Cours d'Informatique Commune

Maths Sup

F. Fayard

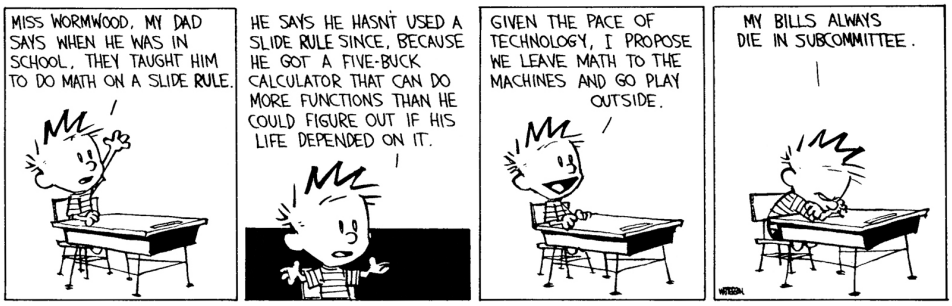
15 septembre 2022

La version de ce document est la A22B6EC.

Table des matières

Chapitre 1

Valeur, type, variable



1.1 Valeur, type

Le langage Python manipule des valeurs de différents *types*. Nous rencontrerons d'abord les types numériques : les *entiers* ainsi que les *nombres flottants* que l'on utilise pour représenter les réels. Nous verrons ensuite les *chaines de caractères*, les *booléens* et les *tuples*.

1.1.1 Nombre entier

Nous utiliserons Python le plus souvent dans ce qu'on appelle le shell ou la boucle interactive. Ce mode est aussi appelé « Repl » pour : Read, Evaluate, Print, Loop. Autrement dit, lorsqu'on entre une expression, Python la lit, l'évalue, affiche le résultat, et est prêt pour l'interaction suivante.

On écrit les entiers de manière naturelle. D'une manière générale, on obtient le type d'une valeur grâce à la fonction `type`.

```
In [1]: 42
Out[1]: 42

In [2]: type(42)
Out[2]: int
```

Le type des entiers est donc `int`. Les opérateurs usuels d'addition « `+` », de soustraction « `-` », de multiplication « `*` » et d'exponentiation « `**` » sont disponibles pour créer des *expressions* qui sont *évaluées* par l'interpréteur. Ces opérateurs possèdent différents niveaux de priorité. L'exponentiation est prioritaire sur la multiplication. L'addition et la soustraction ont la priorité la plus basse.

```
In [3]: 2 + 3 * 5
Out[3]: 17

In [4]: -1 + 2 ** 8
Out[4]: 255
```

On utilise les parenthèses pour grouper différentes sous-expressions lorsque les calculs que l'on souhaite effectuer diffèrent de ceux fixés par les règles de priorité.

```
In [5]: (2 + 3) * 5
Out[5]: 25
```

Les différentes règles de priorité sont parfois subtiles. Il est donc souhaitable, pour des raisons de lisibilité, d'ajouter des parenthèses dès lors que l'évaluation de notre expression repose sur leur connaissance fine. N'oubliez jamais qu'un programme est écrit pour être lu non seulement par un ordinateur, mais aussi par des humains, qu'ils soient programmeurs ou correcteurs de concours. Par exemple, on n'écrira pas `2 ** 2 ** 3` mais plutôt l'une des deux expressions suivantes :

```
In [6]: (2 ** 2) ** 3
Out[6]: 64

In [7]: 2 ** (2 ** 3)
Out[7]: 256
```

Ces opérateurs sont des opérateurs *binaires* : ils nécessitent deux arguments. Le PEP8, qui fixe les règles de bon usage en Python, recommande de mettre un espace de part et d'autre de tels opérateurs. Cependant, lorsqu'on construit des expressions mélangeant des opérateurs ayant différents niveaux de priorité, il est parfois plus lisible d'omettre cet espace autour des opérateurs ayant la priorité la plus forte. Par exemple, on écrira `2**10 - 1`.

Le symbole « `-` », utilisé comme opérateur binaire de soustraction, est aussi utilisé pour la négation. Dans ce cas, c'est un opérateur *unaire* ne nécessitant qu'une opérande.

```
In [8]: -2 * 3
Out[8]: -6
```

Contrairement à ce qui se passe dans la plupart des autres langages comme le C ou OCaml, Python peut représenter des nombres aussi grands que l'on souhaite. Prenons l'exemple du n-ième nombre de Mersenne définit par $M_n \coloneqq 2^n - 1$. Il est courant de chercher des nombres premiers parmi ces entiers. Le dixième nombre de Mersenne qui est premier est M_{89} et son calcul ne pose aucun problème à Python.

```
In [9]: 2**89 - 1
Out[9]: 618970019642690137449562111
```

Python offre deux types de division. Commençons par la division entière. Rappelons le théorème de la division euclienne sur $\mathbb{Z}$:

Proposition 1.1.1

Soit $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$. Alors il existe un unique couple $(q, r) \in \mathbb{Z}^2$ tel que

$$a = qb + r \quad \text{et} \quad 0 \leqslant r < b.$$

q est appelé *quotient* de la division euclidienne de a par b, et r son *reste*.

Par exemple $7 = 2 \times 3 + 1$, donc 2 est le quotient de la division euclidienne de 7 par 3 et son reste est 1. De même $-7 = (-3) \times 3 + 2$, donc -3 est le quotient de la division euclidienne de -7 par 3 et son reste est 2. En Python, on obtient le quotient de la division euclidienne de a par b avec `a // b` et son reste avec `a % b`.

```
In [10]: -7 // 3
Out[10]: -3
```

```
In [11]: -7 % 3
Out[11]: 2
```

La division par 0 est une erreur et lève ce qu'on appelle une *exception*. Même s'il est possible de rattraper les exceptions, nous ne le ferons pas dans ce cours et une division par 0 aura pour effet de produire l'erreur suivante :

```
In [12]: 1 // 0
ZeroDivisionError: integer division or modulo by zero
```

Python offre aussi une division plus classique, notée `/`. Elle produit une valeur d'un type différent : celui des nombres flottants.

```
In [13]: 3 / 2
Out[13]: 1.5

In [14]: type(1.5)
Out[14]: float
```

1.1.2 Nombre flottant

Les nombres flottants sont utilisés pour représenter les nombres réels. Comme tous les langages de programmation, Python utilise le « . » comme séparateur décimal. Pour calculer une valeur approchée de la circonférence d'un cercle de diamètre 2, on entre donc :

```
In [1]: 3.14 * 2.0
Out[1]: 6.28
```

Commençons par remarquer que les opérateurs `+`, `-`, `*`, `/` et `**` sont disponibles pour les nombres flottants. On peut par ailleurs mélanger flottants et entiers dans les calculs. Comme pour les entiers, la division par `0.0` lève une exception.

Pour simplifier l'écriture de grands et de petits nombres, on utilise la notation scientifique. Ainsi, l'âge de l'univers étant estimé à 13.8 milliards d'années et la vitesse de la lumière étant de l'ordre de 3.0×10^8 mètres par seconde, le calcul suivant nous montre qu'il est impossible d'observer des endroits de l'univers à une distance supérieure à 1.31×10^{26} mètres de la terre :

```
In [2]: 13.8e9 * 365 * 24 * 60 * 60 * 3e8
Out[2]: 1.3055904e+26
```

Pour le calcul de la circonférence du cercle de diamètre 2, on a approché plus haut π par 3.14. On pourrait bien sûr utiliser une approximation plus précise comme 3.14159,

```
In [3]: 2 * 3.14159
Out[3]: 6.28318
```

mais la précision disponible avec Python n'est pas illimitée. En première approximation, on peut considérer que Python ne peut travailler qu'avec des nombres flottants ayant une précision de 16 chiffres significatifs : tout excès de précision est ignoré.

```
In [4]: 1234567890.12345678 - 1234567890.1234567
Out[4]: 0.0
```

Le premier nombre possède 18 chiffres significatifs alors que le second en possède 17. Avant même d'effectuer la soustraction, les deux nombres sont arrondis au même nombre : le résultat final est donc nul. La situation est en fait plus complexe que cela, car tout comme les entiers, les flottants ne sont pas représentés en interne en base 10 mais en base 2. Ne soyez donc pas surpris si, en faisant vos propres essais, vous avez parfois l'impression que Python garde 16 chiffres significatifs, parfois 17.

Pour les mêmes raisons, le résultat de chaque opération arithmétique est arrondi. Cela conduit à des résultats surprenants comme le calcul suivant qui n'est pas égal à 10^{-16} comme on pourrait s'y attendre.

```
In [5]: (1.0 + 1.0e-16) - 1.0
Out[5]: 0.0
```

En effet, $1.0 + 10^{-16}$ possède 17 chiffres significatifs. Il est arrondi à 1.0 avant d'effectuer la soustraction qui donne donc 0. Comme les flottants sont stockés en base 2 et non en base 10, même les nombres décimaux les plus simples ne sont pas représentables exactement. On peut donc avoir des résultats surprenants :

```
In [6]: 0.1 + 0.2 - 0.3
Out[6]: 5.551115123125783e-17
```

Vous comprendrez pourquoi les logiciels de comptabilité ne travaillent pas en interne avec des nombres flottants. Ils ont cependant de nombreuses qualités et sont utilisés en simulation numérique ainsi qu'en intelligence artificielle. On n'oubliera cependant jamais que des arrondis sont effectués à chaque opération et nous verrons que les erreurs accumulées peuvent parfois devenir significatives et fausser complètement un résultat.

Lorsqu'on mélange des entiers et des flottants dans une expression, une conversion préalable des entiers vers les flottants est réalisée automatiquement. Cette conversion automatique est un choix raisonnable, car contrairement aux nombres décimaux, les nombres entiers qui ne sont pas trop grands (disons, ceux qui s'écrivent avec moins de 16 chiffres) sont représentables de manière exacte par des flottants. La conversion des flottants vers les entiers est aussi possible. Cependant, comme elle fait perdre de l'information, il faut la demander explicitement en utilisant la fonction `int`. Cette fonction arrondit un flottant au premier entier rencontré lorsqu'on se rapproche de 0.

```
In [7]: int(2.718)
Out[7]: 2

In [8]: int(-2.718)
Out[8]: -2
```

De manière générale, chaque type numérique possède une fonction associée pour forcer une conversion.

Les fonctions usuelles sont disponibles dans la bibliothèque `math`. On y trouve par exemple la fonction `floor` qui, pour chaque nombre x, renvoie sa partie entière dont on rappelle la définition ci-dessous.

Proposition 1.1.2

Soit $x \in \mathbb{R}$. Il existe un unique $n \in \mathbb{Z}$ tel que

$$n \leqslant x < n+1.$$

Cet entier est appelé *partie entière* de x et est noté $\lfloor x \rfloor$.

Pour charger la bibliothèque `math`, on utilise l'instruction suivante :

```
In [9]: import math

In [10]: math.floor(2.718)
Out[10]: 2

In [11]: math.floor(-2.718)
Out[11]: -3
```

De même, on définit la partie entière supérieure d'un réel.

Proposition 1.1.3

Soit $x \in \mathbb{R}$. Il existe un unique $n \in \mathbb{Z}$ tel que

$$n-1 < x \leqslant n.$$

Cet entier est appelé *partie entière supérieure* de x et est noté $\lceil x \rceil$.

La fonction `ceil` de la même bibliothèque permet d'y accéder.

```
In [12]: math.ceil(2.718)
Out[12]: 3

In [13]: math.ceil(-2.718)
Out[13]: -2
```

On peut calculer la racine carrée d'un nombre avec la fonction `sqrt`, abréviation de « square root ». Les autres fonctions usuelles sont aussi disponibles.

```
In [14]: math.sqrt(2.0)
Out[14]: 1.4142135623730951

In [15]: math.exp(1.0)
Out[15]: 2.718281828459045
```

Le logarithme naturel ln (`log` en Python), le logarithme en base 10 (`log10`) et le logarithme en base 2 (`log2`) sont aussi disponibles. Bien sûr, les fonctions trigonométriques circulaires cos, sin et tan sont présentes, tout comme la constante π.

```
In [16]: math.cos(math.pi / 17)
Out[16]: 0.9829730996839018
```

La principale devise de Python est « batteries included ». Autrement dit, de nombreuses bibliothèques (« libraries » en anglais), sont disponibles. Nous venons d'utiliser notre première bibliothèque : le module `math`. Il existe de nombreuses manières de les rendre accessibles. La plus simple est d'écrire `import` suivi du nom de la bibliothèque. Les fonctions et constantes seront alors disponibles, préfixées par le nom du module. Si vous souhaitez seulement en utiliser certaines sans avoir à taper à chaque fois le nom du module, vous pouvez entrer la commande :

```
In [17]: from math import cos, pi

In [18]: cos(pi / 17)
Out[18]: 0.9829730996839018
```

Il est d'ailleurs possible d'importer tous les composants du module `math` avec la commande

```
In [19]: from math import *
```

C'est cependant une opération dangereuse, car si on importe ainsi plusieurs modules, on ne sait rapidement plus d'où viennent nos fonctions. En pratique, il est préférable d'utiliser `import math`, quitte à renommer le module en un nom plus court. On utilise pour cela la commande :

```
In [20]: import math as ma

In [21]: ma.cos(ma.pi / 3)
Out[21]: 0.5000000000000001
```

1.1.3 Chaine de caractères

Python nous permet de travailler avec du texte. Pour cela, on utilise des chaines de caractères que l'on encadre en utilisant soit des « " », soit des « ' ».

```
In [1]: "hello, world"
Out[1]: 'Hello, world'

In [2]: 'Ça dépend, ça dépasse.'
Out[2]: 'Ça dépend, ça dépasse.'
```

Python permet d'utiliser des accents dans les chaines de caractères. Vous pouvez même utiliser des caractères espagnols, allemands, russes, hébreux, arabes ou chinois. Bref, tous les caractères UNICODE sont supportés. De nombreux caractères « spéciaux » existent. Par exemple, le retour à la ligne est un « caractère » que l'on obtient en écrivant « `\n` ». De même, la tabulation est un caractère que l'on obtient en écrivant « `\t` ». La plupart des éditeurs de texte affichent la tabulation en la remplaçant par 2 ou 4 espaces, mais il est important d'être conscient que dans les fichiers textes, c'est un caractère à part entière. Comme il est difficile de le distinguer d'une succession d'espaces, on convient en général de ne pas l'utiliser : beaucoup d'éditeurs de texte insèrent des espaces plutôt qu'un caractère de tabulation lorsque l'on utilise la touche « TAB » de notre clavier. Enfin, si vous souhaitez utiliser des guillemets dans une chaine de caractères et que votre choix du délimiteur vous en empêche, vous pouvez l'insérer avec « `\"` » ou « `\'` ». Par exemple :

```
In [3]: "What do you mean \"ew\"? I don't like Spam!"
Out[3]: 'What do you mean "ew"? I don't like Spam!'
```

Il est possible d'obtenir la longueur d'une chaine de caractères en utilisant la fonction `len`.

```
In [4]: len("hello, world")
Out[4]: 12
```

Les chaines de caractères ont leur propre type : le type `str`.

```
In [5]: type("Il suffit pas d'y dire, y faut aussi y faire.")
Out[5]: str
```

On ne confondra pas les chaines de caractères et les entiers. En particulier, si on essaie d'ajouter un entier à une chaine de caractères en écrivant « `2 + "2"` », on obtient une erreur de type. Cependant, on peut utiliser le symbole `+` entre deux chaines de caractères, ce qui a pour effet de les concaténer :

```
In [6]: "Tic" + "Tac"
Out[6]: 'TicTac'
```

De la même manière, il est possible de multiplier une chaine de caractères par un entier.

```
In [7]: "G" + 10 * "o" + "al"
Out[7]: 'Gooooooooooal'
```

On peut convertir une chaine de caractères en un entier ou un nombre flottant. C'est une conversion de type et il suffit pour cela d'appliquer la fonction portant le nom du type désiré à notre chaine.

```
In [8]: int("2")
Out[8]: 2

In [9]: float("13.1")
Out[9]: 13.1

In [10]: int("2") * float("13.1")
Out[10]: 26.2
```

Si la chaine de caractères ne peut être interprétée comme une valeur du type demandée, une exception sera levée. La conversion inverse est possible avec la fonction `str` :

```
In [11]: "Fahrenheit " + str(45) + str(1)
Out[11]: 'Fahrenheit 451'
```

Le standard ASCII associe une valeur entre 0 et 127 à chacun des caractères les plus courants. Le tableau ci-dessous donne ces valeurs, les cases grisées représentant des caractères non imprimables. Par exemple, le caractère A est associé à la valeur 65.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30				!	"	#	$	%	&	’
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Pour obtenir cet entier, on utilise la fonction `ord`.

```
In [12]: ord('A')
Out[12]: 65
```

On peut obtenir un caractère à partir de son code ASCII grâce à la fonction `chr`.

```
In [13]: chr(65)
Out[13]: 'A'
```

1.1.4 Booléen

Python possède un type booléen qui n'a que deux valeurs distinctes : `True` et `False`.

```
In [1]: True
Out[1]: True

In [2]: type(True)
Out[2]: bool
```

Les opérateurs logiques usuels « et », « ou » et « non » sont disponibles. On remarquera que le « ou » est bien le ou inclusif, comme en mathématiques.

```
In [3]: True and True
Out[3]: True

In [4]: True and False
Out[4]: False

In [5]: True or True
Out[5]: True

In [6]: not True
Out[6]: False
```

Pour savoir si deux valeurs sont égales, on utilise le symbole « == ».

```
In [7]: 1 + 1 == 2
Out[7]: True
```

La valeur renvoyée par un test d'égalité est un *booléen*. Attention à ne jamais utiliser de test d'égalité entre deux flottants, car du fait de l'arithmétique de ces derniers, certains résultats sont surprenants !

```
In [8]: 0.1 + 0.2 == 0.3
Out[8]: False
```

En général, deux valeurs de types différents ne sont pas égales.

```
In [9]: 2 == "2"
Out[9]: False
```

Les opérateurs `!=`, `<`, `>`, `<=`, `>=` sont aussi disponibles.

`x == y`	x est égal à y
`x != y`	x est différent de y
`x < y`	x est strictement inférieur à y
`x > y`	x est strictement supérieur à y
`x <= y`	x est inférieur ou égal à y
`x >= y`	x est supérieur ou égal à y

```
In [10]: 3 <= 3.14 and 3.14 <= 4
Out[10]: True
```

Pour la comparaison des chaines de caractères, l'ordre utilisé est l'ordre lexicographique, chaque caractère étant ordonné dans l'ordre de la table Ascii/Unicode.

```
In [11]: "OL" > "OM"
Out[11]: False
```

Faites attention à l'ordre de ces caractères. Les minuscules sont bien évidemment dans l'ordre alphabétique, tout comme les majuscules, mais la lettre « Z » est avant la lettre « a » et donc de manière surprenante `"Zorro" < "algèbre"`.

Exercice 1

⇨ En utilisant le tableau Ascii, classer ces chaines de caractère dans l'ordre lexicographique : `"9"`, `"34"`, `"Maison"`, `"la"` et `"laisser"`.

1.1.5 Tuple

Afin de grouper plusieurs valeurs, Python propose un type appelé `tuple`.

```
In [1]: (1.0, 2.0)
Out[1]: (1.0, 2.0)

In [2]: "Teddy", "Riner", 1989
Out[2]: ("Teddy", "Riner", 1989)

In [3]: type((2.0, 1.0))
Out[3]: tuple
```

Les parenthèses regroupant ces valeurs sont optionnelles. Il arrive qu'on utilise des tuples pour grouper des valeurs n'ayant pas le même type, comme dans notre second exemple.

1.2 Programmation impérative

1.2.1 Variable

Les valeurs déjà calculées peuvent être gardées en mémoire afin de les utiliser plus tard. Pour cela, on utilise des variables. Les types que nous avons vus jusqu'ici seront plus tard décrits comme *immuables* et lorsqu'on travaille avec de tels types, une variable peut être conceptualisée par une boite portant un *nom* et contenant une *valeur*. Afin de stocker une valeur dans une boite, on utilise le symbole d'*affectation* « `=` ».

```
In [1]: a = 6
```

À gauche du symbole d'affectation, on place le nom de la boite qui doit être utilisée. À droite, on doit trouver une expression qui sera évaluée en une valeur. Cette valeur sera alors stockée dans la boite.

On accède ensuite à la valeur mémorisée en utilisant le nom de la variable. Lors de l'évaluation de chaque expression, les noms de variables sont remplacés par les valeurs qu'elles contiennent.

```
In [2]: a * (a + 1)
Out[2]: 42
```

Exercice 2

⇨ Dans cet exercice on s'interdit d'utiliser l'exponentiation « `**` ».

1. Montrer que l'on peut calculer a^8 avec 3 multiplications.
2. Donner une manière de calculer a^7 avec 4 multiplications.

Une fois qu'une variable est définie, il est possible de la redéfinir en utilisant une nouvelle valeur.

```
In [3]: a = 7

In [4]: a = a + 1

In [5]: a
Out[5]: 8
```

Pour l'entrée `a = a + 1`, le membre de droite est d'abord évalué pour produire la valeur 8. Cette valeur est ensuite stockée dans la variable `a`. L'ancienne valeur est « écrasée » et il n'est plus possible d'y accéder. Ce type d'instruction nous rappelle que le symbole d'affectation est dissymétrique, contrairement au symbole d'égalité utilisé en mathématiques. En particulier, l'instruction « `a + 1 = a` » n'a aucun sens et sera signalée par Python comme une erreur. Remarquons que c'est bien une valeur qui est stockée dans une variable. En particulier, si l'on définit « `b = a` » et que l'on change ensuite la valeur de `a`, celle de `b` reste inchangée.

Exercice 3

⇨ La méthode de Héron est une méthode historique pour obtenir une valeur approchée de la racine carrée d'un nombre $a > 0$. Pour cela, on définit la suite (u_n) par

$$u_0 \coloneqq a, \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad u_{n+1} \coloneqq \frac{u_n + \frac{a}{u_n}}{2}.$$

Déterminer la plus petite valeur de n pour laquelle la précision sur les nombres flottants ne permet plus de distinguer u_n de u_{n+1} lors du calcul de $\sqrt{2}$.

Python est un langage de programmation à *typage dynamique* : une même variable peut à un moment donné stocker un entier et plus tard une chaine de caractères. Le type d'une variable, c'est-à-dire le type de la valeur stockée par cette variable est donc autorisé à changer lors de l'exécution d'un programme. Cette manière de programmer rend cependant les programmes plus difficiles à lire et nous éviterons de le faire.

Pour les noms de variables, nous nous limiterons aux noms composés de lettres minuscules (`a-z`) et majuscules (`A-Z`), ainsi qu'au caractère « tiret du bas » ou « underscore » (`_`) disponible sur la touche 8 des claviers français. L'utilisation de chiffres (`0-9`) à la fin d'un nom est autorisée. On évitera d'utiliser les accents dans les noms de variables. Choisir judicieusement le nom de ses variables est un art qu'il est important de cultiver. Les noms de variables courts ont l'avantage d'être rapides à taper et à lire. On les utilisera donc pour stocker des valeurs que nous utiliserons souvent. Les noms de variables longs ont l'avantage d'être plus descriptifs. On les utilisera donc pour faire référence à des valeurs que nous utiliserons plus rarement. Pour des noms de variables composés de plusieurs mots, on utilise souvent un underscore comme dans `nb_eleves` ou une lettre majuscule comme dans `nbEleves`.

1.2.2 État du système

Contrairement aux expressions dont la finalité est de produire une valeur, une affectation a pour effet de changer l'état des variables. On dit qu'elle agit par *effet de bord*. Pour représenter l'état du système, nous utiliserons la notation suivante `{a: 2, b: 7}`. Elle signale que la variable `a` contient la valeur 2 tandis que `b` contient la valeur 7. La programmation *impérative* consiste à écrire une succession d'instructions pour changer l'état de la machine. Le langage machine, qui est utilisé par les processeurs, fonctionne de cette manière. C'est une des raisons pour lesquelles ce style est central dans de nombreux langages de programmation. C'est le cas pour Python, et c'est un style que nous adopterons souvent dans ce cours. Afin de visualiser l'état dans lequel se trouve la machine, on le décrira sur une ligne de commentaire. Ces lignes commencent par le caractère `#` et sont ignorées par Python.

Si par exemple les variables `a` et `b` contiennent respectivement 2 et 7, les instructions suivantes modifient l'état de la machine comme suit :

```
# etat {a: 2, b: 7}
In [1]: a = b
In [2]: b = a
# etat {a: 7, b: 7}
```

En particulier, ces deux instructions n'ont pas eu pour effet d'échanger le contenu des variables `a` et `b`. La première instruction a eu pour effet d'écraser la valeur contenue dans `a` qui est alors définitivement perdue. Si on possède un verre d'eau et un verre de vin, le meilleur moyen pour échanger le contenu de ces verres est d'utiliser un troisième verre. Pour échanger deux variables, on peut donc utiliser la séquence d'instructions suivante :

```
# etat {a: 2, b: 7}
In [1]: c = a
In [2]: a = b
In [3]: b = c
# etat {a: 7, b: 2, c: 2}
```

Notons que Python permet d'utiliser les `tuple` pour effectuer des affectations « simultanées ». Par exemple après l'instruction

```
In [4]: a, b = 7, 2
```

`a` contient 7 et `b` contient 2. Puisque l'expression de droite est évaluée avant l'affectation, cette construction est très utile pour échanger le contenu de deux variables.

```
In [5]: a, b
Out[5]: (7, 2)

In [6]: a, b = b, a

In [7]: a, b
Out[7]: (2, 7)
```

En pratique, on réservera ces affectations simultanées aux cas où plusieurs affectations les unes à la suite des autres ne permettent pas d'obtenir un résultat similaire.

Notons qu'il est possible de supprimer une variable avec l'instruction `del`, mais nous nous contenterons d'ignorer les variables dont nous n'avons plus l'utilité.

On notera parfois $\mathcal{E}_0, \mathcal{E}_1, \ldots$ l'état du système à différentes étapes de l'exécution de notre code. On utilisera aussi la convention suivante : si `a` est une variable, a_k sera la valeur contenue par cette dernière quand le système est dans l'état $\mathcal{E}_k$. Par exemple

```
# etat0 {a: a0, b: b0}
In [8]: a = a + b
# etat1 {a: a0 + b0, b: b0}
```

signifie qu'après notre affectation $a_1 = a_0 + b_0$ et $b_1 = b_0$.

1.2.3 Entrée, sortie

Le langage Python permet d'interagir avec l'utilisateur en demandant d'entrer des valeurs avec lesquelles il va travailler puis en affichant les résultats de son calcul.

Pour afficher une valeur, on utilise la fonction `print`. On l'utilise pour afficher des chaines de caractères, mais aussi des entiers ou des nombres flottants.

```
In [1]: print("hello, world")
hello, world

In [2]: print(2**10)
1024
```

La fonction `print` travaille par effet de bord. Elle a pour effet de changer l'état du système, à savoir ce qui est affiché par la *console* de l'ordinateur. Il est essentiel de bien faire la différence entre l'expression `2**10` qui s'évalue en 1024 et l'appel `print(2**10)` qui affiche 1024 sur la console. Cet appel renvoie `None`, l'unique valeur du type `NoneType` qui est renvoyée par les fonctions travaillant par effet de bord. On ne voit pas cette valeur sur une ligne `Out` car le shell a pour habitude de ne jamais l'afficher. La différence peut paraitre subtile lorsque l'on travaille avec Python en mode interactif mais elle existe bien :

```
In [3]: 2**10
Out[3]: 1024

In [4]: print(2**10)
1024
```

La fonction `print` peut être utilisée avec plusieurs valeurs, séparées par des virgules : elles sont affichées sur une même ligne, les unes à la suite des autres.

```
In [5]: nb_eleves = 43

In [6]: print("Il y a", nb_eleves, "éleves dans la classe.")
Il y a 43 élèves dans la classe.
```

Par défaut, un retour à la ligne est automatiquement ajouté après chaque appel à `print`. Pour éviter cela, on peut utiliser l'option `end` et remplacer le caractère « `\n` », par la chaine de votre choix. Le plus courant est d'utiliser une chaine vide.

```
In [7]: print("hello, ", end="")
   ...: print("world")
hello, world
```

Enfin, lorsque vous voulez afficher plusieurs valeurs sur une même ligne en les séparant par des virgules, Python va ajouter un espace entre chaque valeur. Cela peut être utile, mais dans les cas où vous ne le souhaitez pas, vous pouvez construire les chaines de caractères à la main.

```
In [8]: n = 3
```

```
In [9]: print("Sup" + str(n) + " rocks!")
Sup3 rocks!
```

La fonction `input` permet quant à elle de demander des valeurs à l'utilisateur. Quelle que soit la valeur attendue, c'est sous la forme d'une chaine de caractères que Python la renvoie au programmeur. Il convient donc d'effectuer explicitement une conversion lorsque l'on souhaite une valeur d'un autre type.

```
In [10]: nom = input("Quel est votre nom ? ")
Quel est votre nom ? Teddy Riner

In [11]: entree = input(nom + ", quelle est votre année de naissance ? ")
Teddy Riner, quelle est votre année de naissance ? 1989

In [12]: annee = int(entree)
In [13]: print("Vous aurez", 2024 - annee, "ans l'année des jeux de Paris.")
Vous aurez 35 ans l'année des jeux de Paris.
```

Bien qu'elle renvoie une valeur, on dit aussi que la fonction `input` travaille par effet de bord, car elle attend une entrée de l'utilisateur. C'est la seule fois dans ce cours où vous verrez cette fonction. Nous sommes en 2022, les téléphones ont des interfaces tactiles et la reconnaissance vocale commence à marcher. Tout cela pour vous dire qu'il vaut mieux laisser gérer l'interface utilisateur par des personnes maitrisant ces technologies. Comme nous travaillerons en mode interactif, `print` et `input` nous seront de toute façon le plus souvent inutiles et nous vous demandons de les laisser de côté, sauf si on vous demande explicitement de les utiliser.

Chapitre 2

Flot d'exécution

2.1 Programmation procédurale

La *programmation procédurale* consiste à découper un programme en fonctions ou procédures élémentaires afin de rendre le programme modulaire. Chaque fonction a une responsabilité bien déterminée. Cela permet la réutilisation du programme ainsi défini : on dit que l'on *factorise* le code. Ainsi, il est plus facile de faire évoluer notre programme en remplaçant par exemple une fonction par une version plus efficace.

2.1.1 Fonction

Dans sa forme la plus simple, une fonction prend en entrée une valeur et en renvoie une autre. Par exemple, la fonction

```
1 def carre(n):
2     return n * n
```

prend en entrée la valeur n et renvoie n^2. On utilise ensuite la fonction de la manière suivante :

```
In [1]: carre(3)
Out[1]: 9
```

Bien entendu, il est possible d'utiliser le résultat renvoyé par une fonction à l'intérieur d'une expression.

```
In [2]: carre(3) + carre(4)
Out[2]: 25
```

Une fonction peut prendre en entrée plusieurs paramètres :

```
1 def somme(a, b):
2     return a + b
```

```
In [3]: somme(3, 5)
Out[3]: 8
```

Les fonctions que nous avons vues jusqu'à présent sont dites *pures*, dans la mesure où elles ne changent pas l'état du système.

Une fonction peut aussi ne rien renvoyer (en pratique elles renvoient `None`, mais c'est un détail que nous pouvons ignorer pour le moment). Elle fonctionne alors par effet de bord ; on dit que c'est une *procédure*. On peut par exemple afficher du texte sur la console :

```
def greetings(nom):
    print("Hello", nom)
```

```
In [4]: greetings("Paul")
Hello Paul
```

2.1.2 Liste

Bien que les listes n'aient pas de lien avec la programmation procédurale, nous les introduisons ici afin d'avoir des exemples plus intéressants dans la suite de ce chapitre. Une liste est une succession ordonnée de valeurs. Pour définir une liste, on énumére ses éléments entre crochets, en les séparant par des virgules. Les listes ont leur type `list` et il est possible de connaitre leur longueur à l'aide de la fonction `len`.

```
In [1]: note = [9, 10, 14]

In [2]: type(note)
Out[2]: list

In [3]: len(note)
Out[3]: 3
```

Si t est une liste de longueur n, ses valeurs sont indexées de 0 à $n-1$ et il est possible d'accéder directement à la valeur d'indice k grâce à `t[k]`. On peut imaginer que ses valeurs sont stockées dans un tableau les unes à la suite des autres : une liste peut ainsi avoir un accès direct à son k-ième élément.

```
In [4]: note[0]
Out[4]: 9

In [5]: moyenne = (note[0] + note[1] + note[2]) / len(note)

In [6]: moyenne
Out[6]: 11.0
```

Si l'on dépasse les bornes d'une liste, Python lève l'exception « list index out of range ». Par exemple `note[3]` va lever une telle exception. Même s'il est possible d'avoir des listes contenant des objets de types différents, en pratique, nous n'utiliserons que des listes constituées d'objets du même type.

Notons que les chaines de caractères ont un comportement comparable aux listes : si s est une chaine de caractères, `s[k]` permet d'accèder au caractère d'indice k. À noter que contrairement à de nombreux langages, il n'existe pas de type « caractère » et `s[k]` est tout simplement une chaine de caractères de longueur 1.

Les listes peuvent contenir d'autres listes. Par exemple, pour représenter une matrice, on utilise le plus souvent une liste formée des listes de ses vecteurs ligne. Par exemple, pour représenter la matrice

$$M \coloneqq \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix} \in \mathcal{M}_{2,3}(\mathbb{R})$$

on utilise :

```
In [7]: m = [[0, 1, 2], [3, 4, 5]]
```

On accède à l'élément $m_{i,j}$ à l'aide de `m[i][j]`. Si `m` représente une matrice à q lignes et p colonnes alors $0 \leqslant i < q$ et $0 \leqslant j < p$, contrairement à l'usage mathématique où $1 \leqslant i \leqslant q$ et $1 \leqslant j \leqslant p$. Si l'on souhaite récupérer le nombre de lignes et de colonnes, il suffit d'écrire :

```
In [8]: q = len(m)

In [9]: p = len(m[0])
```

2.1.3 Ordre d'évaluation

Lors de l'évaluation d'une fonction comportant des expressions comme arguments, Python évalue ces expressions avant d'appeler la fonction. Par exemple, si l'on évalue l'expression `f(2 + 3)`, Python va d'abord évaluer `2 + 3` en `5` puis appeler la fonction `f` avec l'argument `5`. Presque tous les langages de programmation fonctionnent de cette manière et seuls certains langages fonctionnels de niche comme Haskell ont un comportement différent.

Les opérateurs `and` et `or` ont la particularité de fonctionner différemment. Étant donné que `a and b` est faux dès que `a` est faux, l'opérateur `and` évalue d'abord sa première opérande. Dans le cas où celle-ci s'évalue en `False`, *la seconde opérande n'est pas évaluée* et la valeur `False` est renvoyée. On dit que l'opérateur `and` est *paresseux* (*lazy* en anglais). Cette particularité est importante, notamment lorsque l'évaluation de la seconde opérande peut provoquer une erreur si la première est fausse. Par exemple, si $x = 0$, l'expression `x != 0 and 1 / x <= 1` ne lève pas d'exception et s'évalue en `False`. De même, l'opérateur `or` évalue d'abord sa première opérande. Si le résultat de cette évaluation est `True`, la seconde opérande n'est pas évaluée et le résultat est `True`. Si par contre, l'évaluation de la première opérande est `False`, la seconde opérande est évaluée.

Exercice 1

⇨ Quel est le résultat de l'expression `k < len(t) and t[k] == 1` si les variables `k` et `t` contiennent respectivement les valeurs 3 et `[1, 1, 0, 1]` ? Et si `k` contient la valeur 2 ? Si elle contient la valeur 4 ?

2.2 Programmation structurée

En programmation impérative, l'ordre dans lequel les différentes instructions sont exécutées est essentiel. Jusqu'à présent, nous avons écrit des programmes dans lesquels les instructions s'exécutaient les unes après les autres, toujours dans le même ordre. Afin de changer cet ordre, les programmes sont capables d'effectuer des sauts dans ce flot d'instructions. Les premiers langages de programmation utilisaient une instruction nommée `goto` qui leur permettait, sous condition, de sauter d'un endroit à l'autre du programme. Bien que totalement adaptée à la manière dont fonctionne un processeur, cette instruction est beaucoup trop permissive et a abouti à l'écriture de « code spaghetti », difficilement compréhensible par des humains, et donc source de nombreux bugs. Dans un célèbre article publié en 1968 sous le titre « Goto statement considered harmful », Edsger Dijkstra a plaidé pour l'utilisation essentielle d'instructions conditionnelles (`if`) et de boucles (`for`/`while`). Parallèlement, on a montré que ces structures de contrôle suffisent pour l'écriture des programmes les plus complexes. Les années 1970 donnent ainsi naissance à la *programmation structurée*.

2.2.1 Branchement

L'instruction `if` permet de soumettre l'exécution d'une instruction ou d'un bloc d'instructions à une condition.

```
def banquier(solde):
    if solde < 0:
        print("Vous êtes à découvert.")
        print("Veuillez passer à la banque.")
    print("Bonne journée.")
```

```
In [1]: banquier(100)
Bonne journée.

In [2]: banquier(-10)
Vous êtes à découvert.
Veuillez passer à la banque.
Bonne journée.
```

Le bloc d'instructions soumis à condition est délimité par l'*indentation*. Par rapport à l'instruction `if`, on décale d'un même nombre d'espaces chaque instruction faisant partie de ce bloc. Par convention, nous choisissons une indentation de 4 espaces. L'instruction suivant la fin du bloc doit avoir le même niveau d'indentation que l'instruction `if`. Le programme précédent vous souhaitera donc une bonne journée quel que soit l'état de votre compte.

Exercice 2

⇨ Expliquer ce que font les deux fonctions suivantes.

```
def foo(n):
    if n % 2 == 1:
        n = n - 1
```

```
    print(n)

def bar(n):
    if n % 2 == 1:
        n = n - 1
        print(n)
```

Il est possible d'exécuter un autre bloc d'instructions dans le cas où la condition n'est pas vérifiée.

```
def salutation(est_femme):
    if est_femme:
        genre = "Madame"
    else:
        genre = "Monsieur"
    return "Bonjour " + genre + "."
```

```
In [3]: salutation(True)
Out[3]: 'Bonjour Madame.'

In [4]: salutation(False)
Out[4]: 'Bonjour Monsieur.'
```

Enfin, il est possible d'exécuter différents blocs si l'on a plusieurs conditions.

```
def bac(note):
    if note >= 16:
        print("Mention Tres Bien.")
    elif note >= 14:
        print("Mention Bien.")
    elif note >= 12:
        print("Mention Assez Bien.")
    elif note >= 10:
        print("Vous avez votre Bac.")
    else:
        print("Same player shoot again!")
```

```
In [5]: bac(13)
Mention Assez Bien.
```

Dans ce cas, seul le bloc correspondant à la première condition qui est vraie est exécuté.

