack.append(2)
15     return n
```

Proposition 2.50. *Pour tout $n \in \mathbb{N}$, la fonction 91 de McCarthy satisfait l'égalité suivante (d'où son nom) :*

$$M(n) = \begin{cases} 91 & \textit{si } n \leq 100, \\ n - 10 & \textit{sinon.} \end{cases}$$

Démonstration. Si $n \geq 101$, il est clair par la définition donnée dans l'énoncé que $M(n) = n - 10$. Pour montrer la deuxième partie de l'égalité, nous allons montrer que pour tout $m \in [\![0, 9]\!]$, si $n \in [\![10m, 100]\!]$, alors $M(n) = 91$. Pour cela, fonctionnons par induction sur m.

Pour cas de base prenons le cas $m = 9$. Soit alors $n \in [\![90, 100]\!]$. Notons $n' = n + 11 \in [\![101, 111]\!]$. Calculons alors :

$$M(n) = M(M(n + 11)) = M(M(n')) = M(n' - 10) = M(n + 11 - 10) = M(n + 1),$$

ou encore :

$$M(90) = M(91) = \ldots = M(100) = M(101) = 91.$$

Procédons ensuite par induction et supposons que le résultat est vrai pour un certain $m \in [\![1, 9]\!]$. Fixons alors $n \in [\![10(m - 1), 100]\!]$ et notons $n' = n + 11 \in [\![10m, 111]\!]$. À nouveau calculons :

$$M(n) = M(M(n + 11)) = M(M(n')).$$

Distinguons le cas où $n' > 100$ (i.e. $n > 89$) et appliquons le fait que $M(n') = n' - 10 = n + 1$ pour déduire $M(n) = M(n + 1)$ et le cas où $n' \leq 100$ pour déduire le fait que $M(n') = 91$. Dans les deux cas, il existe un $m \in [\![90, 100]\!]$ tel que $M(n) = M(m)$. Or M est constante et vaut 91 sur $[\![90, 100]\!]$, ce qui conclut la preuve. □

Nous pouvons donc écrire la fonction suivante :

```
1 def mccarthy_opti(n):
2     return 91 if n <= 101 else n-10
```

Proposition 2.51. *Les trois fonctions proposées dans cet exercice s'exécutent en temps* $\Theta(1)$.

Démonstration. Il est évident que `mccarthy_opti` s'exécute en temps constant mais il est moins clair au premier abord que ce résultat tient également pour `mccarthy_rec` ainsi que pour `mccarthy_nonrec`. Cependant, il est clair que pour $n \geq 101$, ces deux fonctions s'exécutent en temps constant. Or par définition de Θ, seul le comportement asymptotique nous intéresse et ce dernier est borné par une constante. □

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).

Résolution. Il faut ici modifier la fonction `partition` de manière à ce que cette dernière s'occupe de regrouper tous les éléments égaux au pivot, et de séparer les plus petits à gauche et les plus grands à droite. Cependant, il faut maintenir dans deux variables l'indice de début de la sous-liste égale au pivot, et l'indice de fin de cette même sous-liste. La fonction `partition` doit alors renvoyer cette paire d'indices au lieu d'un unique indice :

```
def partition(vect, beg, end):
    pivot = v[beg]
    i = beg-1
    j = beg+1
    k = end+1
    while j < k:
        if v[j] < pivot:
            i += 1
            swap(vect, i, j)
            j += 1
        elif v[j] > pivot:
            k -= 1
            swap(vect, k, j)
        else:
            j+=1
    return i+1, k-1
```

La fonction `swap` est simplement définie comme suit (et permet de permuter deux entrées du tableau) :

```
def swap(vect, i, j):
    vect[i], vect[j] = vect[j], vect[i]
```

La fonction `quicksort` quant à elle fonctionne de manière très similaire à la version vue en cours, à la différence près qu'elle doit faire usage des deux indices renvoyés par `partition`, i.e. les sous-listes à traiter vont respectivement de l'indice `beg` à `middle_beg-1` et de l'indice `middle_end+1` à `end` :

```
def quicksort(vect, beg, end):
```

```
    if beg < end:
        middle_beg, middle_end = partition(vect, beg, end)
        quicksort(vect, beg, middle_beg-1)
        quicksort(vect, middle_end+1, end)
```

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.

Résolution. Le principe derrière cet exercice est le suivant : puisqu'à chaque appel à `partition`, un nouvel élément est placé à la bonne position, à chaque itération il faut chercher si l'élément cherché est à gauche ou à droite du pivot et itérer sur ce côté-là uniquement. Il faut cependant passer par une copie du vecteur `vect` sur lequel le tri *partiel* sera exécuté afin de s'assurer que `vect` reste inchangé :

```
def percentil_rec(vect, k):
    n = len(vect)
    if not (0 <= k < n):
        raise ValueError('k doit être inférieur à n')
    copy = vect[:]
    _percentil_rec(copy, k, 0, n-1)
    return copy[k]

def _percentil_rec(vect, k, beg, end):
    p = partition(vect, beg, end)
    if p == k:
        return
    if p < k:
        _percentil_rec(vect, k, p+1, end)
    else:
        _percentil_rec(vect, k, beg, p-1)
```

La version récursive ci-dessus est presque identique à la fonction `quicksort` mais ne fait au plus qu'un unique appel récursif (au lieu de 2 pour `quicksort`). Il s'agit donc de *fausse* récursivité, et peut être remplacée par une boucle `while` :

```
def percentil(vect, k):
    n = len(vect)
    if not (0 <= k < n):
        raise ValueError('k doit être inférieur à n')
    copy = vect[:]
    beg = 0
    end = n-1
    p = partition(copy, beg, end)
    while p != k:
        if p < k:
```

```
            beg = p+1
        else: #  p > k
            end = p-1
        p = partition(copy, beg, end)
    return copy[p]
```

Il est assez évident que cette fonction s'exécute en $\mathcal{O}(n \log n)$ en moyenne puisqu'elle nécessite moins d'opérations que `quicksort` qui, lui, en requiert $\sim 2n \log n$ en moyenne.

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.

Résolution. Il suffit d'adapter l'algorithme quicksort classique en changeant les comparaisons pour prendre en compte `key(v[i])` au lieu de `v[i]` :

```
class Quicksort:
    def __init__(self, v):
        self.v = v

    def sort(self, key):
        self.key = key
        self.quicksort(0, len(self.v)-1)

    def _partition(self, beg, end):
        pivot = self.key(self.v[beg])
        j = beg+1
        k = end+1
        while j < k:
            if self.key(self.v[j]) <= pivot:
                j += 1
            elif self.key(self.v[j]) > pivot:
                k -= 1
                self.swap(j, k)
        self.swap(vect, j, beg)
        return j

    def swap(self, i, j):
        self.v[i], self.v[j] = self.v[j], self.v[i]

    def _quicksort(vect, beg, end):
        if beg < end:
            idx = self._partition(vect, beg, end)
            self._quicksort(vect, beg, idx-1)
            self._quicksort(vect, idx+1, end)
```

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 ?

Résolution. `argsort` est un cas particulier de l'exercice précédent puisque si on définit une fonction `key` par `key(i) = array[i]`, alors cela revient à trier le tableau `indices` en utilisant la fonction `key` :

```
1 def argsort(vect):
2     to_sort = list(range(len(vect)))
3     def key(idx):
4         return vect[idx]
5     Quicksort(to_sort).sort(key)
6     return to_sort
```

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.

Résolution. Faisons une classe `CustomPivotQuicksort` dont le constructeur prend en paramètre un vecteur (à trier *inplace*) et une fonction de choix de pivot ayant pour attributs le vecteur à trier, la fonction de choix de pivot et un entier `counter` contenant le nombre de fois que la méthode `_quicksort` a été appelée :

```
class CustomPivotQuicksort:
    def __init__(self, vect, pivot_choice_fct):
        self.vect = vect
        self.choose_pivot = pivot_choice_fct

    @property
    def counter(self):
        return self._counter
```

La méthode privée `_quicksort` (appelée par la méthode publique `sort`) s'écrit de manière naturelle avec comme seul ajout l'incrément de `_counter` :

```
    def sort(self):
        self._counter = 0
        self._quicksort(0, len(self.vect)-1)

    def _quicksort(self, beg, end):
        if beg >= end:
            return
        self._counter += 1
        pivot_idx = self._partition(beg, end)
        self._quicksort(beg, pivot_idx-1)
        self._quicksort(pivot_idx+1, end)
```

La méthode privée `_partition` est semblable à ce qui a été écrit jusqu'à présent avec comme seul ajout les deux premières lignes qui servent respectivement à choisir le pivot

et à le placer en première position :

```
    def _partition(self, beg, end):
        pivot_idx = self.choose_pivot(self.vect, beg, end)
        self.swap(beg, pivot_idx)
        pivot = self.vect[beg]
        j = beg
        k = end+1
        while j < k:
            j = self._first_bigger_than(j, end, pivot)
            k = self._first_smaller_than(beg, k, pivot)
            #print(j, k, end='')
            if j < k:
                self.swap(j, k)
        self.swap(beg, k)
```

Les méthodes `swap`, `_first_bigger_than` et `_first_smaller_than` sont données par :

```

    def swap(self, i, j):
        self.vect[i], self.vect[j] = self.vect[j], self.vect[i]

    def _first_bigger_than(self, beg, end, pivot):
        beg += 1
        while self.vect[beg] <= pivot:
            if beg == end:
                break
            beg += 1
        return beg

    def _first_smaller_than(self, beg, end, pivot):
        end -= 1
        while pivot < self.vect[end]:
            if beg == end:
                break
            end -= 1
```

La fonction de choix de pivot demandée (dont la signature est `function(vector, beg, end)`) peut s'écrire comme suit :

```
def median_of_3(vect, beg, end):
    mid = (beg+end) // 2
    if vect[beg] <= vect[mid]:
        if vect[end] <= vect[beg]:
            return beg
        else:
            return mid if vect[mid] <= vect[end] else end
    else:
        if vect[beg] <= vect[end]:
```

```
            return beg
        else:
            return end if vect[mid] <= vect[end] else mid
```

et trier un vecteur `v` donné se fait comme ceci (attention le vecteur `v` est bien modifié par l'appel à `sorter.sort()`) :

```
sorter = CustomPivotQuicksort(v, median_of_3)
sorter.sort()
assert all(v[i] <= v[i+1] for i in range(len(v)-1))
print(sorter.count, 'calls to _quicksort')
```

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.

Résolution. Nous pouvons réutiliser la classe `CustomPivotQuicksort` écrite ci-dessus et uniquement écrire une autre méthode de choix du pivot implémentant cet algorithme. La fonction `median` prend en paramètre un vecteur de taille ≤ 5 et en détermine la médiane. Pour ce faire, nous pouvons tout simplement trier le vecteur et en retourner l'élément du milieu. Le tri proposé ici est un *bubble sort* mais la fonction `median` s'exécute en $\mathcal{O}(1)$ puisque la taille du vecteur est bornée.

```
def median(vector, beg, end):
    for i in range(beg, end-1):
        for j in range(i, end-1):
            if vector[j] > vector[j+1]:
                vector[j], vector[j+1] = vector[j+1], vector[j]
    return vector[(beg+end) // 2]
```

Il nous faut ensuite écrire la fonction (récursive) `median_of_medians` (à donner en paramètre au constructeur de `CustomPivotQuicksort`) qui fait appel à `median` pour le cas de base :

```
def median_of_medians(vector, beg=None, end=None):
    if beg is not None and end is not None:
        vector = vector[beg:end]
    n = len(vector)
```

```
    if n <= 5:
        return median(vector, 0, n)
    new_vector = [0] * (n//5)
    for i in range(n//5):
        new_vector[i] = median(vector, i*5, min((i+1)*5, n))
    return median_of_medians(new_vector)
```

Notons qu'une copie (au moins partielle) de `vector` est faite pour éviter de modifier le tableau en train d'être trié par `Quicksort`. Ce traitement n'est pas obligatoire et peut être retiré (ce qui rendra le code plus efficace) : il a été écrit dans le cas présent pour restreindre les modifications du tableau à trier à l'algorithme `Quicksort` en tant que tel, et pas à l'algorithme de sélection de pivot.

Séance 19 — Programmation dynamique (partie 1)

Exercice 19.1. Écrivez une fonction `get_change` prenant les paramètres :
- `coins`, une liste triée des valeurs de pièces possibles ;
- `value`, un entier,

et renvoyant le plus petit nombre de pièces nécessaire pour 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]
```

Résolution. Nous pourrions être tentés d'appliquer la solution naïve suivante (une approche gloutonne dans laquelle on prend autant de pièces de valeur maximale que possible jusqu'à arriver à la somme désirée) :

```
1 def get_change_naive(coins, value):
2     ret = 0
3     idx = len(coins)-1
4     while idx >= 0:
5         ret += value // coins[idx]
6         value %= coins[idx]
7         idx -= 1
8     return ret
```

Cependant cette approche ne garantit pas que le nombre de pièce est minimal. En effet si nous prenons des pièces des valeurs suivantes : `[1, 3, 4]`, alors le nombre de pièces minimum pour atteindre la valeur 6 est 2 (car il faut deux pièces de 3) alors que la méthode `get_change_naive` renverra 3 (une pièce de 4 et deux pièces de 1). Il nous faut donc utiliser la programmation dynamique pour être sûr d'obtenir une solution optimale.