Exercice 3

⇨ Une agence de voyages propose un voyage organisé où l'on peut s'inscrire en groupe. Le prix par personne est dégressif selon le nombre de personnes : 80 euros pour une ou deux personnes, 70 euros pour 3 à 5 personnes, 60 euros pour 6 à 9 personnes et 50 euros à partir de 10 personnes. On souhaite écrire une fonction ayant pour argument le nombre n de personnes et renvoyant le prix total pour l'ensemble du groupe.

1. Écrire une fonction qui effectue au plus 3 comparaisons à chaque exécution.
2. Écrire une nouvelle fonction qui effectue au plus 2 comparaisons.

2.2.2 Boucle for

Il est possible de répéter plusieurs fois la même séquence d'instructions en utilisant une boucle `for`. Comme pour l'instruction `if`, le bloc d'instructions à exécuter dans la boucle est indenté. La première instruction ne faisant pas partie de la boucle doit utiliser le même niveau d'indentation que la ligne du `for`.

```
def the_shining(n):
    for _ in range(n):
        print("All work and no play")
        print("makes Jack a dull boy.")
    print("Jack Torrance")
```

```
In [1]: the_shining(3)
All work and no play
makes Jack a dull boy.
All work and no play
makes Jack a dull boy.
All work and no play
makes Jack a dull boy.
Jack Torrance
```

Comme le nombre de fois où le corps de la boucle s'exécute est connu avant de rentrer dans la boucle, on parle de boucle *inconditionnelle*. En particulier, nous sommes certains d'en sortir avant même d'y rentrer ; on dit qu'elles sont *bornées*.

Pour calculer le n-ième terme de la suite (u_n) définie par

$$u_0 \coloneqq \alpha \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad u_{n+1} \coloneqq \cos(u_n)$$

on peut utiliser le programme suivant :

```
1 import math
2
3 def suite(alpha, n):
4     u = alpha
5     for _ in range(n):
6         u = math.cos(u)
7     return u
```

```
In [1]: suite(1.0, 1)
Out[1]: 0.5403023058681398

In [2]: suite(1.0, 10)
Out[2]: 0.7442373549005569

In [3]: suite(1.0, 100)
Out[3]: 0.7390851332151608
```

Exercice 4

⇨ On définit la suite de Fibonacci par

$$F_0 \coloneqq 0, \quad F_1 \coloneqq 1, \quad \text{et} \quad \forall n \in \mathbb{N}, \quad F_{n+2} \coloneqq F_{n+1} + F_n.$$

Écrire une fonction `fibo(n)` renvoyant F_n. Notre fonction pourra utiliser deux variables `a` et `b` contenant respectivement les valeurs F_k et F_{k+1}.

Il est souvent utile d'avoir une variable prenant des valeurs entières successives lors d'une boucle. Ainsi, dans le programme suivant, la variable `k` va prendre successivement les 10 valeurs : $0, 1, 2, 3, \ldots, 9$.

```
1 def table(n):
2     for k in range(10):
3         print(k, "*", n, "=", k * n)
```

```
In [4]: table(8)
0 * 8 = 0
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
```

Remarquons que dans les exemples précédents, `_` désigne un nom de variable qu'il est coutume d'utiliser en Python lorsque sa valeur ne nous est pas utile.

Les boucles `for` sont très utiles pour calculer des sommes. Par exemple, pour calculer les premiers termes de la suite (u_n) définie par

$$\forall n \in \mathbb{N}, \quad u_n \coloneqq \sum_{k=1}^{n} \frac{1}{k^2},$$

on utilise la fonction suivante :

```
def suite(n):
    s = 0.0
    for k in range(1, n + 1):
        s = s + 1 / k**2
    return s
```

Dans cette fonction, on dit que `s` est un *accumulateur*.

```
In [1]: suite(1)
Out[1]: 1.0

In [2]: suite(10)
Out[2]: 1.5497677311665408

In [3]: suite(100)
Our[3]: 1.6349839001848923
```

De manière générale, la boucle `for k in range(a, b)` permet à la variable `k` de prendre successivement les valeurs $a, a+1, \dots, b-1$. Dans notre cas, k va prendre les valeurs $1, 2, \dots, n$ pour ajouter les valeurs $1, 1/2^2, \dots, 1/n^2$ à s.

Plus généralement, si $\delta > 0$, `range(a, b, delta)` est utilisé pour boucler sur les entiers $a, a+\delta, a+2\delta$ jusqu'au plus grand entier de la forme $a + k\delta$ strictement inférieur à b.

Exercices 5

⇨ Donner dans chacun des cas suivants les valeurs générées par le `range` :

1. `range(7)`
2. `range(2, 5)`
3. `range(3, 7, 2)`

⇨ Écrire une fonction permettant de calculer la somme de tous les nombres impairs entre 1 et n inclus.

⇨ Écrire une fonction permettant de calculer $n!$.

2.2.3 Réduction

Si l'on souhaite calculer la somme des éléments d'une liste d'entiers a, on peut initialiser une variable `acc` à 0 et lui ajouter successivement tous les éléments de a. On obtient alors la fonction :

```
def somme(a):
    acc = 0
    for i in range(len(a)):
        acc = acc + a[i]
    return acc
```

Pour prouver que cette fonction nous renvoie bien la somme des éléments de a, on définit, pour tout $i \in [\![0, n]\!]$

$$\mathcal{H}_i \coloneqq \text{« La variable } \texttt{acc} \text{ contient la valeur } \sum_{k=0}^{i-1} a_k \text{. »}$$

$\mathcal{H}_0$ est vraie avant de rentrer dans la boucle, et si $\mathcal{H}_i$ est vraie au début du corps de la boucle, alors $\mathcal{H}_{i+1}$ est vraie à la fin du corps de la boucle. Cela prouve que $\mathcal{H}_n$ est vraie en sortie de boucle et donc que la fonction renvoie bien la

valeur souhaitée

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

On dit que $\mathcal{H}_i$ est un *invariant de boucle*.

On peut de même écrire une fonction calculant le produit des éléments d'une liste. Cette fois, la variable `prod` est initialisée à 1. En effet, 0 était l'élément neutre pour l'addition, puisque pour tout $v \in \mathbb{N}$, $0 + v = v$. Son équivalent pour la multiplication est 1, puisque pour tout $v \in \mathbb{N}$, $1 \times v = v$. On écrit donc :

```
def produit(a):
    prod = 1
    for i in range(len(a)):
        prod = prod * a[i]
    return prod
```

Dans ce cas, l'invariant de boucle est

$$\mathcal{H}_i \coloneqq \text{« La variable } \texttt{prod} \text{ contient la valeur } \prod_{k=0}^{i-1} a_k. \text{ »}$$

et prouve que la fonction renvoie bien le produit des éléments de a.

Exercice 6

⇨ Écrire une fonction prenant en entrée une liste de booléens et renvoyant `True` si tous ces booléens sont égaux à `True`, et `False` sinon.

De la même manière, on peut écrire une fonction calculant le plus grand élément d'une liste non vide d'entiers.

```
def maximum(a):
    v_max = a[0]
    for i in range(1, len(a)):
        v_max = max(v_max, a[i])
    return v_max
```

On peut aussi adapter le programme afin qu'il nous renvoie l'indice de ce maximum :

```
def indice_maximum(a):
    v_max = a[0]
    i_max = 0
    for i in range(1, len(a)):
        if a[i] > v_max:
            v_max = a[i]
            i_max = i
    return i_max
```

2.2.4 Boucle while

Les boucles `for` nous ont permis d'exécuter plusieurs fois un bloc d'instructions dans le cas où le nombre d'itérations est connu avant de rentrer dans la boucle. Lorsque ce nombre n'est pas connu à priori, typiquement lorsque l'on doit exécuter un bloc tant qu'une condition est vérifiée, on utilise l'instruction `while`.

Supposons que l'on souhaite calculer la racine carrée entière de $n \in \mathbb{N}$, c'est-à-dire le plus grand $a \in \mathbb{N}$ tel que $a^2 \leqslant n < (a+1)^2$. Autrement dit, on souhaite calculer $\lfloor \sqrt{n} \rfloor$, mais sans utiliser de nombre flottant. Pour cela, on initialise a à 0 et on l'incrémente de 1 tant que $a^2 \leqslant n$. Dès que ce n'est plus le cas, on renvoie $a - 1$ qui est la valeur cherchée. On obtient ainsi le code :

```
def int_sqrt(n):
    a = 0
    while a * a <= n:
        a = a + 1
    return a - 1
```

```
In [1]: int_sqrt(15)
Out[1]: 3
```

Exercices 7

⇨ Après avoir remarqué que

$$ne^{-n} \xrightarrow[n\to+\infty]{} 0,$$

écrire un programme prenant en entrée $\varepsilon > 0$ et permettant de trouver le plus petit entier $n \in \mathbb{N}^*$ tel que $ne^{-n} \leqslant \varepsilon$.

⇨ Montrer comment une boucle `for`

```
for k in range(a, b):
    bloc...................
    ..........d'instructions
```

peut s'écrire à l'aide d'une boucle `while`.

Contrairement aux boucles inconditionnelles pour lesquelles on est assuré de sortir de la boucle, il est possible qu'une boucle conditionnelle ne termine jamais. On dit alors que le programme part en *boucle infinie*.

```
1 def un_jour_sans_fin():
2     while True:
3         print("This is Groundhog day!")
```

```
In [2]: un_jour_sans_fin()
This is Groundhog day!
This is Groundhog day!
This is Groundhog day!
...
```

Vous pouvez interrompre un tel programme en appuyant à la fois sur la touche « Ctrl » et la touche « c ».

Une boucle `while` a donc le défaut de ne pas être assurée de terminer. Il est cependant essentiel de pouvoir prouver que dans les conditions normales d'exécution, votre boucle se termine bien. Pour cela, on cherche souvent une grandeur entière positive qui diminue strictement à chaque itération. Comme il n'existe pas de suite infinie strictement décroissante d'entiers positifs, on aura ainsi prouvé que la boucle termine. Une telle grandeur est appelé un *variant*.

Le calcul du pgcd par l'algorithme d'Euclide est basé sur le principe suivant : si $a, b \in \mathbb{N}$, le pgcd de a et b est a lorsque $b = 0$ et est égal au pgcd de b et du reste de la division euclidienne de a par b lorsque $b > 0$. Le programme suivant permet donc de calculer ce pgcd.

```
1 def pgcd(a, b):
2     while b > 0:
3         a, b = b, a % b
4     return a
```

```
In [3]: pgcd(15, 21)
Out[3]: 3
```

Exercice 8

⇨ Prouver que le programme précédent termine.

2.2.5 Boucles imbriquées

Il est possible d'imbriquer les boucles les unes dans les autres. Vous pouvez par exemple générer les tables de multiplication très facilement de la manière suivante :

```
1 def tables(n):
2     for a in range(2, n + 1):
3         for b in range(2, n + 1):
4             print(a, "*", b, "=", a * b)
```

```
In [1]: tables(4)
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
```

Comme les produits `2 * 3` et `3 * 2` sont égaux, ont peut chercher à limiter ces produits aux cas où $a \leqslant b$. On écrit alors :

```
def tables_bis(n):
    for a in range(2, n + 1):
        for b in range(a, n + 1):
            print(a, "*", b, "=", a * b)
```

```
In [2]: tables_bis(4)
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 3 = 9
3 * 4 = 12
4 * 4 = 16
```

Les boucles imbriquées nous seront utiles pour parcourir les éléments d'une matrice. Notons au passage, qu'il est possible de mélanger les boucles `for` et `while`. Supposons par exemple qu'étant donnée une matrice de 0 et de 1, on souhaite calculer le nombre de lignes possèdant au moins un 1.

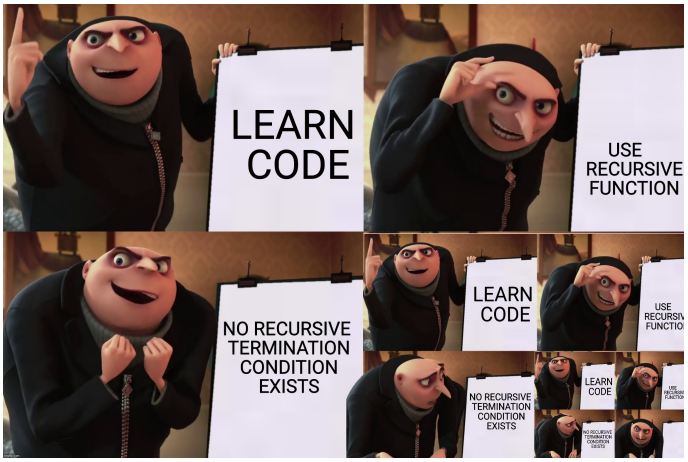
```
def nb_lignes_avec_un(m):
    q = len(m)
    p = len(m[0])
    nb = 0
    for i in range(q):
        j = 0
        while j < p and m[i][j] != 1:
            j = j + 1
        if j < p:
            nb = nb + 1
    return nb
```

Exercice 9

⇨ Écrire une fonction prenant en entrée une matrice de 0 et de 1 et renvoyant l'indice d'une des lignes possédant le plus de 1.

Chapitre 3

Fonction



3.1 Fonction

3.1.1 Fonction

La syntaxe générale d'une fonction en Python est

```
def nom_fonction(arg1, ..., argn):
    bloc...................
    .........d'instructions
```

Le bloc d'instruction a pour vocation soit :
— de calculer une nouvelle valeur qui est renvoyée à l'aide de l'instruction `return`.
— d'avoir un effet de bord comme afficher du texte sur la sortie standard.

Les arguments `arg1`, ..., `argn` sont appelés *arguments formels*. On appelle une fonction à l'aide de la syntaxe `nom_fonction(arg1, ..., argn)` ; les valeurs des expressions `arg1`, ..., `argn` sont appelées *arguments effectifs*.

Une fonction renvoie toujours *une unique* valeur. On utilise pour cela l'instruction `return`. Lorsqu'on souhaite seulement effectuer un effet de bord, on renvoie `None`, ce que Python fait automatiquement s'il ne rencontre pas de `return`.

Python étant un langage de programmation à *typage dynamique*, nous n'avons pas besoin de préciser, ni les types des arguments, ni le type de la valeur de retour. Cette caractéristique du langage nous permet d'avoir des fonctions acceptant des arguments effectifs de types différents :

```
def f(x):
    return x + 1
```

```
In [1]: f(2)
Out[1]: 3

In [2]: f(2.0)
Out[2]: 3.0
```

L'idée est que f peut accepter comme argument n'importe quel type que l'on peut ajouter à 1. En Python, les types `int` et `float` sont de bons candidats. On peut dire que la fonction f accepte un nombre et renvoie un nombre. On dit que Python fonctionne avec le principe du « duck typing » dont la devise est : « If it walks like a duck and it quacks like a duck, then it must be a duck ». Autrement dit, dans notre cas, si $f(x)$ a un sens, c'est que x est un nombre.

Cependant, le plus souvent, nous définirons des fonctions qui ont vocation à être utilisées avec des paramètres d'un type donné. Dans ce cas, la valeur de retour est généralement aussi d'un type déterminé. Par exemple, la fonction

```
def est_pair(n):
    return n % 2 == 0
```

est pensée pour prendre en entrée un entier ; elle renvoie alors un booléen. On dit que la signature de cette fonction est `est_pair(n: int) -> bool`. En Python, si le bloc d'instructions commence par une chaine de caractères, elle est utilisée comme documentation. L'utilisation des triples `"` permet de rentrer des chaines de caractères qui s'étirent sur plusieurs lignes. Il est coutume d'utiliser de telles chaines pour la documentation ; on les appelle *docstrings*.

```
def est_pair(n):
    """est_pair(n: int) -> bool"""
    ans = (n % 2 == 0)
    return ans
```

Cette signature est présente uniquement à titre de documentation et n'est pas lue par Python. Par exemple, rien n'empêche la fonction

```
def suivant(n):
    """suivant(n: int) -> int"""
    return n + 1
```

d'être appelée avec un nombre flottant. Cependant, les fonctions que nous écrirons ne s'utiliseront qu'avec les types suggérés par la docstring.

Une fonction ne peut renvoyer qu'une seule valeur, ce qui est parfois problématique. Supposons par exemple que l'on souhaite écrire une fonction qui nous donne l'heure en fonction du nombre de secondes qui se sont écoulées depuis minuit. On doit pour cela renvoyer trois entiers : h, m et s. Pour cela, on choisit de renvoyer un tuple formé de 3 entiers. Une affectation simultanée permet de déconstruire le tuple lors de l'appel d'une telle fonction.

```
def heure(n):
    """heure(n: int) -> tuple[int, int, int]"""
    s = n % 60
    n = n // 60
    m = n % 60
    h = n // 60
    return h, m, s
```

```
In [3]: h, m, s = heure(42000)
```

3.1.2 Les fonctions comme valeurs

Les fonctions sont des valeurs comme les autres. Leur type est `function`.

```
def next(n):
    """next(n: int) -> int"""
    return n + 1
```

```
In [1]: type(next)
Out[1]: function
```

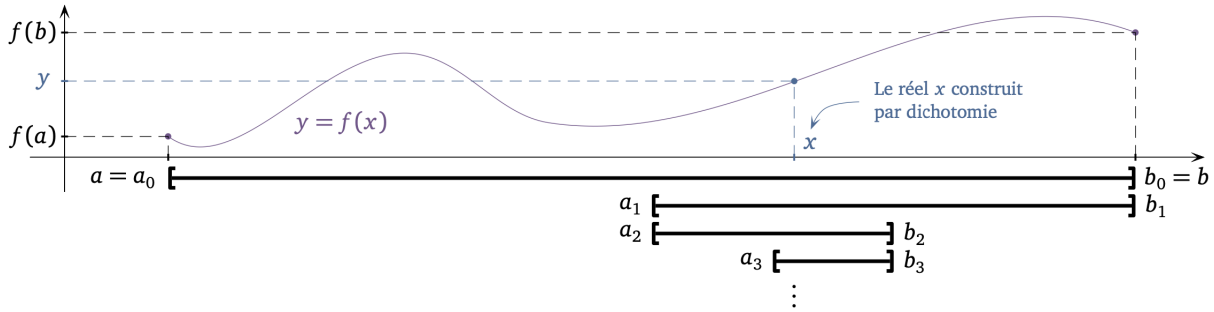
En particulier, il est possible de passer une fonction en argument d'une autre fonction. On peut par exemple définir la fonction suivante qui calcule une approximation de la dérivée d'une fonction.

```
def f(x):
    """f(x: float) -> float"""
    return x**2 - 2

def derive(f, x, eps):
    """derive(f: function, x: float, eps: float) -> float"""
    return (f(x + eps) - f(x)) / eps
```

```
In [2]: derive(f, 1.0, 1.0e-6)
Out[2]: 2.0000009999243673
```

Exercice 1

⇨ Écrire une fonction `dichotomie(f: function, a: float, b: float, y: float, eps: float) -> float` qui prend en argument une fonction continue f telle que $f(a)$ et $f(b)$ sont de part et d'autre de y et renvoyant un couple (u, v) tel que $0 \leqslant v - u \leqslant \varepsilon$ et tel qu'il existe $x \in [u, v]$ tel que $f(x) = y$.



On utilisera pour cela l'algorithme de *dichotomie* qui consiste à définir deux suites (a_n) et (b_n) en commençant par poser $a_0 \coloneqq a$, $b_0 \coloneqq b$. Pour tout $n \in \mathbb{N}$, une fois a_n et b_n définis, on définit a_{n+1} et b_{n+1} de la manière suivante : on commence par calculer $f(c_n)$ où $c_n \coloneqq (a_n + b_n)/2$ et
— si $f(a_n) - y$ et $f(c_n) - y$ sont de signes distincts, on pose $a_{n+1} \coloneqq a_n$ et $b_{n+1} \coloneqq c_n$.
— sinon, on pose $a_{n+1} \coloneqq c_n$ et $b_{n+1} \coloneqq b_n$.
Alors les suites (a_n) et (b_n) sont telles que pour tout $n \in \mathbb{N}$, il existe un $x \in [a_n, b_n]$ tel que $f(x) = y$. De plus $b_n - a_n$ tend vers 0 lorsque n tend vers $+\infty$.

3.1.3 Assertion, test unitaire

Afin de déceler au plus tôt la présence de bugs dans nos programmes, il est important d'écrire des jeux de tests unitaires. Ces tests exécutent une fonction pure avec des arguments dont la valeur de retour est connue ; ils vérifient ainsi leur conformité. Par exemple, pour vérifier que la fonction

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    fac = 1
    for k in range(1, n + 1):
        fac = fac * k
    return fac
```

nous renvoie bien la factorielle de n pour tout $n \geqslant 0$, on pourra vérifier que `factorielle(0)` renvoie bien 1, `factorielle(2)` renvoie bien 2 et `factorielle(5)` renvoie bien 120. Pour cela, on écrira

```
assert factorielle(0) == 1
assert factorielle(2) == 2
assert factorielle(5) == 120
```

en dessous de la définition de notre fonction. De manière générale, le mot clé `assert` est suivi d'une expression qui doit s'évaluer en un booléen : si ce booléen s'évalue en `True`, l'instruction `assert` ne fait rien ; sinon, elle lève une exception, ce qui se traduit par une erreur.

3.1.4 Sortie anticipée

Une fonction peut posséder plusieurs `return` dans son corps. Dans ce cas, le premier `return` rencontré fait sortir de la fonction. Il est courant d'utiliser cette propriété afin d'exprimer simplement certaines boucles. Par exemple, la fonction suivante teste si une chaine de caractères possède un e.

```
1 def possede_un_e(s):
2     """possede_un_e(s: str) -> bool"""
3     for k in range(len(s)):
4         if s[k] == 'e':
5             return True
6     return False
```

Dans cette fonction, la boucle a pour tâche de vérifier les caractères un à un. Si la chaines de caractères s possède un « e », il existe un indice $k \in [\![0, n-1]\!]$ pour lequel `s[k]` est égal à `'e'` ; la fonction va exécuter la ligne 5, renvoyer `True` et sortir immédiatement. Si la chaine de caractères ne contient pas la lettre « e », la condition ligne 4 n'est jamais satisfaite ; on sort alors de la boucle et la dernière instruction renvoie `False`.

```
In [1]: s = "Puis, à la fin, nous saisirons pourquoi tout fut bati à partir d'un
    carcan si dur, d'un canon si tyrannisant. Tout naquit d'un souhait fou, d'un
    souhait nul : assouvir jusqu'au bout la fascination du cri vain, sortir du
    parcours rassurant du mot trop subit, trop confiant, trop commun, n'offrir au
    signifiant qu'un goulot, qu'un boyau, qu'un chas, si aminci, si fin, si aigu
    qu'on y voit aussitot sa justification."

In [2]: possede_un_e(s)
Out[2]: False
```

Ce style de progammation, dans lequel il existe plusieurs points de sortie d'une fonction, peut devenir plus difficile à comprendre. Il est donc découragé d'en abuser, car c'est une source de bugs. Cependant, dans un exemple comme celui-ci, son usage est pleinement justifié.

Remarquons qu'une sortie anticipée casse le caractère inconditionnel d'une boucle `for`, puisqu'on ne sait plus avant de rentrer dans la boucle quel va être le nombre d'itérations. Cependant, elle garde son caractère borné et est toujours assurée, soit de terminer totalement, soit d'être interrompue par un `return`.

Exercice 2

⇨ 1. Écrire une fonction `est_sous_mot_position(sm: str, s: str, k: int) -> bool` prenant deux chaines de caractères `sm` et `m` et renvoyant `True` si `sm` est un sous-mot de `s` commençant à la position k, et `False` sinon. Par exemple `est_sous_mot_position("th", "python", 2)` doit renvoyer `True`. On supposera que le mot `m` est assez grand pour contenir le sous-mot `sm` à partir de la position k.

2. En déduire une fonction `est_sous_mot(sm: str, s: str) -> bool` renvoyant `True` si `sm` est un sous-mot de `m` et `False` sinon.

Notons que l'instruction `break` permet de sortir d'une boucle `for/while` (la plus intérieure s'il y en a plusieurs imbriquées), sans sortir de la fonction. Son utilisation peut se justifier dans quelques rares cas, par exemple si l'on souhaite écrire la fonction `est_sous_mot(sm: str, s: str) -> bool` de l'exercice précédent sans utiliser une fonction auxiliaire :

```
1 def est_sous_mot(sm, s):
2     """est_sous_mot(sm: str, s: str) -> bool"""
3     m = len(sm)
```

```
4       n = len(s)
5       for k in range(n - m + 1):
6           found = True
7           for i in range(m):
8               if sm[i] != s[k + i]:
9                   found = False
10                  break
11          if found:
12              return True
13      return False
```

```
In [3]: est_sous_mot("thon", "python")
Out[3]: True
```

On l'utilisera cependant avec parcimonie, car il rend la compréhension d'un algorithme plus délicate. Il sera en général beaucoup plus simple d'utiliser une fonction auxiliaire utilisant une sortie anticipée.

3.2 Variable locale et globale

3.2.1 Variable locale

Il est possible de créer des variables à l'intérieur d'une fonction. Ces variables sont *locales* à la fonction et sont détruites une fois sorti de cette dernière.

```
1 def puissance_quatre(n):
2     """puissance_quatre(n: int) -> int"""
3     c = n * n
4     return c * c
```

```
In [1]: puissance_quatre(2)
Out[1]: 16

In [2]: c
NameError: name 'c' is not defined
```

Si la variable c est définie avant l'appel de la fonction, sa valeur est masquée lors de l'appel et on la retrouve une fois sorti de la fonction :

```
In [3]: c = 5

In [4]: puissance_quatre(2)
Out[4]: 16

In [5]: c
Out[5]: 5
```

Pour comprendre ce phénomène, il est important de comprendre la notion de masquage des variables : une fois dans la fonction, ligne 4, juste avant d'exécuter l'instruction `return`, l'état du système est le suivant :

```
# sous-état local fonction  {c: 4, n: 2}
# sous-état global          {c: 5}
```

À ce moment précis, l'état du système est la superposition de deux sous-états : le sous-état du dessus a été créé lors de l'appel de notre fonction et celui du dessous correspond au sous-état au moment de l'appel. Cette superposition de sous-états est rendue possible par *la pile d'appels*. La variable à laquelle on accède est par défaut celle qui se situe dans le sous-état *actif*, celui qui est le plus haut sur la pile. Par exemple, une fois à l'intérieur de notre fonction, la variable globale c contenant 5 est masquée par la variable locale de même nom, contenant 4 : c'est cette dernière à laquelle on accède. Une fois l'appel terminé, le sous-état local à l'appel est supprimé et on retrouve notre état initial.

```
# sous-état global          {c: 5}
```

Ces sous-états sont agencés comme une pile d'assiettes : lorsqu'on appelle une fonction, une nouvelle « assiette » est empilée au sommet de la pile ; lorsque cet appel se termine, cette assiette est supprimée.

Ce mécanisme permet à la fonction `puissance_quatre` de n'avoir aucun effet sur l'état au niveau de l'appel. Notons d'ailleurs que la variable n est locale à la fonction : elle est initialisée avec l'argument effectif passé lors de son appel. Comme cette variable est locale, on peut la modifier sans craindre d'effet de bord.

```
1 def puissance_quatre(n):
2     """puissance_quatre(n: int) -> int"""
3     n = n * n
4     return n * n
```

```
In [6]: n = 2

In [7]: puissance_quatre(n)
Out[7]: 16

In [8]: n
Out[8]: 2
```

3.2.2 Variable globale

On appelle *variable globale* toute variable définie dans le niveau le plus bas de la pile. Les *variables locales* sont celles définies dans les niveaux supérieurs. Si elles ne sont pas masquées, il est toujours possible d'accéder en lecture aux variables globales.

```
1 b = 1
2
3 def f(a):
4     """f(a: int) -> int"""
5     return a + b
```

```
In [1]: f(2)
Out[1]: 3
```

Lorsque l'on est dans la fonction f pour calculer `f(2)`, ligne 5, juste avant le `return`, l'état du système est donné par

```
# sous-état local fonction  {a: 2}
# sous-état global                {b: 1}
```

et l'accès à la variable globale b est possible.

Cependant, par défaut, il n'est pas possible de changer la valeur d'une variable qui n'est pas locale. Afin de pouvoir modifier une telle variable, il convient de la déclarer à l'intérieur de la fonction comme *globale* à l'aide du mot clé `global`.

```
1  compteur = 0
2
3  def carre_mission_impossible(n):
4      """carre_mission_impossible(n: int) -> int"""
5      global compteur
6      compteur = compteur + 1
7      if compteur <= 2:
8          return n * n
9      else:
10         return 0
```

```
In [2]: carre_mission_impossible(2)
Out[2]: 4

In [3]: carre_mission_impossible(2)
```

```
Out[3]: 4

In [4]: carre_mission_impossible(2)
Out[4]: 0
```

Cette fonction, si vous l'utilisez, s'autodétruira après 2 appels. Ce genre d'effet est très difficilement compréhensible pour son utilisateur. On dit qu'elle est *impure* car elle a un effet de bord. Comme elle ne renvoie pas `None`, ce comportement est surprenant. C'est pourquoi, la modification de variables globales est un style de programmation à proscrire.

3.2.3 Composition de fonctions

Une fonction peut elle-même appeler une autre fonction. On peut ainsi définir une fonction calculant les coefficients binomiaux à l'aide d'une fonction calculant la factorielle d'un entier.

```
1  def factorielle(n):
2      """factorielle(n: int) -> int"""
3      fac = 1
4      for k in range(1, n + 1):
5          fac = fac * k
6      return fac
7
8  def binome(k, n):
9      """binome(k: int, n: int) -> int"""
10     return factorielle(n) // (factorielle(k) * factorielle(n - k))
```

Lors de l'exécution, la composition de fonctions fait apparaitre une succession de sous-états sur plusieurs niveaux. Voyons cela sur un exemple simple :

```
1  def puissance_deux(n):
2      """puissance_deux(n: int) -> int"""
3      u = n * n
4      return u
5
6  def puissance_quatre(n):
7      """puissance_quatre(n: int) -> int"""
8      u = puissance_deux(n)
9      v = puissance_deux(u)
10     return v
```

```
In [1]: n = 2

In [2]: puissance_quatre(n)
Out[2]: 16
```

Lors du calcul de `puissance_quatre(n)`, ligne 9, on appelle `puissance_deux(u)` et à l'intérieur de cette fonction, ligne 4, juste avant le `return u`, le système est dans l'état suivant :

```
# sous-état local puissance_deux    {n: 4, u: 16}
# sous-état local puissance_quatre  {n: 2, u: 4}
# sous-état global                  {n: 2}
```

La superposition de ces sous-états forme la pile d'appels.

3.3 Programmation récursive

Une fonction *récursive* est une fonction qui s'appelle elle-même. Cette possibilité donne naissance à un style d'algorithmes qu'on appelle programmation récursive. L'idée essentielle derrière ce style est de *réduire* la résolution d'un problème à la résolution de problèmes similaires de tailles strictement inférieures. Pour que ce principe fonctionne, il faut d'une part spécifier des problèmes de tailles élémentaires, qu'on appelle *cas de base*, et donner leurs solutions ; il faut s'assurer d'autre part que les réductions précédentes finissent toujours par rencontrer de tels cas.

3.3.1 Fonction récursive pure

Commençons par un classique de la programmation récursive : le calcul de $n!$.
— *réduction* : Si $n \geqslant 1$, alors $n! = n \times (n-1)!$.
— *cas de base* : Sinon $n = 0$ et $0! = 1$.
La traduction en Python est immédiate.

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    if n >= 1:
        return n * factorielle(n - 1)
    else:
        return 1
```

```
In [1]: factorielle(5)
Out[1]: 120
```

Sur le même principe, nous allons programmer la division euclidienne d'un entier $a \in \mathbb{N}$ par $b \in \mathbb{N}^*$ en n'utilisant que des comparaisons, des additions et des soustractions.
— *réduction* : Si $a \geqslant b$, on note q et r le quotient et le reste de la division euclidienne de $a - b$ par b. Alors

$$a - b = qb + r, \quad \text{donc} \quad a = (q+1)b + r.$$

On en déduit que $q + 1$ et r sont le quotient et le reste de la division euclidienne de a par b.
— *cas de base* : Sinon $0 \leqslant a < b$ et le quotient de la division euclidienne de a par b est 0 et son reste est a.
On obtient ainsi la fonction :

```
def division(a, b):
    """division(a: int, b:int) -> tuple[int, int]"""
    if a >= b:
        q, r = division(a - b, b)
        return (q + 1, r)
    else:
        return (0, a)
```

```
In [2]: division(23, 7)
Out[2]: (3, 2)
```

Exercice 3

⇨ Définir de manière récursive la fonction `puissance(x: int, n: int) -> int` calculant x^n pour $n \in \mathbb{N}$. On utilisera le fait que $x^0 = 1$ et que si $n \geqslant 1$, alors $x^n = x^{n-1}x$.

Tout comme une boucle `while` peut être infinie, une fonction récursive peut ne jamais terminer. Prenons l'exemple de la fonction

```
def est_pair(n):
    """est_pair(n: int) -> bool"""
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return est_pair(n - 2)
```

qui détermine si un entier n est pair ou non. Elle fonctionne parfaitement pour un entier $n \geqslant 0$, mais si on appelle `est_pair(-1)`, la fonction va appeler successivement `est_pair` avec les valeurs $-3, -5, -7$, etc. Elle ne terminera jamais et on obtiendra l'erreur

```
In [3]: est_pair(-1)
RecursionError: maximum recursion depth exceeded in comparison
```

Il faudra donc être attentif, lorsqu'on définit une fonction récursive, à ce que tous les cas se réduisent à un cas de base en un nombre fini d'appels.

L'algorithme d'*exponentiation rapide* permet de calculer efficacement x^n pour $n \in \mathbb{N}$. Cet algorithme se base sur les deux remarques suivantes :

— *réduction* : Si $n > 0$, pour calculer x^n, on effectue la division euclidienne de n par 2. Il existe donc $p \in \mathbb{N}$ et $r \in \{0, 1\}$ tel que $n = 2p + r$. On remarque ensuite que
  — si $r = 0$, c'est-à-dire si n est pair, on a $x^n = (x^p)^2$.
  — si $r = 1$, c'est-à-dire si n est impair, on a $x^n = x(x^p)^2$.

— *cas de base* : On a $x^0 = 1$.

Ces deux remarques conduisent à l'algorithme suivant :

```
1  def expo_rapide(x, n):
2      """expo_rapide(x: int, n: int) -> int"""
3      if n == 0:
4          return 1
5      else:
6          p = n // 2
7          y = expo_rapide(x, p)
8          if n % 2 == 0:
9              return y * y
10         else:
11             return x * y * y
```

L'algorithme d'exponentiation rapide permet de calculer x^n de manière plus efficace que l'algorithme naïf. Le calcul de x^n se fait de manière naïve en $n-1$ multiplications. Cependant, une récurrence immédiate montre qu'avec l'algorithme d'exponentiation rapide, le calcul de x^n pour $n := 2^p$ nécessite seulement $2 + p$ multiplications. Ainsi, l'algorithme naïf a besoin de 1023 multiplications pour calculer x^{1024} alors que l'algorithme d'exponentiation rapide en a besoin seulement de 12, car $1024 = 2^{10}$.

Cet exemple nous permet de réaliser qu'une fonction récursive va le plus souvent créer une pile d'appels conséquente. Par exemple, lors de l'appel de `expo_rapide(3, 4)`, on va finir par appeler récursivement `expo_rapide(3, 0)`. Lors de cet appel, une fois à la ligne 4, juste avant le `return 1`, la pile d'appels est dans l'état suivant :

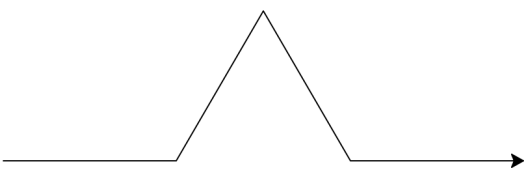
```
# sous-état local expo_rapide(3, 0)  {x: 3, n: 0}
# sous-état local expo_rapide(3, 1)  {x: 3, n: 1, p: 0}
# sous-état local expo_rapide(3, 2)  {x: 3, n: 2, p: 1}
# sous-état local expo_rapide(3, 4)  {x: 3, n: 4, p: 2}
# sous-état global                   {}
```

Lors de l'appel précédent `est_pair(-1)` qui ne terminait pas, l'erreur renvoyée était d'ailleurs liée à cette pile d'appels qui était devenue trop grande. On parle de *débordement de la pile d'appels*, ou de *stackoverflow* en anglais.

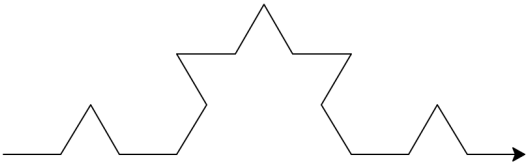
3.3.2 Fonction récursive impérative

Nous allons continuer en programmant le flocon de Von Koch. La génération 0 de ce flocon est un segment de longueur a,

la première génération est la figure suivante, le « triangle » central étant équilatéral,



puis la seconde génération est :



Le but est d'écrire un programme dessinant la n-ième génération du flocon de Von Koch de longueur a. On remarque évidemment le caractère récursif de sa définition.

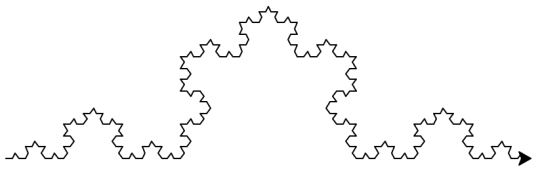
— *réduction* : Si $n \geqslant 1$, on dessine un flocon de Von Koch de longueur $a/3$ et de génération $n-1$, puis on tourne à gauche de 60 degrés, on dessine de nouveau le même flocon de Von Koch, on tourne à droite de 120 degrés, on dessine à nouveau un flocon de Von Koch, on tourne à gauche de 60 degrés et on dessine un dernier flocon.

— *cas de base* : Si $n = 0$, on trace un segment de longueur a.