L'idée derrière cet exercice peut se formuler comme ceci : pour trouver le nombre minimum de pièces sommant à v en utilisant uniquement des pièces à valeur dans $c \in \mathbb{N}^n$, il suffit d'ajouter une pièce à une somme plus petite déjà trouvée. Nous allons donc devoir itérativement déterminer la solution au problème sur des valeurs de plus en plus grandes jusqu'à arriver à v. Formulons ce problème de manière plus formelle et rigoureuse.

Fixons un vecteur $c = (c_1, \ldots, c_n) \in \mathbb{N}^n$ de pièces.

Définition 19. Soit $v \in \mathbb{N}$. Un vecteur $x = (x_1, \ldots, x_n) \in \mathbb{N}^n$ est dit *solution au problème*

actuel pour v si :

$$\sum_{k=1}^{n} x_k c_k = v.$$

Une solution x est dite *optimale* si pour toute solution y au problème actuel pour v, on a $|x| \leq |y|$, où $|\cdot|$ désigne la somme des composantes :

$$|\cdot| : \mathbb{N}^n \to \mathbb{N} : x \mapsto \sum_{k=1}^{n} x_k.$$

Notons N_v le nombre minimum de pièces de c nécessaire pour sommer à v, i.e. si x est une solution optimale au problème actuel pour v, alors $|x| = N_v$.

Lemme 2.52. *Soient* $v_1, v_2 \in \mathbb{N}$. *Alors :*

$$N_{v_1+v_2} \leq N_{v_1} + N_{v_2}.$$

Démonstration. Soient x et y solutions optimales pour respectivement v_1 et v_2. Alors $(x + y) \coloneqq (x_1 + y_1, \ldots, x_n + y_n) \in \mathbb{N}^n$ est une solution au problème pour $v_1 + v_2$. En effet :

$$\sum_{k=1}^{n} (x+y)_k c_k = \sum_{k=1}^{n} x_k c_k + \sum_{k=1}^{n} y_k c_k = v_1 + v_2.$$

De plus il est clair que pour tout $k \in [\![1, n]\!]$: $(x + y)_k = x_k + y_k \geq x_k \geq 0$.

Dès lors, si z est une solution optimale au problème pour $v_1 + v_2$, par définition nous savons que $N_{v_1+v_2} = |z| \leq |x + y| = |x| + |y|$. □

Proposition 2.53. *Si* $x \in \mathbb{N}^n$ *est solution optimale au problème pour* $v \in \mathbb{N}$ *tel que* $v > 1$, *alors il existe* $k \in [\![1, n]\!]$ *tel que* $v' = v - c_k > 0$ *et* $y \in \mathbb{N}^n$ *tel que* y *est sol optimale pour* v' *satisfaisant :*

$$x = y + e_k = (y_1, \ldots, y_{k-1}, y_k + 1, y_{k+1}, \ldots, y_n).$$

Démonstration. Supposons par l'absurde que x soit une solution optimale pour v mais :

$$\forall k \in [\![1, n]\!] : (v > c_k \Rightarrow \forall y \text{ solution optimale pour } v - c_k : x \neq y + e_k).$$

Il existe nécessairement un $k \in [\![1, n]\!]$ tel que $v > c_k$ et $x_k \gneqq 0$ puisque $v > 1$. Pour ce k, $x - e_k = (x_1, \ldots, x_{k-1}, x_k - 1, x_{k+1}, \ldots, x_n)$ est une solution pour $v - c_k$ mais n'est pas optimale par hypothèse. Dès lors il existe une solution optimale $z \in \mathbb{N}^n$ pour $v - c_k$ telle que :

$$N_{v-c_k} = |z| \lneqq |x - e_k| = |x| - 1 = N_v - 1.$$

Cependant, cette inégalité contredit le lemme précédent qui dit (en particulier) que $N_{v-c_k} \geq N_v - 1$ pour tout k tel que $v > c_k$. Nous avons donc une contradiction, la proposition est donc vraie. □

Corollaire 2.54.

$$N_v = \begin{cases} 0 & \text{si } v = 0, \\ 1 + \min\limits_{\substack{k \in [\![1,n]\!] \\ \text{s.t. } v > c_k}} N_{v-c_k} & \text{sinon.} \end{cases}$$

Démonstration. Immédiat par la proposition précédente. □

Écrivons maintenant le code ! Nous allons calculer la relation de récurrence donnée dans le corollaire ci-dessus. Pour cela, il faut maintenir une table (donc une liste) de taille $v+1$ telle que l'entrée d'indice `j` contient N_j, i.e. le nombre minimum de pièces nécessaire pour sommer à j. Nous savons que `table[0] = 0` par ce même corollaire.

À partir de là, il nous faut parcourir cette liste de l'indice `1` à `value` (compris) et appliquer la relation de récurrence, i.e. aller trouver la valeur minimale aux entrées `v-coins[k]` et y ajouter `1`.

```python
def get_change(coins, value):
    INF = +float('inf')
    table = [INF] * (value+1)
    table[0] = 0
    for v in range(1, value+1):
        for coin_idx, coin in enumerate(coins):
            sub_sum = v-coin
            if sub_sum < 0:
                break
            new_sol = table[sub_sum]+1
            if new_sol < table[v]:
                table[v] = new_sol
    return None if table[-1] == INF else table[-1]
```

Proposition 2.55. *La fonction* `get_change` *effectue* $\mathcal{O}(nv)$ *instructions.*

Démonstration. La valeur de N_v est calculée pour tout $v \in [\![0, v]\!]$, et chacune de ces valeurs est calculée en $\leq n$ comparaisons, pour un total de $\mathcal{O}(nv)$ comparaisons. Le nombre d'instruction total est proportionnel au nombre de comparaisons, ce qui conclut la preuve. □

Bonus : Écrivons une classe `Change` qui représentera une solution au problème :

```python
class Change:
    def __init__(self, coins, counts):
        assert len(coins) == len(counts)
        self.coins = coins
        self.counts = counts

    def copy(self):
        return Change(self.coins, self.counts.copy())

    def add_coin(self, coin):
        self.counts[self.coins.index(coin)] += 1

```

```
    def nb_coins(self, coin):
        return self.counts[self.coins.index(coin)]

    def __len__(self):
        return sum(self.counts)

    def __lt__(self, other):
        return len(self) < len(other)
```

Utilisons cette classe pour modifier la fonction `get_change` de manière à conserver en mémoire quelles pièces sont utilisées dans chaque solution (en fait nous stockons x en entier et pas uniquement $|x|$ comme précédemment) :

```
def get_change_details(coins, value):
    table = [None] * (value+1)
    table[0] = Change(coins, [0]*len(coins))
    for v in range(1, value+1):
        for coin_idx, coin in enumerate(coins):
            sub_sum = v-coin
            if sub_sum < 0 or table[sub_sum] is None:
                continue
            new_sol = table[sub_sum].copy()
            new_sol.add_coin(coin)
            if table[v] is None or new_sol < table[v]:
                table[v] = new_sol
    return None if table[-1] is None else table[-1].counts
```

Notez que si on veut ajouter à cela une limite sur le nombre de pièces disponibles, on peut s'y prendre comme suit :

```
def get_change_details_limits(coins, limits, value):
    table = [None] * (value+1)
    table[0] = Change(coins, [0]*len(coins))
    for v in range(1, value+1):
        for coin_idx, coin in enumerate(coins):
            sub_sum = v-coin
            if sub_sum < 0 or table[sub_sum] is None:
                continue
            new_sol = table[sub_sum].copy()
            if new_sol.nb_coins(coin)+1 > limits[coin_idx]:
                continue
            new_sol.add_coin(coin)
            if table[v] is None or new_sol < table[v]:
                table[v] = new_sol
    return None if table[-1] is None else table[-1].counts
```

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".`

Résolution. Commençons par déterminer la longueur de la sous-séquence commune la plus longue, nous adapterons cette solution après afin de retrouver a sous-séquence en question.

Pour cela, prenons deux séquences $x = (x_1, \dots, x_m)$ et $y = (y_1, \dots, y_n)$ et pour $i \in [\![0, m]\!], j \in [\![0, n]\!]$, notons $L_{i\,j}$ la longueur de la sous-séquence commune la plus longue entre $x^i = (x_1, \dots, x_i)$ et $y^j = (y_1, \dots, y_j)$.

Lemme 2.56. *Pour deux séquences x et y de longueur respective m et n, la relation de récurrence suivante tient :*

$$L_{i\,j} = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ 1 + L_{i-1\,j-1} & \text{si } x_i = y_j, \\ \max\{L_{i-1\,j}, L_{i\,j-1}\} & \text{sinon.} \end{cases}$$

Démonstration. Puisque x^0 et y^0 sont des séquences vides, la plus longue sous-séquence commune entre x^i (ou y^j) et toute autre séquence est de longueur nulle, i.e. $L_{i\,j} = 0$.

Si $x_i = y_j$, alors la sous-séquence commune la plus longue entre x^i et y^j est la sous-séquence dont les $L_{i-1\,j-1}$ premiers éléments donnent la sous-séquence commune la plus longue entre x^{i-1} et y^{j-1}, et finissant par $x_i = y_j$.

Finalement, $i > 0, j > 0$ et si $x_i \neq y_j$, alors la sous-séquence commune la plus longue entre x^i et y^j est soit la sous-séquence commune la plus longue entre x^{i-1} et y^j, soit la sous-séquence commune la plus longue entre x^i et y^{j-1} (en fonction de laquelle est la plus longue). □

Nous pouvons donc trouver la longueur de cette sous-séquence comme suit :

```python
class LongestCommonSequence:
    def __init__(self, seq1, seq2):
        self.seq1 = seq1
        self.seq2 = seq2
        self.lengths = [[0]*(self.n+1) for i in range(self.m+1)]

    @property
    def m(self):
        return len(self.seq1)

    @property
    def n(self):
        return len(self.seq2)

    def find(self):
        for i in range(self.m+1):
```

```
            for j in range(self.n+1):
                if i == 0 or j == 0:
                    pass # lengths[i,j] = 0
                elif self.seq1[i-1] == self.seq2[j-1]:
                    self.lengths[i][j] = self.lengths[i-1][j-1] + 1
                else:
                    self.lengths[i][j] = max(
                        self.lengths[i-1][j],
                        self.lengths[i][j-1]
                    )
        return self.lengths[-1][-1]
```

Maintenant nous voulons reconstruire la sous-séquence en question sur base de la matrice `lengths`. Pour cela remarquons que la sous-séquence commune la plus longue n'est augmentée que dans le cas $x_i = y_j$. Dès lors, si à chaque étape de la construction de la matrice `lengths`, on stocke dans lequel des 3 cas nous somme (ou de manière équivalent, si on note de quelle *direction* on vient), alors nous pouvons juste suivre ces directions en sens inverse jusqu'à atteindre une extrémité de la matrice.

Le constructeur et la méthode `find` se voient donc modifiés comme suit :

```
UP = 0
LEFT = 1
UPPER_LEFT = 2

class LongestCommonSequence:
    def __init__(self, seq1, seq2):
        self.seq1 = seq1
        self.seq2 = seq2
        self.lengths = [[0]*(self.n+1) for i in range(self.m+1)]
        self.directions = [[None]*(self.n+1) for j in range(self.m+1)]

    @property
    def m(self):
        return len(self.seq1)

    @property
    def n(self):
        return len(self.seq2)

    def find(self):
        for i in range(self.m+1):
            for j in range(self.n+1):
                if i == 0 or j == 0:
                    pass # lengths[i,j] = 0
                elif self.seq1[i-1] == self.seq2[j-1]:
                    self.lengths[i][j] = self.lengths[i-1][j-1] + 1
                    self.directions[i][j] = UPPER_LEFT
                else:
                    if self.lengths[i-1][j] > self.lengths[i][j-1]:
```

```
                    self.lengths[i][j] = self.lengths[i-1][j]
                    self.directions[i][j] = UP
                else:
                    self.lengths[i][j] = self.lengths[i][j-1]
                    self.directions[i][j] = LEFT
        return self._find_longest()
```

Il nous reste à écrire la méthode `_find_longuest` en suivant la remarque ci-dessus :

```
    def _find_longest(self):
        i = self.m
        j = self.n
        ret = ''
        while i > 0 and j > 0:
            if self.directions[i][j] == UP:
                i -= 1
            elif self.directions[i][j] == LEFT:
                j -= 1
            else:
                i -= 1
                j -= 1
                ret = self.seq1[i] + ret
        return ret
```

Séance 20 — Programmation dynamique (partie 2)

Exercice 20.1. Étant donnée une séquence de valeurs réelles $x = (x_1, \dots, x_n)$, trouvez une sous-séquence contiguë de somme maximale. Par exemple, la séquence suivante :

$$x = (1, 2, 4, -4, 10, -5, -6, -12)$$

admet la sous-séquence de somme maximale suivante :

$$x' = (1, 2, 4, -4, 10).$$

Résolution.

Lemme 2.57. *Soit $x \in \mathbb{R}^n$ une séquence de valeurs réelles. Si $i, j \in [\![1, n]\!]$ désignent les indices de départ et de fin maximisant la somme, i.e. si $\sum_{k=i}^{j} x_k$ est maximale, alors :*

1. *soit $i = 1$, soit $x_{i-1} \leq 0$;*
2. *soit $j = n$, soit $x_{j+1} \leq 0$.*

Autrement dit, soit ces indices correspondent aux limites du vecteur, soit ils séparent des éléments positifs et des éléments négatifs.

Démonstration. Il est clair que si $i > 1$ et $x_{i-1} > 0$, alors $\sum_{k=i-1}^{j} x_k > \sum_{k=i}^{j} x_k$ et donc $(x_i, \dots, x_j)$ n'est pas de somme maximale. De manière similaire, si $j < n$ et $x_{j+1} > 0$, alors $\sum_{k=i}^{j+1} x_k > \sum_{k=i}^{j} x_k$ et donc $(x_i, \dots, x_j)$ n'est pas de somme maximale. □

Nous allons alors procéder de la manière suivante : en partant du début de la séquence, ajoutons tous les éléments positifs que l'on trouve. Dès que l'on tombe sur un nombre négatif, regardons la somme de ces nombres. Si la somme des nombres négatifs est (en valeur absolue) plus grande que la somme des nombres positifs, alors on sauve la séquence des nombres positifs (si elle est plus grande que la plus grande somme trouvée jusqu'à présent), et sinon, on regarde si la somme des nombres positifs qui suivent permettent d'augmenter la somme totale.

```
1  def find_max_sub_seq(v):
2      best_sum = -float('inf')
3      n = len(v)
4      k = 0
5      # on ignore tous les nombres négatifs au début
6      while k < len(v) and v[k]< 0:
7          k += 1
8      # on ignore tous les nombres négatifs à la fin
9      while n >= k and v[n-1] < 0:
10         n -= 1
11     pos_sum = 0
12     neg_sum = 0
13     tmp_start = k # d'où on part pour la somme en cours
14     i, j = k, n
15     while k < n:
16         while k < n and v[k] >= 0:
17             pos_sum += v[k]
18             k += 1
```

```
19          # Si somme plus grande que la meilleure jusqu'à  présent
20          if pos_sum > best_sum:
21              best_sum = pos_sum
22              i = tmp_start
23              j = k
24          # on accumule tous les nombres négatifs
25          while k < n and v[k] < 0:
26              neg_sum += v[k]
27              k += 1
28          # si |neg_sum| > pos_sum
29          if pos_sum + neg_sum <= 0:
30              pos_sum = 0
31              tmp_start = k
32          else:
33              # sinon on regarde si on peut faire encore plus grand avec la
                    suite
34              pos_sum += neg_sum
35          neg_sum = 0
36      return v[i:j]
```

Proposition 2.58. *La fonction* `find_max_sub_seq` *trouve la sous-séquence contiguë maximale en* $\Theta(\ell)$ *opérations si* ℓ *désigne la taille du conteneur.*

Démonstration. La fonction parcourt le vecteur de gauche à droite à l'aide de l'indice `k` qui va de 0 à `n` (qui est, quant à lui, initialisé à ℓ). De plus, à `k` fixé, le traitement de l'entrée `v[k]` se fait en temps $\Theta(1)$: si `v[k]` est positif, alors il est ajouté à l'accumulateur `pos_sum` et s'il est négatif, alors il est ajouté à l'accumulateur `neg_sum`. Il reste les traitements conditionnels (l. 20-23 et 29-34) qui sont, eux également exécutés au plus ℓ fois et dont le temps d'exécution ne dépend pas de la taille du vecteur.

Le nombre total d'opérations est donc $\Theta(\ell) + \mathcal{O}(\ell) = \Theta(\ell)$. □

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.

Résolution.

Lemme 2.59. *Soient* $x = (x_1, \dots, x_m)$ *et* $y = (y_1, \dots, y_n)$ *deux séquences. Notons* D_{ij} *la distance d'édition entre* $x^i = (x_1, \dots, x_i)$ *et* $y^j = (y_1, \dots, y_j)$*. Alors* D_{ij} *satisfait la relation suivante :*

$$D_{ij} = \begin{cases} i + j & \text{si } i = 0 \text{ ou } j = 0, \\ \min\{D_{i-1\,j-1}, D_{i-1\,j} + 1, D_{i\,j-1} + 1\} & \text{si } x_i = y_j, \\ 1 + \min\{D_{i-1\,j-1}, D_{i-1\,j}, D_{i\,j-1}\} & \text{sinon.} \end{cases}$$

Démonstration. Si $i = 0$, alors il faut bien faire j insertions pour passer de $x^0 = ()$ à y^j, et inversement, si $j = 0$, alors il faut i suppression pour passer de x^i à $y^j = ()$. Ces deux cas

ne sont pas exclusifs : il est possible que $i = j = 0$ en quel cas la distance est bien de 0. Ces deux cas peuvent se réécrire en $D_{i\,j} = i + j$ si $i = 0$ ou $j = 0$.

Prenons maintenant $i > 0$ et $j > 0$ tels que $x_i = y_j$. Pour passer de x^i à y^j, il faut distinguer trois cas :

1. soit on transforme x^{i-1} en y^{j-1} et on garde $x_i = y_j$ en place (donc $D_{i-1\,j-1}$ opérations) ;
2. soit on passe de x^i à y^{j-1} et puis on insère y_j (donc $D_{i\,j-1} + 1$ opérations) ;
3. soit on passe de x^{i-1} à y^{j-1} et puis on insère y_j (donc $D_{i-1\,j-1} + 1$ opérations).

Puisque la distance d'édition est le nombre *minimum* d'ajouts/modifications/suppressions nécessaire pour passer d'une séquence à l'autre, on a bien :

$$D_{i\,j} = \min\{D_{i-1\,j-1}, D_{i-1\,j} + 1, D_{i\,j-1} + 1\}.$$

Si maintenant $i > 0$ et $j > 0$ avec $x_i \neq y_j$, pour passer de x^i à y^j, il faut à nouveau distinguer trois cas :

1. soit on retire x_i et puis on passe de x^{i-1} à y^j (donc $D_{i-1\,j} + 1$ opérations) ;
2. soit on passe de x^i à y^{j-1} et puis on insère y_j (donc $D_{i\,j-1} + 1$ opérations) ;
3. soit on passe de x^{i-1} à y^{j-1} et puis on insère y_j (donc $D_{i-1\,j-1} + 1$ opérations).

À nouveau, on a bien :

$$D_{i\,j} = 1 + \min\{D_{i-1\,j}, D_{i\,j-1}, D_{i-1\,j-1}\}.$$

□

Remarque. *Définissons :*

$$\delta_{i\,j} = \begin{cases} 1 & \textit{si } x_i \neq y_j, \\ 0 & \textit{sinon.} \end{cases}$$

Nous pouvons maintenir reformuler la conclusion du lemme ci-dessus comme suit :

$$D_{i\,j} = \begin{cases} i + j & \textit{si } i = 0 \textit{ ou } j = 0, \\ \min\{D_{i-1\,j-1} + \delta_{i\,j}, D_{i-1\,j} + 1, D_{i\,j-1} + 1\} & \textit{sinon.} \end{cases}$$

Comme précédemment, construisons itérativement cette matrice D en partant du cas $i = 0$ ou $j = 0$ jusqu'à arriver à $D_{m\,n}$ qui est la distance d'édition entre x et y :

```
1  def edit_distance(x, y):
2      m = len(x)
3      n = len(y)
4      D = [[0]*(n+1) for i in range(m+1)]
5      for i in range(1, m+1):
6          D[i][0] = i
7      for j in range(1, n+1):
8          D[0][j] = j
9      for i in range(m):
10         for j in range(n):
11             delta_ij = 1-int(x[i] == y[j])
```

```
            D[i+1][j+1] = min(
                D[i][j]  + delta_ij,
                D[i+1][j] + 1,
                D[i][j+1] + 1
            )
    return D[m][n]
```

Remarque. *L'algorithme de Needleman-Wunsch est une variante de cet algorithme que vous verrez en bloc 3 (au cours INFO-F208) dans le cadre d'alignements de séquences biologiques (e.g. alignement de protéines). L'algorithme N-W utilise un système de pénalité sur les substitutions : certaines substitutions sont plus* coûteuses *que d'autres et donc augmentent davantage la variante de la distance d'édition.*

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`.

Résolution. À nouveau, commençons par trouver un algorithme qui trouve la longueur de la plus grande sous-séquence palindromique.

Définition 20. Une sous-séquence x' de x est un palindrome si $(x'_1, \ldots, x'_m) = (x'_m, \ldots, x'_1)$ (pour $m = |x'|$). On dit que x' est un *sous-palindrome* de x.

Un sous-palindrome x' de x est dit *maximal* si tout sous-palindrome y de x satisfait $|y| \leq |x'|$.

Définition 21. Pour $x = (x_1, \ldots, x_n)$ une séquence quelconque de taille finie n, pour $1 \leq i \leq j \leq n$, on note $x^{ij} = (x_i, \ldots, x_j)$. Notons L_{ij} la taille d'un sous-palindrome maximal de x^{ij}.

Lemme 2.60. *Soit $x = (x_1, \ldots, x_n)$ une séquence quelconque de taille finie. Alors pour toute paire (i, j) telle que $1 \leq i \leq j \leq n$:*

$$1 \leq L_{ij} \leq j - i + 1.$$

Démonstration. Puisque toute sous-séquence y de x^{ij} telle que $|y| = 1$ est un palindrome, on sait que $L_{ij} \geq 1$. De plus, puisque toute sous-séquence y de x^{ij} satisfait $|y| \leq |x^{ij}| = j - i + 1$ (par définition d'une sous-séquence), on sait que $L_{ij} \leq j - i + 1$. □

Remarque. *La borne supérieure sur L_{ij} ne peut être atteinte que lorsque x^{ij} est un palindrome puisque la seule sous-séquence y de x^{ij} satisfaisant $|y| = |x^{ij}|$ est $y = x^{ij}$.*

Lemme 2.61. *Soit $x = (x_1, \ldots, x_n)$ une séquence quelconque de taille finie. L satisfait la relation suivante :*

$$L_{ij} = \begin{cases} 1 & \text{si } i = j, \\ 2 & \text{si } j = i + 1 \text{ et } x_i = x_j, \\ L_{i+1\,j-1} + 2 & \text{si } j > i + 1 \text{ et } x_i = x_j, \\ \max\{L_{i\,j-1}, L_{i+1\,j}\} & \text{sinon.} \end{cases}$$

Démonstration. Il est clair que si $i = j$, alors $L_{ij} = 1$ par le lemme précédent.

Si $j = i + 1$ et $x_i = x_j$, alors x^{ij} est un sous-palindrome de taille 2, i.e. $L_{ij} \geq 2$. Or par le lemme précédent, on sait que $L_{ij} \leq j - i + 1 = i + 1 - i + 1 = 2$. On en déduit $L_{ij} = 2$.

Supposons maintenant que $1 \leq i \lneq j \leq n$ (et $j > i + 1$) tels que $x_i = x_j$. Nous savons qu'il existe un sous-palindrome y de $x^{i+1\,j-1}$ tel que $|y| = L_{i+1\,j-1}$ (par définition de L). Dès lors, $y' = (x_i, y_1, \ldots, y_{L_{i+1\,j-1}}, x_j)$ est un sous-palindrome de x^{ij}, et donc $L_{ij} \geq |y'| = |y| + 2 = L_{i+1\,j-1} + 2$. Dans l'autre sens, prenons $y = (y_1, \ldots, y_{L_{ij}})$ un sous-palindrome maximal de x^{ij}. On sait que $|y| = L_{ij} \geq L_{i+1\,j-1} + 2 \geq 2$. La sous-séquence $y' = (y_2, \ldots, y_{L_{ij}-1})$ est bien définie et est un sous-palindrome de $x^{i+1\,j-1}$. Dès lors $L_{ij} = |y| = |y'| + 2 \leq L_{i+1\,j-1} + 2$. On en déduit donc que $L_{ij} = L_{i+1\,j-1} + 2$.