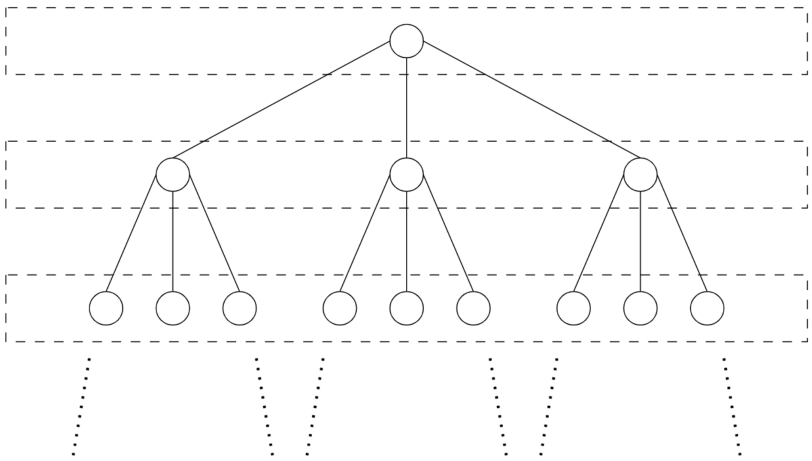
On obtient ainsi le programme suivant :

```
1  import turtle as lg
2
3  def koch(a, n):
4      if n == 0:
5          lg.forward(a)
6      else:
7          koch(a / 3, n - 1)
8          lg.left(60)
9          koch(a / 3, n - 1)
10         lg.right(120)
11         koch(a / 3, n - 1)
12         lg.left(60)
13         koch(a / 3, n - 1)
```

Le tracé de la 4^e génération nous donne



Ce programme récursif est l'occasion de présenter l'arbre d'appels d'une fonction récursive. Au sommet, nous avons la racine qui représente l'appel initial à notre fonction. Cette fonction va elle-même s'appeler plusieurs fois et donc générer des appels représentés dans l'arbre avec une profondeur de 1.

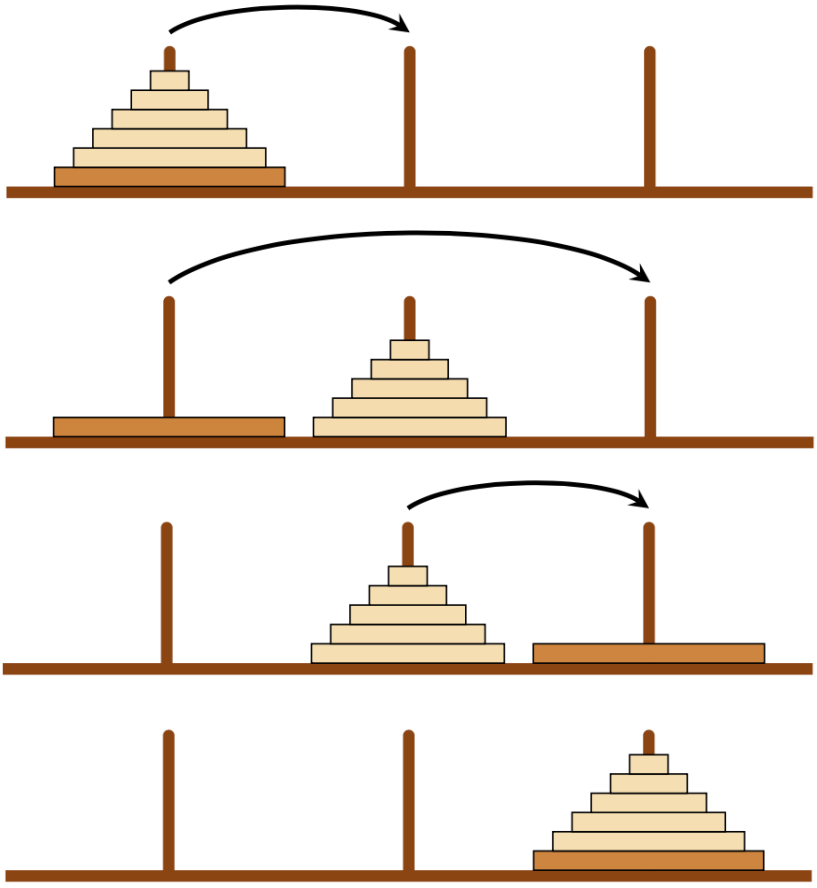


Ces fonctions s'appellent elles-mêmes plusieurs fois, et ainsi de suite.

Nous allons maintenant nous atteler à la résolution d'un grand classique des jeux mathématiques : le jeu des tours de Hanoï, inventé par le mathématicien Edouard Lucas. Ce jeu est constitué de trois tiges sur lesquelles sont enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous positionnés sur la première tige, du plus grand qui est en dessous, au plus petit. L'objectif est de déplacer tous ces disques sur la troisième tige en respectant les règles suivantes :

— On ne peut déplacer qu'un disque à la fois.
— On ne peut pas poser un disque sur un disque de diamètre inférieur.

Raisonnons par récurrence : pour pouvoir déplacer le dernier disque, on déplace les $n-1$ disques qui le couvrent sur la tige centrale. Une fois ces déplacements effectués, nous pouvons déplacer le dernier disque sur la troisième tige. Il reste alors à déplacer les $n-1$ autres disques vers la troisième tige.



Tout est dit : pour pouvoir déplacer n disques de la tige 1 vers la tige 3, il suffit de savoir déplacer $n-1$ disques de la tige 1 vers la tige 2 puis de la tige 2 vers la tige 3. Autrement dit, il suffit de généraliser le problème de manière à décrire le déplacement de n disques de la tige i à la tige k en utilisant la tige j comme pivot. Ceci conduit à la fonction suivante :

```
1 def hanoi(n, i, j, k):
2     """hanoi(n: int, i: int, j: int, k: int) -> NoneType"""
3     if n == 0:
4         return None
5     else:
6         hanoi(n - 1, i, k, j)
7         print("Déplacer le disque au sommet de la tige", i, "vers la tige", k)
8         hanoi(n - 1, j, i, k)
```

```
In [1]: hanoi(3, 1, 2, 3)
Déplacer le disque au sommet de la tige 1 vers la tige 3
Déplacer le disque au sommet de la tige 1 vers la tige 2
Déplacer le disque au sommet de la tige 3 vers la tige 2
Déplacer le disque au sommet de la tige 1 vers la tige 3
```

```
Déplacer le disque au sommet de la tige 2 vers la tige 1
Déplacer le disque au sommet de la tige 2 vers la tige 3
Déplacer le disque au sommet de la tige 1 vers la tige 3
```

Exercice 4

⇨ Déterminer le nombre de mouvements utilisés par l'algorithme précédent pour résoudre le problème des tours de Hanoï à n disques.

3.3.3 Fonctions mutuellement récursives

Il est possible de définir des fonctions *mutuellement récursives* : l'exemple le plus simple serait

$$\begin{cases} f(0) \coloneqq 1 \\ g(0) \coloneqq 0 \\ \forall n \in \mathbb{N}, \quad f(n+1) \coloneqq g(n) \\ \forall n \in \mathbb{N}, \quad g(n+1) \coloneqq f(n) \end{cases}$$

Voici le code Python mettant en oeuvre ces fonctions :

```python
def f(n):
    """f(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return g(n - 1)

def g(n):
    """g(n: int) -> int"""
    if n == 0:
        return 0
    else:
        return f(n - 1)
```

Exercice 5

⇨ De quelles fonctions élémentaires f et g sont-elles des implémentations tordues et inefficaces ?

Nous n'aurons pas très souvent besoin de définir des fonctions mutuellement récursives, mais c'est quand même nécessaire de temps en temps.

Chapitre 4

Structure Séquentielle - Informatique Commune

Si l'on dispose d'une collection ordonnée d'objets comme des relevés de température effectués chaque jour d'un mois ou une liste de vêtements, il est naturel de les regrouper à l'aide d'un objet informatique que l'on appelle *collection*. Un peu comme il existe différentes manières de ranger ses pantalons, il existe différentes façons d'organiser une collection dans la mémoire de l'ordinateur. En informatique, une telle organisation est appelée une *structure de données*.

Prenons l'exemple d'une collection de pantalons que l'on souhaite ranger dans notre placard. La manière la plus courante de les organiser est de les placer en *pile* comme ci-dessous :

Une autre manière de les ranger est d'avoir une succession de compartiments dans lesquels on peut ranger chaque pantalon. En informatique, une telle organisation est appelée un *tableau*.

Chaque organisation a ses avantages et ses inconvénients :

— Le tableau à l'avantage de donner un accès direct à tous les pantalons. Cependant, il ne permet pas d'en rajouter de nouveau.

— La pile permet facilement d'ajouter un nouveau pantalon dans notre collection, en le plaçant au sommet, mais elle ne permet l'accès direct qu'au pantalon placé sur le dessus. Si vous souhaitez accéder soigneusement à un pantalon qui se trouve en dessous, il est nécessaire de dépiler un à un chaque pantalon.

Python propose une structure de données qu'il appelle *liste* et qui essaie de faire la synthèse des deux qualités essentielles des structures de données précédentes : la possibilité d'avoir un accès direct à chaque objet et celle d'ajouter facilement un objet en fin de collection. L'analogie la plus proche en terme de rangement est la suivante

où l'on imagine que les pantalons doivent toujours être placés sur les tiges situées les plus à gauche.

4.1 Liste

4.1.1 Liste

Une liste est une succession ordonnée de n valeurs. La manière la plus simple de définir une liste est d'énumérer ses éléments en les plaçant entre crochets et en les séparant par des virgules. Par exemple, si l'on souhaite définir la liste des températures moyennes en France des 10 premiers jours de juillet 2022, on écrit :

```
In [1]: t = [23, 25, 23, 23, 25, 24, 24, 23, 27, 22]
```

La longueur d'une liste t, c'est-à-dire le nombre d'éléments qu'elle contient, est obtenue avec `len(t)`. On la notera souvent $|t|$. Si t est une liste de longueur n, ses éléments sont indexés de 0 à $n-1$. On imaginera un tableau possédant n cases et contenant les éléments de la collection. Voici par exemple une représentation du tableau précédent de longueur $n = 10$.

23	25	23	23	25	24	24	23	27	22
0	1	2	3	4	5		$\cdots$		$n-1$

Il est important de remarquer que l'indice de la première case est 0 (et non 1 comme dans certains langages de programmation) et que celui de la dernière case est $n-1$ (et non n). On a un accès direct à l'élément d'indice k pour

tout $k \in [\![0, n[\![$ grâce à `t[k]`. Si l'on tente d'accéder à un élément `t[k]` pour $k \geqslant n$, Python lève une exception qui se solde par une erreur.

Python permet aussi d'accéder aux éléments de la liste en les indexant « par la fin ». On utilise pour cela des indices strictement négatifs $k \in [\![-n, 0[\![$.

23	25	23	23	25	24	24	23	27	22
$-n$	$-n+1$		$\cdots$		-5	-4	-3	-2	-1

Cependant, cette manière d'accéder aux listes est à priori proscrite aux concours. On se permettra cependant d'utiliser la syntaxe bien pratique `t[-1]` permettant d'accéder au dernier élément d'une liste.

Les listes Python peuvent contenir des valeurs de types différents. Elles peuvent même contenir d'autres listes.

```
In [2]: t = [9, 3.14159, "Hello", True, [3, 8]]
```

Cependant, contrairement aux `tuple`, l'utilisation des listes se fera essentiellement avec des valeurs ayant le même type. Quels que soient les éléments qu'elle contient, le type d'une liste est `list`, mais dans la signature d'une fonction, on notera `list['a]` pour désigner le type d'une liste d'éléments de même type `'a`.

4.1.2 Parcours de liste

Nous avons vu dans le second chapitre comment parcourir une liste. La manière la plus conventionnelle est d'effectuer une boucle sur un entier k qui va prendre successivement les indices admissibles pour la liste. Par exemple, la fonction suivante va tester si un entier x est un élément de la liste t :

```
1 def est_present(x, t):
2     """est_present(x: int, t: list[int]) -> bool"""
3     for i in range(len(t)):
4         if t[i] == x:
5             return True
6     return False
```

Il est cependant possible d'écrire la même fonction de manière plus concise, en itérant non par sur les indices de la liste, mais directement sur la liste elle-même. Pour cela, on utilise la syntaxe « `for y in t` » : dans ce cas, la boucle va itérer sur les éléments de la liste et y va prendre successivement les valeurs $t_0, t_1, \ldots, t_{n-1}$. La fonction `est_present` peut donc s'écrire ainsi :

```
1 def est_present(x, t):
2     """est_present(x: int, t: list[int]) -> bool"""
3     for y in t:
4         if y == x:
5             return True
6     return False
```

Le fait qu'on puisse écrire « `for y in t` » fait de la liste `t` un objet *itérable*. D'autres objets Python possèdent cette propriété : les chaines de caractères et les tuples. La fonction `possede_un_e` vue au chapitre précédent s'écrit donc :

```
1 def possede_un_e(s):
2     """possede_un_e(s: str) -> bool"""
3     for c in s:
4         if c == 'e':
5             return True
6     return False
```

Les deux styles ont chacun leurs avantages et leurs inconvénients. L'utilisation d'une liste en tant qu'objet itérable a l'avantage de la concision, mais l'utilisation d'un indice et d'un `range` est parfois nécessaire.

Nous avons vu dans le second chapitre des algorithmes de réduction permettant de calculer la somme, le produit ou le maximum des éléments d'une liste. Ces algorithmes sont essentiels et nous ne les redétaillerons pas ici. Nous allons plutôt détailler deux nouveaux algorithmes : l'algorithme de Horner et la méthode de recherche dichotomique dans une liste triée.

Algorithme de Horner

Les listes sont de bons candidats pour représenter les polynômes. Le polynôme $P \coloneqq p_0 + p_1 X + \cdots + p_n X^n$ sera ainsi représenté par la liste $[p_0, p_1, \ldots, p_n]$ de longueur $n+1$. Si x est un nombre, nous allons voir différents algorithmes pour calculer $P(x)$. Si nous nous autorisons l'exponentiation `**`, nous obtenons une première implémentation de cet algorithme :

```
def eval0(p, x):
    """eval0(p: list[int], x: int) -> int"""
    ans = 0
    for k in range(len(p)):
        ans = ans + p[k] * x**k
    return ans
```

Cependant, si on s'intéresse à la performance notre fonction, l'utilisation de `**` est problématique, car il n'existe pas d'instruction calculant la puissance d'un entier sur les processeurs. Python va donc devoir générer du code dont nous n'avons pas le contrôle et dont la performance est donc difficile à estimer. Si nous programmons nous-mêmes une fonction naïve calculant x^k, nous obtenons l'implémentation suivante :

```
def puiss(x, n):
    """puiss(x: int, n: int) -> int"""
    ans = 1
    for _ in range(n):
        ans = ans * x
    return ans

def eval1(p, x):
    """eval1(p: list[int], x: int) -> int"""
    ans = 0
    for k in range(len(p)):
        ans = ans + p[k] * puiss(x, k)
    return ans
```

On peut s'intéresser au nombre de multiplications effectuées par notre algorithme. Le calcul de x^k par la fonction `puiss` nécessite k multiplications, donc le calcul de `p[k] * puiss(x, k)` nécessite $k+1$ multiplications, ce qui donne un cout de

$$C_1(n) \coloneqq \sum_{k=0}^{n} (k+1) = \frac{(n+1)(n+2)}{2}$$

multiplications. Il est possible de faire plus efficace en utilisant le fait qu'on a déjà calculé x^k lorsque l'on a besoin de calculer x^{k+1}. On utilise donc une variable `xk` qui va accumuler le produit des x et contenir x^k.

```
def eval2(p, x):
    """eval2(p: list[int], x: int) -> int"""
    ans = 0
    xk = 1
    for k in range(len(p)):
        ans = ans + p[k] * xk
        xk = xk * x
    return ans
```

Cet algorithme de nécessite que $C_2(n) \coloneqq 2(n+1)$ multiplications. Il est donc bien plus efficace que notre premier algorithme puisque

$$\frac{C_2(n)}{C_1(n)} = \frac{4}{n+2} \xrightarrow[n\to+\infty]{} 0.$$

Donc, plus n devient grand, plus la différence de performance entre les deux algorithmes va se faire sentir. On peut faire encore mieux en remarquant que

$$\begin{aligned} p_n x^n + p_{n-1} x^{n-1} + p_{n-2} x^{n-2} + \cdots + p_1 x + p_0 &= (((p_n x + p_{n-1})x + p_{n-2})x + \cdots + p_1)x + p_0 \\ &= ((((0x + p_n)x + p_{n-1})x + p_{n-2})x + \cdots + p_1)x + p_0. \end{aligned}$$

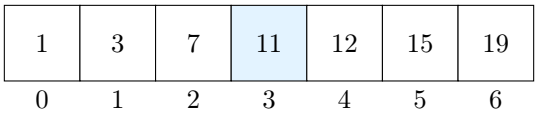
On aboutit à l'algorithme suivant, appelé algorithme de Horner

```python
def eval3(p, x):
    """eval3(p: list[int], x: int) -> int"""
    m = len(p)
    ans = 0
    for k in range(m):
        ans = ans * x + p[m - 1 - k]
    return ans
```

et qui nécessite seulement $C_3(n) \coloneqq n+1$ multiplications, soit deux fois moins que l'algorithme précédent.

Recherche dichotomique

Nous avons vu plus haut comment effectuer une recherche linéaire dans une liste de longueur n en effectuant au plus $C_1(n) \coloneqq n$ comparaisons. Nous allons voir qu'il est possible de faire beaucoup plus efficace lorsque cette liste est triée dans l'ordre croissant. Prenons l'exemple de la liste croissante `t = [1, 3, 7, 11, 12, 15, 19]`.



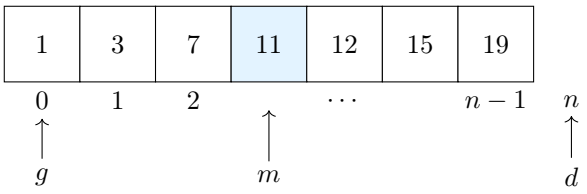
Si nous cherchons l'élément x, on peut commencer à le comparer à $t_3 = 11$.

— Si $x = t_3$, il est présent dans la liste et l'algorithme se termine.

— Si $x < t_3$, puisque la liste est triée dans l'ordre croissant, s'il est présent dans la liste, il est à gauche de 11. Il suffit donc de le chercher parmi les 3 premiers éléments. On va donc comparer x à $t_1 = 3$ et on recommence notre procédé.

Soit x va être égal à un moment à l'un des t_k et le procédé se termine, soit notre sous-liste de travail devient vide ce qui prouve que x ne fait pas partie de la liste initiale.

Pour implémenter notre algorithme, nous allons utiliser deux variables initialisées à $g \coloneqq 0$ et $d \coloneqq n$ (la longueur de notre liste). À chaque itération, la tranche de recherche sera l'ensemble des éléments ayant un indice k tel que $g \leqslant k < d$. On va considérer l'indice $m \coloneqq \lfloor (g+d)/2 \rfloor$ et comparer x à t_m.



— Si $x = t_m$, il est présent dans la liste et l'algorithme se termine.

— Si $x < t_m$, on continue notre recherche dans la tranche $g \leqslant k < m$.

— Si $x > t_m$, on continue notre recherche dans la tranche $m+1 \leqslant k < d$.

Notre algorithme continue tant que la tranche de recherche est non vide, c'est-à-dire tant que $g < d$.

```python
def dichotomie(x, t):
    """dichotomie(x: int, t: list[int]) -> bool"""
    g = 0
    d = len(t)
    while g < d:
        m = (g + d) // 2
        if x == t[m]:
            return True
        elif x < t[m]:
            d = m
        else:                # Cas où x > t[m]
            g = m + 1
    return False
```

Sur notre exemple possédant 7 éléments, l'algorithme de recherche linéaire nécessitait au plus 7 comparaisons alors que la recherche dichotomique nécessite au plus 3 comparaisons. Plus généralement, on montre facilement par récurrence que si t possède $n := 2^p - 1$ éléments, la recherche dichotomique nécessite au plus $p = \log_2(n+1)$ comparaisons. Sur un tableau de $1\ 023 = 2^{10} - 1$ éléments, cela fait seulement 10 comparaisons alors qu'une recherche linéaire peut en nécessiter 1 023. On montrera dans le chapitre sur la complexité que la recherche dichotomique sur un tableau de taille n quelconque nécessite au plus de l'ordre de $C_2(n) := \log_2 n$ comparaisons.

Exercice 1

⇨ Montrer que dans l'algorithme précédent, $d - g$ est un variant de boucle, et donc que l'algorithme termine.

4.1.3 Création de liste

Nous avons vu comment créer des listes contenant quelques éléments. Si l'on souhaite créer des listes plus grandes, diverses méthodes existent.

Concaténation

Comme pour les chaines de caractères, il est possible de concaténer deux listes.

```
In [1]: [7, 2, 1] + [3, 5, 2]
Out[1]: [7, 2, 1, 3, 5, 2]
```

On peut même multiplier des listes par un entier.

```
In [2]: [1, 2, 3] * 3
Out[2]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Cela nous sera notamment utile pour créer une liste formée de n éléments identiques.

```
In [3]: n = 10

In [4]: [0] * n
Out[4]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Compréhension

Pour créer des listes plus complexes, on utilise ce qu'on appelle une *compréhension*.

```
In [1]: n = 5

In [2]: t = [k**2 for k in range(n)]

In [3]: t
Out[3]: [0, 1, 4, 9, 16]
```

Il est possible de créer des listes de listes en imbriquant des listes définies en compréhension.

```
In [4]: [[10 * i + j for i in range(4)] for j in range(3)]
Out[4]: [[0, 10, 20, 30], [1, 11, 21, 31], [2, 12, 22, 32]]
```

Si l'on souhaite, on peut même ne garder que les éléments vérifiant une certaine condition.

```
In [5]: a = [k**2 for k in range(n) if k % 3 != 1]

In [6]: a
Out[6]: [0, 4, 9]
```

Slicing et copie

Il est enfin possible de créer une nouvelle liste en dupliquant la tranche d'une liste déjà existante. On appelle cette opération le *slicing*.

```
In [1]: t = [7, 1, 3, 9]

In [2]: a = t[1:3]

In [3]: a
Out[3]: [1, 3]
```

Plus généralement, si t est de longueur n et $(i,j) \in [\![0,n]\!]^2$ sont tels que $i \leqslant j$, alors `t[i:j]` est la liste composée des objets $t_i, \dots, t_{j-1}$. Lorsque l'indice i est omis, la valeur 0 est utilisée. Si l'indice j est omis, c'est la longueur de la liste qui est utilisée. On peut aussi utiliser la forme plus avancée `t[i:j:p]` où $p > 0$ est le pas : cette tranche est alors formée des valeurs d'indices $i + kp$ pour $i \leqslant i + kp < j$.

Exercice 2

⇨ Étant donné une liste `t`, comment obtenir la liste de tous les éléments d'indices impairs ?

Si l'on souhaite faire une copie d'une liste `t`, on peut utiliser la syntaxe `c = t[:]`. Nous verrons cependant plus tard dans ce chapitre pourquoi cette technique fonctionne très bien pour des listes de booléens, d'entiers ou de flottants mais pose problème pour des listes de listes. Si l'on veut faire une copie de liste de listes, le mieux est d'utiliser la fonction `deepcopy` du module `copy`.

```
In [4]: t = [[1, 2, 3], [4, 5, 6]]

In [5]: import copy

In [6]: c = copy.deepcopy(t)
```

4.1.4 Modification des éléments

Les éléments d'une liste sont accessibles en lecture et en écriture. Il est donc possible de changer leurs éléments.

```
In [1]: note = [9, 10, 14]

In [2]: note[1] = 11

In [3]: note[2] = note[2] + 2

In [4]: note
Out[4]: [9, 11, 16]
```

Une liste de longueur n se comporte donc comme n variables avec lesquelles on peut travailler. Supposons par exemple que l'on souhaite calculer les termes de la suite de Catalan définie par

$$c_0 \coloneqq 1, \quad \text{et} \quad \forall n \in \mathbb{N}, \quad c_{n+1} \coloneqq \sum_{k=0}^{n} c_{n-k} c_k.$$

Les premiers termes de cette suite sont $1, 1, 2, 5, 14, 42$, etc. Pour calculer le n-ième terme de cette suite, on va créer un tableau t de longueur $n+1$ dans lequel on va ranger petit à petit les valeurs de c_k pour $0 \leqslant k \leqslant n$.

```
1  def catalan(n):
2      """catalan(n: int) -> int"""
3      t = [None] * (n + 1)
4      t[0] = 1
5      for i in range(1, n + 1):
6          c = 0
7          for k in range(i):
8              c = c + t[i - 1 - k] * t[k]
9          t[i] = c
10     return t[n]
```

Exercice 3

⇨ Quel est le nombre de multiplications nécessaire au calcul de c_n ?

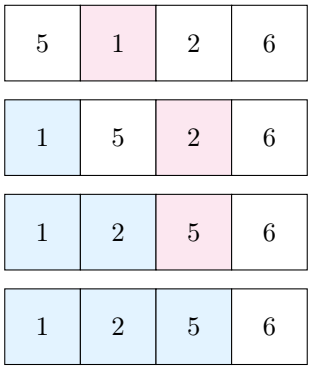
Nous savons que, grâce à la dichotomie, la recherche d'un élément dans une liste triée est considérablement plus rapide que dans une liste quelconque. Nous avions cependant passé sous silence la manière dont on peut trier une liste. De nombreux algorithmes sont dédiés à cette tâche. Les premiers que nous allons étudier s'effectuent « en place », c'est-à-dire qu'ils fonctionnent en effectuant une succession d'échanges d'éléments. Nous utiliserons pour cela la fonction

```
1 def swap(t, i, j):
2     """swap(t: list[int], i: int, j: int) -> NoneType"""
3     t[i], t[j] = t[j], t[i]
```

qui échange les éléments t_i et t_j.

Tri par sélection

Le tri par sélection est le plus simple des algorithmes de tri et fonctionne de la manière suivante :
— On cherche d'abord l'élément le plus petit du tableau et on l'échange avec l'élément d'indice 0.
— On cherche ensuite le plus petit élément d'indice $i \geqslant 1$ du tableau et on l'échange avec l'élément d'indice 1.
On continue ainsi jusqu'à la fin du tableau. Cet algorithme est appelé « tri par sélection » car il fonctionne en sélectionnant de manière répétée le plus petit élément qui n'a pas encore été trié. Par exemple, si on l'applique sur la liste `t = [5, 1, 2, 6]`, on passe par les étapes suivantes :



À chaque étape, les cases bleues représentent les éléments triés ; ils sont donc à la bonne place dans notre tableau. La case rose représente le plus petit élément parmi ceux qui ne sont pas encore triés. Afin de rendre notre programme modulaire, on commence par écrire une fonction `indice_minimum(t: list[int], i: int) -> int` qui renvoie l'indice de l'élément le plus petit parmi les éléments $t_i, t_{i+1}, \ldots, t_{n-1}$.

```
1  def indice_minimum(t, i):
2      """indice_minimum(t: list[int], i: int) -> int"""
3      j_min = i
4      for j in range(i + 1, len(t)):
5          if t[j] < t[j_min]:
6              j_min = j
7      return j_min
8
9  def tri_selection(t):
10     """tri_selection(t: list[int]) -> NoneType"""
11     for i in range(len(t) - 1):
12         j = indice_minimum(t, i)
13         swap(t, i, j)
```

Si l'on souhaite calculer le nombre de comparaisons effectuées par cet algorithme, on constate que pour tout i tel que $0 \leqslant i < n-1$, l'appel `indice_minimum(t, i)` effectue $n-1-i$ comparaisons ligne 5. Cet algorithme effectue donc au total

$$C_s(n) \coloneqq (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$$

comparisons.

Le tri par sélection est une méthode de tri simple qui est facile à comprendre et qui possède les propriétés suivantes :
— *Le nombre de comparaisons ne dépend que de la taille du tableau.* Cette propriété peut être à notre désavantage dans certaines situations. Par exemple, le temps d'exécution du tri par sélection sera le même sur un tableau déjà trié et sur un tableau de nombres aléatoires.

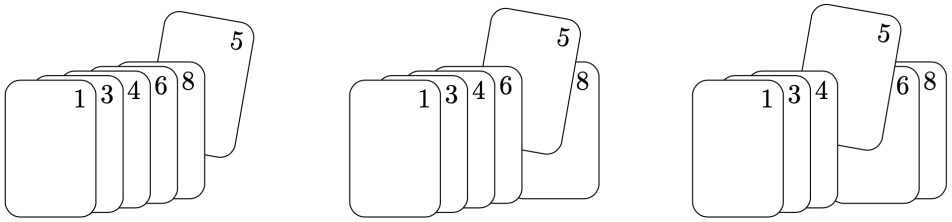
— *Le mouvement des données est minimal.* Chacune des boucles en i n'effectue qu'un échange d'éléments du tableau. Quelle que soit la liste à trier, il y a donc exactement $n-1$ échanges dans un tri par sélection.

Exercice 4

⇨ Donner un invariant de la boucle présente ligne 11 prouvant que le tri par sélection effectue bien un tri de la liste t.

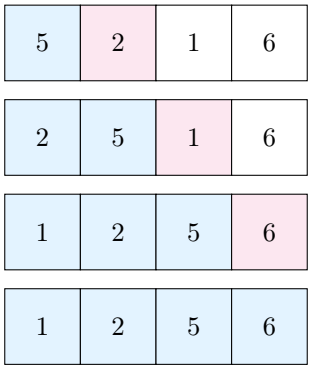
Tri par insertion

Le tri par insertion est l'algorithme que les gens utilisent généralement lorsqu'ils veulent trier les cartes qu'ils ont en main. On insère chaque carte, de la gauche vers la droite, dans la partie déjà triée qui se trouve sur la gauche. Dans une implémentation informatique de cet algorithme, on a besoin d'insérer la carte dans la partie déjà triée en effectuant des échanges successifs qui font remonter petit à petit la carte à insérer à sa position.



Insertion de la carte 5 dans une main déjà triée

Par exemple, si on l'applique sur la liste `t = [5, 2, 1, 6]`, on passe par les étapes suivantes :



À chaque étape, les cases bleues représentent la main triée et la case rose représente la carte que l'on insère par échanges successifs dans notre main triée. Contrairement au tri par sélection, les éléments bleus ne sont pas toujours dans leur position finale puisqu'il est possible que ces éléments doivent plus tard faire place à des éléments plus petits qui seront insérés.

```
1 def tri_insertion(t):
2     """tri_insertion(t: list[int]) -> NoneType"""
3     for i in range(1, len(t)):
4         j = i - 1
5         while j >= 0 and t[j] > t[j + 1]:
6             swap(t, j, j + 1)
7             j = j - 1
```

Si l'on souhaite calculer le nombre de comparaisons effectuées par cet algorithme, on constate que pour tout i tel que $1 \leqslant i < n$, la boucle de la ligne 6 effectue au plus i comparaisons. Cet algorithme effectue donc au total au plus

$$C_i(n) \leqslant 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

comparaisons. Contrairement au tri par sélection, ce nombre de comparaisons dépend non seulement de la taille de la liste, mais aussi de ses éléments.

— Si on effectue un tri par insertion sur une liste déjà triée, il effectue exactement $n-1$ comparaisons.

— Si on trie une liste triée dans l'ordre décroissant, on va effectuer exactement $n(n-1)/2$ comparaisons.

On peut démontrer qu'en moyenne, le tri par insertion effectue un nombre de comparaisons de l'ordre de $n^2/4$.

Le tri par insertion marche assez bien pour des tableaux qui ne sont pas aléatoires et que l'on rencontre souvent en pratique. Pour des tableaux « partiellement triés », il nécessite de l'ordre de n comparaisons. Nous nous contenterons

d'une idée assez floue de ce que signifie ce terme ; on peut dire cependant que les tableaux suivants sont partiellement triés :

— Un petit tableau qui a été ajouté à la fin d'un gros tableau trié.
— Un tableau avec seulement peu d'éléments qui ne sont pas au bon endroit.
— Un tableau dont tous les éléments ne sont pas loin de leur position finale.

En résumé, le tri par insertion est une excellente méthode pour les tableaux qui sont partiellement triés et pour les petits tableaux. Comme ces tableaux arrivent dans des parties intermédiaires d'autres algorithmes de tri plus avancés, nous le retrouverons dans des implémentations efficaces de tels algorithmes.

4.1.5 Ajout et suppression d'éléments

Si nous revenons à notre analogie où une liste fonctionne comme un porte-pantalon, l'image suivante correspond à une liste de 3 éléments :

Il est aisé d'ajouter un quatrième pantalon à côté du dernier. Les listes permettent d'effectuer efficacement cette opération avec la méthode `append`.

```
In [1]: lst = [7, 2, 1]

In [2]: lst.append(5)

In [3]: lst
Out[3]: [7, 2, 1, 5]
```

La syntaxe de cette fonction diffère de ce que nous avons vu pour le moment. Cette syntaxe est héritée de ce qu'on appelle la *programmation orientée objet* et c'est la raison pour laquelle on parle de *méthode* et non de *fonction*. Cependant, la différence entre ces deux notions sera purement syntaxique en prépa où nous n'étudierons pas la programmation orientée objet : il faut simplement imaginer que ce que nous écrivons `lst.append(5)` aurait pu s'écrire `append(lst, 5)`.

Remarquons que les instructions `t = t + [x]` et `t.append(x)` ajoutent toutes les deux x à la fin de t. Cependant, la première instruction crée une nouvelle liste alors que la seconde modifie la liste déjà existante. Outre le fait qu'une modification de liste est bien plus efficace que la création d'une nouvelle liste, ces deux opérations ne sont pas interchangeables. En pratique, la première instruction n'est presque jamais utile et son utilisation doit être considérée avec beaucoup de suspicion.

L'opération inverse, qui consiste à enlever le dernier élément d'une liste, est aussi disponible grâce à la méthode `pop`. Cette méthode agit de deux manières :

— Elle fonctionne par effet de bord et enlève le dernier élément de la liste.
— Elle renvoie la valeur de cet élément.

```
In [4]: x = lst.pop()

In [5]: lst
Out[5]: [7, 2, 1]

In [6]: x
Out[6]: 5
```

Bien entendu, si la valeur de cet élément ne nous intéresse pas, il est possible d'appeler `lst.pop()` et d'ignorer la valeur renvoyée. Enfin, si la liste `lst` est vide, un appel à `lst.pop()` va lever une exception et donc signaler une erreur.

Notons enfin l'existence d'une méthode `extend` qui fonctionne de manière similaire à la méthode `append` mais qui ajoute tous les éléments d'une liste à la fin d'une liste existente.

```
In [7]: lst.extend([9, 3, 5])

In [8]: lst
Out[8]: [7, 2, 1, 9, 3, 5]
```

Afin de mettre en oeuvre la méthode `append`, supposons que l'on souhaite écrire une fonction qui renvoie la liste de tous les nombres premiers inférieurs ou égaux à n. Nous remarquons que si $k \geqslant 2$ et si nous avons la liste de tous les nombres premiers strictement inférieurs à k, il est facile de savoir si k est premier : il suffit de voir si un des nombres premiers dont on dispose divise k. Ce principe nous permet d'écrire l'algorithme suivant

```
1  def admet_diviseur(n, t):
2      """admet_diviseur(n: int, t: list[int]) -> bool"""
3      for p in t:
4          if n % p == 0:
5              return True
6      return False
7
8  def liste_premiers(n):
9      """liste_premiers(n: int) -> list[int]"""
10     lst = []
11     for k in range(2, n + 1):
12         if not admet_diviseur(k, lst):
13             lst.append(k)
14     return lst
```

On obtient ainsi

```
In [9]: liste_premiers(20)
Out[9]: [2, 3, 5, 7, 11, 13, 17, 19]
```

Tri fusion

L'algorithme de tri que nous allons étudier dans cette section est basé sur une opération simple appelée *fusion* : regrouper deux listes ordonnées en une liste ordonnée plus grande. Cette opération conduit à une méthode de tri récursive de la famille « diviser pour régner » appelée tri fusion. Pour trier une liste, on divise la liste en deux parties, on trie les deux parties de manière récursive et on fusionne les deux résultats.

entrée	1	5	2	9	0	8	4	3
moitié de gauche triée	1	2	5	9	0	8	4	3
moitié de droite triée	1	2	5	9	0	3	4	8
fusion	0	1	2	3	4	5	8	9

Nous nous intéressons d'abord à l'algorithme de fusion. Pour cela, nous créons une liste vide à laquelle nous allons ajouter petit à petit les éléments des listes t_1 et t_2 par ordre croissant. Une fois qu'une des listes est épuisée, il suffit d'ajouter les éléments de l'autre liste.

```
1  def fusion(t1, t2):
2      """fusion(t1: list[int], t2: list[int]) -> list[int]"""
3      t = []
4      i1 = 0
5      i2 = 0
6      while i1 < len(t1) and i2 < len(t2):
7          if t1[i1] < t2[i2]:
8              t.append(t1[i1])
9              i1 = i1 + 1
10         else:
```

```
            t.append(t2[i2])
            i2 = i2 + 1
    t.extend(t1[i1:])
    t.extend(t2[i2:])
    return t
```

Une fois l'algorithme de fusion écrit, le tri fusion s'écrit facilement.

— *cas de base* : Si la liste est vide ou ne possède qu'un seul élément, elle est triée.

— *réduction* : Si la liste est de de taille n, on la découpe en deux listes de tailles respectives $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$. On trie ces listes de manière récursive que l'on fusionne ensuite.

On obtient alors l'implémentation suivante :

```
def tri_fusion(t):
    """tri_fusion(t: list[int]) -> list[int]"""
    n = len(t)
    if n <= 1:
        return t[:]
    n1 = n // 2
    t1 = tri_fusion(t[0:n1])
    t2 = tri_fusion(t[n1:n])
    t = fusion(t1, t2)
    return t
```

Nous verrons dans le chapitre sur la complexité que l'algorithme de tri fusion utilise de l'ordre de $n \log n$ comparaisons.

Tri rapide

Le tri rapide est aussi un algorithme de la famille « diviser pour régner ». Il fonctionne en partitionnant le tableau en deux sous-tableaux et en triant ces tableaux de manière indépendante. On commence par choisir au hasard un élément de notre tableau qu'on appelle *pivot* grâce à la fonction `choice` du module `random`.

— On génère ensuite le tableau `smaller` des éléments de t qui sont strictement inférieurs au pivot.

— De même, on génère le tableau `equal` des éléments de t qui sont égaux au pivot.

— Enfin, on génère le tableau `greater` des éléments de t qui sont strictement supérieurs au pivot.

On trie ensuite de manière récursive les tableaux `smaller` et `greater` avant de concaténer les tableaux obtenus.

```
import random

def quicksort(t):
    """quicksort(t: list[int]) -> list[int]"""
    if len(t) <= 1:
        return t[:]
    pivot = random.choice(t)
    smaller = [x for x in t if x < pivot]
    equal = [x for x in t if x == pivot]
    greater = [x for x in t if x > pivot]
    return quicksort(smaller) + equal + quicksort(greater)
```

Nous verrons en exercice la « vraie » version du tri rapide qui ne crée pas de tableau auxiliaire mais s'effectue en place. Cette version effectue dans le pire des cas de l'ordre de $n^2/2$ comparaisons, mais on peut montrer qu'en moyenne, elle effectue seulement de l'ordre de $n \log n$ comparaisons.

4.1.6 Les objets Python

Le lecteur attentif se sera peut-être rendu compte que les listes avaient un comportement étrange lorsqu'elles étaient passées en argument d'une fonction. Pour mettre en valeur ce comportement, il peut être utile de jouer avec les deux fonctions suivantes :

```
def f(n):
    """f(n: int) -> NoneType"""
    n = n + 1

def g(t):
    """g(n: list[int]) -> NoneType"""
```

```
7       for k in range(len(t)):
8           t[k] = t[k] + 1
```

Les deux essais suivants nous montrent bien que les entiers et les listes d'entiers réagissent de manière différente.

```
In [1]: a = 5

In [2]: f(a)

In [3]: a
Out[3]: 5
```

```
In [4]: a = [5, 5, 5]

In [5]: g(a)

In [6]: a
Out[6]: [6, 6, 6]
```

Cette différence de comportement peut d'ailleurs s'observer sans fonction. Pour la comprendre, nous avons besoin de revenir sur la notion d'*objet* et de *variable* en Python. L'idée même selon laquelle une variable est une boite contenant une valeur va d'ailleurs être mise à mal : c'était une simplification de la réalité que les listes viennent de mettre en valeur.

```
In [7]: a = [7, 2, 1]

In [8]: b = a

In [9]: a.append(5)

In [10]: a
Out[10]: [7, 2, 1, 5]

In [11]: b
Out[11]: [7, 2, 1, 5]
```

Si les variables a et b étaient réellement des boites contenant des valeurs, une modification de a n'aurait aucune influence sur b. Comme nous venons de voir sur cet exemple, ce n'est pas le cas. Pour comprendre, ce phénomène, il est bon de détailler la notion d'objet.

En Python, toutes les données sont représentées par des objets, que ce soient des entiers, des nombres flottants, des booléens, des chaines de caractères, des tuples, des listes et même des fonctions. Chaque objet possède un identifiant, un type et une valeur : on dit qu'en Python les valeurs sont *boxées*.

— L'*identifiant* d'un objet est l'adresse mémoire à laquelle il est stocké ; on utilise aussi le nom de *pointeur*. Il ne change pas au cours de la vie de l'objet et deux objets distincts ont des identifiants distincts. Nous verrons cependant que deux objets différents peuvent avoir la même valeur.

— Le *type* d'un objet est aussi une propriété qui ne change pas au cours de sa vie. Pour l'instant, nous avons vu essentiellement les types `bool`, `int`, `float`, `string`, `tuple` et `list`.

— Enfin, un objet possède une *valeur* qui peut changer où non au cours de sa vie, selon son type. Les valeurs des objets de type `bool`, `int`, `float`, `string` et `tuple` ne peuvent pas changer. On dit que ces types sont *immuables*. Le seul moyen d'avoir une nouvelle valeur est de créer un nouvel objet. Contrairement aux autres, le type `list` est *mutable* : les objets de type `list` ont une valeur qui peut changer au cours de leur vie.

Comme nous l'avons observé, une variable n'est pas une boite dans laquelle on met une valeur. C'est en fait une boite dans laquelle on met l'identifiant de l'objet auquel elle est associée. Par exemple, l'instruction

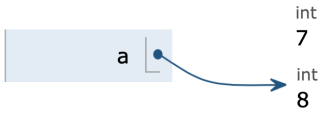
```
In [12]: a = 7
```

crée un objet de type `int` dont la valeur est 7 et associe la variable a à cet objet.

Lorsqu'on écrit ensuite

```
In [13]: a = a + 1
```

Python crée un nouvel objet de type `int` dont la valeur est 8 et associe le nom a à ce nouvel objet.

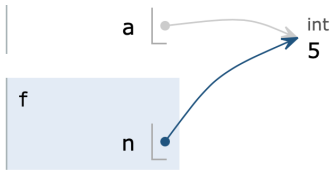


L'ancien objet de valeur 7 existe toujours, mais plus aucune variable ne pointe vers lui. Lors du prochain passage du *ramasse-miette* (*garbage collector* en anglais), cet objet sera supprimé et la mémoire qu'il utilise sera de nouveau disponible.

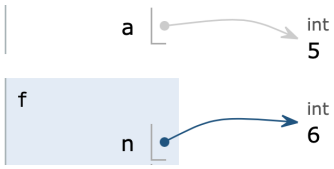
Nous pouvons maintenant comprendre pourquoi l'appel de la fonction f définie plus haut n'a aucun effet sur la variable a.

```
1 def f(n):
2     n = n + 1
3
4 a = 5
5 f(a)
```

Lorsqu'on est à la ligne 5 et qu'on appelle la fonction f, l'objet de valeur 5 est passé à la fonction et une variable locale n est créée.



L'instruction `n = n + 1` de la ligne 2 crée un nouvel objet de valeur 6 vers lequel pointe la variable n.

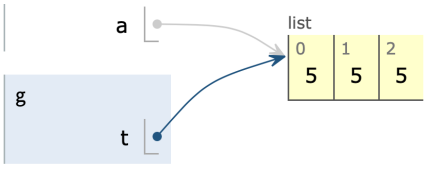


Lorsqu'on sort de la fonction, la variable n est détruite et a pointe toujours vers un objet de valeur 5.

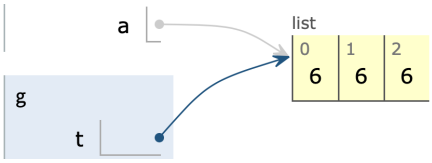
Le cas de la fonction g est différent.

```
1 def g(t):
2     for k in range(len(t)):
3         t[k] = t[k] + 1
4
5 a = [5, 5, 5]
6 g(a)
```

Lorsqu'on est à la ligne 6 et qu'on appelle la fonction g, l'objet de valeur `[5, 5, 5]` est passé à la fonction et une variable locale t est créée. Nous nous trouvons donc dans la situation où deux variables pointent vers le même objet ; ce phénomène est appelé *aliasing*.



Contrairement à ce qui se passait dans le cas précédent où l'instruction `n = n + 1` créait un nouvel objet, l'instruction `t[k] = t[k] + 1` fait muter la liste t et l'on se retrouve dans l'état suivant :



Lorsqu'on va sortir de la fonction, la variable t sera détruite et a pointera toujours vers la liste qui a été modifiée par notre fonction.

Exercice 5

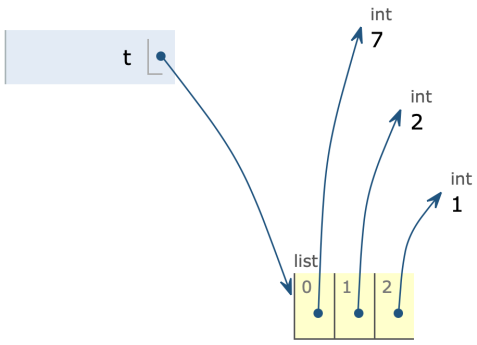
⇨ À quelle valeur est associée a après le script suivant ?

```
a = [7, 2, 1]
b = a
b.append(5)
```

La réalité est encore plus complexe que cela. Comme nous l'avions dit, les cases d'une liste se comportent comme des variables. Donc lorsqu'on écrit

```
1 t = [7, 2, 1]
```

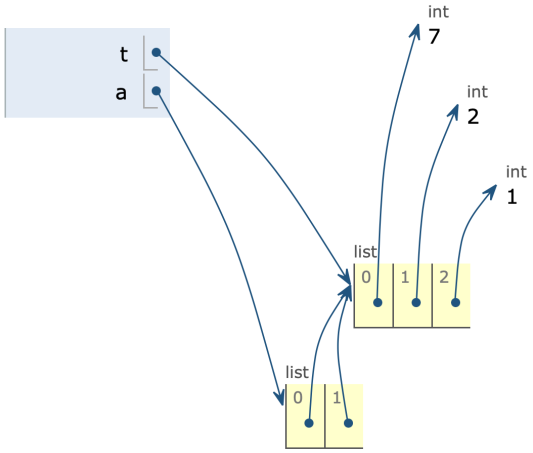
ce sont en fait les objets suivants qui sont créés.



La représentation qui a été faite plus haut ne pose cependant aucun problème puisque les entiers sont des valeurs immuables. Cependant, une telle représentation devient cruciale lorsqu'on commence à manipuler des listes d'objets mutables, comme des listes de listes.

```
1 t = [7, 2, 1]
2 a = [t, t]
3 t.append(5)
```

Juste avant l'exécution de la ligne 3, voici les relations entre les différents objets.



Après ce script, a va donc être associé à la valeur `[[7, 2, 1, 5], [7, 2, 1, 5]]`.

Exercice 6

⇨ On souhaite créer une liste de n listes. Si l'on écrit

```
1 n = 3
2 a = [[]] * n
3 a[0].append(7)
```

la liste a va être associée à la valeur `[[7], [7], [7]]`. Cependant, si on écrit

```
1 n = 3
2 a = [[] for _ in range(n)]
3 a[0].append(7)
```

la liste a va être associée à la valeur `[[7], [], []]`. Dessiner dans les deux cas les différents objets juste avant l'exécution de la 3^e ligne.

On retiendra de l'exercice précédent que si l'on souhaite créer une liste de n zéros, l'expression `[0] * n` fonctionne parfaitement, mais si l'on veut créer une liste de n listes vides, il ne faut surtout pas utiliser `[[]] * n` mais plutôt `[[] for _ in range(n)]`.

En résumé, on retiendra les points essentiels suivants :
— En Python, les données sont représentées par des *objets*.
— Certains objets sont de types *immuables* : `bool`, `int`, `float`, `str`, `tuple`. D'autres, comme `list`, sont *mutables*. Si une variable est associée à un objet d'un type immuable, on peut l'imaginer comme une boite contenant la valeur de cet objet. Cependant, si une variable est associée à un objet d'un type mutable, il est essentiel de se souvenir qu'elle pointe vers cet objet. Lorsque deux variables pointent vers un même objet, on dit qu'il y a *aliasing*.
— Une affectation `a = expr` fait pointer `a` vers l'objet en lequel `expr` s'évalue. Si `a` est associée à une liste, les instructions `a.append(x)`, `a.pop()` et `a[k] = expr` font muter cette liste. Le slicing `a[i:j]` crée une nouvelle liste dont les cases sont associées aux objets correspondants de la liste `a`. On comprend maintenant pourquoi l'instruction `c = t[:]` est à éviter lorsque `t` est une liste de listes.
— En Python, le passage des paramètres se fait *par objet* : à l'intérieur d'une fonction, les paramètres sont des variables locales associées aux objets passés lors de l'appel de la fonction. Si ces objets sont mutables, la fonction peut agir par effet de bord et changer leur valeur. Ces valeurs seront ensuite observables par l'appelant.

Exercice 7

⇨ 1. Quelle est la valeur de `a` après le script suivant ?

```
1 a = [7, 2, 3, 5, 1]
2 b = a[1:4]
3 b[0] = b[0] + 1
```

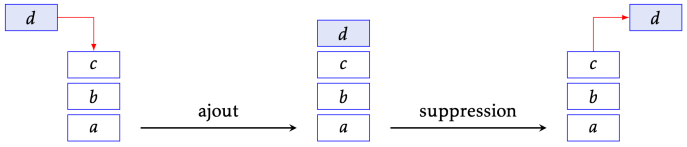
2. Et après le script suivant ?

```
1 a = [[7], [2], [3], [5], [1]]
2 b = a[1:4]
3 b[0].append(9)
```

4.2 Structures séquentielles

4.2.1 Pile

Une *pile* (*stack* en anglais) est une structure de données linéaire qui se distingue par ses conditions d'accès et d'ajout d'éléments : c'est le principe du « dernier arrivé, premier servi » (principe du LIFO pour Last In, First Out). Un peu comme une pile d'assiettes, c'est la dernière assiette posée sur une pile qui sera la première utilisée.



Une réalisation concrète de cette structure de données fournit, outre la structure, les fonctions suivantes :

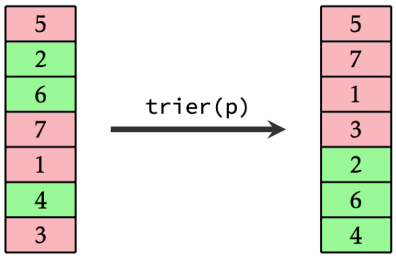
— Une fonction de création d'une pile vide.
— Une fonction déterminant si une pile est vide.
— Une fonction permettant d'empiler un élément au sommet de la pile.
— Une fonction permettant de dépiler et de renvoyer l'élément au sommet d'une pile non vide.
— Une fonction permettant de connaitre l'élément en haut d'une pile non vide.

La signature d'une pile d'entiers est donc la suivante :

```
stack_new() -> stack[int]
stack_is_empty(s: stack[int]) -> bool
stack_push(s: stack[int], x: int) -> NoneType
stack_pop(s: stack[int]) -> int
stack_peek(s: stack[int]) -> int
```

Exercices 8

⇨ Écrire une fonction `trier(p: stack[int]) -> NoneType` qui prend en argument une pile p d'entiers et qui modifie l'ordre de ses éléments de sorte qu'en fin de traitement les nombres pairs soient placés sous les nombres impairs. On pourra pour ce faire créer et utiliser une ou plusieurs piles auxiliaires.



⇨ Expliquer comment implémenter une pile à l'aide d'une liste Python. On proposera une implémentation des 5 fonctions dont la spécification est donnée plus haut.

Comme nous avons pu le voir dans l'exercice précédent, la structure de liste Python est tellement bien adaptée à la structure de pile que lorsqu'on aura besoin d'une pile, on utilisera une liste et on se limitera à l'usage des méthodes `append`, `pop` et à l'accès au dernier élément par `s[-1]`.

4.2.2 File

Une *file* (*queue* en anglais) est une structure de données linéaire fonctionnant sur le principe du FIFO (First In, First Out). On peut l'imaginer horizontale : on rajoute des éléments par la droite et on les enlève par la gauche. Cette fois, l'analogie se fait avec une file d'attente. Les clients arrivent par la droite et la caisse est à gauche. Le prochain client servi étant celui qui attend depuis le plus longtemps.

Une réalisation concrète de cette structure fournit les fonctions suivantes :
— Une fonction de création d'une file vide.
— Une fonction déterminant si un file est vide.
— Une fonction permettant d'enfiler un élément à droite de la file.
— Une fonction permettant de défiler et de renvoyer l'élément à gauche d'une file non vide.
— Une fonction permettant de connaitre l'élément à gauche d'une file non vide.

La signature d'une file d'entiers est donc :

```
queue_new() -> queue[int]
queue_is_empty(q: queue[int]) -> bool
queue_push(q: queue[int], x: int) -> NoneType
queue_pop(q: queue[int]) -> int
```

```
queue_peek(q: queue[int]) -> int
```

Exercice 9

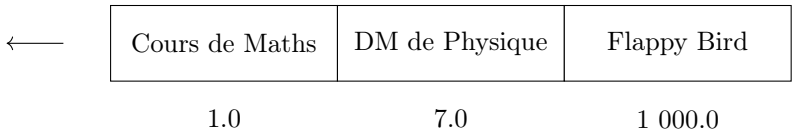
⇨ Expliquer comment implémenter une file à l'aide d'une liste Python. On proposera une implémentation des 5 fonctions dont la spécification est donnée plus haut. Pourquoi cette implémentation est-elle inefficace ?

En pratique, on n'utilisera donc pas de liste Python lorsqu'on aura besoin d'une file. On utilisera plutôt une « double ended queue » ou « deque » telle qu'elle est disponible dans le module `collections`.

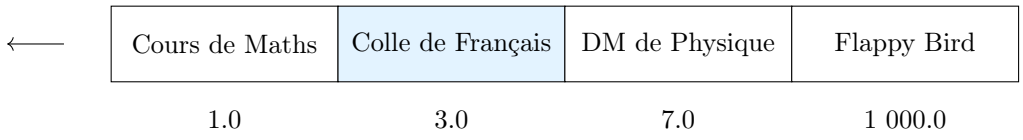
```
import collections

file = collections.deque() # Pour créer une file vide
n = len(file)              # Pour connaitre la longueur de la file
file.append(x)             # Pour enfiler un élément à droite
x = file.popleft()         # Pour défiler un élément à gauche
```

4.2.3 File de priorité

Une *file de priorité* (*priority queue* en anglais) est une structure de donnée linéaire où chaque élément de la file possède une priorité. Intuitivement, c'est avec une telle structure de données que vous fonctionnez lorsqu'on vous donne du travail. Supposons que vous ayez plusieurs tâches à accomplir, listées par importance décroissante : travailler le cours de maths du jour, finir le DM de physique pour la semaine prochaine et faire un gros score à flappy bird. On se représente cet ensemble de tâches comme ceci



la flèche nous indiquant que la prochaine tâche à accomplir est de relire votre cours de maths. Les nombres flottants placés en dessous de chaque tâche sont des priorités : plus ce nombre est bas, plus la tâche est prioritaire. Si vous apprenez que votre prochaine colle de français a lieu dans trois jours, il faut vous ménager du temps pour la préparer. Vous insérez donc cette nouvelle tâche dans la file avec une priorité de 3.0.



Une réalisation concrète d'une file de priorité fournit donc les fonctions suivantes :

- Une fonction de création d'une file de priorité vide.
- Une fonction déterminant si une file de priorité est vide.
- Une fonction permettant d'enfiler un élément, accompagné d'une priorité.
- Une fonction permettant de défiler l'élément ayant la priorité la plus basse.
- Une fonction permettant de connaitre l'élément ayant la priorité la plus basse.

La signature d'une file de priorité d'entiers est donc :

```
pqueue_new() -> pqueue[int]
pqueue_is_empty(q: pqueue[int]) -> bool
pqueue_push(q: pqueue[int], x: int, p: float) -> NoneType
pqueue_pop(q: pqueue[int]) -> tuple[int, float]
pqueue_peek(q: pqueue[int]) -> tuple[int, float]
```

Exercice 10

⇨ Expliquer comment implémenter une file de priorité d'entiers à l'aide d'une liste du tuples composés d'un entier et d'un nombre flottant.

Cette implémentation est cependant loin d'être efficace. Lorsque nous aurons besoin d'une meilleure efficacité, nous pourrons utiliser le module `heapq`.

```
import heapq

filep = []                      # Pour créer une file de priorité vide
n = len(filep)                  # Pour connaitre la longueur de la file
heapq.heappush(filep, (p, x)) # Pour insérer un élément x de priorité p
p, x = heapq.heappop(filep)   # Pour défiler un élément de priorité minimale
```

4.2.4 Dictionnaire

Un *dictionnaire* présente de nombreuses similitudes avec les listes, si ce n'est qu'au lieu d'accéder aux éléments par le biais d'un indice, on y accède par le biais d'une *clé*. On crée un dictionnaire en suivant la syntaxe `{c1: v1, ..., cn: vn}` où $c_1, \dots, c_n$ sont des clés, nécessairement deux à deux distinctes, et $v_1, \dots, v_n$ les valeurs qui leur sont associées. Ainsi, `{}` crée un dictionnaire vide.

```
In [1]: prof = {"Info": "Fayard", "Maths": "Fayard", "Physique": "Villegas"}

In [2]: prof["Maths"]
Out[2]: 'Fayard'
```

Si d est un dictionnaire et c est une clé :
- L'expression `c in d` renvoie un booléen indiquant si la clé c est présente dans le dictionnaire.
- L'expression `d[c]` renvoie la valeur associée à la clé si celle-ci est présente dans le dictionnaire.
- L'instruction `d[c] = v` crée une nouvelle association si la clé n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon.
- L'instruction `del d[c]` supprime l'entrée associée à la clé c dans le dictionnaire.
- On peut enfin connaitre le nombre de clés présentes dans un dictionnaire avec la fonction `len(d)`.

Le type d'un dictionnaire est `dict`. En pratique les clés seront toutes d'un même type et les valeurs seront aussi toutes du même type. Dans la signature des fonctions, on notera `dict[str, int]` un dictionnaire dont les clés sont des chaines de caractères et les valeurs sont des entiers.

Il est possible d'itérer sur les clés d'un dictionnaire avec la construction `for k in d.keys()`, mais on peut aussi directement itérer sur les clés et les valeurs avec la construction `for k, v in d.items()`. Par exemple :

```
In [3]: for k, v in prof.items():
   ...:     print("Le cours de", k, "est enseigné par M.",  v)
Le cours de Info est enseigné par M. Fayard
Le cours de Maths est enseigné par M. Fayard
Le cours de Physique est enseigné par M. Villegas
```

Ces deux méthodes sont celles préconisées par le programme de prépa, mais il est plus naturel en Python d'utiliser directement `for k in d` qui a le même effet que `for k in d.keys()`. Le même programme s'écrit alors :

```
In [4]: for k in prof:
   ...:     print("Le cours de", k, "est enseigné par M.",  prof[k])
```

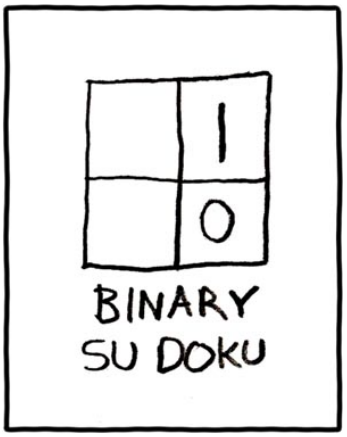
Exercice 11

⇨ Écrire une fonction `occ(lst: list[str]) -> dict[str, int]` renvoyant un dictionnaire `d` tel que si `s` est une chaine de caractères, `d[s]` est le nombre d'occurrences de `s` dans la liste `lst`.

Il est possible de simuler un ensemble à l'aide d'un dictionnaire. Un ensemble d'entiers `s` sera tout simplement un dictionnaire `dict[int, NoneType]`. On ajoutera un entier `a` à cet ensemble à l'aide de l'instruction `s[a] = None` et on le supprimera avec l'instruction `del s[a]`.

Chapitre 5

Représentation des données



Un ordinateur possède une mémoire vive, appelée RAM pour « Random Access Memory ». C'est cette mémoire qui matérialise l'état du système. Concrètement, les barrettes mémoire contiennent des milliards de condensateurs qui peuvent être chargés ou déchargés. Lorsqu'un condensateur est chargé, il représente le *bit* 1. S'il est déchargé, il représente le bit 0.

condensateur	0	1	2	3	4	5	6	7	8	9	10	...
état	0	1	0	0	1	0	1	1	0	0	1	...

Une succession de 8 bits est appelée un *octet* : c'est la plus petite quantité de mémoire adressable par un ordinateur. Les quantités de mémoire se comptent en kilooctets ($1\,000 \approx 2^{10}$ octets), mégaoctets ($10^6 \approx 2^{20}$ octets), gigaoctets ($10^9 \approx 2^{30}$ octets) et téraoctets ($10^{12} \approx 2^{40}$ octets).

Dans ce chapitre, nous allons voir comment une succession de 0 et de 1 peut être utilisée pour représenter des entiers et des nombres flottants.

5.1 Les entiers

5.1.1 Décomposition en base b

L'écriture de l'entier 1984 décrit un nombre formé de 4 unités, 8 dizaines, 9 centaines et 1 millier :

$$1984 = 4 \times 10^0 + 8 \times 10^1 + 9 \times 10^2 + 1 \times 10^3.$$

Le choix de faire des paquets en utilisant des puissances de 10 est cependant arbitraire. On peut tout aussi bien décider d'utiliser des puissances de 2, 12, 16 ou 60. Si l'on choisit d'utiliser des puissances de b, on dit qu'on décompose notre nombre en base b.

Proposition 5.1.1

Soit $b \geqslant 2$ un entier et $w \in \mathbb{N}$. Alors, pour tout $n \in [\![0, b^w[\![$, il existe un unique w-uplet $(d_0, d_1, \ldots, d_{w-1})$ d'éléments de $[\![0, b[\![$ tel que

$$n = \sum_{k=0}^{w-1} d_k b^k.$$

Remarques

- On parle de *décomposition en base b* de l'entier n. Les d_k sont appelés *chiffres* de n en base b.
- Les chiffres correspondant aux plus grandes puissances de b sont dits *plus significatifs* ou de *poids fort*. Ceux qui correspondent aux petites puissances de b sont dits *moins significatifs* ou de *poids faible*.
- Pour tout $n \in \mathbb{N}^*$, il existe un unique $w \in \mathbb{N}^*$ et un unique w-uplet $(d_0, d_1, \ldots, d_{w-1})$ d'éléments de $[\![0, b[\![$ tel que $d_{w-1} \neq 0$ et $n = \sum_{k=0}^{w-1} d_k b^k$. On écrit
$$n = \underline{d_{w-1} \cdots d_1 d_0}_b.$$
Lorsqu'on parle de *la* décomposition de n en base b, c'est de cette écriture qu'il s'agit. Par exemple $13 = \underline{13}_{10}$ car $13 = 3 \times 10^0 + 1 \times 10^1$ et $13 = \underline{1101}_2$ car $13 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$. Par convention, la décomposition de 0 est vide, quelle que soit la base.
- En base 2, les valeurs possibles pour un chiffre sont 0 et 1 ; on parlera indifféremment de *bit* ou de chiffre. Étant donné l'importance de la base 2 en informatique, il est bon de connaitre les premières puissances de 2.

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1 024

- Si la base est supérieure à dix, on a un problème pour l'écriture : il y a moins de chiffres usuels que de chiffres de la base. Le seul cas que l'on rencontre en pratique est celui de la base 16 dite *hexadécimale*. La convention est d'utiliser les lettres de `A` à `F` pour représenter les chiffres de 10 à 15.
- Les bases 2, 16 et dans une moindre mesure 8, sont couramment utilisées en informatique. Par conséquent, il est possible d'écrire les littéraux directement dans ces bases. En Python, la syntaxe est :

```
In [1]: 0b10010
Out[1]: 18

In [2]: 0xff
Out[2]: 255

In [3]: 0o77
Out[3]: 63
```

- Si l'on veut obtenir les chiffres de 137 en base 10, on commence par écrire que $137 = 13 \times 10 + 7$: on effectue la division euclidienne de 137 par 10. Le chiffre de poids faible d_0 est donc 7, et avant cela on a les chiffres de 13. De même, $13 = 1 \times 10 + 3$, donc $d_1 = 3$, et on continue en remplaçant 13 par 1. Enfin $1 = 0 \times 10 + 1$, donc $d_2 = 1$. On remplace 1 par 0 et on a terminé car 0 n'a « pas de chiffre ». On a donc $137 = \underline{137}_{10}$.
- Si au contraire on dispose de la liste des chiffres en base b et que l'on souhaite obtenir n, le plus efficace est de remarquer que

$$\sum_{k=0}^{w-1} d_k b^k = d_0 + b\left(d_1 + b\left(d_2 + \cdots b\left(d_{w-2} + b d_{w-1}\right)\right)\right).$$

Cette écriture est à la base de l'algorithme de Horner : on part de 0 et on multiplie successivement notre valeur par 10 avant de lui ajouter d_k, pour toutes les valeurs de k allant en décroissant de $w-1$ à 0. Si l'on souhaite calculer $\underline{137}_{10}$, on effectue donc les calculs

$$0 \xrightarrow[\times 10+1]{} 1 \xrightarrow[\times 10+3]{} 13 \xrightarrow[\times 10+7]{} 137.$$

Exercices 1

⇨ Calculer $\underline{1000110}_2$ et $\underline{C7}_{16}$.

⇨ Donner l'écriture binaire de 59 et de 31. Quel phénomène général peut-on remarquer dans le deuxième cas ?

⇨ Comment s'écrit $\underline{102}_3$ en base 5 ?

La fonction `eval_lsd(b: int, d: list[int]) -> int` (`lsd` pour least significant digit) renvoie l'entier dont l'écriture en base b est

$$\underline{d_{w-1}\dots d_0}_b$$

en utilisant l'algorithme de Horner.

```
def eval_lsd(b, d):
    """eval_lsd(b: int, d: list[int]) -> int"""
    w = len(d)
    n = 0
    for k in range(w):
        n = n * b + d[w - 1 - k]
    return n
```

La fonction `digits_lsd(b: int, n: int) -> list[int]` renvoie la liste des chiffres de n en base b, le chiffre le moins significatif étant en premier.

```
def digits_lsd(b, n):
    """digits_lsd(b: int, n: int) -> list[int]"""
    d = []
    while n > 0:
        d.append(n % b)
        n = n // b
    return d
```

Ces deux fonctions sont bien entendu à connaitre sur le bout des doigts.

Proposition 5.1.2

Soit $b \geqslant 2$ un entier. Un entier $n > 0$ s'écrit avec $w = 1 + \lfloor \log_b(n) \rfloor$ chiffres en base b.

Les opérations d'addition et de multiplication apprises à l'école primaire fonctionnent tout aussi bien en base $b \geqslant 2$ qu'en base 10. Entrainez-vous avec les exercices suivants pour vous convaincre de cela.

Exercices 2

⇨ Effectuer l'addition $\underline{100110}_2 + \underline{1011}_2$ en base 2, c'est-à-dire sans jamais convertir un nombre en base 10.

⇨ Effectuer la multiplication $\underline{100110}_2 \times \underline{1011}_2$ en base 2.

⇨ Écrire les tables de multiplication en base 3. Calculer le produit $\underline{1022}_3 \times \underline{221}_3$ en travaillant en base 3.

La décomposition en base 2 nous permet de revenir sur l'algorithme d'exponentiation rapide dont nous avons donné une version récursive dans le chapitre sur les fonctions et dont nous donnons ici une version itérative. Étant donné $n \in \mathbb{N}$, on effectue sa décomposition en base 2

$$n = \sum_{k=0}^{w-1} d_k 2^k$$

et on remarque que pour tout x

$$x^n = x^{\sum_{k=0}^{w-1} d_k 2^k} = \prod_{k=0}^{w-1} \left[x^{(2^k)}\right]^{d_k}.$$

Comme $d_k \in \{0,1\}$, quel que soit $k \in [\![0, w-1]\!]$, le terme entre crochets est soit présent dans le produit (si $d_k = 1$) soit absent (si $d_k = 0$). De plus, il est aisé de calculer les valeurs successives de x^{2^k} car chaque terme est le carré du précédent. On obtient ainsi l'algorithme d'exponentation rapide dans sa version itérative.

```
def expo(x, n):
    """expo(x: int, n: int) -> int"""
    ans = 1
    y = x
    m = n
    while m > 0:
        if m % 2 == 1:
            ans = ans * y
        y = y * y
        m = m // 2
    return ans
```

5.1.2 Représentation mémoire des entiers non signés

La décomposition en base 2 nous donne un moyen de représenter les nombres positifs à l'aide d'une séquence de bits. Comme cette décomposition s'effectue uniquement pour les entiers positifs, on parle de représentation des entiers *non signés*. C'est cette représentation que les processeurs utilisent pour manipuler les entiers non signés. Il ne sont cependant capables que de travailler avec des entiers ayant une largeur w fixée. Aujourd'hui, un processeur 64 bits peut travailler avec des entiers codés sur 8, 16, 32 ou 64 bits.

Proposition 5.1.3

Pour une largeur de $w \in \mathbb{N}$, le plus grand entier non signé représentable est $2^w - 1$.

Avec 8 bits, on peut coder tous les entiers entre 0 et $2^8 - 1 = 255$. Avec 16 bits, on peut coder tous les entiers entre 0 et $2^{16} - 1 = 65\,535$. Avec 32 bits on peut coder tous les entiers entre 0 et $2^{32} - 1 = 4\,294\,967\,295$, soit quelques milliards. Avec 64 bits on peut coder tous les entiers entre 0 et quelques milliards de milliards, ce qui est suffisant pour de nombreuses applications.

Ces représentations sur une largeur fixe ont cependant un défaut : la somme et le produit de deux entiers représentables ne sont pas toujours représentables. Par exemple, avec une largeur de 8 bits, 250 et 12 sont représentables, mais $250 + 12$ ne l'est pas. Lorsque ce problème survient, on dit qu'il y a un *dépassement de capacité* (*overflow* en anglais) et le résultat obtenu est calculé modulo 256. Dans notre exemple, le calcul sur 8 bits de $250 + 12$ donnera donc 6 ! Certains langages comme le C ne cachent pas cette caractéristique des processeurs et le programmeur a la responsabilité de s'assurer que ce problème n'arrive jamais (ou d'agir en conséquence). Python travaille quant à lui avec des entiers de taille variable. L'avantage est qu'il peut manipuler des entiers aussi grands que l'on souhaite ; on ne risque pas de dépassement de capacité. L'inconvénient est que les opérations usuelles sur ces entiers sont bien plus lentes qu'en C et que leur temps d'exécution dépend de la taille des entiers.

5.1.3 Représentation mémoire des entiers signés

Les entiers considérés pour le moment étaient supposés positifs, mais les processeurs proposent bien évidemment de travailler avec des types *signés*, qui permettent de représenter des valeurs négatives. D'une manière ou d'une autre, il est clair que le signe nous « coutera » un bit : il faut stocker l'information + ou −. Il est assez naturel d'imaginer la stratégie suivante :

— Le bit le plus significatif détermine le signe : un 1 signifie que le nombre est positif, un 0 qu'il est négatif.
— La valeur absolue du nombre est stockée de manière standard sur les autres bits.

Implicitement, nous considérons ici que l'on travaille avec des entiers d'une largeur w fixée. Ainsi, l'expression *bit le plus significatif* désigne le bit d_{w-1} de poids maximal dans cette largeur, ce qui explique pourquoi il peut être égal à zéro.

Bien que cette méthode paraisse raisonnable, elle a deux défauts :

— Le nombre 0 a deux représentations : $00\ldots0$ et $10\ldots0$. Cela a pour conséquence de ne pouvoir stocker que $2^w - 1$ valeurs différentes, par exemple les entiers de $-(2^{w-1} - 1)$ à $2^{w-1} - 1$ inclus. On perd une place puisque zéro en prend deux.
— Les opérations arithmétiques usuelles ne sont pas très simples à effectuer : essentiellement, pour ajouter deux nombres, on est obligé de regarder leur bit de poids fort pour déterminer leur signe et de distinguer les cas.

En réalité, aucun ordinateur n'utilise cette représentation pour les entiers signés. L'immense majorité utilise la représentation par *complément à deux*.

Proposition 5.1.4

Soit $w \in \mathbb{N}$. Pour tout $n \in [\![-2^{n-1}, 2^{n-1}[\![$, il existe un unique w-uplet $(b_0, b_1, \dots, b_{w-1})$ de bits tel que

$$n = \left(\sum_{k=0}^{w-2} b_k 2^k\right) - b_{w-1}2^{w-1}.$$

Remarques

⇨ Le bit b_{w-1} est nul si et seulement si $n \geqslant 0$. Si tel est le cas $(b_0, \dots, b_{w-1})$ est la décomposition de n en base 2. Sinon $n < 0$, $b_{w-1} = 1$ et $(b_0, \dots, b_{w-1})$ est la décomposition de $n + 2^w$ en base 2.

⇨ On appelle valeur en *complément à deux* de la suite de bits $(b_0, \dots, b_{w-1})$ l'entier

$$\left(\sum_{k=0}^{w-2} b_k 2^k\right) - b_{w-1}2^{w-1}.$$

On dit que le bit de poids fort a un poids négatif.

⇨ Une même suite de bits $(b_0, \dots, b_{w-1})$ dans une largeur w, correspondra donc à deux entiers différents : $\sum_{k=0}^{w-1} b_k 2^k$ en non signé et $\sum_{k=0}^{w-2} b_k 2^k - b_{w-1}2^{w-1}$ en signé par complément à deux. Fixons par exemple $w := 4$, et notons respectivement $\mathrm{v}_4(b_3b_2b_1b_0)$ et $\mathrm{vs}_4(b_3b_2b_1b_0)$, les valeurs non signées et signées associées à une suite de bits.

- $\mathrm{vs}_4(0000) = \mathrm{v}_4(0000) = 0$.
- $\mathrm{vs}_4(0100) = \mathrm{v}_4(0100) = 4$.
- $\mathrm{vs}_4(1100) = -8 + 4 = -4$ et $\mathrm{v}_4(1100) = 8 + 4 = 12$.
- $\mathrm{vs}_4(1111) = -8 + 4 + 2 + 1 = -1$ et $\mathrm{v}_4(1111) = 8 + 4 + 2 + 1 = 15$.

Exercice 3

⇨ On fixe $w := 8$.

1. Quel est le plus grand entier non signé représentable, c'est-à-dire la plus grande valeur $\mathrm{v}_8(bits)$ que l'on peut obtenir ?
2. Quels sont les plus petits et plus grands entiers signés représentables ?
3. Quelle suite de bits donne $\mathrm{vs}_8(bits) = 0$? 127 ? -1 ? -128 ?

Proposition 5.1.5

Pour une largeur $w \in \mathbb{N}$
- Le plus grand entier signé représentable en complément à 2 est $2^{w-1} - 1$.
- Le plus petit entier signé représentable en complément à 2 est -2^{w-1}.

Remarques

⇨ *Explosion d'Ariane 5* : Le vol inaugural de la fusée Ariane 5 a eu lieu le 4 juin 1996. Comme le montre l'illustration ci-dessous, il s'est terminé, un peu moins de 37 secondes après le décollage, par ce que nous appellerons pudiquement

un RUD (*Rapid Unplanned Dissassembly*).

Dommage.

La fusée et son chargement avaient couté 500 millions de dollars. La commission d'enquête a rendu son rapport au bout de deux semaines. Il s'agissait d'une erreur de programmation dans le système inertiel de référence. À un moment donné, un nombre codé en virgule flottante qui représentait la vitesse horizontale de la fusée par rapport à la plateforme de tir était converti en un entier signé sur 16 bits. Malheureusement, le nombre en question était plus grand que 32 767 et la conversion a été incorrecte.

```
L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                ((1.0/C_M_LSB_BH) *
                                G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;
```

Extrait du code source (en ADA) d'Ariane 5. On peut voir un certain nombre de conversions d'un entier 32 bits vers un entier 16 bits avec protection contre les dépassements de capacité, et, soulignée en rouge, une conversion non protégée d'un flottant vers un entier 16 bits.

⇨ L'année 2012 a été marquée par une contribution majeure au patrimoine culturel de l'humanité : la vidéo *Gangnam Style*. À cette époque, le nombre de vues d'une vidéo YouTube était codé sur un entier 32 bits signé. Bien évidemment, ce choix, qui limite le nombre de vues à 2 147 483 647, n'était absolument pas adapté à un chef-d'œuvre de cette ampleur. Début 2014, il est devenu évident qu'on allait bientôt avoir un dépassement de capacité. Fort heureusement, YouTube a apporté la modification nécessaire à temps : les vues sont maintenant codées sur un entier signé de 64 bits, ce qui laisse de la marge, la valeur maximale étant 9 223 372 036 854 775 807. *Baby shark* peut rester tranquille pour de nombreuses années.