Finalement si $i \lneq j$ et $x_i \neq x_j$, alors prenons y sous-palindrome maximal de $x^{i\,j-1}$ et z sous-palindrome maximal de $x^{i+1\,j}$. En particulier y et z sont des sous-palindromes de x^{ij}, et donc $L_{ij} \geq |y| = L_{i\,j-1}$ et $L_{ij} \geq |z| = L_{i+1\,j}$. Dès lors $L_{ij} \geq \max\{L_{i+1\,j}, L_{i\,j-1}\}$. Prenons maintenant $y = (y_1, \ldots, y_m)$, un sous-palindrome maximal de x^{ij}. On ne peut pas avoir $y_1 = x_i$ et $y_m = x_j$ car y est un palindrome et $x_i \neq x_j$. Dès lors y satisfait au moins une des conditions suivantes :

1. si $y_1 = x_i$, alors $y_m \neq x_j$ et y est un sous-palindrome de $x^{i\,j-1}$, et donc $|y| \leq L_{i\,j-1}$;
2. si $y_m = x_j$, alors $y_1 \neq x_i$ et y est un sous-palindrome de $x^{i+1\,j}$, et donc $|y| \leq L_{i+1\,j}$;
3. si $y_1 \neq x_i$ et $y_m \neq x_j$, alors y est un sous-palindrome de $x^{i+1\,j-1}$. En particulier, $|y| \leq L_{i+1\,j-1}$, mais $L_{i+1\,j-1} \leq \min\{L_{i+1\,j}, L_{i\,j-1}\}$ puisque tout sous-palindrome de $x^{i+1\,j-1}$ est un sous-palindrome de $x^{i+1\,j}$ et de $x^{i\,j-1}$ simultanément.

Puisqu'au moins une des conditions est satisfaite, on déduit que $|y| \leq \max\{L_{i\,j-1}, L_{i+1\,j}\}$ (i.e. la conclusion la moins restrictive des trois conditions est remplie). On en déduit que $L_{ij} = \max\{L_{i+1\,j}, L_{i\,j-1}\}$. □

Il y a cependant une subtilité dans l'implémentation de cette relation : contrairement aux exercices précédents où les entrées (i, j) des matrices calculées ne dépendent que des entrées (i', j') où $i < i'$ et $j < j'$, ici L_{ij} dépend potentiellement de $L_{i+1\,j}$ et $L_{i\,j-1}$. L'ordre dans lequel remplir la matrice L n'est donc pas évident. Une approche à cela est d'utiliser une fonction récursive qui va remplir cette matrice avec les entrées dont elle a besoin dans l'ordre nécessaire (mais tout en conservant les principes de programmation dynamique) :

```python
class SubPalindromeSize:
    def __init__(self, seq):
        self.seq = seq
        self.L = [[None]*self.n for _ in range(self.n)]

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

    def find(self, i=0, j=None):
        if j is None:
            j = self.n-1
        if self.L[i][j] is not None:
            return self.L[i][j]
```

```
        if i == j:
            ret = 1
        elif self.seq[i] == self.seq[j]:
            ret = 2
            if j > i+1:
                ret += self.find(i+1, j-1)
        else:
            ret = max(self.find(i+1, j), self.find(i, j-1))
        self.L[i][j] = ret
        return ret
```

Si nous voulons éviter les appels récursifs, nous pouvons tout de même trouver un ordre de parcours de L nous permettant d'avoir les valeurs nécessaires déjà initialisées. Pour cela, il suffit de se rendre compte que pour calculer L_{ij} (donc la longueur d'un sous-palindrome maximal d'une sous-séquence de longueur $j-i+1$), nous n'avons besoin que des $L_{i'j'}$ tels que $j'-i'+1 \leq j-i$. Dès lors, si nous calculons d'abord toutes les valeurs L_{ii}, ensuite toutes les valeurs L_{ii+1} et puis les valeurs L_{ii+2}, etc. jusque L_{1n}, nous n'avons plus de problème d'ordre d'initialisation :

```
    def find_non_rec(self):
        for length in range(self.n):
            for i in range(self.n-length-1):
                j = i+length
                if i == j:
                    self.L[i][j] = 1
                elif self.seq[i] == self.seq[j]:
                    self.L[i][j] = 2
                    if j > i+1:
                        self.L[i][j] += self.L[i+1][j-1]
                else:
                    self.L[i][j] = max(self.L[i+1][j], self.L[i][j-1])
        return self.L[0][self.n-1]
```

Pour la reconstruction d'un sous-palindrome sur base de la matrice L, inspirons-nous de la solution de l'exercice 19.2 dans lequel nous avions une deuxième matrice dans laquelle nous stockions une valeur représentant dans quel cas nous nous trouvions. Parcourir cette matrice depuis la case $(0, n-1)$ en suivant les *directions* nous permet alors de retrouver un sous-palindrome de taille maximale, et même de marquer les éléments de x qui sont ignorés :

```
ONE_CHAR = 0
TWO_CHARS = 1
EQUAL_CHARS = 2
DIFFERENT_CHARS_LEFT = 3
DIFFERENT_CHARS_RIGHT = 4

class SubPalindrome:
    def __init__(self, seq):
        self.seq = seq
```

```
10          self.L = [[None]*self.n for _ in range(self.n)]
11          self.cases = [[None]*self.n for _ in range(self.n)]
12
13      @property
14      def n(self):
15          return len(self.seq)
16
17      def find(self):
18          for length in range(self.n):
19              for i in range(self.n-length):
20                  j = i+length
21                  if i == j:
22                      self.L[i][j] = 1
23                      self.cases[i][j] = ONE_CHAR
24                  elif self.seq[i] == self.seq[j]:
25                      self.L[i][j] = 2
26                      if j > i+1:
27                          self.L[i][j] += self.L[i+1][j-1]
28                          self.cases[i][j] = EQUAL_CHARS
29                      else:
30                          self.cases[i][j] = TWO_CHARS
31                  else:
32                      if self.L[i+1][j] > self.L[i][j-1]:
33                          self.L[i][j] = self.L[i+1][j]
34                          self.cases[i][j] = DIFFERENT_CHARS_RIGHT
35                      else:
36                          self.L[i][j] = self.L[i][j-1]
37                          self.cases[i][j] = DIFFERENT_CHARS_LEFT
38          return self.reconstruct()
39
40      def reconstruct(self):
41          i = 0
42          j = self.n-1
43          chars = [None] * self.n
44          while i <= j:
45              if self.cases[i][j] == ONE_CHAR:
46                  chars[i] = self.seq[i]
47                  i += 1
48              elif self.cases[i][j] == TWO_CHARS:
49                  chars[i:j+1] = self.seq[i:j+1]
50                  i += 1
51                  j -= 1
52              elif self.cases[i][j] == EQUAL_CHARS:
53                  chars[i] = self.seq[i]
54                  chars[j] = self.seq[j]
55                  i += 1
56                  j -= 1
57              elif self.cases[i][i] == DIFFERENT_CHARS_LEFT:
58                  chars[j] = '_'
```

```
59                 j -= 1
60             else:
61                 chars[i] = '_'
62                 i += 1
63         return ''.join(chars)
```

Le constructeur a juste une addition : la création de l'attribut `cases` ; la méthode `find` est l'équivalent de `SubPalindromeSize.find_non_rec` avec les modifications de`cases` en plus ; et finalement `reconstruct` parcourt la matrice `cases` en recréant le sous-palindrome associé.

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]`.

Résolution. Ici nous pourrions avoir envie d'écrire simplement `return [D(n, k) for k in range(n)]` et de définir une autre fonction `D(n, k)`, mais en réalité nous n'en avons pas besoin, voire mieux : nous avons besoin de toutes les valeurs en même temps pour avancer.

Tout comme pour l'énumération du nombre de sous-ensembles, procédons par récurrence. Nous savons que si $S_n = |\mathcal{P}([\![1, n]\!])|$, alors S_n satisfait la relation de récurrence suivante :

$$S_n = \begin{cases} 1 & \text{si } n = 0, \\ 2S_n & \text{sinon.} \end{cases}$$

Maintenant considérons les nombres $0 \leq m \leq n$ et regardons comment se comporte le nombre de sous-ensembles de $[\![1, m]\!]$ dont la somme vaut k modulo n (notons bien que m apparaît ici uniquement dans l'ensemble dont on prend les sous-ensembles mais c'est bien toujours n dans le modulo). Si nous avons un ensemble, disons $S \in \mathcal{P}([\![1, m]\!])$, nous savons qu'il y a une unique valeur ℓ telle que S somme à k modulo n. Mais que se passe-t-il si nous ajoutons $m + 1$ à cet ensemble ? Nous savons que $\sum_{s \in S} s + m + 1 = k + m + 1 \mod n$. Dès lors, si nous définissons $T_{m,\ell}$ comme le nombre de sous-ensembles de $[\![1, m]\!]$ qui somment à k modulo n, nous observons aisément que $T_{0,0} = 1$ et $T_{0,\ell} = 0$ pour $\ell > 0$. De plus :

$$T_{m,k} = T_{m-1,k} + T_{m-1,k'},$$

où k' est l'unique valeur $0 \leq k' < n$ telle que $k' = k - m \mod n$. Nous pouvons dès lors implémenter la fonction `D` demandée comme ceci :

```python
def D(n: int) -> list[int]:
    T = [[0]*n for _ in range(n+1)]
    T[0][0] = 1
    for m in range(1, n+1):
        for ell in range(n):
            T[m][ell] = T[m-1][ell] + T[m-1][(ell-m)%n]
    return T[n]
```

Remarque. *Il est important de noter qu'en Python, le modulo de nombres entiers est obligatoirement positif, i.e.* `m % n` *est l'unique nombre* `k` *tel que* `0 <= k < n` *et* `m-k` *est*

un multiple de `n`*. En particulier, dans le code ci-dessus,* `ell <= m` *donc* `(ell-m) <= 0`*. Cependant* `(ell-m)%n` *est bien un indice positif.*

Proposition 2.62. *La fonction* `D` *s'exécute en* $\Theta(n^2)$ *opérations.*

Démonstration. La boucle sur `m` va s'exécuter exactement n fois et la boucle sur `ell` également. Puisque le traitement qui y est fait ne dépend pas de n (i.e. se fait en $\Theta(1)$), nous déduisons que `D` effectue $n^2 \cdot \Theta(1) = \Theta(n^2)$ opérations. □

Notons qu'ici la complexité en temps et en espace sont toutes deux quadratiques, mais il est tout à fait possible de s'en sortir avec $\Theta(n)$ en espace. En effet à l'étape m, seules les lignes m et $m-1$ sont considérées. D'ailleurs la ligne j ne sera plus jamais considérée après que m ait pris la valeur $j+1$. Nous pouvons dès lors nous en sortir avec uniquement deux lignes (pour un total de $2n = \Theta(n)$ cases de la mémoire).

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`.

Résolution. Il nous faut exprimer ce problème de manière récursive. Notons Z_n la longueur de la plus longue séquence de zéros consécutifs dans l'écriture binaire de n. Il est clair que si n est impair, alors $Z_n = Z_{\lfloor n/2 \rfloor}$. Sinon, notons k la plus grande puissance de 2 qui divise n (i.e. $2^k \mid n$ mais $2^{k+1} \nmid n$). Dans ce cas nous avons :

$$Z_n = \max\left\{k, Z_{\frac{n}{2^k}}\right\}. \tag{3.1}$$

Cette relation peut s'implémenter telle quelle comme suit :

```
def zeros_consecutifs_semi_rec(n):
    if n == 0:
        return 1
    if n & 1:
        n //= 2
        return 0 if n == 0 else zeros_consecutifs_semi_rec(n)
    ret = 0
    while n & 1 == 0:
        ret += 1
        n //= 2
    return max(ret, zeros_consecutifs_semi_rec(n))
```

Il est également possible de s'assurer que chaque appel à la fonction `zeros_consecutifs` s'exécute en $\mathcal{O}(1)$. Pour cela, il faut ajouter un paramètre `current` à la fonction qui sert à calculer k dans l'expression 3.1 :

```
def zeros_consecutifs(n, current=0):
    if n == 0:
        return 1
    m = n >> 1
    if m == 0:
        ret = 0
    else:
```

```
        if n & 1 == 0:
            ret = zeros_consecutifs(m, current+1)
        else:
            ret = zeros_consecutifs(m, 0)
    return max(current, ret)
```

Une autre manière de voir cette implémentation est de la considérer comme une version récursive de la fonction itérative suivante :

```
def zeros_consecutifs_iteratif(n):
    if n == 0:
        return 1
    ret = 0
    current = 0
    while n > 0:
        if n&1:
            current = 0
        else:
            current += 1
            if current > ret:
                ret = current
        n //= 2
    return ret
```

En effet, la variable `current` compte combien de zéros consécutifs ont été rencontrés jusqu'à présent et la variable `ret` contient la valeur maximale que `current` a atteint.

Proposition 3.1. *La fonction* `zeros_consecutifs` *effectue* $\Theta(\log n)$ *instructions.*

Démonstration. Il est clair que la boucle `while` de ce code va s'exécuter $\sim \log_2 n$ fois (au moins $\lfloor \log_2 n \rfloor$ et au plus $1 + \lceil \log_2 n \rceil$ fois). Si on note C_n le nombre total d'opérations, il est clair par cette observation que $C_n = \Omega(\log n)$, et puisque le traitement de `n`, `current` et `ret` s'exécute en temps $\mathcal{O}(1)$ [1], nous pouvons déduire que $C_n = \Theta(\log n)$. □

Utilisation de la builtin `bin`

Plusieurs soumissions utilisaient la fonction *builtin* `bin` de Python qui prend un entier `n` en paramètre et renvoie un `str` contenant la représentation binaire de `n`. Il était inutile d'utiliser cette fonction puisque tout entier est *déjà* représenté en binaire et que cette représentation peut être directement manipulée par les opérateurs binaires `&`, `>>`, `<<`, etc.