Proposition 5.1.6

Pour toute largeur $w \in \mathbb{N}$ et toute suite de bits $(b_0, \ldots, b_{w-1})$, on a

$$\mathrm{v}_w(bits) \equiv \mathrm{vs}_w(bits) \mod 2^w.$$

Remarques

⇨ Cette propriété est la raison d'être de la représentation par complément à deux.

⇨ Nous n'allons pas rentrer dans les détails qui ne nous intéressent pas vraiment, mais l'avantage principal de la représentation en complément à deux, illustré par l'exercice ci-dessous, est qu'on peut utiliser essentiellement le même circuit physique pour les opérations arithmétiques sur les entiers signés et non signés.

Exercice 4

⇨ 1. Poser les additions binaires suivantes :

$$\begin{array}{rr} & 1011 \\ + & 0111 \\ \hline \end{array} \qquad \begin{array}{rr} & 1001 \\ + & 0011 \\ \hline \end{array} \qquad \begin{array}{rr} & 1001 \\ + & 1011 \\ \hline \end{array} \qquad \begin{array}{rr} & 0101 \\ + & 0011 \\ \hline \end{array}$$

2. Interpréter ces additions comme des opérations sur des entiers non signés de 4 bits. On ne gardera donc que les quatre bits les moins significatifs du résultat. Mathématiquement, cela revient à faire quoi ?
3. Reprendre la question précédente en faisant cette fois une interprétation signée, en complément à deux, sur quatre bits.
4. Quel critère simple, portant sur les deux bits de retenue de poids les plus forts, peut-on utiliser pour déterminer s'il y a eu un « vrai » dépassement de capacité ? Par « vrai » dépassement de capacité, on entend une situation dans laquelle le résultat mathématique de l'opération n'est pas représentable avec la largeur fixée.

5.2 Les nombres flottants

5.2.1 Représentation mémoire des flottants

Pour écrire un nombre réel de manière approchée, les physiciens ont pris l'habitude d'utiliser l'écriture scientifique. Par exemple

$$\mathrm{e}^{\pi} \approx 23.14 = 2.314 \times 10^1 = \left(2 + \frac{3}{10} + \frac{1}{10^2} + \frac{4}{10^3}\right) \times 10^1.$$

On dit qu'un tel nombre est représenté avec 4 chiffres significatifs. Remarquons que contrairement aux entiers naturels, qui admettent tous une décomposition en base b, seuls les nombres décimaux peuvent s'écrire de manière exacte sous la forme

$$\pm \left(\sum_{k=0}^{p-1} \frac{m_k}{10^k}\right) \times 10^e$$

où $m_k \in [\![0, 9]\!]$. Même certains nombres rationnels comme $1/3$ ne peuvent pas s'écrire de la sorte. Bien entendu, ces remarques faites en base 10 sont aussi valables en base 2, plus familière des ordinateurs.

Définition 5.2.1

Soit $p \in \mathbb{N}^*$. On dit qu'un réel x est un nombre flottant représentable avec une mantisse de p bits lorsqu'il existe $m_0, \dots, m_{p-1} \in \{0, 1\}$ et $e \in \mathbb{Z}$ tels que

$$x = \pm \left(\sum_{k=0}^{p-1} \frac{m_k}{2^k}\right) 2^e.$$

Si x est non nul, il est possible d'imposer $m_0 = 1$; cette écriture est alors unique. L'ensemble des nombres flottants représentables avec une mantisse de p bits est noté $\mathcal{F}_p$.

Remarques

⇨ Par exemple $2.5 = (1 + 0/2 + 1/4) \times 2^1 \in \mathcal{F}_3$.

⇨ Si x est non nul et $m_0 = 1$, cette écriture est appelée *décomposition en base 2 normalisée*. Les autres écritures comme $x = (0 + 1/2 + 1/4) \times 2^4$ sont dites *dénormalisées*

⇨ Tous les entiers compris entre $-2^p + 1$ et $2^p - 1$ sont des éléments de $\mathcal{F}_p$.

Exercice 5

⇨ Montrer que les rationnels $r = \pm a/b$ (avec a et b premiers entre eux) tels que b n'est pas une puissance de 2 ne sont pas des éléments de $\mathcal{F}_p$. En particulier $0.1 = 1/10$ et $1/3$ n'appartiennent pas à $\mathcal{F}_p$.

Proposition 5.2.2

Soit $x \in \mathbb{R}$. Alors il existe un élément f de $\mathcal{F}_p$ minimisant la distance de x à $\mathcal{F}_p$. De plus

$$|x - f| \leqslant u_p\,|x| \quad \text{avec } u_p \coloneqq 2^{-p}.$$

Remarques

- Les éléments de $\mathcal{F}_p$ permettent donc d'approcher n'importe quel réel avec une *erreur relative* inférieure à u_p. Cet élément u_p est appelé *epsilon* de $\mathcal{F}_p$.
- Pour une valeur de x, f est unique sauf dans le cas particulier où x est au milieu de deux éléments successifs de $\mathcal{F}_p$. Dans ce cas, un seul de ces deux éléments a une décomposition en base 2 normalisée telle que $m_{p-1} = 0$. C'est cet élément f qu'on appelle *arrondi* de x à la *précision* $\mathcal{F}_p$.

Définition 5.2.3

Si $p, q \in \mathbb{N}^*$, on note $\mathcal{F}_{p,q}$ l'ensemble formé
- des réels x de la forme
$$x = \pm \left(m_0 + \frac{m_1}{2} + \frac{m_2}{4} + \cdots + \frac{m_{p-1}}{2^{p-1}} \right) 2^e$$
où $m_0, \ldots, m_{p-1} \in \{0, 1\}$ et $-2^{q-1} + 2 \leqslant e \leqslant 2^{q-1} - 1$.
- des éléments $+\infty$ et $-\infty$.
- de l'élément noté NaN (Not a Number)

Remarques

- Il y a en fait deux zéros, notés 0^+ et 0^-, mais ils sont le plus souvent affichés de la même façon.
- Si x est non nul, le plus souvent, il est possible d'imposer $m_0 = 1$. Ce n'est cependant pas possible pour les nombres de la forme
$$x = \pm \left(0 + \frac{m_1}{2} + \frac{m_2}{4} + \cdots + \frac{m_{p-1}}{2^{p-1}} \right) 2^e$$
pour $e = -2^{q-1} + 2$. De tels nombres sont dits *dénormalisés*.
- Les processeurs actuels proposent en général deux types de nombres flottants.
  - Le format simple précision, codé sur 32 bits, utilise $p = 24$ et $q = 8$. On a alors $u = 2^{-24} \approx 5.9 \times 10^{-8}$.
  - Le format double précision, codé sur 64 bits, utilise $p = 53$ et $q = 11$. On a alors $u = 2^{-53} \approx 1.1 \times 10^{-16}$.

  C'est le format double précision auquel Python nous donne accès avec son type `float`. Le plus petit nombre strictement positif représentable est de l'ordre de 10^{-324} et le plus grand nombre représentable est de l'ordre de 10^{308}.
- Représenter un réel à l'aide d'une succession de bits revient donc à coder son signe, sa mantisse et son exposant. Avec les nombres flottants double précision de la norme IEEE 754, on se donne 64 bits pour stocker ces trois données. Si $x \in \mathcal{F}_{53,11}$ est non nul et $-1022 \leqslant e \leqslant 1023$, on code, dans l'ordre
  - Le *signe*, qui ne nécessite qu'un seul bit : 0 code le signe positif et 1 le signe négatif.
  - L'*exposant*, qui se code sur 11 bits. L'entier e est compris entre -1022 et 1023 on code l'écriture binaire de $e + 1023$ qui est un entier entre 1 et 2046.
  - La *mantisse*, qui se code sur 52 bits : comme $m_0 = 1$, il suffit de coder $m_1, \ldots, m_{52}$.
- Dans la plage de 11 bits dévolue à l'exposant, les entiers $\underline{0\ldots0}_2 = 0$ et $\underline{1\ldots1}_2 = 2047$ ne sont pas utilisés dans l'explication précédente. Le nombre 0 est représenté par un exposant e égal à -1023, donc codé $\underline{0\ldots0}_2 = 0$, avec une mantisse dont tous les bits sont nuls. Il y a bien deux 0, un 0^+ et un 0^- puisque le bit donnant le signe peut prendre les deux valeurs 0 ou 1 (les 63 autres bits sont à zéro). L'exposant décalé égal à 2047 est utilisé pour les situations particulières ($+\infty$, $-\infty$ et d'autres choses plus compliquées comme NaN). Dans ce cas, tous les bits dévolus à l'exposant sont égaux à 1.

Charles Leclerc semble avoir un problème sur l'un de ses capteurs de vitesse

5.2.2 Problèmes liés à l'arithmétique des nombres flottants

Overflow et underflow

Le problème le plus simple à comprendre est celui du dépassement arithmétique (*overflow* en anglais), qui survient lorsqu'on dépasse le plus grand nombre représentable, qui est de l'ordre de 10^{308} en double précision. Dans ce cas, le résultat est $+\infty$ ou $-\infty$.

```
In [1]: a = 2.0**1023

In [2]: a
Out[2]: 8.98846567431158e+307

In [3]: 2.0 * a
Out[3]: inf
```

De même, pour les flottants strictement positifs, il peut y avoir dépassement par valeurs inférieures, ou *underflow*. Dans ce cas, le nombre renvoyé est 0 (plus précisément 0^+ dans notre cas).

```
In [4]: 2.0**(-1074)
Out[4]: 5e-324

In [5]: 2.0**(-1075)
Out[5]: 0.0
```

Des études ont montré que même lors de calculs intermédiaires, de tels nombres n'apparaissaient presque jamais. Ces problèmes peuvent donc essentiellement être ignorés.

Inexactitude de la représentation, arrondis

Les arrondis sont plus problématiques et sont liés au fait que les nombres flottants n'ont qu'un nombre fini de chiffres significatifs. Deux types d'arrondis entrent en jeu.

Le premier est dû au fait que les ordinateurs travaillent en base 2 alors qu'ils échangent le plus souvent des informations avec l'utilisateur et le programmeur en base 10. Par exemple, lorsqu'on écrit

```
In [1]: x = 0.1

In [2]: x
Out[2]: 0.1
```

on pourrait facilement être dupé et croire que 0.1 est représentable de manière exacte en double précision. Or $0.1 = 1/10$ n'appartient à aucun des $\mathcal{F}_p$ puisque 10 n'est pas une puissance de 2. Il n'est donc pas représentable de manière exacte par un nombre flottant, un peu comme $1/7 = 0.142857\underline{142857}$ n'est pas un nombre décimal (le souligné signifie que le groupe de chiffres se répète à l'infini). Si l'on effectue un développement de $1/10$ en base 2, on obtient $1.100\underline{1100} \times 2^{-4}$ (encore une fois, le groupe de chiffres se répète à l'infini). Lorsque 0.1 est entré dans le shell, Python va effectuer son développement en base 2. Comme ce développement est infini et qu'il travaille en double précision, il ne va garder que les 53 premiers bits et arrondir 0.1 à un nombre légèrement différent, avant de stocker ce nombre dans x. C'est un *arrondi de conversion* de bases. Pour afficher la valeur de x à l'utilisateur, il va le reconvertir en base 10 et un autre arrondi de conversion va faire qu'il affiche 0.1. Mais il ne faut pas oublier que ce n'est pas 0.1 qui est stocké dans x.

Le second problème est plus fondamental : les ensembles $\mathcal{F}_p$ ne sont stables ni par addition, ni par soustraction, ni par multiplication ; encore moins par division. Par exemple

$$x := (1 + 0/2 + 1/4) \times 2^1 \in \mathcal{F}_3 \quad \text{et} \quad y := (1 + 0/2 + 0/4) \times 2^{-2} \in \mathcal{F}_3,$$

mais

$$x + y = (1 + 0/2 + 1/4 + 1/8) \times 2^1 \notin \mathcal{F}_3$$

car $x + y$ possède 4 chiffres significatifs. Pour chaque opération élémentaire (addition, soustraction, multiplication, division) entre deux nombres flottants, le processeur se trouve donc dans l'incapacité de représenter le résultat exact par un nombre flottant : il va devoir effectuer un *arrondi arithmétique*. Même si `+` et `*` restent commutatives, ces arrondis leur font perdre leur associativité. Plus le nombre d'opérations élémentaires va être grand, plus ces arrondis vont nous éloigner du résultat attendu.

Exercice 6

⇨ Observez les lignes suivantes et commentez.

```
In [1]: 1.0 + 2.0**(-53) + (-1.0)
Out[1]: 0.0

In [2]: 1.0 + (-1.0) + 2.0**(-53)
Out[2]: 1.1102230246251565e-16
```

Ces phénomènes ne sont pas anodins et ne doivent pas être pris à la légère. Par exemple, si l'on veut tester l'égalité de deux flottants et qu'il existe une légère différence entre eux due aux arrondis, on aura un résultat surprenant !

```
In [3]: 0.1 + 0.1 + 0.1 == 0.3
Out[3]: False

In [4]: import math

In [5]: math.sqrt(10)**2 == 10.0
Out[5]: False
```

On retiendra qu'il ne faut jamais faire de test d'égalité entre deux nombres flottants. En pratique, on ne se posera pas la question de savoir si `a == b`, mais plutôt de savoir si `abs(a - b) <= eps * abs(a)` où `eps` est un nombre « petit », à choisir selon notre application ($\varepsilon \coloneqq \sqrt{u} \approx 10^{-8}$ est souvent un bon choix).

Nous avons vu pour le moment des calculs où les erreurs introduites par les arrondis étaient négligeables devant les grandeurs manipulées. Mais ce n'est pas toujours le cas. Supposons par exemple que l'on souhaite calculer

$$u_n \coloneqq \int_0^1 x^n \mathrm{e}^x \,\mathrm{d}x$$

pour tout $n \in \mathbb{N}$. Un encadrement élémentaire de l'intégrande nous montre que

$$\forall n \in \mathbb{N}, \quad 0 \leqslant u_n \leqslant \int_0^1 x^n \mathrm{e}\,\mathrm{d}x = \frac{\mathrm{e}}{n+1}$$

ce qui prouve par le théorème des gendarmes que u_n tend vers 0 lorsque n tend vers $+\infty$. Si l'on veut calculer explicitement u_n, on remarque que $u_0 = \mathrm{e} - 1$ et une intégration par partie nous donne

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \mathrm{e} - (n+1)u_n.$$

Le programme suivant permet donc de calculer u_n.

```
1 import math
2
3 def integrale(n):
4     """integrale(n: int) -> float"""
5     u = math.exp(1.0) - 1.0
6     for k in range(n):
7         u = math.exp(1.0) - (k + 1) * u
8     return u
```

```
In [6]: [integrale(n) for n in [0, 5, 10, 20]]
Out[6]: [1.718281828459045, 0.395599547802016,
         0.22800151529345358, -129.26370813285942]
```

Les premières valeurs de u_n calculées sont réalistes, mais u_{20} est totalement faux. Ce phénomène était prévisible, car si $\alpha \in \mathbb{R}$ et (v_n) est la suite définie par

$$v_0 \coloneqq \alpha, \quad \text{et} \quad \forall n \in \mathbb{N}, \quad v_{n+1} \coloneqq \mathrm{e} - (n+1)v_n$$

alors, en définissant l'erreur $\varepsilon_n \coloneqq v_n - u_n$, on obtient facilement $\varepsilon_{n+1} = -(n+1)\varepsilon_n$ et donc

$$\forall n \in \mathbb{N}, \quad \varepsilon_n = (-1)^n n! \varepsilon_0.$$

Si α est une valeur approchée de $e-1$ telle que $|\varepsilon_0| = |\alpha - (e-1)| \propto 10^{-16}$, comme $20! \propto \times 10^{18}$, on en déduit que $|v_{20} - u_{20}| \propto 100$. Donc si l'on effectue une erreur de calcul de l'ordre de 10^{-16} pour u_0, même si les calculs suivants sont exacts, l'erreur absolue obtenue pour le 20^e terme de la suite (u_n) est de l'ordre de 100, ce qui est beaucoup plus grand que l'ordre de grandeur de u_{20} puisque $0 \leqslant u_{20} \leqslant e/21 \approx 0.13$. Certains algorithmes numériques comme celui-ci sont *instables* et rendent les calculs avec des nombres flottants inexploitables. D'autres sont *stables* et peuvent donc être utilisés avec des nombres flottants. L'étude de la stabilité des algorithmes numériques dépasse le cadre du programme des classes préparatoires.

Quelques catastrophes dues à une mauvaise utilisation des nombres flottants

Il y a un nombre de catastrophes qui sont attribuables à une mauvaise gestion de l'arithmétique des nombres flottants. Dans le premier exemple, cela s'est payé en vies humaines.

— *Missile Patriot* : En février 1991, pendant la guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours du scud, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits. Le temps était compté par l'horloge interne du système en dixièmes de seconde. Malheureusement, 1/10 n'a pas d'écriture finie dans le système binaire : $1/10 = 0.1$ (dans le système décimal) $= 0.0001100110011001100110011...$ (dans le système binaire). L'ordinateur de bord arrondissait 1/10 à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque dixième de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui a entrainé une accumulation des erreurs d'arrondi de 0.34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible. Ce qu'il aurait fallu faire c'est redémarrer régulièrement le système de guidage du missile.

— *Bourse de Vancouver* : Un autre exemple où les erreurs de calcul ont conduit à une erreur notable est le cas de l'indice de la Bourse de Vancouver. En 1982, elle a créé un nouvel indice avec une valeur nominale de 1000. Après chaque transaction boursière, cet indice était recalculé et tronqué après le troisième chiffre décimal et, au bout de 22 mois, la valeur obtenue était 524.881, alors que la valeur correcte était 1098.811. Cette différence s'explique par le fait que toutes les erreurs d'arrondi étaient dans le même sens : l'opération de troncature diminuait à chaque fois la valeur de l'indice.

5.3 Caractères et chaines de caractères

5.3.1 Codes ASCII et Unicode

Le code ASCII associe un caractère à chaque entier entre 0 et 127, ce qui correspond à 7 bits non signés. Ces caractères peuvent être classés en trois grandes catégories :

— *Caractères alphanumériques :* les chiffres et les lettres minuscules et majuscules. Seules les lettres utilisées en anglais font partie du code ASCII, donc pas de « é », de « ñ », de « ß »...

— *Autres caractères imprimables :* les signes de ponctuation, quelques symboles (`()+`, `*`, `}`, etc) et l'espace.

— *Caractères non imprimables :* la tabulation, les différents caractères correspondant à un retour à la ligne, le caractère nul, etc.

	0	1	2	3	4	5	6	7	8	9
0										
10										
20										
30				!	"	#	$	%	&	’
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Comme les caractères étaient presque systématiquement codés sur 8 bits, les codes 128 à 255 étaient « libres » : ils ont pendant très longtemps été utilisés pour coder les caractères spécifiques aux différentes langues (signes diacritiques, lettres supplémentaires, etc). Cependant ces extensions posaient deux problèmes :

— Elles n'étaient pas standardisées, puisqu'elles différaient d'une langue à l'autre et qu'il y avait même plusieurs extensions concurrentes pour une même langue. En pratique, jusqu'à la fin des années 2000, il y avait une chance sur deux que tous les caractères accentués contenus dans un mail soient remplacés par une bouillie infâme avant de parvenir au destinataire.
— Elles permettaient plus ou moins de gérer les langues européennes ou au moins les langues basées sur l'alphabet latin, mais étaient totalement inappropriées au chinois, au japonais, etc.

Le standard Unicode a été développé à partir de la fin des années 1980. À l'heure actuelle, il définit des codes pour 144 697 caractères, ce qui permet de gérer l'ensemble des langues, ainsi que de nombreux caractères supplémentaires comme les Emojis, par exemple. Ce standard définit trois représentations binaires UTF-8, UTF-16 et UTF-32, et il est assez complexe : nous ne rentrerons pas dans les détails. Dans tous les cas, les *codepoints* (l'entier associé à un caractère) ne sont pas modifiés pour les caractères appartenant au code ASCII. On peut obtenir un caractère à partir de son code Unicode grâce à la fonction `chr`.

```
In [1]: ord('A')
Out[1]: 65

In [2]: ord('é')
Out[2]: 233
```

On peut obtenir un caractère à partir de son code UNICODE grâce à la fonction `chr`. C'est d'ailleurs le moyen le plus simple d'obtenir des caractères non disponibles sur votre clavier.

```
In [3]: "I " + chr(9829) + " les Lazos."
Out[3]: 'I ♥ les Lazos.'
```

5.3.2 Lecture et écriture dans un fichier

Python permet d'ouvrir des fichiers texte en lecture ou en écriture. Pour Python, un fichier n'est qu'une séquence de caractères. Une fois que le fichier aura été ouvert par le programme, celui-ci maintiendra un marqueur fictif à la position courante qui nous indique où sera lue ou écrit la prochaine séquence. Un fichier est ouvert avec la fonction `open` :

```
f = open("/Users/fayard/Desktop/fichier.txt", 'r')
```

Le premier argument est une chaine de caractères contenant le chemin complet du fichier. Le second argument est le mode d'ouverture du fichier : `'r'` (pour read) pour une ouverture en lecture seule, `'w'` (pour write) pour une ouverture avec les droits d'écriture et enfin `'a'` (pour append) pour une ouverture avec les droits d'écriture et la position du marqueur en fin de fichier. Une fois que le travail dans le fichier sera fini, il faudra prendre soin de bien fermer le fichier avec la commande

```
f.close()
```

Une fois ouvert, la fonction `read` permet de lire le fichier en entier et le renvoie sous la forme d'une chaine de caractères :

```
texte = f.read()
```

Dans la même famille, la fonction `readlines()` permet de lire le fichier en entier, mais renvoie non pas une seule chaine de caractères, mais une liste de chaines de caractères, chaque chaine correspondant à une ligne du fichier. Attention, cette ligne se terminera par le caractère de retour à la ligne.

```
lignes = f.readlines()
```

Mais en général, on lira et on traitera le fichier ligne par ligne avec le fonction `readline()` qui lit une ligne et la renvoie en tant que chaine de caractères. Bien entendu, cette chaine finira aussi par un caractère de retour à la ligne. Après cet appel, le curseur sera positionné au début de la ligne suivante.

```
ligne = f.readline()
```

On saura qu'on est à la fin du fichier lorsque cette méthode nous renverra une chaine de caractères vide.

Mais la méthode sans doute la plus utilisée pour parcourir un fichier en lecture est de remarquer que le descripteur du fichier `f` est un itérable. La boucle

```
for ligne in f:
```

va donc permettre d'effectuer une boucle sur toutes les lignes du fichier. Cette boucle est équivalente à

```
for ligne in f.readlines():
```

mais a l'avantage de faire lire le fichier à notre programme ligne après ligne alors que l'utilisation de `readlines()` va charger le fichier en mémoire en entier avant même de traiter la première ligne.

Il est courant de lire des fichiers textes contenant des données organisées, comme par exemple un fichier Csv (Comma Separated Values) qui est le format le plus simple pour enregistrer les données d'un tableur. Le fichier décrit dans un fichier texte chaque ligne du document, où chaque colonne est séparée par une virgule. Si par exemple, vous souhaitez avoir une liste des élèves de la classe avec leur âge, le fichier Csv correspondant sera simplement

```
Elon,Musk,51
Donald,Knuth,84
Master,Yoda,900
```

Afin de séparer proprement ce type de ligne, on utilisera la fonction `split`. Par exemple, si `ligne = "Elon,Musk,51"`, la commande `data = ligne.split(',')` stockera dans la variable `data` la liste `["Elon", "Musk", "51"]`.

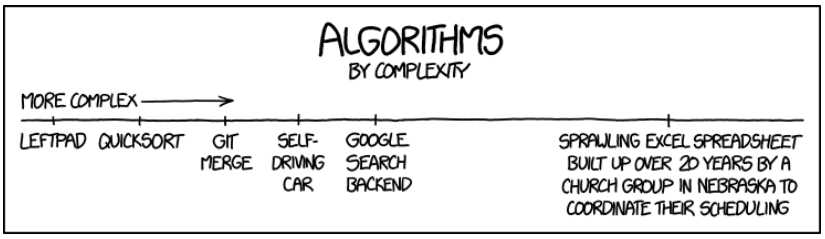
Si le fichier est ouvert en écriture, on peut écrire dedans à l'aide de la méthode `write`.

```
f.write("Ma jolie histoire.")
```

Le programme spécifie que la documentation de ces fonctions doit vous être rappellée mais il est important de savoir les utiliser proprement une fois ce rappel fait.

Chapitre 6

Complexité



6.1 Complexité

6.1.1 Notation mathématique

Définition 6.1.1

Soit (u_n) et (v_n) deux suites réelles positives

— On dit que $u_n = \mathrm{O}(v_n)$ lorsqu'il existe $B > 0$ et $N \in \mathbb{N}$ tels que

$$\forall n \geqslant N, \quad u_n \leqslant B v_n.$$

— On dit que $u_n = \Omega(v_n)$ lorsqu'il existe $A > 0$ et $N \in \mathbb{N}$ tels que

$$\forall n \geqslant N, \quad u_n \geqslant A v_n.$$

— On dit que $u_n = \Theta(v_n)$ lorsqu'il existe $A, B > 0$ et $N \in \mathbb{N}$ tels que

$$\forall n \geqslant N, \quad A v_n \leqslant u_n \leqslant B v_n.$$

Remarques

⇨ On a $u_n = \Theta(v_n)$ si et seulement si $u_n = \mathrm{O}(v_n)$ et $u_n = \Omega(v_n)$.

⇨ La relation Θ est une relation d'équivalence sur l'ensemble des suites positives. En particulier, elle est symétrique.

⇨ Ces définitions sont asymptotiques : Elles ne dépendent pas des premiers termes de ces suites.

⇨ Lorsque nous ferons des calculs de complexité, nous travaillerons avec des suites (v_n) strictement positives. Dans ce cas
- $u_n = \mathrm{O}(v_n)$ si et seulement si il existe $B > 0$ tel que : $\forall n \in \mathbb{N}, \quad u_n \leqslant B v_n$.
- $u_n = \Omega(v_n)$ si et seulement si il existe $A > 0$ tel que : $\forall n \in \mathbb{N}, \quad u_n \geqslant A v_n$.
- $u_n = \Theta(v_n)$ si et seulement si il existe $A, B > 0$ tels que : $\forall n \in \mathbb{N}, \quad A v_n \leqslant u_n \leqslant B v_n$.

Ces caractérisations n'ont plus besoin du « à partir d'un certain rang ».

⇨ Il faut bien retenir les interprétations intuitives :
- « $u_n = \mathrm{O}(v_n)$ » signifie « u_n est au plus de l'ordre de grandeur de v_n ».
- « $u_n = \Omega(v_n)$ » signifie « u_n est au moins de l'ordre de grandeur de v_n ».
- « $u_n = \Theta(v_n)$ » signifie « u_n et v_n sont du même ordre de grandeur ».

⇨ Si $v_n = \Theta(w_n)$, alors une suite u_n est un O, un Ω ou un Θ de v_n si et seulement si c'est un O, un Ω ou un Θ de w_n. En particulier, puisque quel que soit $b > 1$, $\log_b n = \Theta(\ln n)$, dire que $u_n = \Theta(\log_2 n)$ est équivalent à dire que $u_n = \Theta(\ln n)$. On écrira simplement $u_n = \Theta(\log n)$.

Proposition 6.1.2

Soit (u_n) une suite positive et (v_n) une suite strictement positive. On suppose que

$$\frac{u_n}{v_n} \xrightarrow[n\to+\infty]{} l \in \mathbb{R}_+ \cup \{+\infty\}.$$

Alors
- $u_n = \mathrm{O}(v_n)$ si et seulement si $l < +\infty$.
- $u_n = \Omega(v_n)$ si et seulement si $l > 0$.
- $u_n = \Theta(v_n)$ si et seulement si $0 < l < +\infty$.

Exercice 1

⇨ Déterminer les relations de comparaison entre les suites de terme général
- $u_n := \lfloor \ln n \rfloor$ et $v_n := \ln n$.
- $u_n := 12n^2 + 3n \log n - n$ et $v_n := n^2$.
- $u_n := 12n^2$ et $v_n := n^{17}$.
- $u_n := 12n^2$ et $v_n := n^2$.

6.1.2 Type de ressource

Étudier la complexité d'un algorithme, c'est s'intéresser aux ressources qu'il consomme pour effectuer sa tâche, et plus précisément à la manière dont cette consommation évolue lorsque la taille des données augmente. Les principales ressources auxquelles on peut s'intéresser sont :
- le *temps* de calcul.
- l'*espace* mémoire.
- l'*énergie*, qui prend une importance de plus en plus grande à cause de son impact sur
  - l'*autonomie*, principalement dans les téléphones.
  - le *bilan écologique* et le *cout monétaire* des calculs, principalement à l'échelle d'un datacenter.
- les *données échangées* sur le réseau, qui peuvent être un facteur limitant.

L'étude de la complexité se fait le plus souvent de manière *asymptotique*, c'est-à-dire en faisant tendre la taille des données vers l'infini.

Définition 6.1.3

On dit que la complexité $C(n)$ d'un algorithme est
- *constante* lorsque $C(n) = \Theta(1)$.
- *logarithmique* lorsque $C(n) = \Theta(\log n)$.
- *linéaire* lorsque $C(n) = \Theta(n)$.
- *quasi-linéaire* lorsque $C(n) = \Theta(n \log n)$.
- *quadratique* lorsque $C(n) = \Theta\left(n^2\right)$.
- *polynomiale* lorsqu'il existe $\alpha \in \mathbb{N}$ tel que $C(n) = \Theta\left(n^\alpha\right)$.
- *exponentielle* lorsqu'il existe $\alpha > 1$ tel que $C(n) = \Theta\left(\alpha^n\right)$.
- *factorielle* lorsque $C(n) = \Theta\left(n!\right)$.

Quand on s'intéresse à la complexité temporelle d'un algorithme, on cherche à évaluer comment son temps d'exécution évolue quand on fait tendre la taille des données n vers l'infini. Comme le temps de calcul dépend de nombreux

facteurs difficiles à contrôler comme le langage, l'implémentation ou la machine, on se concentre sur le *nombre d'instructions élémentaires* exécutées. Il faut donc :

— Déterminer le nombre de fois où chaque instruction est exécutée.
— Déterminer si chacune de ces instructions est *élémentaire* ou non. Pour les instructions qui ne sont pas élémentaires, estimer leur complexité.
— Sommer toutes ces complexités et tenter éventuellement d'en tirer des conclusions sur le temps de calcul.

Ce qui caractérise une opération élémentaire, c'est qu'elle s'exécute en temps constant. Dans l'idéal, il serait bon de ne considérer comme élémentaire que les instructions assembleur exécutées par le processeur. En Python, les opérations suivantes s'effectuent en temps constant :

— Utiliser les opérateurs `not`, `and`, et `or` sur les booléens.
— Ajouter, soustraire, multiplier, diviser, comparer deux entiers ou deux flottants. Le fait que Python travaille avec des entiers de taille variable rend ces opérations non élémentaires, mais on supposera que les entiers que nous manipulons restent de taille raisonnable (disons qu'ils sont représentables sur 64 bits) ce qui a pour conséquence le fait que les opérations arithmétiques sur ces derniers restent élémentaires.
— Calculer la longueur d'une liste ou d'une chaine de caractères avec `len(t)`, accéder à ou modifier l'élément d'indice i d'une liste par `t[i]`, accéder au caractère d'indice i d'une chaine de caractères par `s[i]`.
— Ajouter ou enlever un élément à la fin d'une liste grâce aux méthodes `append` et `pop`.
— Enfiler ou défiler un élément sur une file du module `collections`. Tester si une clé fait partie d'un dictionnaire, obtenir la valeur associée à une clé, créer, mettre à jour ou supprimer une association dans un dictionnaire.
— Affecter une variable.
— Appeler une fonction.

Cependant, les opérations suivantes ne s'effectuent pas en temps constant :

— Si x est un entier, le calcul de x^n par `x ** n` se fait en $\Theta(\log n)$.
— La concaténation `u + v` de deux chaines de caractères ou de deux listes u et v s'effectue en $\Theta(|u| + |v|)$.
— La création d'une liste de taille n par `[x] * n` s'effectue en $\Theta(n)$.
— La méthode `u.extend(v)` s'effectue en $\Theta(|v|)$.
— Le slicing `t[a:b:p]` s'effectue en un temps proportionnel à la longueur de la liste créée.

Dans de nombreux exemples, il arrive qu'on vous rappelle quelles sont les opérations élémentaires que l'on doit prendre en compte pour le calcul de la complexité. Par exemple, lors de l'étude de tris, il est courant de ne prendre en compte que le nombre de comparaisons. Dans ce cas, les autres opérations doivent être ignorées. Cependant, comme on détermine les complexités à un Θ près, la spécification exacte des opérations élémentaires à prendre en compte n'a en général aucune influence sur le résultat final.

6.1.3 Complexité dans le pire des cas

On donne toujours la complexité d'un algorithme en fonction de la taille n de l'entrée. Pourtant, la plupart des algorithmes peuvent avoir un temps d'exécution très variable entre deux entrées de même taille. La fonction suivante, qui cherche si un élément x appartient ou non à la liste t de longueur n, s'exécute en temps constant si le premier élément de a vaut x, et en temps proportionnel à n si x n'appartient pas à t.

```
def appartient(x, t):
    """appartient(x: int, t: list[int]) -> bool"""
    for i in range(len(t)):
        if t[i] == x:
            return True
    return False
```

Par défaut, nous nous intéresserons toujours à la complexité *dans le pire des cas* : autrement dit, le $C(n)$ cherché est le nombre maximum d'opérations élémentaires pour traiter une entrée de taille n. Dans ce cadre, la fonction précédente a une complexité en $\Theta(n)$.

Définition 6.1.4

Si un algorithme nécessite $C(d)$ opérations élémentaires pour s'exécuter sur une donnée d, on appelle

— *complexité dans le pire des cas* et on note $C_{\max}(n)$, le nombre maximal d'opérations élémentaires nécessaires pour traiter une donnée de taille n.

$$C_{\max}(n) \coloneqq \max_{|d|=n} C(d).$$

— *complexité dans le meilleur des cas* et on note $C_{\min}(n)$, le nombre minimal d'opérations élémentaires

nécessaires pour traiter une donnée de taille n.

$$C_{\min}(n) \coloneqq \min_{|d|=n} C(d).$$

Exemple

⇨ Si on compte comme opération élémentaire le nombre de tests `==`, la fonction `appartient` a une complexité dans le pire et dans le meilleur des cas de

$$C_{\max}(n) = n = \Theta(n), \qquad C_{\min}(n) = 1 = \Theta(1).$$

Le pire cas est obtenu lorsque le tableau ne contient pas l'élément x et le meilleur des cas est obtenu lorsque l'élément x est contenu dans la case d'indice 0 du tableau t.

6.1.4 Complexité en moyenne

Certains algorithmes peuvent être très lents dans une petite proportion de cas « pathologiques » et très efficaces sur les autres. Il peut alors être intéressant de calculer la *complexité en moyenne* de l'algorithme, c'est-à-dire l'espérance du temps de calcul pour une certaine loi de probabilité sur les données.

Ce type de complexité est plus délicat à étudier que la complexité dans le pire des cas, et ce pour deux raisons.
— Il n'est pas toujours évident de définir une loi de probabilité sur les données, et encore moins une loi de probabilité pertinente. Dans l'exemple précédent, quelle peut bien être la probabilité que le premier élément soit égal à x ?
— Une fois que l'on a fixé la loi de probabilité, les calculs sont en général beaucoup plus délicats que pour le pire cas. Il peut même être nécessaire de faire des mathématiques très difficiles.

Définition 6.1.5

Si un algorithme nécessite $C(d)$ opérations élémentaires pour s'exécuter sur une donnée d, et si $\mathbb{P}$ est une mesure de probabilité sur l'ensemble des données de taille n, on appelle *complexité en moyenne* et on note $C_{\text{moy}}(n)$ le nombre

$$C_{\text{moy}}(n) \coloneqq \sum_{|d|=n} \mathbb{P}(d) C(d)$$

Remarques

⇨ Dans le cas où il existe un nombre fini m de données $d_1, \dots, d_m$ de taille n, on prend le plus souvent la loi de probabilité uniforme. Dans ce cas

$$C_{\text{moy}}(n) \coloneqq \frac{1}{m} \sum_{k=1}^{m} C(d_k).$$

⇨ On a bien évidemment

$$C_{\text{moy}}(n) = \mathrm{O}(C_{\max}(n)) \quad \text{et} \quad C_{\text{moy}}(n) = \Omega(C_{\min}(n)).$$

⇨ L'exemple le plus connu, et sans doute le plus important, d'algorithme pour lequel il est pertinent de faire une analyse en moyenne est celui du *tri rapide*. En effet, il est en $\Theta\left(n^2\right)$ dans le pire des cas mais en $\Theta(n \log n)$ en moyenne si le tableau d'entrée est dans un ordre aléatoire. De plus, étant donné qu'il se fait en place, il est souvent plus efficace que le tri fusion qui est pourtant en $\Theta(n \log n)$ dans le pire cas.

⇨ Voici les complexités des différents algorithmes de tri que nous avons vu :

Méthode	$C_{\min}(n)$	$C_{\text{moy}}(n)$	$C_{\max}(n)$
Tri sélection	$\Theta\left(n^2\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^2\right)$
Tri insertion	$\Theta\left(n\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n^2\right)$
Tri fusion	$\Theta\left(n \log n\right)$	$\Theta\left(n \log n\right)$	$\Theta\left(n \log n\right)$
Tri rapide	$\Theta\left(n \log n\right)$	$\Theta\left(n^2\right)$	$\Theta\left(n \log n\right)$

6.1.5 Complexité temporelle et temps de calcul

Après avoir analysé la complexité temporelle d'un algorithme, on dispose d'une information du type $T(n) = \Theta(n \log n)$. En théorie, cela ne nous dit absolument rien du temps de calcul réel pour une valeur donnée de n, mais en pratique on peut en tirer quelques informations.

Valeur de la constante cachée : Pour un algorithme raisonnablement simple, on peut supposer que la constante multiplicative cachée dans le Θ est de l'ordre de 10, voire de 100. Pour certains algorithmes très sophistiqués, elle peut cependant être très grande et rendre l'algorithme beaucoup moins efficace qu'on ne le pense, voire inutilisable en pratique.

Traduction d'une opération élémentaire : Certaines des opérations élémentaires vues plus haut (ajouter deux flottants, par exemple) correspondent directement à une instruction processeur. D'autres sont plus complexes et correspondront à plusieurs instructions (une bonne centaine pour faire un `append` en Python).