Voici un exemple d'utilisation de `bin`

```
def zeros_consecutifs_bin(n, pos=2, current=0):
    s = bin(n)
    if pos >= len(s):
```

1. En réalité, il faudrait s'assurer que l'instruction `n&1` s'exécute bien en temps $\mathcal{O}(1)$, ce qui est garanti sur des entiers de taille bornée mais ce qui est moins trivial sur des entiers qui peuvent être arbitrairement grands. Nous pouvons ici toutefois supposer que ce n'est pas un problème puisque, peu importe la taille effective de `n`, seul le bit de poids le plus faible nous intéresse. Il est donc tout à fait possible de vérifier la parité d'un nombre arbitrairement grand en temps constant.

```
        return current
    bit = s[pos]
    if bit == '1':
        ret = zeros_consecutifs_bin(n, pos+1, 0)
    else:
        ret = zeros_consecutifs_bin(n, pos+1, current+1)
    return max(current, ret)
```

Proposition 3.2. *Cette fonction effectue* $\Theta(\log(n)^2)$ *instructions.*

Démonstration. Par le même raisonnement que précédemment, on sait qu'il y a au plus $\lfloor \log n \rfloor \sim \log n$ appels récursifs à la fonction, et chaque appel récursif s'exécute en $\Theta(\log n)$ à cause de l'appel à `bin`. Le nombre total d'instructions est donc bien $\Theta(\log(n)^2)$. □

Une implémentation alternative faisant également usage de `bin` :

```
def zeros_consecutifs_bin_2(n, current=0):
    if n == 0:
        return current
    bit = bin(n)[-1]
    if bit == '1':
        ret = zeros_consecutifs_bin_2(n//2, 0)
    else:
        ret = zeros_consecutifs_bin_2(n//2, current+1)
    return max(current, ret)
```

Proposition 3.3. *Cette fonction effectue également* $\Theta(\log(n)^2)$ *instructions.*

Démonstration. Il faut ici un petit peu plus de travail pour montrer cela puisque maintenant la fonction `bin` est appelée sur une valeur différente à chaque appel récursif. Si nous notons à nouveau C_n le nombre d'instructions effectuées pour le paramètre n, alors nous avons $C_n = \Theta(1)$ pour $n \leq 1$ et pour $n > 1$ nous avons :

$$C_n = \Theta(\log n) + C_{\lfloor \frac{n}{2} \rfloor}.$$

En effet, le $k$ème appel récursif se fait en $\Theta(\log(\lfloor \frac{n}{2^k} \rfloor))$ puisqu'il y a $\Theta(\log(\lfloor \frac{n}{2^k} \rfloor))$ instructions pour l'appel à `bin` et $\Theta(1)$ instructions comme dans la solution récursive initiale.

Nous pouvons en déduire :

$$C_n = \Theta(\log n) + C_{\lfloor \frac{n}{2} \rfloor} = \Theta(\log n) + \Theta\left(\log \frac{n}{2}\right) + C_{\lfloor \frac{n}{4} \rfloor} = \cdots = \sum_{k=0}^{\ell-1} \Theta\left(\log \frac{n}{2^k}\right) + C_{\lfloor \frac{n}{2^\ell} \rfloor}$$

pour tout $\ell \leq \lfloor \log n \rfloor$. À nouveau, lorsque ℓ atteint cette borne :

$$C_n = \sum_{k=0}^{\lfloor \log n \rfloor - 1} \Theta\left(\log \frac{n}{2^k}\right) = \Theta\left(\sum_{k=0}^{\lfloor \log n \rfloor - 1} \log \frac{n}{2^k}\right).$$

Or :

$$\sum_{k=0}^{\lfloor \log n \rfloor - 1} \log \frac{n}{2^k} = \sum_{k=0}^{\lfloor \log n \rfloor - 1} (\log n - k) \sim \sum_{k=1}^{\lfloor \log n \rfloor} k \sim \frac{\log(n)^2}{2}.$$

Nous pouvons alors conclure que :

$$C_n = \Theta(\log(n)^2).$$

□

Le one-liner qui claque

Pour pouvoir faire les malins, nous pouvons aussi implémenter la fonction à l'aide d'un *one-liner* (qui a ici été écrit sur plusieurs lignes pour la lisibilité) :

```
def zc_oneliner(n, current=0):
    return 1 if n == 0 else max(
            current,
            0 if n<2 else zc_oneliner(n>>1, 0 if n&1 else current+1)
    )
```

Exercice coté 2. Écrivez une version récursive et non-récursive d'un parcours préfixé et suffixé d'un arbre d-aire.

Résolution. La classe `Node` de cet exercice utilise une liste pour contenir tous les fils d'un sommet. Gardons en tête que l'ordre de ces sommets dans la liste est important : lors d'un parcours on veut traiter ces sommets fils dans l'ordre dans lequel ils se situent au sein de la liste.

Versions récursives

Pour l'implémentation récursive, les deux fonctions sont fortement similaires : la seule différence est que lors du parcours préfixé, le sommet actuel est traité *avant* d'effectuer les appels récursifs sur les fils alors que lors du parcours postfixé, le sommet est traité *après* ses fils. Le parcours préfixé peut s'implémenter comme suit :

```
def prefix_rec(node):
    traitement(node.value)
    for child in node:
        prefix_rec(child)
```

Et le parcours postifxé peut s'implémenter comme suit :

```
def postfix_rec(node):
    for child in node:
        postfix_rec(child)
    traitement(node.value)
```

Il faut noter que la boucle `for child in node` va bien itérer sur tous les sommets fils dans l'ordre de la liste de sommets.

Versions non-récursives

Tout comme dans les exercices de la séance 7, lors de la dérécursification des fonctions de parcours, il faut maintenir un stack explicitement pour simuler les appels récurisfs, et lors du traitement de chaque sommet, les fils doivent être insérés sur le stack dans l'ordre inverse de l'ordre de stockage. En effet, un stack étant un LIFO, les sommets sont récupérés sur le stack dans l'ordre inverse de leur ajours. Dès lors pour les récupérer du fils le plus à gauche jusqu'au fils le plus à droite, il faut les ajouter sur le stack du plus à droite jusqu'au plus à gauche.

Pour cela, il faut itérer sur les sommets en utilisant la *builtin* `reversed` qui va appeler la méthode `__reversed__` de la classe `Node` et ajouter les sommets dans cet ordre-là.

Le parcours préfixé dérécursifié ressemble très fort à la solution de l'exercice 7.3 :

```
def prefix_non_rec(node):
    stack = [node]
    while len(stack) > 0:
        node = stack.pop()
        traitement(node.value)
        for child in reversed(node):
            stack.append(child)
```

Les trois différences sont les suivantes :

— le `print` de la valeur du sommet est remplacé par un appel à la fonction `traitement` pour être plus générique ;
— les `push` sur le stack ne sont pas faits explicitement sur les sommets mais sont faits à l'intérieur d'une boucle ;
— il ne faut pas vérifier si les fils valent `None` ou non puisque seuls les fils existants sont stockés dans la liste.

Pour le parcours postfixé, il faut ajouter une variable supplémentaire sur le stack à côté de chaque sommet nous permettant de déterminer si les fils du sommet en question ont déjà été traités ou non. Cette variable vaut `False` si les sommets fils n'ont pas encore été traités (et donc s'il faut les ajouter sur le stack) et vaut `True` si les sommets fils ont déjà été traités (et donc s'il faut traiter ce sommet-là) :

```
1  def postfix_non_rec(node):
2      stack = [(False, node)]
3      while len(stack) > 0:
4          is_treating, node = stack.pop()
5          if is_treating:
6              traitement(node.value)
7          else:
8              stack.append((True, node))
9              for child in reversed(node):
10                 stack.append((False, child))
```

Lors d'un `pop` sur le stack, nous récupérons une paire `(is_treating, node)` et la suite de l'exécution va dépendre de ce premier booléen : s'il est à `True`, alors il suffit de traiter le sommet en appelant la fonction `traitement` ; alors que s'il est à `False`, alors il faut ajouter tous les sommets fils sur le stack (toujorus dans l'ordre inverse, comme précédemment) avec le booléen à `False` (puisque les fils des fils n'ont pas encore été traités), mais il faut également ajouter le sommet qui vient d'être récupéré, mais cette fois en passant le booléen à `True` puisque ses fils auront été traités la prochaine fois qu'il sera récupéré du stack.

Cependant, comme mentionné ci-dessus, le stack est un LIFO. Dès lors pour s'assurer que le sommet en question ne sera re-récupéré du stack que lorsque ses sommets auront été traités, il faut l'ajouter au stack *avant* ses enfants (i.e. avant la boucle `for`).

> **Proposition 3.4.** *Les quatre fonctions ci-dessus effectuent exactement n appels à la fonction* `traitement`*. En supposant que cette dernière s'exécute en $\Theta(1)$, les quatre fonctions ci-dessus effectunt $\Theta(n)$ opérations.*

Démonstration. Comme déjà vu à maintes reprises précédemment, le parcours visite une unique fois chaque sommet et les traite tous une unique fois. □

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 :

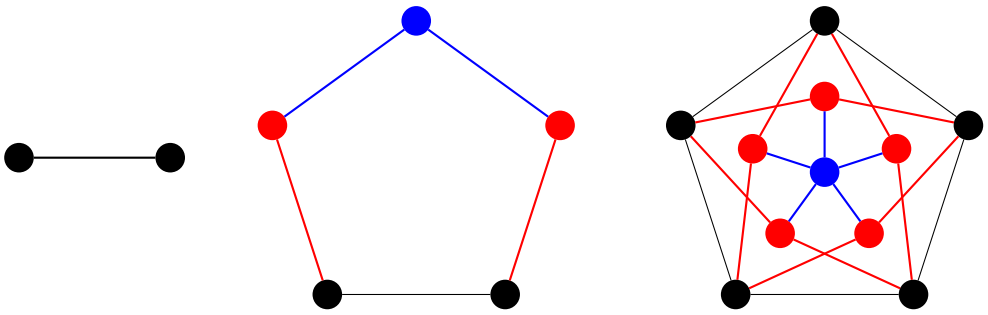
— $V' = V \cup \{v'_1, \ldots, v'_n, w\}$;
— $\forall i, j \in [\![1, n]\!] : \{v_i, v_j\} \in E' \iff \{v_i, v_j\} \in E$;
— $\forall i, j \in [\![1, n]\!] : \{v'_i, v_j\} \in E' \iff \{v_i, v_j\} \in E$;
— $\forall i \in [\![1, n]\!] : \{v'_i, w\} \in E'$.

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):
```

```
        self.vertex(i).add_edge(self.vertex(j))
        self.vertex(j).add_edge(self.vertex(i))
        self.nb_edges_ += 1
```

Résolution. Voici les trois premiers graphes de Mycielski :



(a) $M_0 = K_2$ (b) $M_1 = M(K_2)$ (c) $M_2 = M(M_1)$

Il est clair que si $G = (V, E)$ est un graphe à n sommets et m arêtes, la création de $M(G)$ requiert la création de $n + 1$ nouveaux sommets (les n rouges et l'unique bleu). Si les sommets noirs sont les sommets $\{v_1, \dots, v_n\}$, alors les sommets rouges sont les sommets $\{v'_1, \dots, v'_n\}$ et le sommet noir v_i est lié au sommet rouge v'_j si et seulement si v_i et v_j sont adjacents dans G.

La modification d'un graphe G en son mycielskien $M(G)$ se fait donc en trois étapes :

1. la création des sommets v'_i et du sommet w ;
2. la création des arêtes entre les sommets v_i et v'_j ;
3. la création des arêtes entre w et tous les sommets v'_j.

La reformulation suivante nous permet d'exprimer la création de mycielskien en ne jouant que sur les indices des sommets :

1. ajouter les sommets $v_{n+1}, \dots, v_{2n+1}$ à G ;
2. pour chaque arête $\{v_i, v_j\}$ (pour $i, j \in [\![1, n]\!]$), ajouter l'arête $\{v_{i+n}, v_j\}$ (car v_{i+n} représente v'_i) ;
3. pour tout $i \in [\![1, n]\!]$, ajouter l'arête $\{v_{i+n}, v_{2n+1}\}$ (car v_{2n+1} représente w).

La fonction suivante *transforme* le graphe donné en paramètre en son mycielskien en suivant les trois étapes ci-dessus :

```
def mycielskian(G):
    """
    Transform G into M(G)
    """
    n = G.n
    for _ in range(n+1):
        G.add_vertex()
    for i in range(n):
        v_i = G.vertex(i)
        for v_j in v_i.neighbours:
            j = v_j.idx
            if j >= n:
```