Cycle processeur : Vu de l'extérieur, l'état d'un processeur évolue de manière discrète. Il est dans un certain état à l'instant t_n, dans un certain état à l'instant $t_{n+1} = t_n + h$ et dans un état non défini entre ces deux instants. Ce h est la durée d'un *cycle*, c'est-à-dire l'inverse de la *fréquence d'horloge* que les constructeurs communiquent. Comme vous le savez peut-être, la fréquence d'un processeur actuel varie entre 1 GHz et 5 GHz, et un cycle prend donc de 0.2ns à 1ns.

Temps pour exécuter une instruction : Exécuter une instruction prend au moins un cycle (un processeur peut cependant exécuter plusieurs instructions en parallèle sur un même cœur), mais peut prendre beaucoup plus longtemps. Quelques exemples :

— Ajouter deux entiers, comparer deux flottants : 1 cycle.
— Une division entière : Une vingtaine de cycles.
— Le calcul du cosinus d'un nombre flottant : Une centaine de cycles.
— Accès mémoire : entre 1 et 1000 cycles.

Une recette de cuisine : En étant optimiste

— La constante cachée vaut 2 (très optimiste).
— Chaque opération élémentaire donne 5 instructions (optimiste).
— On a un IPC (nombre d'instructions exécutées par cycle d'horloge) de 2 (très raisonnable).
— Un cycle prend 0.2ns (optimiste).

On arrive alors à un temps de calcul de $f(n)$ nanosecondes si la complexité est en $\Theta(f(n))$. C'est possible si par exemple on programme bien une multiplication matricielle en assembleur, voire en C. Si au contraire on travaille dans un langage « lent » comme Python et si l'algorithme se prête moins à un traitement efficace en machine, on peut facilement être mille fois plus lent. On pourra donc retenir la recette suivante : *Si la complexité temporelle est en $\Theta(f(n))$, pour des valeurs de n « pas trop petites », le temps de calcul sera le plus souvent compris entre $f(n)$ nanosecondes et $f(n)$ microsecondes.*

	10	100	1 000	10 000	1 000 000	10^9
$\Theta(\log n)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{n})$	1ns	10ns	100ns	100ns	1mics	1ms
$\Theta(n)$	10ns	100ns	1mics	10mics	1ms	1s
$\Theta(n \log n)$	10ns	100ns	10mics	100mics	10ms	10s
$\Theta(n^2)$	100ns	10mics	1ms	100ms	10 min	10 ans
$\Theta(n^3)$	1mics	1ms	1s	10 min	10 ans	∞
$\Theta(2^n)$	1mics	∞	∞	∞	∞	∞

Ordre de grandeur du temps de calcul pour quelques valeurs de n et complexités usuelles.

On est ici très optimiste : il faut par exemple comprendre que pour une complexité en $\Theta(n \log n)$ avec $n = 10^6$, on prendra *au mieux quelques dizaines de millisecondes*. Pour une version pessimiste, il faut essentiellement tout multiplier par mille.

6.2 Calcul de complexité temporelle

Dans cette partie, on suppose que l'on connait la complexité de toutes les fonctions prédéfinies qu'on utilise (on sait donc que `len` est en $\Theta(1)$ par exemple), et que l'on souhaite calculer la complexité temporelle dans le pire cas d'une fonction. Autrement dit, on cherche une estimation asymptotique du nombre $C(n)$ d'opérations effectuées dans le pire cas, idéalement sous la forme d'un Θ, ou d'un « bon » O. Si nous parlons d'un bon « O », c'est que si vous prouvez rigoureusement que la complexité du tri par insertion est en $O(n!)$, vous n'avez certes pas commis d'erreur, mais vous n'avez pas non plus obtenu de points.

6.2.1 Algorithme itératif

Boucles for

Pour une boucle `for`, il faut simplement sommer le nombre d'opérations pour chacune des itérations.

Commençons par l'exemple du tri par sélection, dont le code nous est rappelé ci-dessous :

```
def swap(t, i, j):
    """swap(t: list[int], i: int, j: int) -> NoneType"""
    t[i], t[j] = t[j], t[i]

def indice_minimum(t, i):
    """indice_minimum(t: list[int], i: int) -> int"""
    j_min = i
    for j in range(i + 1, len(t)):
        if t[j] < t[j_min]:
            j_min = j
    return j_min

def tri_selection(t):
    """tri_selection(t: list[int]) -> NoneType"""
    for i in range(len(t) - 1):
        j = indice_minimum(t, i)
        swap(t, i, j)
```

La fonction `swap` s'effectue en $\Theta(1)$ car elle n'effectue que deux accès mémoire et deux affectations. Si on note n la longueur du tableau, la fonction `indice_minimum` effectue quant à elle $n-1-i$ passages dans la boucle. Comme les opérations à l'intérieur de la boucle sont élémentaires, sa complexité est donc en $\Theta(n-1-i)$. La complexité de la fonction `tri_selection` est donc en

$$\begin{aligned} C(n) = \sum_{i=0}^{n-2} [\Theta(n-1-i) + \Theta(1)] &= \sum_{i=0}^{n-2} \Theta(n-1-i) = \Theta\left(\sum_{i=0}^{n-2}(n-1-i)\right) \\ &= \Theta((n-1)+\cdots+2+1) = \Theta\left(\frac{n(n-1)}{2}\right) \\ &= \Theta(n^2). \end{aligned}$$

L'algorithme de tri sélection est donc un algorithme de complexité quadratique. Remarquons que nous aurions trouvé le même résultat si nous avions seulement compté le nombre de comparaisons.

Attention à bien voir la différence entre des boucles *successives* qui donnent une complexité en $\Theta(n)$ si les corps de boucle sont des instructions élémentaires

```
for i in range(n):
    corps de..............  # On passe n fois ici
    ..............boucle
for i in range(n):
    corps de..............  # On passe n fois ici
    ..............boucle
```

et des boucles *imbriquées* qui donnent une complexité en $\Theta(n^2)$ dans la même situation.

```
for i in range(n):
    for j in range(n):
        corps de..............  # On passe n^2 fois ici
        ..............boucle
```

Prenons désormais un exemple un peu plus général. On suppose maintenant que f est une fonction de signature `f(t: list[int], i: int) -> int`, et on considère la fonction :

```
def g(t):
    """g(t: list[int]) -> NoneType"""
    n = len(t)
```

```
4       for i in range(n):
5           j = f(t, i)
6           t[i] = j
```

La ligne 3 n'est exécutée qu'une seule fois, et prend un temps unitaire. Les lignes 5 et 6, le *corps* de la boucle, sont exécutées n fois chacune. La ligne 6 est une opération élémentaire, mais la ligne 5 contient un appel à une fonction inconnue f avec le paramètre i. Le nombre total d'instructions exécutées dans cette boucle est donc

$$C(n) = \sum_{i=0}^{n-1} [1 + T(i)] .$$

où $T(i)$ correspond au nombre d'instructions pour le calcul de `f(t, i)`.

— Si f s'exécute en temps constant, par exemple si

```
def f(t, i):
    """f(t: list[int], i: int) -> int"""
    return t[i]
```

alors chaque itération est en $\Theta(1)$ et on obtient au total

$$C(n) = \sum_{i=0}^{n-1} [1 + \Theta(1)] = \sum_{i=0}^{n-1} \Theta(1) = \Theta(n).$$

— Le temps d'exécution de `f(t, i)` pourrait aussi dépendre de `n`, tout en restant indépendant de `i`. Par exemple

```
def f(t, i):
    """f(t: list[int], i: int) -> int"""
    k = 0
    n = len(t)
    for j in range(n):
        if t[j] < t[i]:
            k += 1
    return k
```

Dans ce cas, chacun des termes de la somme est un $\Theta(n)$, et l'on obtient donc

$$C(n) = \sum_{i=0}^{n-1} [1 + \Theta(n)] = \sum_{i=0}^{n-1} \Theta(n) = \Theta\left(\sum_{i=0}^{n-1} n\right) = \Theta\left(n^2\right).$$

— Considérons maintenant la fonction `f` suivante, qui s'exécute en temps $\Theta(i)$.

```
def f(t, i):
    """f(t: list[int], i: int) -> int"""
    k = 0
    for j in range(i):
        if t[j] < t[i]:
            k += 1
    return k
```

— Le calcul de la complexité totale ne pose pas de problème

$$C(n) = \sum_{i=0}^{n-1} [1 + \Theta(i)] = \sum_{i=0}^{n-1} \Theta(i) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2).$$

— Comme on a toujours $i < n$, la fonction f s'exécute en temps $\mathrm{O}(n)$. Si l'on souhaite seulement une *majoration* de la complexité totale, on peut donc écrire

$$C(n) = \sum_{i=0}^{n-1} [1 + \mathrm{O}(n)] = \sum_{i=0}^{n-1} \mathrm{O}(n) = \mathrm{O}\left(\sum_{i=0}^{n-1} n\right) = \mathrm{O}(n^2).$$

Au point précédent, on avait obtenu une estimation plus précise : notre « grand-O » est en fait un Θ. Ce ne sera pas toujours le cas. Par exemple, si la fonction f s'exécute en temps $\Theta(2^i)$, alors une majoration brutale donnerait

$$C(n) = \sum_{i=0}^{n-1} \left[1 + \Theta\left(2^i\right)\right] = \sum_{i=0}^{n-1} \Theta\left(2^i\right) = \sum_{i=0}^{n-1} \mathrm{O}\left(2^n\right) = \mathrm{O}\left(\sum_{i=0}^{n-1} 2^n\right) = \mathrm{O}(n2^n).$$

C'est correct, car c'est bien une *majoration*, mais elle est grossière. En réalité, on a

$$C(n) = \sum_{i=0}^{n-1} \Theta\left(2^i\right) = \Theta\left(\sum_{i=0}^{n-1} 2^i\right) = \Theta(2^n - 1) = \Theta(2^n).$$

Dans les exemples précédents, nous avons à plusieurs reprises dû trouver un ordre de grandeur de la somme $\sum_{k=0}^{n-1} k$. Comme nous sommes amenés à manipuler des sommes de ce type, on pourra utiliser directement les résultats suivants :

Proposition 6.2.1

Soit $\alpha, \beta \geqslant 0$ et $\gamma > 1$. Alors

$$\sum_{k=0}^{n} k^\alpha = \Theta(n^{\alpha+1}), \qquad \sum_{k=1}^{n} k^\alpha \log^\beta k = \Theta(n^{\alpha+1} \ln^\beta n) \qquad \text{et} \qquad \sum_{k=0}^{n} \gamma^k = \Theta(\gamma^n).$$

Boucle while

La différence avec une boucle `for` est qu'on ne connait pas *à priori* le nombre d'itérations. Le plus souvent, on sera amené à le majorer, mais il faudra faire attention à ne pas être trop grossier.

Revenons à l'exemple de la recherche d'un élément d'une liste triée par dichotomie, dont nous avons donné le code dans le chapitre sur les listes.

```
def dichotomie(x, t):
    """dichotomie(x: int, t: list[int]) -> bool"""
    g = 0
    d = len(t)
    while g < d:
        m = (g + d) // 2
        if x == t[m]:
            return True
        elif x < t[m]:
            d = m
        else:                # Cas où x > t[m]
            g = m + 1
    return False
```

On note g_k et d_k les valeurs respectives de `g` et `d` lors du passage d'indice k dans la boucle. On a $g_0 \coloneqq 0$ et $d_0 \coloneqq n$ et, dans les deux cas où x n'est pas trouvé à l'itération k, on montre que

$$d_{k+1} - g_{k+1} \leqslant \frac{d_k - g_k}{2}$$

Autrement dit, la largeur $l_k \coloneqq d_k - g_k$ de la tranche de recherche est au moins divisée par 2 à chaque itération. On en déduit que

$$l_k \leqslant \frac{n}{2^k}.$$

Dès que $l_k \leqslant 1$, on sort de la boucle au plus tard à l'itération suivante. Or

$$\frac{n}{2^k} \leqslant 1 \iff 2^k \geqslant n \iff \log_2(2^k) \geqslant \log_2 n \iff k \geqslant \log_2 n \iff k \geqslant \lceil \log_2 n \rceil.$$

Cet algorithme effectue donc au plus $1 + \lceil \log_2 n \rceil$ itérations. La complexité dans le pire des cas de la recherche dichotomique est donc en $\mathrm{O}(\log n)$. Comme ces itérations sont toutes effectuées si la liste ne contient pas l'élément recherché, on se convaincra que cette complexité est en fait en $\Theta(\log n)$. D'autre part, la complexité dans le meilleur des cas est en $\Theta(1)$, ce cas étant atteint lorsque l'élément x est exactement au milieu de notre liste.

Remarque

⇨ Attention, il ne faut pas oublier de prendre en compte le temps nécessaire à évaluer la condition de la boucle. Pour prendre un exemple un peu idiot, la fonction suivante s'exécute en temps $\Theta(|u|^2)$.

```python
def somme(t):
    """somme(t: list[int]) -> int"""
    s = 0
    for x in t:
        s = s + x
    return s

def f(u, borne):
    """f(u: list[int], borne: int) -> int"""
    i = 0
    while i < len(u) and somme(u[:i]) < borne: # Evaluation en O(i) !!!
        i += 1                                # Corps de la boucle en O(1)
    return i
```

6.2.2 Algorithme récursif

Pour la complexité temporelle, les algorithmes récursifs font naturellement apparaitre une formule de récurrence. Diverses techniques que nous allons voir nous permettent d'en déduire le comportement asymptotique.

Dans les cas les plus simples, on commencera par établir une forme close pour $C(n)$. Prenons l'exemple de la fonction suivante qui calcule la factorielle d'un entier positif :

```python
def factorielle(n):
    """factorielle(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

On souhaite estimer la complexité de cette fonction en calculant le nombre $C(n)$ de multiplications effectuées. La lecture du code nous donne

$$C(0) = 0, \qquad \text{et} \qquad \forall n \in \mathbb{N}^*, \quad C(n) = C(n-1) + 1.$$

On en déduit que $C(n) = n$ et donc que $C(n) = \Theta(n)$.

Mais rapidement, on se rend compte que les formes closes pour $C(n)$ deviennent complexes. Considérons par exemple l'algorithme d'exponentiation rapide dans sa version récursive :

```python
def expo_rapide(x, n):
    """expo_rapide(x: int, n: int) -> int"""
    if n == 0:
        return 1
    else:
        p = n // 2
        y = expo_rapide(x, p)
        if n % 2 == 0:
            return y * y
        else:
            return x * y * y
```

Si on note $C(n)$ le nombre de multiplications effectuées pour le calcul de x^n, on a

$$C(0) = 0, \quad \text{et} \quad \forall n \in \mathbb{N}^*, \quad C(n) = \begin{cases} C(\lfloor n/2 \rfloor) + 1 & \text{Si } n \text{ est pair,} \\ C(\lfloor n/2 \rfloor) + 2 & \text{Si } n \text{ est impair.} \end{cases}$$

Cette récurrence devient plus facile à appréhender si l'on décompose n en base 2. Si $n = \underline{d_{w-1} \dots d_1 d_0}_2$, on a

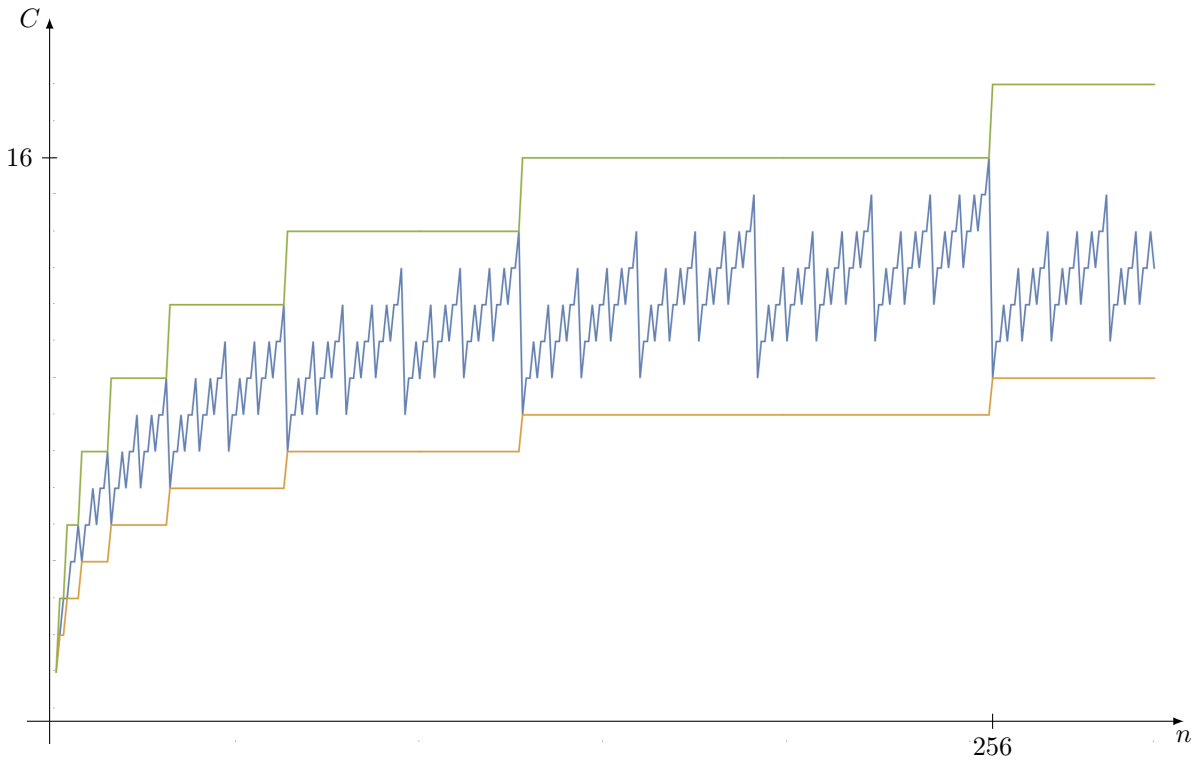
$$C(\underline{d_{w-1} \dots d_1 d_0}_2) = \begin{cases} C(\underline{d_{w-1} \dots d_1}_2) + 1 & \text{Si } d_0 = 0, \\ C(\underline{d_{w-1} \dots d_1}_2) + 2 & \text{Si } d_0 = 1. \end{cases}$$

En notant $z(n)$ le nombre de 0 dans la décomposition de n en base 2 et $u(n)$ le nombre de 1 dans cette même décomposition, on en déduit que $C(n) = z(n) + 2u(n)$. Autrement dit, si on définit $b(n) := z(n) + u(n)$ comme le

nombre de chiffres dans la décomposition en base 2 de n, on a $C(n) = b(n) + u(n)$. Puisque $b(n) = 1 + \lfloor \log_2 n \rfloor$ et $1 \leqslant u(n) \leqslant b(n)$, on en déduit que

$$2 + \lfloor \log_2 n \rfloor \leqslant C(n) \leqslant 2 + 2\lfloor \log_2 n \rfloor,$$

ce qui nous permet de conclure que $C(n) = \Theta(\log n)$. Le graphe ci-dessous représente au centre, en bleu, la courbe d'évolution de $C(n)$, encadrée par les deux fonctions obtenues dans l'inégalité précédente.



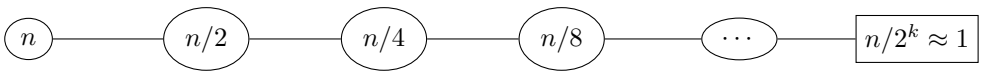
Le fait qu'un programme aussi simple ait une complexité au comportement aussi erratique nous permet de réaliser qu'il est illusoire d'obtenir des formes closes de $C(n)$ pour la plupart des programmes que nous rencontrerons. C'est pourquoi, on se contentera d'une estimation asymptotique. Malheureusement, l'obtention rigoureuse de ces estimations est extrêment technique ; on se permettra donc de sacrifier une trop grande rigueur mathématique. Les deux techniques suivantes nous seront très utiles.

— *Technique de la sommation télescopique* : Elle consiste à « résoudre » la relation de récurrence en se permettant quelques approximations. Dans notre cas, on confondra $\lfloor n/2 \rfloor$ et $n/2$ et on écrira $C(n) = C(n/2) + \Theta(1)$, puis

$$\begin{aligned} C(n) - C(n/2) &= \Theta(1) \\ C(n/2) - C(n/4) &= \Theta(1) \\ \vdots &= \vdots \\ C(n/2^{k-1}) - C(n/2^k) &= \Theta(1) \end{aligned}$$

où k est tel que $n/2^k \approx 1$, c'est-à-dire $k \approx \log_2 n$. En sommant ces égalités, on obtient $C(n) - C(1) = \Theta(1) \log_2 n$, donc $C(n) = \Theta(\log n)$.

— *Technique de l'arbre d'appels* : Cette technique commence par faire une esquisse de l'arbre d'appels de notre fonction récursive :



Puisque $n/2^k \approx 1$, on en déduit que $k \approx \log_2 n$. La profondeur de notre arbre d'appels est donc de l'ordre de $\log_2 n$. Pour chaque appel, on calcule ensuite sa contribution propre à la complexité totale. Autrement dit, on estime les opérations élémentaires effectuées par cet appel en ignorant celles effectuées par ses enfants. Dans notre cas, chaque appel a une complexité propre en $\Theta(1)$. On somme ensuite ces données sur l'ensemble des noeuds de l'arbre. Dans notre cas, on obtient une complexité totale en $C(n) = \Theta(1) \log_2 n = \Theta(\log n)$.

Ces deux techniques sont sur le fond assez semblables, mais la technique de l'arbre d'appels montrera rapidement sa supériorité lorsque les arbres seront plus complexes, comme nous allons le voir dans l'exemple suivant.

Nous allons revenir sur l'algorithme de tri fusion dont nous rappelons le code ici. On commence par remarquer que la fonction `fusion(t1, t2)` a une complexité en $\Theta(|t_1| + |t_2|)$. En effet, chaque passage dans la boucle `while` ajoute un élément parmi les listes t_1 et t_2 à notre liste t. Les appels à `extend` ajoutent les éléments restants une fois qu'une des deux listes est épuisée.

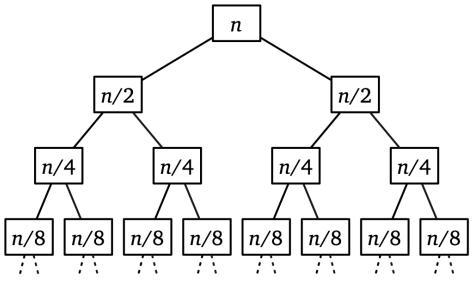
```
def fusion(t1, t2):
    """fusion(t1: list[int], t2: list[int]) -> list[int]"""
    t = []
    i1 = 0
    i2 = 0
    while i1 < len(t1) and i2 < len(t2):
        if t1[i1] < t2[i2]:
            t.append(t1[i1])
            i1 = i1 + 1
        else:
            t.append(t2[i2])
            i2 = i2 + 1
    t.extend(t1[i1:])
    t.extend(t2[i2:])
    return t
```

```
def tri_fusion(t):
    """tri_fusion(t: list[int]) -> list[int]"""
    n = len(t)
    if n <= 1:
        return t[:]
    n1 = n // 2
    t1 = tri_fusion(t[0:n1])
    t2 = tri_fusion(t[n1:n])
    t = fusion(t1, t2)
    return t
```

Si on note $C(n)$ la complexité de la fonction `tri_fusion`, on a donc

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + \Theta(n).$$

Pour obtenir une estimation asymptotique de $C(n)$, nous allons utiliser la technique de l'arbre d'appels. La racine représente l'appel initial à `tri_fusion` avec une liste de taille n, qui appelle récursivement notre fonction avec des listes de tailles $n/2$, qui elles-mêmes appellent récursivement notre fonction avec des listes de taille $n/4$, etc.



On tombe sur un cas de base lorsque la taille des listes est inférieure ou égale à 1. La hauteur h de cet arbre vérifie donc $n/2^h \approx 1$, ce qui nous donne $h \approx \log_2 n$. Pour estimer la complexité de l'appel initial, il suffit désormais de prendre en compte le cout propre de chaque appel, c'est-à-dire le cout de la fusion. Pour simplifier le calcul, on va sommer ligne par ligne.

— Sur la première ligne, on compte une fusion pour une liste de taille n ; ce cout est de $\Theta(n)$.
— Sur la seconde ligne, on compte 2 fusions pour des listes de taille $n/2$; ce cout est de $2\Theta(n/2) = \Theta(n)$.
— Sur la troisième ligne, on compte 4 fusions pour des listes de taille $n/4$; ce cout est de $4\Theta(n/4) = \Theta(n)$.

On constate que sur chaque ligne, le cout est de $\Theta(n)$. Comme l'arbre est de hauteur $\log_2 n$, on en déduit que la complexité du tri fusion est de $C(n) = \Theta(n) \log_2 n = \Theta(n \log n)$.

6.3 Calcul de complexité spatiale

6.3.1 Algorithme itératif

La complexité en espace mesure la quantité de mémoire de travail utilisée par l'algorithme. On ne compte pas la taille des données.

La fonction suivante calcule le n-ième nombre de Fibonacci pour $n \geqslant 1$:

```
1 def fibo(n):
2     """fibo(n: int) -> int"""
3     t = [None] * (n + 1)
4     t[0] = 0
5     t[1] = 1
6     for i in range(2, n + 1):
7         t[i] = t[i - 1] + t[i - 2]
8     return t[n]
```

Elle a une complexité en espace en $\Theta(n)$ puisqu'elle alloue un tableau de taille $n + 1$ pour réaliser ses calculs. Sa complexité en temps est également linéaire.

La fonction suivante effectue le même calcul :

```
1 def fibo(n):
2     """fibo(n: int) -> int"""
3     u = 0
4     v = 1
5     for _ in range(n):
6         u, v = v, u + v
7     return u
```

Elle a également une complexité temporelle linéaire, mais sa complexité spatiale est constante, puisqu'elle utilise uniquement deux variables entières.

De nombreux algorithmes « échangent de l'espace contre du temps » : pour obtenir une meilleure complexité temporelle, on accepte une moins bonne complexité spatiale. C'est par exemple le cas de tous les algorithmes de programmation dynamique, que nous étudierons ultérieurement. C'est souvent efficace en pratique, mais il y a un certain nombre de seuils qu'il est couteux de dépasser : les données ne rentrent plus dans le cache, les données ne rentrent plus en mémoire vive, les données ne rentrent plus dans le stockage de masse local, etc.

Taille des données et capacité des mémoires : Il faut avoir en tête quelques ordres de grandeur :
- Un entier ou un flottant prennent en général 8 octets en mémoire.
- Un ordinateur typique a quelques gigaoctets de mémoire vive.
- Ce même ordinateur peut disposer de quelques téraoctets de mémoire de stockage, mais si l'on doit utiliser cette mémoire pour les calculs, les performances peuvent facilement être divisées par mille, voire un million.

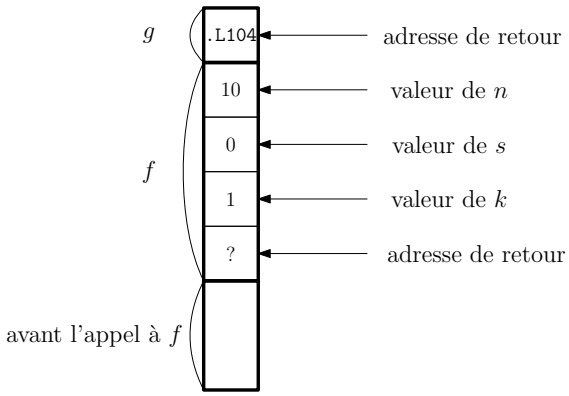
6.3.2 Algorithme récursif

Pile d'appels

Considérons les deux fonctions suivantes

```
1 def g(n):
2     """g(n: int) -> int"""
3     return n * n
4
5 def f(n):
6     """f(n: int) -> int"""
7     s = 0
8     for k in range(1, n + 1):
9         s = s + g(k)     # .L104
10    return s
```

Voici l'état de la pile d'appels lors du premier passage ligne 9 dans le calcul de $f(10)$.



Que se passe-t-il « en vrai » quand on calcule $f(2)$? f crée des variables locales, leur donne une valeur, puis appelle g avec l'argument 1. Pour cela, elle cède le contrôle d'exécution à g et l'exécution « saute » au début du code de g. Quand g aura fini son calcul, il faudra qu'elle rende la main à f : mais l'exécution de f doit reprendre là où elle s'est arrêtée, et dans l'environnement qui était valable à ce moment : `s` doit valoir 0, `k` doit valoir 1, etc. Il faut donc que f sauvegarde un certain nombre d'informations avant d'appeler g.

Cette sauvegarde d'informations se fait sur la *pile d'appels*. Vu de loin, un élément (on parle de *stackframe* ou *bloc d'activation*) de la pile contient toutes les informations relatives à un appel donné d'une fonction donnée :

— Les arguments de la fonction
— Les variables locales de la fonction.
— Le point auquel l'exécution du programme devra reprendre quand l'appel sera terminé.

Quand une fonction est appelée, elle crée une *stackframe* au sommet de la pile, qu'elle supprimera juste avant de renvoyer son résultat et de passer le contrôle au point qui était sauvegardé. À tout moment de l'exécution du programme, la hauteur de la pile (en nombre de *frames*) est donc égale au nombre de fonctions actives, c'est-à-dire ayant été appelées et n'ayant pas encore retourné leur résultat.

Une remarque sur la terminologie : le nom de « pile » n'est bien sûr pas anodin, il y a bien une très forte analogie avec la structure de données que nous avons vue. On empile au moment des appels, on dépile au moment des retours.

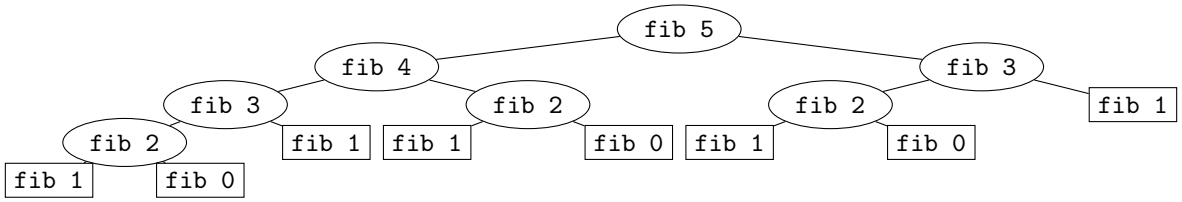
Pile et fonction récursive

Pour l'instant, on a considéré le cas d'une fonction f qui appelle une fonction g. Mais rien n'empêche bien sûr f de s'appeler elle-même : ce sera le cas si f est récursive. Dans ce cas, il y aura *un bloc d'activation par appel actif de f*. C'est logique, puisque chacun de ses appels possède ses propres variables locales, et son propre compteur de programme étant donné que chacun des appels en est à un point différent de son exécution.

Pour visualiser ce type de situation, il est plus intéressant de réfléchir en termes d'*arbres d'appels*. Considérons une fonction calculant les termes de la suite de Fibonacci de manière récursive et naïve. Ici, « naïve » est à comprendre au sens de *scandaleusement mauvais et contraire aux bonnes mœurs*.

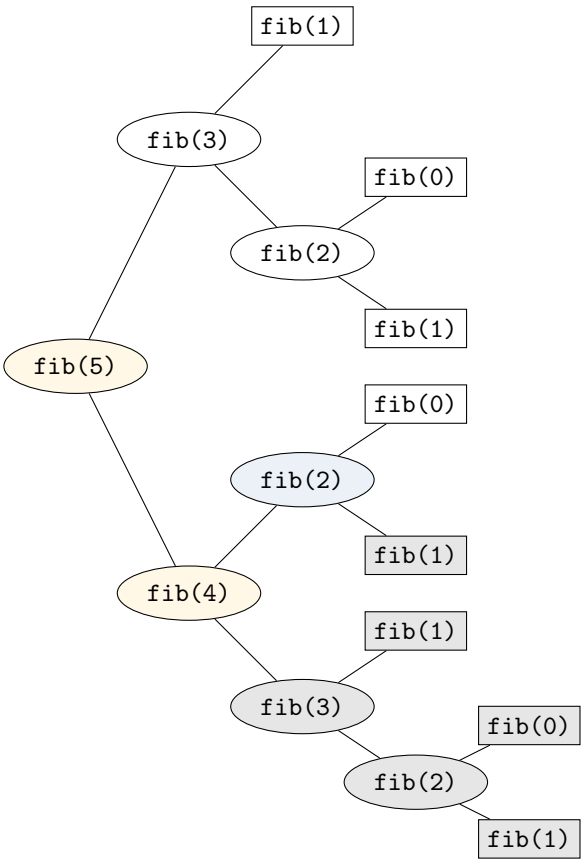
```
1 def fib(n):
2     """fib(n: int) -> int"""
3     if n <= 1:
4         return n
5     else:
6         return fib(n - 1) + fib(n - 2)
```

Un appel à `fib(5)` produit un appel à `fib(4)` et un à `fib(3)`, qui produisent eux-mêmes d'autres appels, etc. La situation est très bien résumée par l'arbre d'appels suivant pour `fib(5)`.



Voici l'évolution de la pile d'appels lors du calcul de `fib(5)`.

```
    fib(5)
    fib(5) | fib(4)
    fib(5) | fib(4) | fib(3)
    fib(5) | fib(4) | fib(3) | fib(2)
    fib(5) | fib(4) | fib(3) | fib(2) | fib(1)
    fib(5) | fib(4) | fib(3) | fib(2)
    fib(5) | fib(4) | fib(3) | fib(2) | fib(0)
    fib(5) | fib(4) | fib(3) | fib(2)
    fib(5) | fib(4) | fib(3)
    fib(5) | fib(4) | fib(3) | fib(1)
    fib(5) | fib(4) | fib(3)
    fib(5) | fib(4)
    fib(5) | fib(4) | fib(2)
    fib(5) | fib(4) | fib(2) | fib(1)
--> fib(5) | fib(4) | fib(2)
    fib(5) | fib(4) | fib(2) | fib(0)
    fib(5) | fib(4) | fib(2)
    fib(5) | fib(4)
    fib(5)
    fib(5) | fib(3)
    fib(5) | fib(3) | fib(2)
    fib(5) | fib(3) | fib(2) | fib(1)
    fib(5) | fib(3) | fib(2)
    fib(5) | fib(3) | fib(2) | fib(0)
    fib(5) | fib(3) | fib(2)
    fib(5) | fib(3)
    fib(5) | fib(3) | fib(1)
    fib(5) | fib(3)
    fib(5)
```



Il faut avoir de cet arbre une vision dynamique : on en effectue un parcours en profondeur. Quand on est en train de visiter un nœud, les appels actifs sont ceux correspondant au nœud actuel ainsi qu'à tous ses ancêtres, c'est-à-dire tous les nœuds situés sur le chemin qui le relie à la racine. La hauteur maximale de la pile est égale à la hauteur de l'arbre. Ici, cette hauteur est de l'ordre de n alors que le temps d'exécution est exponentiel.

Exercice 2

⇨ Montrer que la complexité temporelle de la fonction `fib` est en

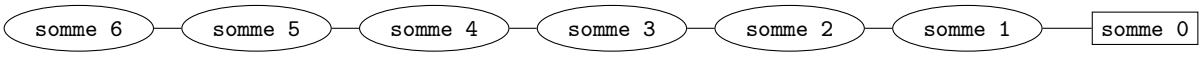
$$\Theta(\varphi^n)$$

où $\varphi \coloneqq (1+\sqrt{5})/2$ est le nombre d'or.

Cependant, si l'on s'intéresse à une fonction ayant une structure d'appels plus simple, les choses peuvent être différentes.

```python
def somme(n):
    """somme(n: int) -> int"""
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

Ici, l'arbre d'appels obtenu est en fait linéaire. Voici par exemple l'arbre d'appels pour `somme(6)` :



Le problème, qui n'est pas forcément évident quand on regarde le code, est qu'un appel à `somme(n)` demande un espace mémoire proportionnel à n : c'est la hauteur maximale de la pile d'appels. De plus, l'espace disponible sur la pile est bien plus limité que l'espace mémoire total. Le résultat :

```
In [1]: somme(3000)
RecursionError: maximum recursion depth exceeded in comparison
```

Notons que le temps de calcul n'est pas le problème : le *stackoverflow* se produit presque instantanément.

Proposition 6.3.1

La complexité en espace d'une fonction récursive est au moins égale à sa profondeur maximale de récursion, c'est-à-dire à la profondeur de son arbre d'appels.

Remarques

⇨ Elle peut bien sûr être supérieure à cette profondeur, typiquement si la fonction crée des listes auxiliaires.

⇨ Par défaut, l'espace disponible sur la pile est assez faible, de l'ordre de quelques mégaoctets. On peut donc faire un *stackoverflow* bien avant d'épuiser la mémoire disponible.

Ici, cette complexité linéaire en espace est problématique : une fonction itérative aura clairement une utilisation mémoire constante, sauf si l'on s'amuse à stocker tous les résultats intermédiaires.

```
def somme(n):
    """somme(n: int) -> int"""
    s = 0
    for k in range(1, n + 1):
        s += k
    return s
```

Et le problème disparait :

```
In [2]: somme(3000)
Out[2]: 4501500
```

Chapitre 7

Correction

« Beware of bugs in the above code ; I have only proved it correct, not tried it. »

— Donald Knuth (1938–)

THE PROBLEM WITH AVERAGING STAR RATINGS

7.1 Correction

7.1.1 Spécification d'une fonction

La *spécification* d'une fonction, c'est le contrat qu'elle doit respecter. On peut la découper ainsi :

— *Entrées :*

— *Nombre et types des arguments* : par exemple, « cette fonction prend en entrée une liste t d'entiers et un entier x ».

  - *Préconditions* : une ou plusieurs conditions qui doivent être vérifiées par les entrées pour que la fonction s'exécute correctement. Par exemple, « les éléments de la liste doivent être distincts », « la liste doit être triée par ordre croissant », « l'entier passé en argument est positif », etc. Si ces préconditions ne sont pas vérifiées, la fonction peut avoir un comportement arbitraire. Autrement dit, elle fait ce qu'elle veut, par exemple renvoyer un résultat arbitraire, planter, formater le disque dur, envoyer la totalité des images présentes sur votre ordinateur à tous vos contacts de messagerie, etc.
- *Sorties :*
  - *Type du résultat* : par exemple, « cette fonction renvoie un entier », ou « cette fonction renvoie un tuple formé d'un entier et d'une liste d'entiers », ou « cette fonction renvoie `None` ».
  - *Valeur du résultat* : si la fonction renvoie une « vraie » valeur (différente de `None`), que doit vérifier cette valeur ? Par exemple, « la fonction renvoie $n!$ où n est l'argument » ou « la fonction renvoie une liste contenant les mêmes éléments que l'argument, mais triée par ordre croissant ».
  - *Effets secondaires* (éventuellement) : Tous les effets que la fonction a sur le monde, en dehors du renvoi de son résultat. Typiquement, modification de l'argument ou d'une variable globale, affichage sur l'écran, suppression de toutes les données de l'ordinateur, envoi de missiles intercontinentaux, etc. Par exemple : « après l'exécution de la fonction, le tableau passé en argument est trié et contient les mêmes éléments qu'au départ ».