```
13                break
14            G.link(i+n, j)
15        G.link(i+n, 2*n)
```

Notons la présence de la condition `if j >= n: break` aux lignes 12-13 qui permettent de s'assurer que l'on ne crée pas d'arêtes entre deux sommets rouges.

Afin de créer le $n$ème graphe de Mycielski, il suffit de construire $M_0 = K_2$, le graphe complet à deux sommets et puis d'appeler n fois la fonction `mycielskian` sur ce graphe :

```
def mycielski(n):
    Mn = DynamicUndirectedGraph(2)
    Mn.link(0, 1) # M0
    k = 0
    while k < n:
        mycielskian(Mn)
        k += 1
    return Mn
```

Proposition 3.5. *La céation de M_n demande $\Theta(3^n)$ opérations.*

Démonstration. Notons V_n le nombre de sommets de M_n et E_n le nombre d'arêtes de M_n. Les deux relations de récurrence suivante découlent directement de la définition du mycielskien :

$$\begin{cases} V_{n+1} &= 2V_n + 1, \\ E_{n+1} &= 3E_n + V_n. \end{cases}$$

En effet, les sommets de M_{n+1} sont composés des V_n sommets de M_n, des V_n copies de ces sommets et du sommet w additionel. La relation sur le nombre d'arêtes peut se trouver simplement en les comptant. En effet dans les arêtes de M_{n+1}, il y a :

- E_n arêtes de M_n ;
- pour chaque sommet v de M_n, $\deg(v)$ nouvelles arêtes ;
- V_n arêtes liant w aux sommets v'_j.

Par la proposition 2.37, nous pouvons conclure que $E_{n+1} = 3E_n + V_n$.

De là, nous pouvons déduire que $V_n = 3 \cdot 2^n - 1$. En effet :

$$3{\cdot}2^0{-}1 = 2 = V_0 \qquad \text{et} \qquad 2V_n{+}1 = 2(3{\cdot}2^n{-}1){+}1 = 3{\cdot}2^{n+1}{-}2{+}1 = 3{\cdot}2^{n+1}{-}1 = V_{n+1}.$$

Nous pouvons également déterminer l'égalité suivante :

$$\begin{aligned} E_n &= 3^n + \sum_{k=1}^{n} 3^{k-1} V_{n-k} = 3^n + \sum_{k=1}^{n} 3^{k-1} \cdot 3 \cdot 2^{n-k} - \sum_{k=0}^{n-1} 3^k = 3^n + 2^n \sum_{k=1}^{n} \left(\frac{3}{2}\right)^k - \sum_{k=0}^{n-1} 3^k \\ &= 3^n + 2^n \frac{3}{2} \sum_{k=0}^{n-1} \left(\frac{3}{2}\right)^k - \sum_{k=0}^{n} 3^k = 3^n + 2^n \frac{3}{2} 2 \left(\frac{3^n}{2^n} - 1\right) - \frac{1}{2}\left(3^n - 1\right) \\ &= 3^n + 2^n 3 \frac{3^n - 2^n}{2^n} - \frac{1}{2}\left(3^n - 1\right) = 3^n + 3\left(3^n - 2^n\right) + \frac{1}{2}3^n - \frac{1}{2} = 3^n \left(1 + 3 - \frac{1}{2}\right) - 3 \cdot 2^n + \frac{1}{2} \\ &= \frac{1}{2}\left(7 \cdot 3^n + 1\right) - 3 \cdot 2^n \end{aligned}$$

Dès lors il est assez clair que $V_n + E_n \sim \frac{7}{2} \cdot 3^n$.

Pour finir, lorsque la fonction `mycielskian` est appelée sur M_k (pour le transformer en M_{k+1}), elle ajoute les $V_{k+1} - V_k$ sommets manquants et les $E_{k+1} - E_k$ arêtes manquantes, et toutes ces insertions se font en temps $\Theta(1)$. Le nombre total d'opérations est donc :

$$\left(V_0 + E_0 + \sum_{k=0}^{n-1} (V_{k+1} - V_k + E_{k+1} - E_k) \right) \Theta(1) = (V_n + E_n)\,\Theta(1) = \Theta(3^n).$$

□

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.

Résolution.

Le constructeur a besoin de recevoir `m` en paramètre, la valeur qui détermine la taille de la table. Une table `self.T` de taille `m` est initialisée à `None` pour signaler que les cases sont vides :

```
class Dict:
    def __init__(self, m):
        self.T = [None] * m
        self.m = m
        self.n = 0

    def __len__(self):
        return self.n

    @property
    def load_factor(self):
        return self.n / self.m
```

La méthode `__len__` doit renvoyer le nombre d'éléments stockés dans la table, qui est encodé dans l'attribut `self.n`. La propriété `load_factor` doit retourner le facteur de charge qui est donc la proportion de la table qui est non-vide, i.e. $\frac{n}{m}$.

Les fonctions de hachage peuvent s'implémenter comme suit :

```
    def h1(self, k):
        return sum(map(ord, k))

    def h2(self, k):
        ret = 0x1505
        for c in map(ord, k):
            ret = 33*ret + c
        return ret
```

L'insertion d'un élément suit directement le pseudo-code vu au cours théorique : une variable `j` est initialisée à 0 et on incrémente cette valeur tant que $h(k, j) \mod m$ désigne une case non-vide (et que la clef qu'on veut insérer n'apparait pas dans la table). Une fois qu'une telle valeur est trouvée, la *paire* `(clef, valeur)` est insérée à la position $h(k, j) \mod m$.

Il est en effet important de stocker la clef en plus de la valeur car il faut pouvoir retrouver les éléments lors de la recherche ou lors de l'insertion (en cas de remplacement de la valeur stockée actuellement).

```
    def insert(self, k, v):
        h1 = self.h1(k)
        h2 = self.h2(k)
        j = 0
```

```
        while j < len(self.T):
            idx = (h1 + j*h2) % self.m
            if self.T[idx] is None or self.T[idx][0] == k:
                if self.T[idx] is None:
                    self.n += 1
                self.T[idx] = [k, v]
                break
            j += 1
        else:
            raise OverflowError('La table est pleine')
```

La recherche fonctionne de manière similaire : on itère sur une variable `j` jusqu'à trouver un indice $h(k, j) \mod m$ qui contient la clef recherchée. Il faut également noter que si une case vide est trouvée pendant la recherche, il est inutile d'aller plus loin car nous savons que la clef n'apparaîtra pas.

```
    def search(self, k):
        h1 = self.h1(k)
        h2 = self.h2(k)
        j = 0
        while j < len(self.T):
            idx = (h1 + j*h2) % self.m
            if self.T[idx] is None:
                break
            elif self.T[idx][0] == k:
                return self.T[idx][1]
            else:
                j += 1
        raise IndexError(f'Clef inconnue {k}')
```

Les méthodes spéciales `__getitem__` et `__setitem__` appellent respectivement `search` et `insert` :

```
    def __setitem__(self, k, v):
        self.insert(k, v)

    def __getitem__(self, k):
        return self.search(k)
```

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.

Résolution. Il est important de se rendre compte que cette récursion est 4-aire, i.e. chaque appel à la fonction `solve_maze_rec` effectue soit le cas de base, soit 4 appels récursifs. Nous allons dès lors appliquer le canevas de dérécursification d'une fonction d-aire.

Pour cela, nous remarquons que la condition pour le cas de base est :

```
def C(maze, x, y, W, H):
   over = x == W-1 and y == H-1
   is_valid = (0 <= x < W) and (0 <= y < H)
   return over or not is_valid or maze[y][x] != EMPTY
```

et que le traitement de ce cas de base consiste uniquement à marquer la case de fin comme visitée et de potentiellement arrêter les récursions (en renvoyant `True`).

Le pré-traitement `A0` consiste à marquer la case actuelle comme faisant partie du chemin, et le post-traitement effectue l'opération inverse, i.e. remet la case marquée à son état initial.

Les fonction `fi` de modification de l'état reviennent uniquement à partir dans une certaine direction, et il n'y a pas de traitement à faire entre les appels récursifs. Tout cela nous permet donc d'écrire la version dérécursifiée suivante :

```
def solve_maze(maze):
    stack = [(0, (None, None))]
    W = len(maze[0])
    H = len(maze)
    x, y = 0, 0
    while True:
```

```
7           while 0 <= x <= W-1 and 0 <= y <= H-1 and \
8                 (x < W-1 or y < H-1) and \
9                 maze[y][x] == EMPTY:
10              maze[y][x] = '.'
11              stack.append((1, (x, y)))
12              x += DELTA_X[0]
13              y += DELTA_Y[0]
14          if x == W-1 and y == H-1:
15              maze[y][x] = '.'
16              break
17          i, (x, y) = stack.pop()
18          while i == 4:
19              maze[y][x] = EMPTY
20              i, (x, y) = stack.pop()
21          if i == 0:
22              break
23          else:
24              stack.append((i+1, (x, y)))
25              x += DELTA_X[i]
26              y += DELTA_Y[i]
```

Observons que lorsque la fonction `solve_maze_rec` retourne `True`, c'est qu'une solution a été trouvée, et donc la récursivité s'arrête puisque tous les appels récursifs sur la pile vont renvoyer `True`. Dès lors, il faut que la version récursive arrête la recherche dès qu'une solution est trouvée. C'est précisément ce que fait le `break` à la l. 16.

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.

Résolution.

Vecteur creux

Un vecteur creux est un *tableau* dont les entrées sont des paires (i, v) où i est un indice et v est une valeur à enregistrer. Le constructeur doit simplement créer un tel tableau :

```
def __init__(self):
    self.ligne = []
```

Puisque les entrées du tableau sont triées selon la première valeur, on sait que lors d'une insertion on peut utiliser une recherche dichotomique pour savoir où placer (ou remplacer) l'élément voulu. Notons également que si l'indice `j` est plus grand que le plus grand indice stocké dans le tableau (donc `self.ligne[-1][0]`), alors pas besoin de le chercher dans le tableau, il est suffisant d'ajouter simplement la paire `(j, v)` au tableau.

Bien sûr, lors de l'insertion de la paire `(j, v)`, s'il existe un indice `i` tel que `j` est le premier élément de la paire en position `i` (i.e. `self.ligne[i][0] == j`), alors il ne faut pas *ajouter* la paire `(j, v)` au tableau mais bien *remplacer* `self.ligne[i][1]`.

```
def inserer(self, j, v):
    n = len(self.ligne)
    # Si l'élément doit être ajouté à la fin
    if n == 0 or j > self.ligne[-1][0]:
        self.ligne.append((j, v))      # O(1)
        return
    # Sinon on cherche l'endroit où le placer
    idx = self.trouver_indice(j)       # O(log n)
    if self.ligne[idx][0] == j:
        self.ligne[idx] = (j, v)       # O(1)
    else:
        self.ligne.insert(idx, (j, v)) # O(n)
```

La méthode `trouver_indice` fait la recherche dichotomique et est implémentée comme suit :

```
def trouver_indice(self, j):
    lo = 0
    hi = len(self.ligne)-1
    found = False
    mid = lo
    while lo < hi:
        mid = (hi+lo)//2
        if self.ligne[mid][0] == j:
            lo = hi = mid
        elif self.ligne[mid][0] < j:
            lo = mid+1
        else:
            hi = mid
    return lo
```

La recherche peut se faire simplement en utilisant cette même méthode :

```
def rechercher(self, j):
    n = len(self.ligne)
    if n == 0 or j > self.ligne[-1][0]:
        return None
    idx = self.trouver_indice(j) # O(log n)
    if self.ligne[idx][0] == j:
        return self.ligne[idx][1]
    else:
        return None
```

Remarques :

- toute implémentation à l'aide d'un dictionnaire a été sanctionnée car ne correspondait pas à ce qui était demandé ;
- l'insertion et la recherche devaient utiliser une recherche dichotomique ;
- le remplacement d'une valeur devait être géré par la méthode `inserer` ;
- `rechercher` devait retourner `None` si aucun élément de cet indice n'était trouvé, aucune exception ne devait être lancée.

Distance

La fonction `distances` devait déterminer - pour chaque sommet v - quels sommets étaient accessible depuis v, et à quelle distance ils se trouvaient. Pour tout triplet (v, u, d) où v et u sont des sommets tels que u est à distance d de v, il fallait ajouter `u.identifiant` à l'indice (`v.identifiant`, `d`) de la matrice creuse `D`.

Afin de déterminer quels sommets sont accessibles depuis un sommet v, et à quelle distance de v ils se situent, il fallait utilise un parcours en **largeur** (et non en **profondeur**) puisqu'un BFS a la particularité suivante : lorsqu'un sommet u est traité, tous les sommets à distance $d \lneq d(v, u)$ ont **déjà** été traités. Il est donc possible de déterminer aisément la distance entre v et chaque sommet accessible à l'aide d'un unique BFS partant de v.

En supposant que l'on a une variable `sommet` de type `Sommet` qui soit définie (le sommet de départ du BFS), la partie BFS est la suivante :

```
identifiant = sommet.identifiant
queue = deque()
queue.appendleft((sommet, 0))                          # O(1)
sommet.marque = True
# on fait un BFS sur le sommet actuel
while len(queue) > 0:                                  # O(m + n)
    s, d = queue.pop()                                 # O(1)
    if (identifiant + s.identifiant) % 2 == 0:
        l = D.rechercher(identifiant, d)               # O(log n)
        if l is None:
            D.inserer(identifiant, d, [s.identifiant]) # O(1) car on insère
                dans l'ordre croissant
        else:
            l.append(s.identifiant)                    # O(1) (en complexité
                amortie)
    for voisin in s.voisins:
        if not voisin.marque:
            # ne pas oublier de marquer un sommet quand on l'ajoute
            voisin.marque = True
            # ne pas oublier de monter la distance
            queue.appendleft((voisin, d+1))            #O(1)
```

La file est ici un objet de type `collections.deque` (c.f. la doc), mais une implémentation à l'aide d'un objet de type `list` était bien entendu acceptée.

La file contient des paires (s, d) où s est un sommet et d est la distance entre `sommet` et v. Il est nécessaire de sauver la distance d dans la file puisqu'à tout instant des sommets à distance d et $d+1$ peuvent se trouver dans la file.

Ainsi pour chaque paire `(v, d)` traitée pendant ce BFS (qui commence à `sommet`), si l'identifiant de v a la même parité que l'identifiant de `sommet` [1], on regarde s'il y a déjà une liste d'identifiants de sommets dans la matrice creuse `D` à l'indice `(identifiant, d)` (où `identifiant` est donc l'identifiant de `sommet`), si oui on `append` simplement l'identifiant de v à cette liste, et sinon on insère une nouvelle liste contenant uniquement l'identifiant de v à cet indice.

Lorsque le sommet a été traité, on ajoute tous ses voisins non-marqués à la file (en les marquant au passage), comme dans un BFS classique.

Ce bloc de BFS devait simplement être appelé séquentiellement sur tous les sommets de `G` en pensant bien à réinitialiser les marques des sommets avant chaque BFS !

1. Ce que l'on peut vérifier avec `if (identifiant + v.identifiant) % 2 == 0` puisque deux nombres entiers m et n ont la même parité si et seulement si leur somme est paire. En effet si m et n sont pairs, il est clair que leur somme sera paire, si m et n sont impairs, également, et si l'un est pair, et l'autre est impair, alors il est clair que leur somme sera impaire.

Voici la méthode en entier :

```
def distances(G):
    assert isinstance(G, Graphe)
    n = G.n
    sommets = G.sommets
    D = MatriceCreuse(n)
    for i in range(n):                                      # O(n)
        # on réinitialise toutes les marques
        for sommet in sommets:                              # O(n)
            sommet.marque = False
        sommet = sommets[i]
        identifiant = sommet.identifiant
        queue = deque()
        queue.appendleft((sommet, 0))                       # O(1)
        sommet.marque = True
        # on fait un BFS sur le sommet actuel
        while len(queue) > 0:                               # O(m + n)
            s, d = queue.pop()                              # O(1)
            if (identifiant + s.identifiant) % 2 == 0:
                l = D.rechercher(identifiant, d)            # O(log n)
                if l is None:
                    D.inserer(identifiant, d, [s.identifiant]) # O(1) car on
                        insère dans l'ordre croissant
                else:
                    l.append(s.identifiant)                 # O(1) (en
                        complexité amortie)
            for voisin in s.voisins:
                if not voisin.marque:
                    # ne pas oublier de marquer un sommet quand on l'ajoute
                    voisin.marque = True
                    # ne pas oublier de monter la distance
                    queue.appendleft((voisin, d+1))         # O(1)
```

Remarques :

- Un DFS au lieu d'un BFS était incorrect ;
- les marques des sommets doivent être réinitialisées avant chaque BFS ;
- les indices de la matrice creuse étaient à respecter ;
- un marquage des sommets à l'aide d'un dictionnaire comme dans le canevas du cours était accepté à condition qu'il soit suffisamment clair ;
- une recherche des distances ne passant pas par un parcours en largeur (e.g. Floyd-Warshall, voire pire : un backtracking) n'a pas été acceptée ;
- la condition de parité ne doit en aucun cas influencer quels sommets sont considérés dans le parcours, mais uniquement quels sommets sont ajoutés à `D` ;
- lors de l'insertion d'un nouvel identifiant dans une des entrées de `D`, il faut impérativement vérifier que cette entrée est vide, sinon la liste déjà stockée sera remplacée et les données stockées jusque là seront perdues ;
- un unique BFS sur chaque sommet de départ est suffisant : il n'est pas nécessaire de faire un BFS pour chaque sommet de départ **et** chaque distance.

Complexité

Dans le pire des cas, la recherche dans un vecteur creux est en $\mathcal{O}(\log k)$ où k est le nombre d'entrées dans ce vecteur grâce à la recherche dichotomique.

Il faut cependant faire attention pour l'insertion : `list.insert` s'effectue en $\mathcal{O}(k)$ pour k le nombre d'éléments dans la liste puisqu'il faut insérer l'élément à la bonne position et décaler tous les suivants. `list.append` est quant à lui en $\mathcal{O}(1)$ en moyenne (et sa complexité *amortie* dans le pire des cas est également en $\mathcal{O}(1)$). Dès lors, `VecteurCreux.insert` ne s'effectue pas en $\mathcal{O}(\log k)$ dans le pire des cas, mais bien en $\mathcal{O}(k)$. Cependant, le remplacement d'un élément est en $\mathcal{O}(\log k)$ dans le pire des cas, et l'insertion d'un élément d'indice plus grand que tous les indices stockés est en $\mathcal{O}(1)$ car il est traité en priorité.

La fonction `distances` effectue donc un BFS pour chaque sommet, mais le traitement de chacun des sommets requiert une recherche dans un vecteur creux (donc en $\mathcal{O}(\log n)$) et soit un `list.append` (donc en $\mathcal{O}(1)$), soit une insertion dans un vecteur creux qui est en $\mathcal{O}(1)$ car le BFS parcourt les sommets par ordre croissant de distance (qui sert d'indice dans le vecteur creux).

Un BFS traite au plus une fois chaque sommet et chaque arête donc la complexité d'un BFS est $\mathcal{O}(n+m)$ (car les opérations sur la file sont en $\mathcal{O}(1)$ par son implémentation sous la forme d'une liste doublement chaînée). Cependant, le traitement sur chaque sommet doit être ajouté à cette complexité, ce qui donne une complexité en $\mathcal{O}(n \log n + m)$ pour cette implémentation (et non $\mathcal{O}((n+m)\log n)$ puisque le traitement ne se fait que sur les sommets et non sur les arêtes).

Ce BFS étant exécuté sur chacun des n sommets du graphe, la complexité totale de la fonction distance est la suivante : $\mathcal{O}(n(n \log n + m)) = \mathcal{O}(n^2 \log n + nm)$.

Remarques :

- une complexité (correctement justifiée) différente de celle donnée ci-dessus pouvait tout à fait être correcte si elle correspondait à l'implémentation donnée ;
- une complexité doit impérativement être justifiée : (n^3) n'est pas suffisant ;
- Les symboles utilisés dans la complexité doivent être définis ! Il est clair que dans ce contexte, n est le nombre de sommets et m est le nombre d'arêtes puisque ce sont les symboles utilisés dans le code et dans l'énoncé, mais $\mathcal{O}(n^2 t)$ n'est pas valide car t n'a pas été défini ;
- considérer `list.append` comme s'exécutant en $\mathcal{O}(k)$ pour k le nombre d'éléments dans la liste était accepté si explicité ;
- $\mathcal{O}(n^m)$ est **gigantesque** comme nombre d'opérations et n'est absolument pas correct pour un parcours de graphe. Cela impliquerait que traiter un graphe de 100 sommets et 200 arêtes serait de l'ordre de 100 fois plus coûteux que traiter un graphe à 100 sommets et 199 arêtes, ce qui n'est bien évidemment pas le cas. Même si *techniquement* cette réponse est correcte par la définition du O de Landau, elle ne pouvait être acceptée car non-informative sur le nombre réel d'opérations effectuées.

Session — Août 2021

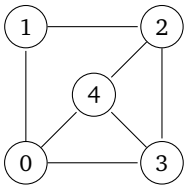
Question d'examen (Q3 août 2021)**.** Soit G un graphe connexe non-dirigé d'ordre n représenté par sa matrice d'adjacence A. On appelle *cycle hamiltonien* un cycle passant une et une seule fois par chacun des sommets de G (sans toutefois obligatoirement passer par toutes les arêtes de G). On dit qu'un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



Résolution. Pour trouver un tel cycle dans le graphe G, nous allons effectuer un backtracking partant du premier sommet ($v = 0$) grâce auquel nous allons construire itérativement tous les chemins partant de v et ne contenant qu'au plus une fois chaque sommet. Notre condition d'arrêt sera la suivante : une fois qu'un chemin de longueur n a été trouvé, c'est qu'il contient tous les sommets de G. Si de plus le dernier sommet de ce chemin est adjacent à v, alors nous avons trouvé un cycle hamiltonien. Attention à bien vérifier cette deuxième condition sinon seul un chemin hamiltonien est trouvé, et l'existence d'un chemin hamiltonien n'implique pas l'existence d'un cycle hamiltonien (savez-vous en trouver un exemple ?).

L'exploration des sommets composant les chemins se fera à l'aide d'un parcours en profondeur somme toute assez classique dans lequel les sommets pris sont marqués (il faut donc bien penser à les démarquer au retour de l'appel récursif) :

```
def _hamilton(A, marks, v, nb_marked):
    if nb_marked == len(A) and A[v][0]:
        return True
    for u in range(len(A)):
        if A[u][v] and not marks[u]:
            marks[u] = True
            if _hamilton(A, marks, u, nb_marked+1):
                return True
            marks[u] = False
    return False

def hamilton(A):
    marks = [True] + ([False] * (len(A)-1))
    return _hamilton(A, marks, 0, 1)
```

Il est clair qu'ici, dès qu'un cycle hamiltonien est trouvé, la pile des appels récursifs va être nettoyée car tous les appels retourneront `True` et donc la condition (l. 7) sera systématiquement vérifiée, menant à une chaîne de `return True`.

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.

Résolution. Cette question d'examen ressemble très fort à l'implémentation itérative de l'exercice coté 1. Il nous faut uniquement considérer la représentation binaire de n puisque les opérations faites dessus à chaque étape correspondent à un modulo 2 et une division par 2. Cette représentation binaire doit, de plus, être parcourue de droite à gauche puisque le cas de base de cette relation de récurrence est le moment où $n = 1$:

```
1  def exponentiate(x, n):
2      ret = 1
3      while n > 0:
4          if n & 1:
5              ret *= x
6              n -= 1
7          else:
8              x *= x
9              n >>= 1
10     return ret
```

Notons que la clause `else` n'est pas nécessaire car si le condition (l. 4) est vérifiée, c'est que n est impair, or n est décrémenté (l. 6) donc en sortant du if, n est obligatoirement pair :

```
1  def exponentiate(x, n):
2      ret = 1
3      while n > 0:
4          if n & 1:
5              ret *= x
6              n -= 1
7          x *= x
```

```
        n >>= 1
    return ret
```

En supposant que toutes les opérations arithmétiques (et logiques pour inclure `&` et `>>`) s'exécutent en $\Theta(1)$, cette implémentation requiert $\Theta(\log n)$ instructions.

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.

Résolution. La relation de récurrence étant déjà donnée, il ne nous reste plus qu'à construire le tableau, déterminer l'ordre d parcours de ce dernier, et implémenter la relation de récurrence. Le tableau doit être une matrice de dimension $(2n+1) \times (n+1)$ afin de pouvoir stocker toutes les valeurs $S(N, k)$ pour $0 \leq N \leq 2n$ et $0 \leq k \leq n$.

Le cas de base est géré dès l'initialisation (i.e. $S[0][0] = 1$ et $S[0][k] = S[N][0]$ pour $N, k > 0$) et la relation de récurrence peut être implémentée en parcourant le tableau ligne par ligne, colonne par colonne puisque lors du calcul de $S[N][k]$, seules les entrées $S[N-1][k]$ et $S[N-1][k-1]$ sont nécessaires :

```
def stirling(n):
    S = [[int(k == N == 0) for k in range(n+1)] for N in range(2*n+1)]
    for N in range(1, 2*n+1):
        for k in range(1, n+1):
            S[N][k] = k*S[N-1][k] + S[N-1][k-1]
    return S[2*n][n]
```

Cette implémentation s'exécute bien en $\mathcal{O}(n^2)$ opérations puisque la table de taille $(2n+1)(n+1) \sim 2n^2$ est créée, et ensuite les $2n^2$ entrées qui ne sont ni sur la première ligne, ni sur la première colonne sont visitées une unique fois et leur valeur est déterminée en $\mathcal{O}(1)$. De plus cette solution propose bien une approche par programmation dynamique puisque chaque entrée est calculée au plus une unique fois, en particulier, une fois calculée, sa valeur est archivée pour être réutilisée plus tard si nécessaire.

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.

Résolution. Ce problème est en réalité un problème de coloration de graphe : deux planètes adjacentes ne peuvent être associées à la même fédération, i.e. deux sommets adjacents ne peuvent être coloriés de la même manière. Il nous faut donc utiliser un backtracking pour trouver une solution (il n'existe aucun algorithme connu pour déterminer si un graphe est k-coloriable en temps polynomial). Ce dernier va créer toutes progressivement les colorations des sommets de G et va avoir comme cas de base le moment où tous les sommets ont une couleur assignée. Le constructeur de la classe doit donc stocker G et k en attributs :