Remarques

⇨ On peut regrouper « effets secondaires » et « valeur du résultat » sous le terme *postconditions*.

⇨ Une fonction ne doit pas avoir d'effets secondaires autres que ceux apparaissant dans sa spécification. Par exemple, la fonction suivante n'est pas une manière acceptable de calculer le maximum d'un tableau.

```
def apres_moi_le_deluge(t):
    """apres_moi_le_deluge(t: list[int]) -> int"""
    for i in range(1, len(t)):
        t[i] = max(t[i], t[i - 1])
    return t[-1]
```

En effet, son exécution ne laisse pas le système dans un état acceptable :

```
In [1]: [7, 2, 9, 0]

In [2]: apres_moi_le_deluge(t)
Out[2]: 9

In [3]: t
Out[3]: [7, 7, 9, 9]
```

⇨ En pratique, on essaie souvent d'éviter les comportements totalement arbitraires. S'il y a un moyen simple et efficace de vérifier que les préconditions sont remplies, on peut préférer faire ces tests pour détecter un éventuel problème le plus tôt possible. Pour cela, on utilisera le mot clé `assert` déjà utilisé pour les tests unitaires. Par exemple, pour une fonction calculant la factorielle, on écrit :

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    assert n >= 0
    f = 1
    for i in range(1, n + 1):
        f = f * i
    return f
```

Ainsi, un appel sur un entier $n < 0$ sera immédiatement détecté comme une erreur :

```
In [1]: factorielle(-1)
AssertionError:
```

Sans le `assert`, la fonction aurait renvoyé un résultat dénué de sens, 1 en l'occurrence, sans se plaindre, ce qui aurait rendu l'erreur beaucoup plus difficile à détecter.

⇨ Attention cependant, ce conseil n'est pas toujours possible à appliquer, car la précondition peut être trop couteuse, voire impossible, à vérifier. Par exemple, si l'on effectue une recherche dichotomique dans une liste triée, algorithme qui s'exécute dans le pire des cas en $\Theta(\log n)$, il serait absurde de vérifier que la liste est bien triée car cette opération a une complexité en $\Theta(n)$. Enfin, il faut avoir conscience que ces assertions ne sont utiles qu'en « production » et il est bien entendu inutile d'en placer dans un écrit de concours, sauf si cela vous est explicitement demandé.

7.1.2 Correction partielle, correction totale

Définition 7.1.1: Terminaison d'une fonction

On dit qu'une fonction *termine* lorsqu'elle renvoie un résultat en un nombre fini d'étapes, quelles que soient les valeurs de ses paramètres vérifiant les préconditions.

Remarque

⇨ Le nombre d'étapes de calcul ne sera en général pas *borné*, puisqu'il dépend de la valeur des paramètres.

Exemple

⇨ La fonction suivante calcule $n!$ pour $n \geqslant 0$.

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

On considère qu'elle termine, car il y a une précondition $n \geqslant 0$. Cependant, si on l'appelle sur un entier $n < 0$, l'appel récursif ne termine pas.

Définition 7.1.2: Correction partielle

Une fonction est dite *partiellement correcte* par rapport à sa spécification lorsque, quelles que soient les valeurs des paramètres vérifiant les préconditions :
- Soit elle renvoie un résultat conforme à la spécification.
- Soit elle ne termine pas.

Définition 7.1.3: Correction totale

Une fonction est dite *totalement correcte*, ou simplement *correcte*, si elle est partiellement correcte et qu'elle termine.

Exemples

⇨ La fonction suivante est partiellement correcte, vis-à-vis de n'importe quelle spécification : il n'y a aucun risque qu'elle renvoie un résultat incorrect.

```
def f(x):
    return f(x)
```

⇨ La fonction suivante est censée calculer x^n pour $x \in \mathbb{Z}$ et $n \in \mathbb{N}$.

```
def puissance(x, n):
    """puissance(x: int, n: int)"""
    if n == 1:
        return x
    else:
        return x * puissance(x, n - 1)
```

Elle est partiellement correcte, mais pas totalement correcte : en effet, `puissance(x, 0)` ne termine pas.

7.2 Algorithme itératif

7.2.1 Terminaison

La preuve de la terminaison d'un programme n'utilisant que des boucles `for` est immédiate. En présence de boucles `while`, prouver la terminaison peut être arbitrairement compliqué : ce problème est *indécidable*, c'est-à-dire qu'il n'existe pas d'algorithme permettant de déterminer si un programme quelconque termine. Cela n'empêche pas de montrer la terminaison dans de nombreux cas particuliers.

Définition 7.2.1: Variant de boucle

Un *variant de boucle* est une quantité :
- entière
- minorée
- qui décroit *strictement* à chaque passage dans une boucle.

Proposition 7.2.2

Si une boucle admet un variant de boucle, alors elle termine.

Remarque

⇨ Une quantité entière, *majorée* et qui *croît* strictement fait aussi l'affaire.

Exemple

⇨ Considérons la fonction suivante.

```
def log2(n):
    """log2(n: int) -> int"""
    i = 0
    x = n
    while x > 1:
        x = x // 2
        i = i + 1
    return i
```

Alors x est un variant de boucle.
- C'est un entier.
- Il est minoré par 1, tant qu'on est dans la boucle.
- En notant x' la valeur en fin d'itération, on a $x' := \lfloor x/2 \rfloor \leqslant x/2 < x$ puisque $x \geqslant 1$, donc il est strictement décroissant.

Cette fonction termine donc. Attention, si l'on remplace la condition de la boucle par `x >= 0`, la terminaison n'est plus assurée, car la décroissance n'est plus stricte.

Exercice 1

⇨ On considère la fonction suivante.

```
def disjoints(u, v):
    """disjoints(u: list[int], v: list[int]) -> bool"""
    iu = 0
    iv = 0
    nu = len(u)
    nv = len(v)
    while iu < nu and iv < nv and u[iu] != v[iv]:
        if u[iu] < v[iv]:
            iu = iu + 1
        else:
            iv = iv + 1
    return iu == nu or iv == nv
```

1. `iu` est-il un variant de boucle ? Même question pour `iv`.
2. Identifier un variant de boucle.
3. Quelle précondition doit être vérifiée par `u` et `v` pour que cette fonction soit « correcte ». Il faut bien sûr commencer par préciser ce que *correcte* signifie ici, en s'aidant du nom de la fonction.

Les variants de boucle ne sont pas le seul outil : fondamentalement, tout type de raisonnement peut être utilisé pour prouver la terminaison d'une fonction.

Exemples

⇨ Considérons la fonction ci-dessous.

```python
def inv_fact(n):
    """inv_fact(n: int) -> int"""
    i = 0
    f = 1
    while f < n:
        i = i + 1
        f = f * i
    return i
```

Une récurrence simple montre qu'après k passages dans la boucle, i vaut k et f vaut $k!$. Comme $k! \to +\infty$, il est alors « clair » que ce programme termine pour toute valeur de n.

⇨ Considérons maintenant le programme suivant.

```python
def syracuse(n):
    """syracuse(n: int) -> int"""
    k = n
    i = 0
    while k != 1:
        i = i + 1
        if k % 2 == 0:
            k = k // 2
        else:
            k = 3 * k + 1
    return i
```

Si vous arrivez à montrer qu'il termine pour toute valeur de n, faites-moi signe.

7.2.2 Correction

Les preuves de correction simples de programmes itératifs reposent sur le principe d'*invariant de boucle*, idée très similaire à celle d'une récurrence mathématique.

Définition 7.2.3: Invariant de boucle

Un *invariant de boucle* est un *prédicat* $\mathcal{I}$ ayant les propriétés suivantes :
- il est vrai avant de rentrer dans la boucle
- si il est vrai au début d'une itération, il reste vrai à la fin de cette itération.

Remarques

⇨ Un invariant de boucle n'a cependant aucune raison d'être vrai en milieu d'itération.

⇨ Pour une boucle *conditionnelle* (boucle `while`) avec une condition $\mathcal{C}$ et un invariant $\mathcal{I}$, on commence par prouver que $\mathcal{I}$ est vérifié avant de rentrer dans la boucle, puis on montre l'hérédité, c'est-à-dire que $(\mathcal{C}$ et $\mathcal{I}) \longrightarrow \mathcal{I}$: si la condition de boucle et l'invariant sont vérifiés en début d'itération, alors l'invariant est vérifié en fin d'itération. En sortie de boucle, (non $\mathcal{C}$) et $\mathcal{I}$ seront alors vrais.

⇨ Pour une boucle *inconditionnelle* (boucle `for`), on rédigera la preuve d'invariant en incorporant l'indice de boucle au prédicat. Pour effectuer la correction d'une boucle du type « `for k in range(a, b)` », on définit les prédicats $\mathcal{I}_a, \dots, \mathcal{I}_b$ et on prouve que :
- $\mathcal{I}_a$ est vérifié avant de rentrer dans la boucle.
- Pour tout $k \in [\![a, b[\![$, si $\mathcal{I}_k$ est vérifié au début de l'itération d'indice k, $\mathcal{I}_{k+1}$ est vérifié en fin d'itération.

En sortie de boucle, $\mathcal{I}_b$ sera alors vrai.

⇨ Comme pour la terminaison, les preuves de correction peuvent être arbitrairement difficiles : la correction d'un programme peut dépendre d'une conjecture mathématique, par exemple.

⇨ Dans les preuves, on notera souvent x la valeur de la variable `x` en début d'itération et x' sa valeur en fin d'itération.

Exemple

⇨ On souhaite montrer que le programme suivant renvoie bien un indice du minimum de t. Pour cela, on considère l'invariant de boucle $\mathcal{H}_i$: « $m = \min(t_0, \dots, t_{i-1})$ et `t[ind] = m` ». On note n la longueur de t.

```python
def ind_min(t):
    """ind_min(t: list[int]) -> int"""
```

```
    ind = 0
    m = t[0]
    for i in range(1, len(t)):
        if t[i] < m:
            m = t[i]
            ind = i
    return ind
```

— $\mathcal{H}_1$ est vérifié avant de rentrer dans la boucle car $ind = 0$ et $m = t_0 = \min(t_0)$.
— Pour $1 \leqslant i \leqslant n-1$, en supposant $\mathcal{H}_i$ vraie en début d'itération, on a deux cas :
  — Si $t_i < \min(t_0, \dots, t_{i-1}) = m$, alors en fin d'itération $m' = t_i$ et $ind' = i$, ce qui est correct puisqu'alors $\min(t_0, \dots, t_i) = t_i$.
  — Sinon, on a $\min(t_0, \dots, t_i) = \min(t_0, \dots, t_{i-1}) = m = t_{ind}$. Or on ne fait rien dans ce cas et l'on a donc $m' = m$ et $ind' = ind$, ce qui est bien correct.

À la fin de l'exécution, $\mathcal{H}_n$ est donc vraie, c'est-à-dire $m = \min(t_0, \dots, t_{n-1}) = \min t$ et `t[ind] = m`. La variable `ind` contient donc bien un indice du minimum de t.

En pratique, dans un cas aussi simple, on se contentera au mieux de donner l'invariant de boucle sans démonstration. Dans des cas plus compliqués, en revanche, l'invariant de boucle est indispensable.

7.2.3 Exemples fondamentaux

Les deux algorithmes présentés ici sont à connaitre absolument.

Exponentiation rapide, version itérative

On considère la fonction `expo(a: int, n: int) -> int`, dont la spécification est :

Précondition : $n \geqslant 0$

Résultat : `expo(a, n)` $= a^n$

```
def expo(a, n):
    """expo(a: int, n: int) -> int"""
    p = n
    x = 1
    b = a
    while p != 0:
        if p % 2 == 1:
            x = x * b
        b = b * b
        p = p // 2
    return x
```

1. Montrer que $x \cdot b^p = a^n$ est un invariant de boucle.
2. En déduire la correction partielle de la fonction.
3. Montrer la correction totale de la fonction.

Recherche dichotomique

On considère l'algorithme suivant :

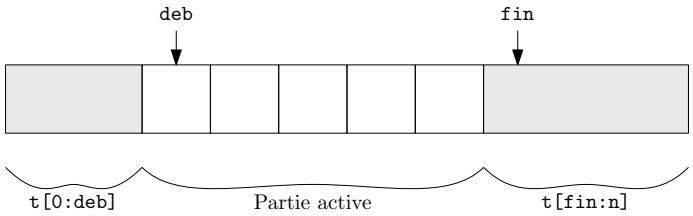
Entrées : un tableau $t = (t_0, \dots, t_{n-1})$ et une valeur x

Précondition : t est trié par ordre croissant

Résultat : un indice $i \in [\![0, n[\![$ tel que $t_i = x$ s'il en existe un, n sinon.

Algorithme 1 Recherche dichotomique dans un tableau trié

fonction RECHERCHE(x, t)
  $deb \leftarrow 0$
  $fin \leftarrow n$
  tant que $fin - deb > 0$ **faire**
    $milieu \leftarrow (deb + fin)//2$ ▷ Division entière
    si $t_{milieu} = x$ **alors**
      renvoyer $milieu$
    sinon si $t_{milieu} < x$ **alors**
      $deb \leftarrow milieu + 1$
    sinon
      $fin \leftarrow milieu$
    fin si
  fin tant que
  renvoyer n
fin fonction



1. Montrer que cet algorithme termine.
2. Montrer que si l'algorithme renvoie un indice $i \neq n$, alors ce résultat est correct.
3. Montrer qu'on a l'invariant suivant : « $x \notin t[0 : deb] \ \cup \ t[fin : n]$ ».
4. En déduire la correction de l'algorithme.
5. L'algorithme reste-t-il totalement ou partiellement correct si
   (a) on remplace la ligne $deb \leftarrow milieu + 1$ par $deb \leftarrow milieu$?
   (b) on remplace la ligne $fin \leftarrow milieu$ par $fin \leftarrow milieu - 1$?

7.3 Algorithme récursif

Une première remarque s'impose : il n'y a pas de différence fondamentale entre un programme écrit à l'aide de boucles ou de manière récursive. En effet, il est facile de remplacer toutes les boucles `while` par des récursions, et toujours possible, mais pas facile en règle générale, de remplacer la récursion par une boucle `while`. En un certain sens, tout ce qui a été dit sur les programmes itératifs s'applique donc aux programmes récursifs, en particulier le caractère arbitrairement difficile des preuves de terminaison.

7.3.1 Principe général

En première approche, la terminaison d'un programme récursif repose sur l'existence d'un certain nombre, potentiellement infini, de cas de base et sur la certitude que toute suite d'appels finit par arriver sur l'un de ces cas de base.

Le cas le plus simple et le plus fréquent est celui d'un programme ayant une définition de ce type :
— $f(0)$ est donné.
— $\forall n \in \mathbb{N},\ f(n+1) \coloneqq g(n, f(n))$ où g est une fonction que l'on sait calculer.
Il est alors immédiat de prouver la terminaison par récurrence, et souvent possible de prouver simultanément la correction.

Exercice 2

⇨ Montrer que le programme suivant termine et calcule $n!$, pour tout entier $n \geqslant 0$.

```
1 def factorielle(n):
2     """factorielle(n: int) -> int"""
3     if n == 0:
```

```
        return 1
    else:
        return n * factorielle(n - 1)
```

Fondamentalement, ce qui rend possible la récurrence est la décroissance stricte de l'argument au cours des appels récursifs. Comme $\mathbb{N}$ n'admet pas de suite infinie strictement décroissante, on peut alors conclure que la suite des appels se termine. En pratique, on justifiera cette décroissance et l'on ne rédigera pas la récurrence. C'est l'équivalent d'un variant de boucle dans un programme impératif.

La correction d'un programme récursif est parfois nettement plus simple à prouver que celle d'un programme itératif équivalent. En effet, la correspondance entre le code et les identités mathématiques qui le justifient peut apparaitre beaucoup plus clairement dans le cas récursif.

```
def expo(x, n):
    """expo(x: int, n: int) -> int"""
    if n == 0:
        return 1
    else:
        if n % 2 == 0:
            return expo(x * x, n // 2)
        else:
            return x * expo(x, n - 1)
```

— Si $n > 0$, $0 \leqslant n-1 < n$ et $0 \leqslant n/2 < n$, donc n décroit strictement au cours des appels jusqu'à être nul : la fonction termine.
— Les identités $x^0 = 1$, $x^{2k} = (x^2)^k$ et $x^n = x \cdot x^{n-1}$ si $n \geqslant 1$ sont correctes, donc la fonction aussi.

Il arrive souvent que la suite des arguments ne soit pas strictement décroissante, soit parce que cela n'a pas de sens, soit parce que c'est simplement faux. Dans ce cas, on peut chercher une fonction φ de l'ensemble des arguments dans $\mathbb{N}$ dont les valeurs décroissent strictement au cours des appels successifs. Autrement dit, on cherche une fonction φ à valeurs dans $\mathbb{N}$ telle que, dès que le programme f contient un appel du type $f(x) \to f(y)$, on ait $\varphi(y) < \varphi(x)$.

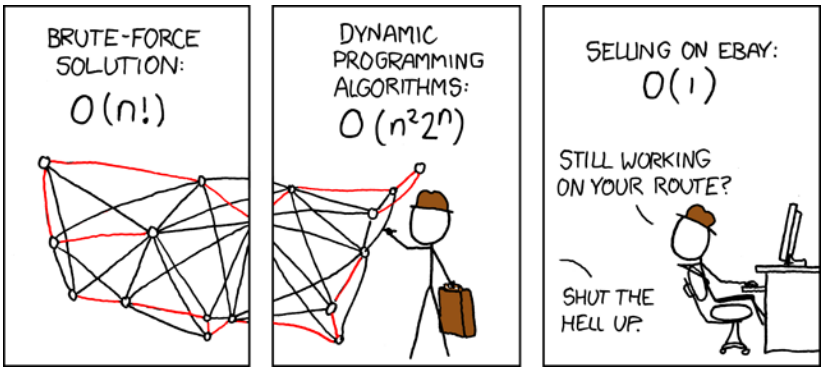
Exercice 3

⇨ Justifier la terminaison de la fonction suivante.

```
def f(n):
    """f(n: int) -> int"""
    if n % 2 != 0 or n == 0:
        return n
    else:
        return f((3 * n) // 2)
```

Chapitre 8

Graphe - Informatique Commune



8.1 Graphe

8.1.1 Graphe non orienté

Définition 8.1.1

On appelle *graphe non orienté* tout couple $G := (S, A)$ où
- S est un ensemble fini non vide dont les éléments sont appelés *noeuds*, ou *sommets*.
- A est un ensemble de paires $\{x, y\}$, où x et y sont deux éléments distincts de S. Ces paires sont appelées *arêtes*, ou *arcs*.

Remarques

⇨ En pratique, l'arête $\{x, y\}$ sera notée $x - y$ ou $y - x$. On utilisera aussi les notations xy ou yx. Intuitivement, une arête $x - y$ permet de passer du sommet x au sommet y et du sommet y au sommet x. Deux sommets reliés par une arête sont dits *adjacents*. On appelle *voisin* d'un sommet x tout sommet adjacent à x.

⇨ Dans notre définition, nous nous sommes interdit les *boucles*, c'est-à-dire les arêtes reliant un sommet à lui-même. Nous n'autorisons pas non plus le fait d'avoir plusieurs arêtes entre deux sommets. Les graphes s'autorisant de telles arêtes sont appelés *multigraphes*.

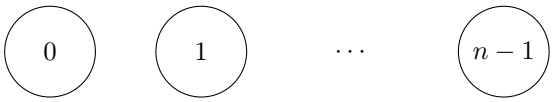
⇨ Dans un graphe non orienté

$$|A| \leqslant \frac{|S|\,(|S|-1)}{2}.$$

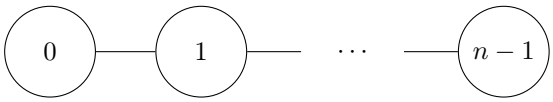
⇨ Lorsqu'on parle d'un graphe à n sommets sans préciser S, on prend $S \coloneqq [\![0, n[\![$.

Exemples

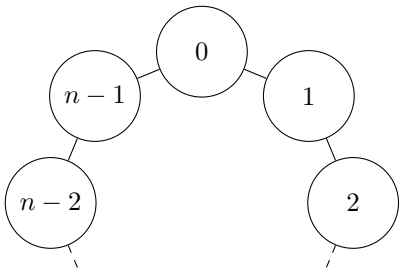
⇨ Le graphe *entièrement déconnecté* possède n sommets et aucune arête.



⇨ Le graphe *chemin* à n sommets $\mathcal{K}_n$ possède une arête entre i et $j \in [\![0, n[\![$ si et seulement si $j = i + 1$.

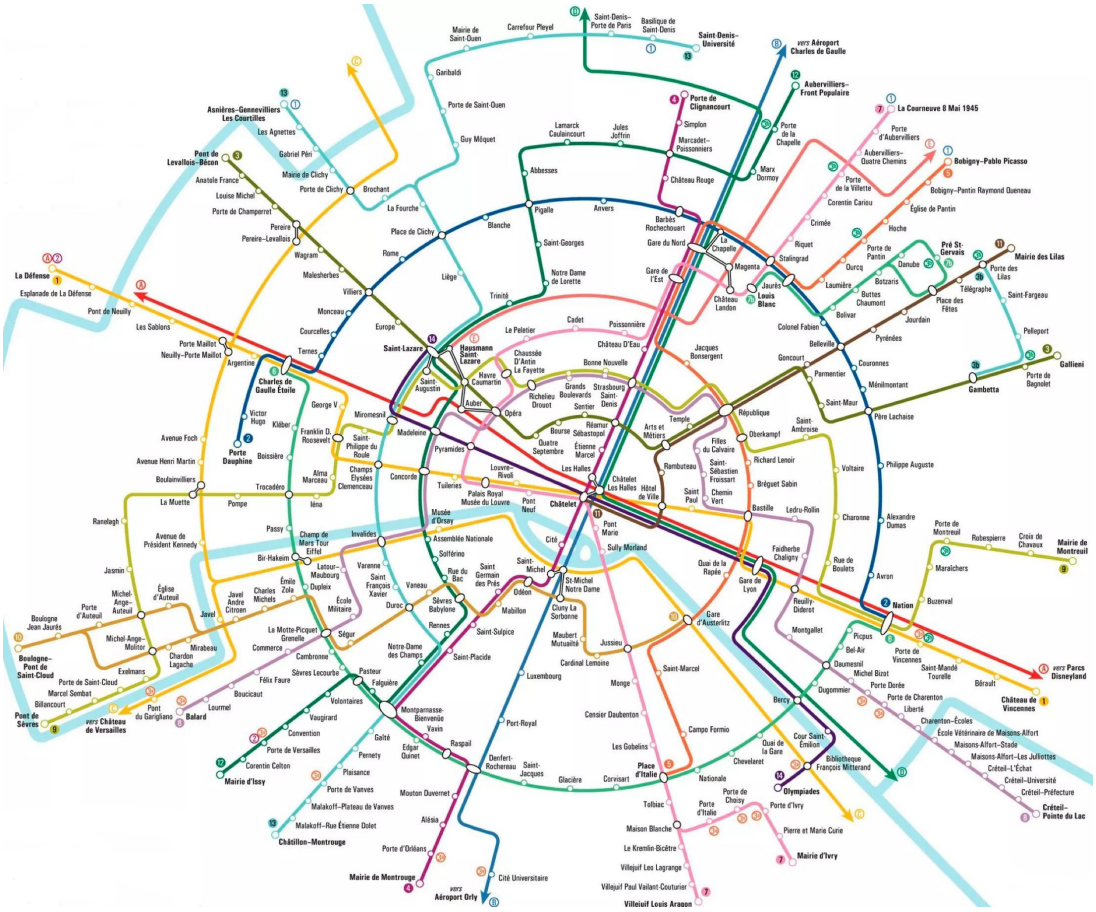


⇨ Le graphe *cycle* à n sommets $\mathcal{C}_n$ (pour $n \geqslant 3$) possède une arête entre i et $j \in [\![0, n[\![$ si et seulement si $j \equiv i+1\ [n]$.



⇨ Le graphe des utilisateurs de Facebook a un sommet pour chaque utilisateur et une arête entre deux sommets lorsque deux utilisateurs sont *amis*. Notons que $|S|$ est de l'ordre de 10^9 et que $|A|$ est de l'ordre de 10^{11}, ce qui pose quelques difficultés algorithmiques.

⇨ Dans le graphe du métro parisien, les sommets représentent les stations de métro et les arêtes, les liaisons entre ces stations.



Définition 8.1.2

On appelle *degré* d'un sommet x le nombre d'arêtes de la forme $x - y$.

Remarque

⇨ Le degré d'un sommet est son nombre de voisins.

Définition 8.1.3

On appelle *chemin* de longueur n toute suite $c \coloneqq z_0, z_1, \ldots, z_n$ de $n+1$ sommets telle que

$$\forall k \in [\![0, n[\![, \quad z_k - z_{k+1} \in A.$$

Les sommets z_0 et z_n sont appelés *extrémités* du chemin et on dit que c *relie* z_0 à z_n.

Remarques

⇨ On peut aussi voir un chemin de longueur n comme une suite de n arêtes consécutives. Un chemin de longueur n est constitué de $n+1$ sommets et de n arêtes.

⇨ On accepte les chemins de longueur nulle, reliant un sommet à lui-même, sans arête.

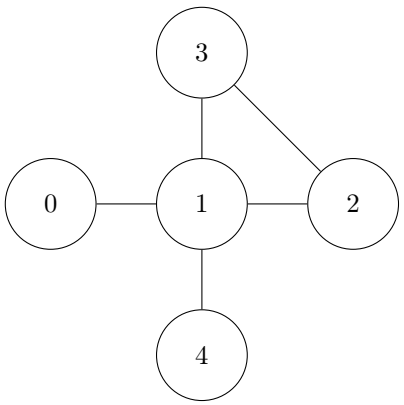
⇨ Dans la suite, on notera $l(c)$ la longueur d'un chemin c.

Définition 8.1.4

Un chemin $z_0, z_1, \ldots, z_n$ est dit
- *élémentaire* lorsqu'il ne passe pas deux fois par le même sommet, c'est-à-dire lorsque les sommets sont deux à deux distincts.
- *simple* s'il ne passe pas deux fois par la même arête, c'est-à-dire lorsque les arêtes $z_k - z_{k+1}$ sont deux à deux distinctes.

Remarque

⇨ Tout chemin élémentaire est simple. Cependant la réciproque est fausse puisque sur le graphe non orienté ci-dessous, le chemin $0 - 1 - 2 - 3 - 1 - 4$ est simple, mais pas élémentaire.



Définition 8.1.5

Un sommet y est dit *accessible* depuis un sommet x lorsqu'il existe au moins un chemin reliant x à y.

Remarque

⇨ Puisqu'on accepte les chemins de longueur nulle, tout sommet est accessible depuis lui-même.

Proposition 8.1.6

Dans un graphe non orienté, la relation d'accessibilité est une relation d'équivalence sur S.

Remarques

⇨ Si y est accessible depuis x, alors x est accessible depuis y. On dit alors que x et y sont *connectés*.

⇨ On appelle *composante connexe* toute classe d'équivalence pour cette relation.

Définition 8.1.7

On dit qu'un graphe non orienté est *connexe* lorsqu'il ne possède qu'une seule composante connexe, c'est-à-dire lorsque tous ses sommets sont connectés.

Définition 8.1.8

Si y est un sommet accessible depuis un sommet x, on appelle distance entre x et y l'entier

$$\mathrm{d}(x,y) \coloneqq \inf\{l(c) : c \text{ est un chemin de } x \text{ à } y\}.$$

Un *chemin de longueur minimale* de x à y est un chemin c de x à y tel que $l(c) = \mathrm{d}(x,y)$.

Remarques

- Lorsque y n'est pas accessible depuis x, la convention est de poser $\mathrm{d}(x,y) \coloneqq +\infty$.
- Conformément à ce qu'on attend d'une distance

$$\begin{aligned} \forall x \in S, &\quad \mathrm{d}(x,x) = 0, \\ \forall x,y \in S, &\quad \mathrm{d}(y,x) = \mathrm{d}(x,y), \\ \forall x,y,z \in S, &\quad \mathrm{d}(x,z) \leqslant \mathrm{d}(x,y) + \mathrm{d}(y,z). \end{aligned}$$

Définition 8.1.9

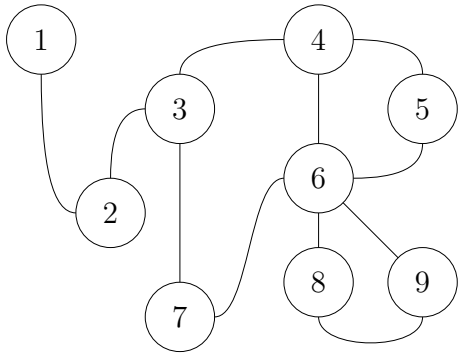
On appelle *cycle* tout chemin simple de longueur non nulle dont les deux extrémités sont identiques.

Remarques

- Dans la définition, il est nécessaire de se limiter aux chemins simples, sinon on pourrait construire des « cycles » dans les graphes en parcourant une même arête dans un sens puis dans l'autre.
- Dans un graphe non orienté, la longueur d'un cycle est supérieure ou égale à 3.
- Un cycle sera dit *élémentaire* lorsque la seule répétition de sommets est celle de ses extrémités. Un cycle est élémentaire si et seulement si il ne contient pas d'autre cycle.
- On dit qu'un graphe est *acyclique* lorsqu'il ne possède pas de cycle.

Exemple

- Dans le graphe suivant, le chemin $4-5-6-9-8-6-4$ est un cycle. Ce graphe possède 4 cycles élémentaires.

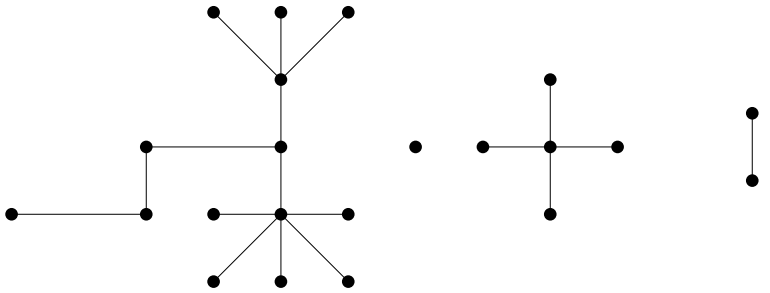


Définition 8.1.10

On appelle *arbre* tout graphe connexe acyclique.

Remarques

⇨ Les graphes suivants sont des arbres.



⇨ Les arbres que nous avons manipulés dans les chapitres précédents sont ce qu'on appelle des *arbres enracinés*. Ce sont des arbres pour lesquels on a choisi un sommet appelé *racine*. La distance d'un sommet à la racine est appelée *profondeur*. Ces arbres sont conventionnellement dessinés de façon à ce que les sommets de même profondeur soient à même hauteur.

Exercice 1

⇨ En choisissant successivement trois racines pour l'arbre de gauche ci-dessus, dessiner l'arbre enraciné ainsi obtenu de manière conventionnelle.

8.1.2 Graphe orienté

Définition 8.1.11

On appelle *graphe orienté* tout couple $G \coloneqq (S, A)$ où
- S est un ensemble fini non vide dont les éléments sont appelés *sommets*.
- A est un ensemble de couples (x, y), où x et y sont deux éléments distincts de S. Ces paires sont appelées *arcs*.

Remarques

⇨ En pratique, l'arc (x, y) sera noté $x \to y$ ou xy. Intuitivement, un arc $x \to y$ permet de passer du sommet x au sommet y mais pas du sommet y au sommet x. S'il y a un arc $x \to y$, on dit que x est un *prédécesseur* de y et que y est un *successeur* de x.

⇨ Dans un graphe orienté

$$|A| \leqslant |S|\,(|S| - 1)\,.$$

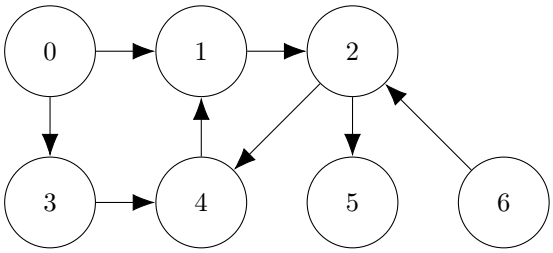
⇨ En pratique, on confondra souvent un graphe non orienté $G \coloneqq (S, A)$ avec son graphe orienté associé G_{o}. Ce dernier possède les mêmes sommets que G. De plus, $x \to y$ est un arc de G_{o} si et seulement si $x - y \in A$. En particulier, dans G_{o}, dès que $x \to y$ est un arc, $y \to x$ en est un autre.

⇨ Les arbres enracinés s'orientent naturellement depuis leur racine : lorsqu'on les dessine de manière conventionnelle, on les oriente du haut vers le bas.

⇨ À un graphe orienté $G \coloneqq (S, A)$, on associe le graphe non orienté G_{no} obtenu en « oubliant » l'orientation des arcs.

Exemples

⇨ Voici un exemple de graphe orienté à 7 sommets. C'est sur cet exemple que nous détaillerons l'exécution de nos algorithmes dans la seconde partie de ce chapitre.



⇨ Le graphe du web possède un sommet pour chaque page web et un arc de x vers y lorsque la page x contient un lien vers la page y. C'est ce graphe que les moteurs de recherche parcourent pour construire leur index. La taille du graphe du web est inconnue mais Google indexe plus de 50 milliards de pages.

⇨ Le graphe des utilisateurs d'Instagram a un sommet pour chaque utilisateur et un arc de x vers y lorsque x est un *follower* de y. Contrairement au graphe des utilisateurs de Facebook qui est non orienté, celui d'Instagram l'est.

Définition 8.1.12

Dans un graphe orienté, on appelle
- *degré entrant* d'un sommet x, le nombre d'arcs de la forme $y \to x$.
- *degré sortant* d'un sommet x, le nombre d'arcs de la forme $x \to y$.
- *degré total* d'un sommet, la somme de son degré entrant et de son degré sortant.

Remarque

⇨ Le degré entrant d'un sommet est son nombre de prédécesseurs et le degré sortant, son nombre de successeurs.

Définition 8.1.13

On appelle chemin de longueur n toute suite $c := z_0, z_1, \dots, z_n$ de $n+1$ sommets telle que

$$\forall k \in [\![0, n[\![\,, \quad z_k \to z_{k+1} \in A.$$

Les sommets z_0 et z_n sont appelés *extrémités* du chemin et on dit que c *relie* z_0 à z_n.

Remarques

⇨ Comme dans les graphes non orientés, on définit la notion de chemin *élémentaire* et de chemin *simple*. Les chemins élémentaires sont simples, mais la réciproque est fausse.

⇨ La notion d'*accessibilité* se définit aussi de la même manière. Cependant, dans un graphe orienté, ce n'est plus une relation d'équivalence sur S. Les notions de connexité et de composante connexe n'ont plus de sens pour ces graphes.

⇨ La notion de distance entre un sommet x et un sommet y est toujours définie. L'inégalité triangulaire reste vraie, mais la distance n'est plus symétrique.

⇨ La notion de cycle se définit toujours de la même manière dans un graphe orienté. Cependant, contrairement à ce qui se passe dans le cas des graphes non orientés, il existe des cycles de longueur 2.

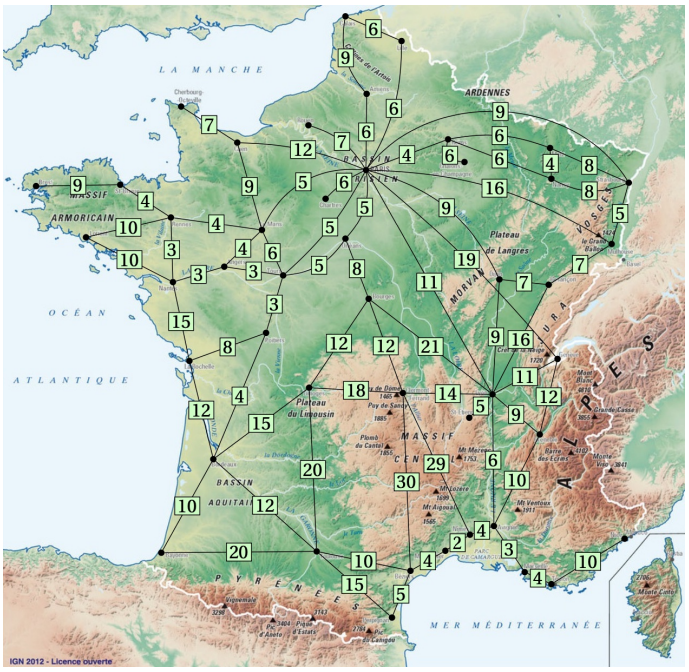
8.1.3 Graphe pondéré

Définition 8.1.14

On appelle *graphe pondéré* la donnée d'un graphe $G := (S, A)$ et d'une application $\rho : A \to \mathbb{R}_+$ appelée *poids*.

Exemple

⇨ Voici le graphe non orienté des connexions ferroviaires françaises, le poids représentant les temps de trajet en dizaines de minutes.



Remarques

- Un graphe pondéré peut être orienté ou non orienté.
- Il est possible de considérer des graphes pondérés avec des fonctions de poids prenant des valeurs négatives. Mais dans ce cours, puisque c'est une condition pour pouvoir appliquer l'algorithme de Dijkstra, nous nous limiterons à des fonctions de poids positives.
- On appelle *poids du chemin* $c \coloneqq z_0, z_1, \ldots, z_n$ le réel positif

$$\rho(c) \coloneqq \sum_{k=0}^{n-1} \rho(z_k z_{k+1}).$$

- On appelle *poids d'un graphe* la somme des poids de ses arêtes.

Définition 8.1.15

Si y est un sommet accessible depuis un sommet x, on définit

$$\delta(x, y) \coloneqq \inf \{\rho(c) : c \text{ est un chemin de } x \text{ à } y\}.$$

Un *chemin de poids minimal* de x à y est un chemin c de x à y tel que $\rho(c) = \delta(x, y)$.

Remarque

- Dans le cas où les poids sont des distances, par exemple si G est le graphe d'un réseau routier, on pourra parler de distance et de plus court chemin. Il ne faut cependant pas confondre le réel $\delta(x, y)$ avec l'entier $\mathrm{d}(x, y)$ représentant le nombre minimal d'arcs entre x et y.

8.1.4 Représentation d'un graphe

Définition 8.1.16

Soit $G \coloneqq (S, A)$ un graphe où $S = [\![0, n[\![$. On appelle matrice d'adjacence la matrice $M \in \mathcal{M}_n(\mathbb{Z})$ définie par

$$\forall i, j \in [\![0, n[\![, \quad m_{i,j} \coloneqq \begin{cases} 1 & \text{si } j \text{ est un successeur de } i, \\ 0 & \text{sinon.} \end{cases}$$

Remarques

- Un graphe est « non orienté » si et seulement si sa matrice d'adjacence est symétrique.
- Puisqu'on interdit les boucles, une matrice d'adjacence n'a que des 0 sur la diagonale.
- On peut représenter un graphe pondéré par une matrice d'adjacence $M \in \mathcal{M}_n(\mathbb{R})$. On prend pour coefficient $m_{i,j}$ la valeur `None` lorsqu'il n'existe pas d'arc de i à j, et le poids $\rho_{i,j}$ de l'arc allant de i à j lorsqu'un tel arc existe.

Définition 8.1.17

Soit $G \coloneqq (S, A)$ un graphe où $S = [\![0, n[\![$. On appelle liste d'adjacence le tableau g de longueur n tel que pour tout $i \in [\![0, n[\![$, g_i est la liste des successeurs $j \in [\![0, n[\![$ de i.

Remarque

- On peut représenter un graphe pondéré par une liste d'adjacence dans laquelle, pour tout sommet i, g_i est la liste des couples $(\rho_{i,j}, j)$ où j est un successeur de i.
- Dans la suite de ce cours, nous utiliserons des listes d'adjacence pour stocker les graphes pondérés.

8.2 Algorithmes sur les graphes

8.2.1 Parcours générique d'un graphe

Supposons que vous êtes enfermé dans un labyrinthe de salles, connectées entre elles par des portes. Nous représentons ce labyrinthe par un graphe dont les sommets sont les salles et les arêtes sont les portes reliant ces salles entre elles. Si l'on souhaite sortir de ce labyrinthe, un réflexe naturel est d'emprunter au hasard les portes que l'on croise. Cependant, cette stratégie possède deux défauts importants : elle ne nous dit pas ce qu'on doit faire lorsqu'on tombe

dans un cul-de-sac et elle ne nous empêche pas de tourner en rond.

Pour résoudre ces deux problèmes, la solution la plus simple est de marquer les salles. Au cours de notre exploration, nous choisirons donc de les placer successivement dans 3 états différents :
— *Inconnu* : C'est l'état dans lequel est une salle qui n'a pas encore été découverte.
— *Découvert* : C'est l'état dans lequel on place une salle lorsqu'on l'a aperçue par une porte.
— *Visité* : C'est l'état dans lequel est une salle dans laquelle nous sommes déjà entrés.
Pour ne pas tourner en rond, il suffit de ne pas entrer dans une salle qui a déjà été visitée. Pour ces salles, on peut choisir de marquer leur sol d'une croix blanche. Et pour savoir que faire lorsqu'on est dans un cul-de-sac, il suffit de garder une trace des salles que l'on a découvertes, mais qui n'ont pas encore été visitées. Pour cela, on conserve avec nous un sac contenant une marque pour chacune d'elles.

Revenons au vocabulaire des graphes. À l'aide de ces deux outils, notre sac ainsi que le marquage des sommets, nous sommes armés pour parcourir l'ensemble des sommets accessibles depuis notre sommet de départ. Ce sommet est appelé *source*. Pour cela, il nous suffit de suivre l'algorithme suivant, que nous appelons « *parcours générique* ».

Algorithme Parcours générique

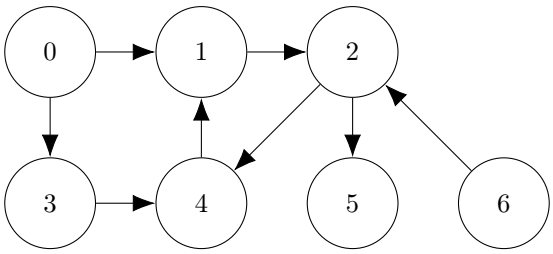
```
mettre le sommet source dans le sac
tant que le sac n'est pas vide faire
    prendre un sommet x dans le sac
    si x n'a pas été visité alors
        marquer le sommet x comme visité
        pour chaque arc xy faire
            mettre le sommet y dans le sac
```

La seule propriété dont notre sac a besoin est qu'on puisse y mettre des sommets pour les extraire plus tard. L'ordre dans lesquels ces sommets sont extraits n'a pas d'importance pour le moment.

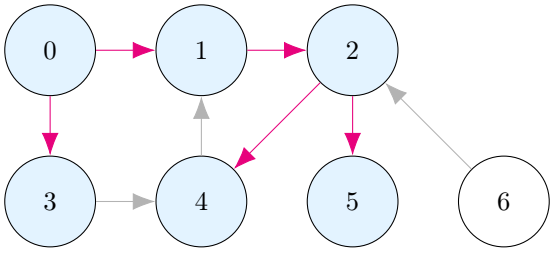
Proposition 8.2.1

L'algorithme de parcours générique visite tous les sommets accessibles depuis la source et uniquement ceux-là.

Supposons que l'on garde en mémoire le sommet depuis lequel on visite chaque sommet, en ne mettant pas le sommet y dans notre sac, mais plutôt le couple (x, y). Un parcours générique mettra ainsi en valeur un arbre, enraciné en la source, couvrant l'ensemble des sommets accessibles. Par exemple, en considérant le graphe ci-dessous,



si les couples que l'on sort du sac sont successivement $(\varnothing, 0)$, $(0, 1)$, $(1, 2)$, $(0, 3)$, $(2, 4)$ et $(2, 5)$, on obtient l'arbre enraciné suivant :



Par la suite, nous utiliserons principalement nos sacs pour y placer des sommets. Cependant, lors de certains raisonnements, il sera parfois utile de faire comme si on y avait placé un couple.

La structure de données que nous allons utiliser pour implémenter notre sac va déterminer l'ordre dans lequel les sommets en sont extraits.

— *Pile* : Si nous utilisons une pile, pour laquelle c'est le dernier sommet qui a été placé dans le sac qui en est extrait, nous obtiendrons ce qu'on appelle un *parcours en profondeur*. Bien que tous les parcours nous permettent d'obtenir l'ensemble des sommets accessibles depuis un sommet, la simplicité de la structure de pile fait que c'est souvent ce parcours que nous utiliserons pour cela. Nous verrons aussi que le parcours en profondeur nous permet de détecter les cycles dans un graphe.
— *File* : Si nous utilisons une file, pour laquelle c'est le premier sommet qui a été placé dans le sac qui en est extrait, nous obtiendrons ce qu'on appelle un *parcours en largeur*. Ce parcours nous sera utile pour trouver le chemin de longueur minimale entre la source et les sommets accessibles depuis cette dernière.
— *File de priorité* : L'utilisation d'une file de priorité nous permettra de découvrir une famille d'algorithmes fonctionnant avec les graphes pondérés. Ils se distinguent par les différentes priorités qu'ils utilisent.
  — *Dijkstra* : L'algorithme de Dijkstra nous permet de trouver le chemin de poids minimal entre la source et les sommets accessibles depuis cette dernière. Pour cela, la priorité utilisée est le poids du chemin qui nous a permis de découvrir le sommet.
  — *Prim* : L'algorithme de Prim nous permet de trouver un arbre de poids minimal couvrant l'ensemble des sommets accessibles. Pour cela, la priorité que nous utiliserons est le poids de l'arc qui nous a permis de découvrir le sommet.

Commençons par étudier le plus simple de ces parcours : le parcours en profondeur.

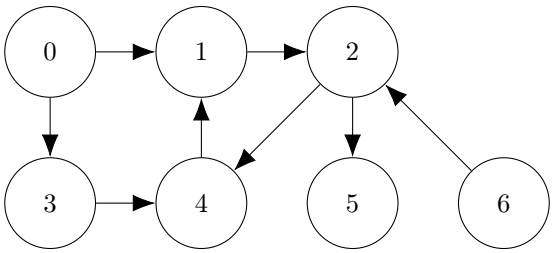
8.2.2 Parcours en profondeur

Version itérative

L'implémentation Python du parcours en profondeur se fait naturellement. On suppose ici que $\mathcal{S} \coloneqq [\![0, n[\![$ et que le graphe est représenté par sa liste d'adjacence. Pour marquer les sommets, nous utilisons le tableau *visité*, de longueur n, dont tous les éléments sont initialisés à `False`. Pour le sac, nous utilisons la liste *pile* que nous utilisons, comme son nom l'indique, comme une pile.