```
class NouvelEspoir:
    def __init__(self, G, k):
        self.G = G
        self.k = k
```

La méthode `solve_galaxy(self)` doit quant à elle initialiser la solution qui va être construite et appeler la méthode de résolution du problème :

```
    def save_galaxy(self):
        self.colours = [None] * len(self.G)
        if not self.solve():
```

```
            print('Aucune solution trouvée')
        else:
            return self.colours
```

Notons que la méthode `solve` renvoie un booléen signalant si une solution a été trouvée ou non. Comme demandé dans l'énoncé, si aucune solution n'a été trouvée, alors un message le signalant est affiché à l'écran.

La méthode `solve` prend donc en paramètre le sommet auquel une couleur est assignée, et va déterminer si cet appel correspond à un cas de base ou non. Si c'en est un (donc si le nombre donné en paramètre ne correspond à aucun sommet du graphe), il faut vérifier si la coloration est valide ou non, ce qui est fait par la méthode `is_solution`. Si par contre `v` correspond bien à un sommet, alors toutes les couleurs lui sont assignées tour à tour et la construction de la solution continue sur le sommet suivant :

```
    def solve(self, v=0):
        if v == len(self.G):
            return self.is_solution()
        for k in range():
            self.colours[v] = k
            if self.solve(v+1):
                return True
            self.colours[v] = None
        return False
```

La méthode `is_solution` doit juste vérifier que toute paire de sommets adjacents ont bien une couleur différente :

```
    def is_solution(self):
        for u in G.vertices():
            for v in G.neighbours(u):
                if self.colours[u] == self.colours[v]:
                    return False
        return True
```

Du point de vue de la complexité, la méthode `solve` va générer tous les vecteurs de coloration possibles. Or il y a exactement k^n tels vecteurs puisque les n sommets peuvent chacun avoir n'importe laquelle des k couleurs. Dès lors la méthode `is_solution` va être appelée k^n fois. Puisque G est représenté de manière dynamique, `G.neighbours(u)` renvoie (en temps constant) un itérateur sur les $\deg(u)$ voisins de u. Or la somme des degrés dans un graphe correspond à deux fois le nombre d'arêtes (par le handshaking lemma ou lemme des poignées de mains, i.e. Proposition 2.37). Dès lors `is_solution` effectue $\mathcal{O}(n+m)$ comparaisons (où m est le nombre d'arêtes de G). Mais par connexité de G, $m \geq n-1$ donc $\mathcal{O}(m+n) = \mathcal{O}(m)$. Finalement nous pouvons en déduire que cette approche requiert $\mathcal{O}(mk^n)$ opérations dans le pire des cas.

Notons qu'ici toutes les combinaisons de couleurs sont générées, même celles ne pouvant pas mener à une solution. De l'élagage (pruning) aurait pu être ajouté à la solution afin de ne considérer pour le sommet `v` que les couleurs n'étant assignées à aucun sommet `u` voisin de `v` tel que `u < v`, mais ceci n'aurait pas permis d'améliorer la complexité dans le

pire des cas.

> **Remarque.** *Il est possible de déterminer si un graphe est k-coloriable en temps $2^n(nk)^{\mathcal{O}(1)}$ dans le pire des cas (ce qui est clairement mieux que k^n), mais l'approche requise dépasse* très largement *le cadre de ce cours et n'était donc pas la solution attendue.*

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.

Résolution. Cet exercice se sépare en deux parties : la première consiste en la recherche du nœud de destination et la seconde qui consiste en l'inversion des valeurs des nœuds sur le chemin entre la racine et le nœud fraichement trouvé.

La fonction `reverse_path` va donc appeler la fonction `find(root, data)` qui renvoie une liste contenant les sommets définissant le chemin entre `root` et le nœud de valeur `data`, dans l'ordre (donc le premier élément est `root` et le dernier est le sommet en question) ; et va ensuite parcourir cette liste et inverser les valeurs. Attention cependant à ne parcourir que la moitié de la liste par symétrie de l'action effectuée (inverser l'ordre) :

```
def find(node, data):
    nodes = [node]
    while node.value != data:
        if node.value < data:
            node = node.right
        else:
            node = node.left
        nodes.append(node)
    return nodes

def reverse_path(root, data):
    nodes = find(root, data)
    for i in range((len(nodes)+1) // 2):
        nodes[i].value, nodes[-i-1].value = nodes[-i-1].value,
            nodes[i].value
```

Notons que la fonction `find` fait juste un parcours classique de BST dans lequel à chaque étape il faut choisir entre visiter le sous-arbre gauche ou le sous-arbre droit en fonction de la valeur du sommet visité et de la valeur recherchée.

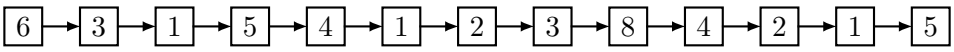
Session — Août 2022

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.

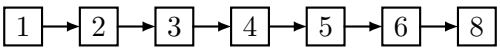
La liste chaînée est implémentée via l'utilisation de la classe `Node` suivante à laquelle vous ne devez pas ajouter des getters/setters/propriétés mais **que vous ne pouvez pas modifier** :

```
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 :



Résolution. L'algorithme *merge sort* (tri fusion) fonctionne de la manière suivante : pour trier un tableau `a` de taille n, soit $n = 1$ en quel cas `a` est déjà trié, soit $n > 1$ donc on peut trier récursivement la partie gauche (d'indices $< \lfloor \frac{n}{2} \rfloor$) et la partie droite (d'indices $\geq \lfloor \frac{n}{2} \rfloor$), et ensuite les deux sous-tableaux triés sont fusionnés en prenant itérativement l'élément minimum de ces tableaux.

Dans le cas présent, `a` n'est pas un tableau mais une liste chaînée. Il faut donc réussir à trouver le milieu, ce qui nécessite plus de travail que simplement diviser la taille par 2. Pour cela, nous allons parcourir notre liste chaînée de deux manières différentes simultanément : une variable `node` va parcourir tous les éléments jusqu'à arriver à la fin de la liste chaînée et un compteur va être incrémenté pour trouver la longueur.

Cependant, nous allons également prendre en paramètre une référence vers le nœud de fin de la liste chaînée afin de ne pas devoir modifier la liste reçue en paramètre :

```
1 def middle(beg, end=None):
2     length = 0
3     node = mid = beg
4     while node is not end:
5         length += 1
6         node = node.next
7         if length % 2 == 0:
```

```
            mid = mid.next
    return mid
```

En avançant le nœud `mid` une itération sur deux, nous savons que `mid` arrivera au milieu de la liste.

Pour implémenter merge sort, il faut donc gérer le cas de base (liste de taille 1) et le cas récursif. Le premier cas doit juste amener au fait de renvoyer une nouvelle liste contenant le même élément que celui reçu en paramètre (donc `return Node(beg.key)`) et le second doit effectuer les appels récursifs et puis procéder à la fusion des deux sous-listes :

```
if length == 1:
    return Node(beg.key)
left = mergesort(beg, mid)
right = mergesort(mid, end)
return merge(left, right)
```

la fusion doit quant à elle fonctionner de la même manière que sur un tableau : on parcourt simultanément les listes gauche et droite en prenant à chaque fois le minimum :

```
def merge(left, right):
    ret = current = None
    while left is not None and right is not None:
        if left.key <= right.key:
            node = Node(left.key)
            left = left.next
        else:
            node = Node(right.key)
            right = right.next
        if ret is None:
            ret = node
        else:
            current.next = node
        current = node
    while left is not None:
        current.next = Node(left.key)
        left = left.next
        current = current.next
    while rightis not None:
        current.next = Node(right.key)
        right = right.next
        current = current.next
    return ret
```

Notons tout de même que les deux dernières boucles `while` (dont seule une peut être exécutée) ne sont pas nécessaire puisque les listes chaînées `left` et `right` sont déjà des copies. Elles peuvent être remplacées par ceci :

```
if left is None:
    current.next = right
```

```
else:
    current.next = left
```

D'ailleurs en utilisant le fait qu'en Python les expressions `None or x` et `x or None` sont évaluées à `x`, nous pouvons écrire simplement `current.next = left or right` (puisque par sortie de la première boucle, au moins une de ces deux variables est à `None`).

Il nous reste encore à gérer la suppression des doublons lors de la fusion (ce qui n'est pas le cas dans le code ci-dessus). Pour cela faisons l'observation suivante : `left` et `right` sont des listes chaînées qui ont été triées par mergesort et qui donc sortent de cette même fonction `merge`. Dès lors chaque valeur ne peut être contenue qu'une unique fois dans `left` et dans `right` (mais une valeur peut apparaître dans les deux). Il nous faut alors séparer les cas où `left.key < right.key`, `left.key > right.key` mais également `left.key == right.key`.

Tout cela mis bout à bout peut donner un code comme le suivant :

```
def mergesort_ll(l: Node):
    return mergesort(l)

def mergesort(beg, end=None):      # trie de beg à end (non compris)
    if beg is end:                 # (sous-)liste vide
        return None
    length = 0
    node = mid = beg
    while node is not end:         # on trouve le milieu
        length += 1
        node = node.next
        if length % 2 == 0:
            mid = mid.next
    if length == 1:                # si un unique élément, pas d'appels
        récursifs
        return Node(beg.key, None)
    left = mergesort(beg, mid)     # sous-liste gauche
    right = mergesort(mid, end)    # sous-liste droite
    return merge_ll(left, right)   # et on fusionne

def merge_ll(left, right):
    ret = current = None
    while None not in (left, right): # tant qu'il reste des éléments à
        fusionner
        x, y = left.key, right.key # on récupère le plus petit à gauche (x)
            et à droite (y)
        if x <= y:                 # on ignore un potentiel doublon
            left = left.next
        if y <= x:
            right = right.next
        node = Node(min(x, y))     # on crée un nouveau noeud
        if ret is None:            # on n'oublie pas le cas de la 1e valeur
            ret = node
```

```
        else:
            current.next = node
        current = node
    current.next = left or right   # on ajoute ce qui reste à gauche (ou à
        droite)
    return ret
```

L'analyse de complexité de cette approche sort du cadre de ce cours.

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.

Résolution. Il nous faut une classe `Graph` représentée de manière dynamique, donc ayant la forme suivante (pour une implémentation, c.f. les séances d'exercices sur les graphes) :

```
class Graph:
    def __init__(self, V, E):
        # peu importe l'implémentation

    def vertices(self):
        # renvoie un itérateur sur G.V
        # O(1)

    def neighbours(self, v):
        # renvoie un itérateur sur les voisins de v
        # O(deg v)
```

Trouver l'excentricité d'un sommet v peut se faire à l'aide d'un parcours en largeur (BFS) depuis ce même sommet v :

```
def excentricite(G, v):
    marks = [False] * len(G)
    q = deque()
    marks[v] = True
    q.appendleft((v, 0))
    ret = 0
    while len(q) > 0:
        v, d = q.pop()
        ret = max(ret, d)
        for u in G.neighbours(v):
            if not marks[u]:
                marks[u] = True
                q.appendleft((u, d+1))
    return ret
```

Dans un BFS, les sommets sont traités par ordre croissant de distance au sommet de départ, mais il nous faut savoir quelle est cette distance précisément. Pour cela, en plus de l'identifiant des sommets, la queue (de type `collections.deque`) doit contenir la distance de v à ces sommets. Puisque l'excentricité de v est la distance maximale entre un sommet $u \in V(G)$ et v, il nous suffit de vérifier à chaque étape si le sommet traité est à une distance plus grande que celle trouvée jusqu'à présent (valeur stockée dans `ret`).

La fonction `rayon` doit quant à elle uniquement trouver la plus petite excentricité, ce qui se fait en appelant la fonction `excentricite` sur tous les sommets et en en prenant le minimum :

```
def rayon(G):
    return min(excentricite(G, v) for v in G.vertices())
```

La complexité du calcul du rayon est donc $\mathcal{O}(nm)$ où n est le nombre de sommets de G et où m est le nombre d'arêtes. En effet la fonction `excentricite` traite une unique fois chaque sommet et chaque arête donc s'exécute en temps $\Theta(m+n)$ (car le traitement de chaque sommet/arête se fait en temps $\mathcal{O}(1)$). Or le graphe est connexe donc $m \geq n-1$, donc $\Theta(m+n) = \mathcal{O}(m)$. Finalement, comme la fonction `excetricite` est appelée exactement une fois par sommet, le traitement total se fait en $n\mathcal{O}(m) = \mathcal{O}(mn)$ opérations.

> **Remarque.** *Remplacer le* `min` *par un* `max` *dans la fonction* `rayon` *permet de trouver le diamètre d'un graphe au lieu de son rayon.*