```
1  def profondeur(g, s):
2      """profondeur(g: list[list[int]], s: int) -> NoneType"""
3      n = len(g)
4      visite = [False for _ in range(n)]
5      pile = [s]
6      while len(pile) != 0:
7          x = pile.pop()
8          if not visite[x]:
9              visite[x] = True
10             for y in g[x]:
11                 pile.append(y)
```

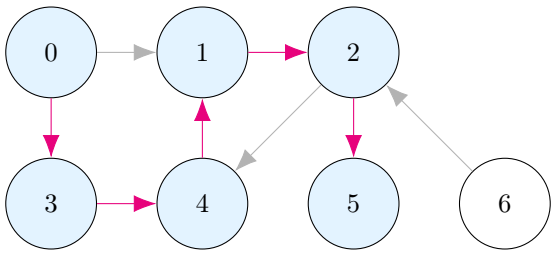
Si l'on souhaite effectuer une action pour chaque sommet, il suffit de définir une fonction `f(x: int) -> NoneType` que l'on appelle juste avant l'instruction `visite[x] = True`. Par exemple, si l'on souhaite afficher à l'écran les sommets visités dans l'ordre de notre parcours, il suffit d'insérer `print(x)` ligne 9.



Illustrons son fonctionnement en détail sur un exemple. Le tableau ci-dessous détaille les différentes étapes du parcours en profondeur du présent graphe à partir du sommet 0. Le contenu de la *pile* est détaillé lors du passage ligne 6, tout comme l'ensemble des sommets marqués dans *visité*.

pile	*visité*	action
`[0]`	$\{\}$	Dépiler 0, le marquer et empiler ses successeurs 1 et 3.
`[1, 3]`	$\{0\}$	Dépiler 3, le marquer et empiler son successeur 4.
`[1, 4]`	$\{0, 3\}$	Dépiler 4, le marquer et empiler son successeur 1.
`[1, 1]`	$\{0, 3, 4\}$	Dépiler 1, le marquer et empiler son successeur 2.
`[1, 2]`	$\{0, 1, 3, 4\}$	Dépiler 2, le marquer et empiler ses successeurs 4 et 5.
`[1, 4, 5]`	$\{0, 1, 2, 3, 4\}$	Dépiler 5 et le marquer. Il n'a pas de successeur.
`[1, 4]`	$\{0, 1, 2, 3, 4, 5\}$	Dépiler 4. Il a déjà été marqué.
`[1]`	$\{0, 1, 2, 3, 4, 5\}$	Dépiler 1. Il a déjà été marqué.
`[]`	$\{0, 1, 2, 3, 4, 5\}$	*pile* est vide donc l'algorithme se termine.

Une fois le parcours terminé, tous les sommets atteignables à partir du sommet 0 ont été marqués, à savoir $0, 1, 2, 3, 4$ et 5. Inversement, le sommet 6, qui n'est pas atteignable à partir de 0 n'a pas été marqué. C'est là une propriété fondamentale du parcours en profondeur. Le graphe ci-dessous met en valeur les sommets visités ainsi que les arcs empruntés lors de ce parcours.



Version récursive

Le parcours en profondeur est un algorithme fondamentalement récursif dont voici une implémentation Python :

```
def profondeur_rec(g, visite, x):
    """profondeur_rec(g: list[list[int]], visite: list[bool],
                      x: int) -> NoneType"""
    if not visite[x]:
        visite[x] = True
        for y in g[x]:
            profondeur_rec(g, visite, y)
```

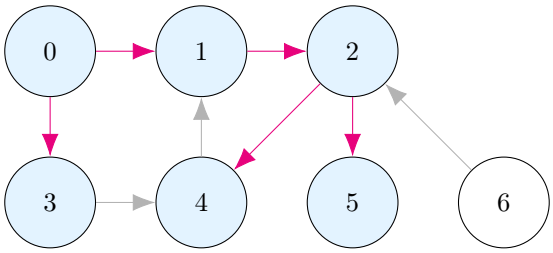
Pour lancer un parcours en profondeur depuis un sommet s, on utilisera la fonction suivante :

```
def profondeur(g, s):
    """profondeur(g: list[list[int]], s: int) -> NoneType"""
    n = len(g)
    visite = [False for _ in range(n)]
    profondeur_rec(g, visite, s)
```

Dans cette version, une pile est toujours présente par l'intermédiaire de la pile d'appels. Contrairement à ce qui se passe dans la version itérative, les sommets sont visités dès qu'ils sont découverts ; en récursif, ces deux états sont donc confondus. Le tableau ci-dessous détaille l'ensemble des sommets marqués dans *visité* au moment de l'appel de `profondeur_rec`. La pile d'appel est aussi représentée dans la colonne « chemin emprunté ».

visité	chemin emprunté	action
$\{\}$	0	Marquer 0, emprunter l'arc $0 \to 1$.
$\{0\}$	$0 \to 1$	Marquer 1, emprunter l'arc $1 \to 2$.
$\{0,1\}$	$0 \to 1 \to 2$	Marquer 2, emprunter l'arc $2 \to 4$.
$\{0,1,2\}$	$0 \to 1 \to 2 \to 4$	Marquer 4, emprunter l'arc $4 \to 1$.
$\{0,1,2,4\}$	$0 \to 1 \to 2 \to 4 \to 1$	Déjà découvert.
$\{0,1,2,4\}$	$0 \to 1 \to 2 \to 4$	Pas d'autre arc, terminé.
$\{0,1,2,4\}$	$0 \to 1 \to 2$	Emprunter l'arc $2 \to 5$.
$\{0,1,2,4,5\}$	$0 \to 1 \to 2 \to 5$	Marquer 5, aucun arc, terminé.
$\{0,1,2,4,5\}$	$0 \to 1 \to 2$	Pas d'autre arc, terminé.
$\{0,1,2,4,5\}$	$0 \to 1$	Pas d'autre arc, terminé.
$\{0,1,2,4,5\}$	0	Emprunter l'arc $0 \to 3$.
$\{0,1,2,4,5\}$	$0 \to 3$	Marquer 3, emprunter l'arc $3 \to 4$.
$\{0,1,2,3,4,5\}$	$0 \to 3 \to 4$	Déjà découvert.
$\{0,1,2,3,4,5\}$	$0 \to 3$	Pas d'autre arc, terminé.
$\{0,1,2,3,4,5\}$	0	Pas d'autre arc, terminé.

On remarque que même si les sommets visités sont les mêmes que dans la version itérative, les arcs empruntés diffèrent : le parcours que l'on vient d'effectuer est un autre parcours en profondeur.



Accessibilité, connexité

Une application immédiate du parcours en profondeur consiste à déterminer s'il existe un chemin entre deux sommets x et y. Pour cela, il suffit de lancer un parcours en profondeur à partir du sommet x puis, une fois qu'il est terminé, de regarder si le sommet y fait partie des sommets visités. Le programme suivant réalise cet algorithme :

```
def existe_chemin(g, x, y):
    """existe_chemin(g: list[list[int]], x: int, y: int) -> bool"""
    n = len(g)
    visite = [False for _ in range(n)]
    profondeur_rec(g, visite, x)
    return visite[y]
```

Le parcours en profondeur est un algorithme très efficace, dont la complexité temporelle est de l'ordre du nombre de sommets du graphe (pour l'initialisation de *visité*) auquel on ajoute le nombre d'arcs qui sont examinés pendant ce parcours. On a donc une complexité temporelle en $O(|S|+|A|)$. En effet, chaque arc $x \to y$ est examiné au plus une fois, à savoir la première fois que la fonction `profondeur_rec` est appelée sur le sommet x : si la fonction `profondeur_rec` est rappelée plus tard sur ce même sommet x, alors il sera trouvé dans *visité* et la fonction se terminera immédiatement. Dans le pire des cas, tous les sommets sont atteignables et le graphe est entièrement parcouru. Le cout est moindre lorsque certains sommets ne sont pas atteignables depuis le sommet de départ.

La complexité spatiale de la version récursive est en $\Theta(|S|)$: cette complexité provient du tableau *visité* dont la taille est $|S|$, auquel on ajoute la taille de la pile d'appels qui reste toujours inférieure au nombre de sommets puisque la succession de sommets passés en argument des appels actifs forme à chaque instant un chemin élémentaire.

On peut aussi tester la connexité d'un graphe non orienté de la même manière. On utilise pour cela la fonction `est_connexe` :

```
def tous_vrai(t):
    """tous_vrai(t: list[bool]) -> bool"""
    for v in t:
        if not v:
            return False
    return True
```

```
7
8  def est_connexe(g):
9      """est_connexe(g: list[list[int]]) -> bool"""
10     n = len(g)
11     visite = [False for _ in range(n)]
12     profondeur_rec(g, visite, 0)
13     return tous_vrai(visite)
```

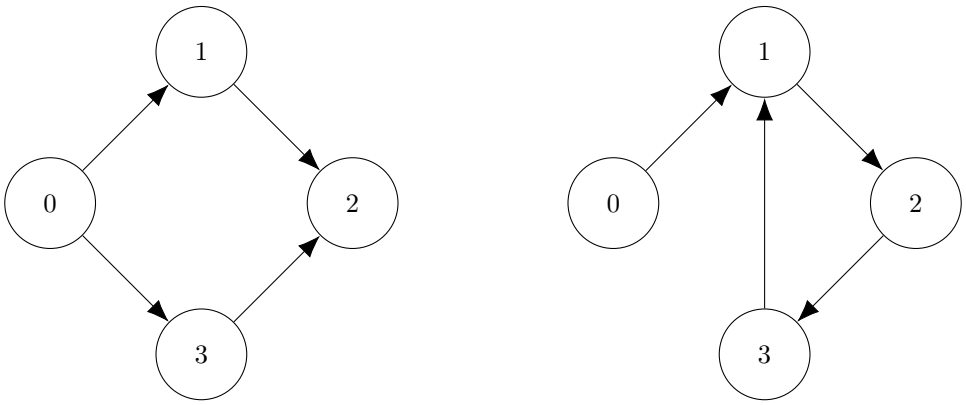
Cet algorithme a de nouveau une complexité temporelle en $O(|S| + |A|)$ et une complexité spatiale en $\Theta(|S|)$.

Exercice 2

⇨ Écrire une fonction qui compte le nombre de composantes connexes d'un graphe non orienté.

Détection de cycle

Le parcours en profondeur permet également de détecter la présence d'un cycle dans un graphe orienté. En effet, puisque l'on marque les sommets avant de considérer leurs voisins, pour justement éviter de tourner en rond dans un cycle, alors on doit pouvoir être à même de détecter leur présence. Il y a cependant une subtilité, car lorsqu'on retombe sur un sommet déjà marqué, on ne sait pas pour autant si l'on vient de découvrir un cycle. Considérons par exemple le parcours en profondeur des deux graphes suivants, à partir du sommet 0 à chaque fois.



Dans le graphe de gauche, on retombe sur le sommet 2. Il n'y a pas de cycle pour autant, mais seulement un chemin parallèle. Dans le graphe de droite, on retombe sur le sommet 1, cette fois à cause d'un cycle. Tel qu'il est écrit, notre parcours en profondeur ne nous permet de pas distinguer ces deux situations. Dans les deux cas, on constate que le sommet est déjà visité sans pouvoir en tirer de conclusion quant à l'existence d'un cycle.

Pour y remédier, on va distinguer dans notre marquage trois sortes de sommets : ceux que l'on n'a pas encore découverts qui seront marqués comme *inconnu*, ceux que l'on a *découvert* mais qui sont toujours présents dans le chemin déterminé par la pile d'appels (ces sommets sont donc *visités* puisque nous utilisons une implémentation récursive dans laquelle ces deux états sont confondus), et ceux qui ne sont plus présents dans la pile d'appels, qu'on marquera comme *fermé*. Le parcours en profondeur est modifié de la manière suivante : lorsqu'on visite un sommet x

- S'il est marqué comme « découvert », c'est qu'on vient de découvrir un cycle.
- S'il est marqué comme « fermé », on ne fait rien.
- S'il est marqué comme « inconnu », on procède ainsi :
  - On marque le sommet x comme « découvert ».
  - On visite tous ses successeurs, récursivement.
  - Enfin, on le marque comme « fermé ».

Comme on le voit, les successeurs du sommet x sont examinés après le moment où x est marqué comme « découvert » et avant le moment où il est marqué comme « fermé ». Ainsi, s'il existe un cycle nous ramenant sur x, on le trouvera comme étant « découvert » et le cycle sera signalé.

Le programme suivant réalise cette détection de cycle. La fonction `possede_cycle_rec(g, x, etat)` est toujours une fonction récursive, mais elle renvoie désormais un résultat, à savoir un booléen indiquant la présence d'un cycle. Enfin, la fonction `possede_cycle` marque tous les sommets comme « inconnu » puis lance un parcours en profondeur à partir de tous les sommets du graphe. Si l'un de ces parcours renvoie `True`, on transmet ce résultat. Sinon, on renvoie `False`.

```
1 INCONNU = 0
2 DECOUVERT = 1
```

```
3  FERME = 2
4
5  def possede_cycle_rec(g, etat, x):
6      """possede_cycle_rec(g: list[list[int]], etat: list[int], x: int) -> bool"""
7      if etat[x] == INCONNU:
8          etat[x] = DECOUVERT
9          for y in g[x]:
10             cycle = possede_cycle_rec(g, etat, y)
11             if cycle:
12                 return True
13         etat[x] = FERME
14         return False
15     else:
16         return etat[x] == DECOUVERT
```

Enfin, dans la version suivante, on cherche à détecter la présence d'un cycle n'importe où dans le graphe. C'est pourquoi on lance un parcours en profondeur à partir de tous les sommets du graphe.

```
1  def possede_cycle(g):
2      """possede_cycle(g: list[list[int]]) -> bool"""
3      n = len(g)
4      etat = [INCONNU for _ in range(n)]
5      for x in range(n):
6          cycle = possede_cycle_rec(g, etat, x)
7          if cycle:
8              return True
9      return False
```

Pour beaucoup de ces sommets, le parcours est déjà passé par là, car ils étaient accessibles depuis des sommets déjà parcourus ; la fonction `parcours_cycle_rec` se termine alors immédiatement sans rien faire. À nouveau, la complexité temporelle de cet algorithme est en $O(|S| + |A|)$, et sa complexité spatiale en $\Theta(|S|)$.

Attention à ne pas oublier que cet algorithme ne fonctionne que pour les graphes orientés. Il nécessite quelques aménagements pour fonctionner avec les graphes non orientés.

8.2.3 Parcours en largeur

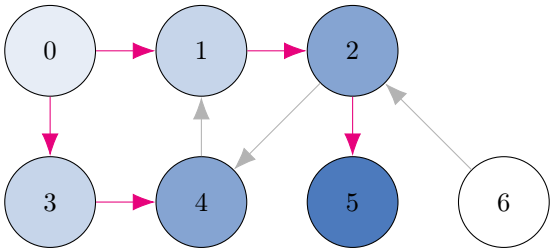
Le parcours en largeur se fait simplement en utilisant une liste pour implémenter le sac de notre parcours générique :

```
1  import collections
2
3  def largeur(g, s):
4      """largeur(g: list[list[int]], s: int) -> NoneType"""
5      n = len(g)
6      visite = [False for _ in range(n)]
7      file = collections.deque()
8      file.append(s)
9      while len(file) != 0:
10         x = file.popleft()
11         if not visite[x]:
12             visite[x] = True
13             for y in g[x]:
14                 file.append(y)
```

Illustrons son fonctionnement sur un notre exemple. Le tableau ci-dessous détaille les différentes étapes du parcours en largeur du graphe précédent à partir du sommet 0. Le contenu de la *file* est détaillé lors du passage ligne 9, tout comme l'ensemble des sommets marqués dans *visité*.

file	*visité*	action
`[0]`	$\{\}$	Défiler 0, le marquer et enfiler ses successeurs 1 et 3.
`[1, 3]`	$\{0\}$	Défiler 1, le marquer et enfiler son successeur 2.
`[3, 2]`	$\{0,1\}$	Défiler 3, le marquer et enfiler son successeur 4.
`[2, 4]`	$\{0,1,3\}$	Défiler 2, le marquer et enfiler ses successeurs 4 et 5.
`[4, 4, 5]`	$\{0,1,2,3\}$	Défiler 4, le marquer et empiler son successeurs 1.
`[4, 5, 1]`	$\{0,1,2,3,4\}$	Défiler 4. Il a déjà été marqué.
`[5, 1]`	$\{0,1,2,3,4\}$	Défiler 5, le marquer. Il n'a pas de successeur.
`[1]`	$\{0,1,2,3,4,5\}$	Défiler 1. Il a déjà été marqué donc on ne fait rien.
`[]`	$\{0,1,2,3,4,5\}$	*file* est vide donc l'algorithme se termine.

Tout comme le parcours en profondeur, le parcours en largeur a visité exactement les sommets atteignables à partir du sommet 0. Voici les sommets visités ainsi que les arcs empruntés lors de ce parcours.



On observe que les arcs ainsi mis en valeur soulignent les chemins de longueur minimale entre la source 0 et les sommets atteignables depuis cette source. On voit que les sommets 1 et 3 sont à une distance de 1 de la source, les sommets 2 et 4 sont à une distance de 2 et le sommet 5 est à une distance de 3.

Comme pour le parcours en profondeur, un même sommet peut apparaitre plusieurs fois dans notre sac, mais le fait qu'on travaille ici avec une *file* fait que les sommets sortent de la file dans le même ordre que celui dans lequel ils y sont entrés. Une fois qu'un sommet y est entré, il est donc inutile de l'enfiler de nouveau. Cette remarque nous permet l'optimisation suivante : au lieu de marquer les sommets lorsqu'ils sont *visités*, c'est-à-dire lorsqu'ils sortent de la file, nous allons les marquer lorsqu'ils sont *découverts*, c'est-à-dire lorsqu'ils y entrent.

```
1  def largeur(g, s):
2      """largeur(g: list[list[int]], s: int) -> NoneType"""
3      n = len(g)
4      decouvert = [False for _ in range(n)]
5      file = collections.deque()
6      decouvert[s] = True
7      file.append(s)
8      while len(file) != 0:
9          x = file.popleft()
10         for y in g[x]:
11             if not decouvert[y]:
12                 decouvert[y] = True
13                 file.append(y)
```

Contrairement à l'algorithme initial qu'on appelle parfois parcours en largeur à marquage *tardif*, cette nouvelle version est appelée parcours en largeur à marquage *précoce*. Avec cette optimisation, les valeurs successives de *file* sont : `[0]`, `[1, 3]`, `[3,2]`, `[2, 4]`, `[4, 5]`, `[5]` et enfin `[]`. On remarque qu'à chaque étape, la file est composée d'une succession de sommets dont la distance à la source est d suivie d'une succession (éventuellement vide) de sommets dont la distance à la source est $d+1$. On observe donc que le parcours en largeur explore le graphe en « cercles concentriques » à partir de la source.

Cette idée de cercles concentriques nous permet une dernière transformation de notre programme dans lequel on va travailler non pas avec un sac fonctionnant comme une file, mais avec deux sacs. À chaque instant, un des sacs, qu'on appelle *courant*, contient des sommets situés à une distance d de la source, tandis que l'autre sac, qu'on appelle *suivant*, contient des sommets à une distance $d+1$ de la source. On examinera ces derniers une fois que le sac *courant* est vide. Par soucis de simplicité, nous utiliserons une pile pour implémenter ces deux sacs, mais nous aurions pu utiliser n'importe quelle structure de donnée séquentielle. À côté de ces deux piles, on utilise un tableau *découvert* qui marque les sommets déjà découverts. Le parcours en largeur procède ainsi :

— Initialement, la source est empilée dans *courant* et on la marque comme *découverte*.
— Tant que la pile *courant* n'est pas vide

— On dépile le sommet x de *courant*.
— Pour chaque successeur y de x, s'il n'a pas été marqué comme *découvert*, on le marque et on l'empile dans *suivant*.
— Si la pile *courant* est vide, on l'échange avec la pile *suivant*.

Si l'on reprend l'exemple de notre graphe exemple et qu'on effectue un parcours en largeur à partir du sommet 0, on obtient le parcours résumé dans le tableau suivant.

vu	*courant*	*suivant*	action
$\varnothing$	`[]`	`[]`	Le sommet 0 est marqué puis empilé dans *courant*.
$\{0\}$	`[0]`	`[]`	On dépile le sommet 0 ; 1 et 3 sont marqués puis empilés dans *suivant*.
$\{0,1,3\}$	`[]`	`[1, 3]`	La pile *courant* est vide ; on échange.
$\{0,1,3\}$	`[1, 3]`	`[]`	On dépile le sommet 3 ; 4 est marqué puis empilé dans *suivant*.
$\{0,1,3,4\}$	`[1]`	`[4]`	On dépile le sommet 1 ; 2 est marqué puis empilé dans *suivant*.
$\{0,1,2,3,4\}$	`[]`	`[4, 2]`	La pile *courant* est vide ; on échange.
$\{0,1,2,3,4\}$	`[4, 2]`	`[]`	On dépile le sommet 2 ; 5 est marqué puis empilé dans *suivant*.
$\{0,1,2,3,4,5\}$	`[4]`	`[5]`	On dépile le sommet 4.
$\{0,1,2,3,4,5\}$	`[]`	`[5]`	La pile *courant* est vide ; on échange.
$\{0,1,2,3,4,5\}$	`[5]`	`[]`	On dépile le sommet 5.
$\{0,1,2,3,4,5\}$	`[]`	`[]`	*courant* est vide donc l'algorithme se termine.

L'implémentation Python de cet algorithme se fait alors naturellement.

```
1  def largeur(g, s):
2      """largeur(g: list[list[int]], s: int) -> NoneType"""
3      n = len(g)
4      decouvert = [False for _ in range(n)]
5      decouvert[s] = True
6      courant = [s]
7      suivant = []
8      while len(courant) != 0:
9          x = courant.pop()
10         for y in g[x]:
11             if not decouvert[y]:
12                 decouvert[y] = True
13                 suivant.append(y)
14         if len(courant) == 0:
15             courant = suivant
16             suivant = []
```

Si l'on souhaite effectuer une action pour chaque sommet à l'aide d'une fonction `f(x: int) -> NoneType`, on l'appellera juste avant d'avoir marqué le sommet avec l'instruction `decouvert[x] = True`, c'est-à-dire aux lignes 5 et 12.

Comme pour le parcours en profondeur, le parcours en largeur a une complexité temporelle en $O(|S|+|A|)$ et une complexité spatiale en $\Theta(|S|)$. En effet, chaque sommet est placé au plus une fois dans la pile *suivant*, la première fois qu'il est rencontré. Donc chaque arc $x \to y$ est examiné au plus une fois, lorsque le sommet x est retiré de l'ensemble courant.

Exercice 3

⇨ Écrire une fonction prenant entrée un graphe et une source et renvoyant le tableau des distances de chaque sommet à la source. Le tableau contiendra `None` pour un sommet qui n'est pas accessible.

8.2.4 Plus court chemin

Dans cette section, on se donne un graphe pondéré $G := (S, A, \rho)$ ainsi qu'une source s. On rappelle que le fonction de poids avec laquelle on travaille est positive. On cherche à déterminer pour chaque sommet x, le poids minimal d'un chemin reliant s à x. Afin d'utiliser une terminologie plus conventionnelle, on imaginera que les poids représentent des distances et on utilisera les termes de distance et de plus court chemin.

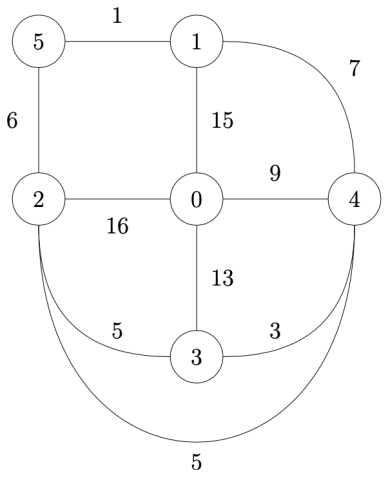
Algorithme de Dijkstra

L'implémentation de l'algorithme de Dijkstra se fait simplement en utilisant une file de priorité pour notre sac dans notre parcours générique. Les priorités sont des distances et c'est ainsi que nous les appelerons dans notre description.
— On commence par insérer la source dans la file avec une distance de 0.

- Tant que la file de priorité n'est pas vide :
  - On récupère l'élément x ayant la plus faible distance δ.
  - Si x n'a pas encore été visité :
    - On le marque comme visité. La distance δ est alors la distance entre la source et x.
    - Pour tous ses successeurs y, on les insère dans la file de priorité avec la distance $\delta + \rho(x \to y)$.

Remarquons qu'un même sommet pourra se retrouver plusieurs fois dans la file de priorité, avec des distances différentes. L'algorithme de parcours ne traitant que les sommets de la file qui n'en sont pas encore sortis, si l'on insère un sommet x avec une distance δ' et qu'il est déjà présent dans la file avec une distance $\delta \leqslant \delta'$, cette insertion n'affecte pas le déroulement de notre programme. Si par contre $\delta' < \delta$, c'est l'ancien élément présent dans la file qui est ignoré. Tout se passe donc comme si la distance δ du sommet x était mise à jour à $\min(\delta, \delta')$.

Afin de se familiariser avec cet algorithme, nous allons l'exécuter sur le graphe suivant, en utilisant le sommet 0 pour source.



- On commence à placer le sommet 0 dans la file avec la distance 0.
- Le seul sommet de la file est le sommet 0. C'est donc celui dont la distance est minimale. On marque ce sommet comme visité puis on regarde ses voisins : 1, 2, 3 et 4. On les place dans la file de priorité avec les distances respectives de 15, 16, 13 et 9.
- Le sommet de la file ayant une distance minimale est 4. Cette distance est de 9. On le marque comme visité, puis on regarde ses voisins : 1, 3 et 2. Le chemin passant par 4 et allant à 1 a une distance de 16 qui n'est pas inférieure à la distance actuelle pour 1. Par contre, le chemin passant par 4 et allant à 3 a une distance de 12 qui est inférieure à la distance actuelle de 13 pour 3. C'est aussi le cas du chemin passant par 4 et allant à 2 dont la distance est 14. On met donc ces distances à jour dans notre file. Les sommets de la file sont donc désormais les sommets 1, 2, et 3 de distances respectives 15, 14 et 12.
- Le sommet de la file ayant une distance minimale est 3. Cette distance est de 12. Aucun des chemins passant par 3 et menant à ses voisins ne permet d'obtenir une meilleure distance que celle que nous avons actuellement.
- Le sommet de la file ayant une distance minimale est 2. On marque ce sommet comme visité, puis on regarde ses voisins : 4, 3, 0 et 5. La distance du sommet 2 étant 14, on insère donc le sommet 5 avec une distance de 20. Les sommets de la file sont désormais les sommets 1 et 5 de distances respectives 15 et 20.
- Le sommet de la file ayant une distance minimale est 1. Cette distance est de 15. On marque ce sommet comme visité, puis on regarde ses voisins : 0, 4 et 5. Le chemin passant par 1 et allant à 5 a une distance de 16 qui est inférieure à la distance temporaire de 20. On met donc à jour cette distance.
- Le sommet de la file ayant une distance minimale est 5. Cette distance est de 16. On marque ce sommet comme visité. L'étude de ses voisins ne donne lieu à aucune mise à jour, car ils ont déjà tous été traités.
- À la boucle suivante, il n'y a plus de sommet dans notre file et l'algorithme s'arrête. Les distances du sommet 0 aux autres sommets du graphe sont donc 0 pour 0, 9 pour 4, 12 pour 3, 14 pour 2, 15 pour 1 et 16 pour 5.

Pour l'implémentation, nous utilisons le module `heapq` de Python.

```python
import heapq

def dijkstra(g, s):
    """dijkstra(g: list[list[tuple[float, int]]], s: int) -> list[float]"""
    n = len(g)
    visite = [False for _ in range(n)]
```

```
    dist = [None for _ in range(n)]
    filep = []
    heapq.heappush(filep, (0.0, s))
    while len(filep) != 0:
        delta, x = heapq.heappop(filep)
        if not visite[x]:
            dist[x] = d
            visite[x] = True
            for rho, y in g[x]:
                heapq.heappush(filep, (delta + rho, y))
    return dist
```

Si nous ne disposons pas de file de priorité, nous pouvons en faire une implémentation à la main (qui ne sera malheureusement pas très efficace). Pour cela, nous allons utiliser deux tableaux *visité* et *dist* dont la longueur est le nombre n de sommets du graphe. Lorsqu'un sommet x est dans l'état *inconnu* c'est-à-dire lorsqu'il n'a pas encore été inséré dans la liste, `dist[x]` est égal à `None` et `visite[x]` est égal à `False`. Lorsqu'un sommet x est dans la file, `dist[x]` contient sa priorité, et `visite[x]` est égal à `False`. Enfin, lorsque x est sorti de la file de priorité, `dist[x]` contient la distance de la source au sommet x et `visite[x]` est égal à `True`. On commence par écrire une fonction `prochain_sommet` qui renvoie le sommet de la file dont la distance est minimale.

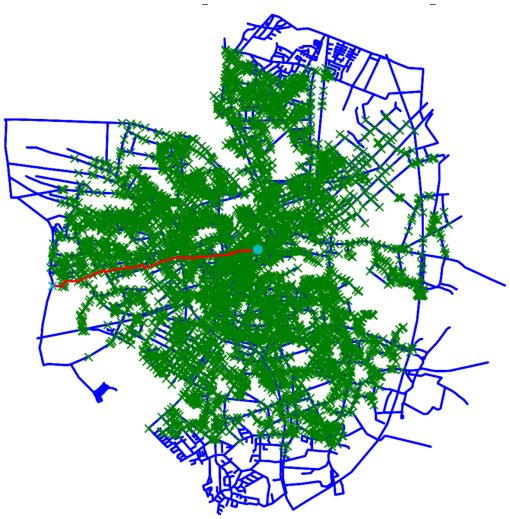
```
def prochain_sommet(dist, visite):
    """prochain_sommet(dist: list[float], visite: list[bool]) -> int"""
    n = len(dist)
    min_value = None
    min_x = None
    for x in range(n):
        if (not visite[x]) and (dist[x] != None) \
                and (min_value == None or dist[x] < min_value):
            min_value = dist[x]
            min_x = x
    return min_x
```

L'algorithme de Dijkstra s'écrit alors naturellement.

```
def dijkstra(g, s):
    """dijkstra(g: list[list[tuple[float, int]]], s: int) -> list[float]"""
    n = len(g)
    visite = [False for _ in range(n)]
    dist = [None for _ in range(n)]
    dist[s] = 0.0
    while True:
        x = prochain_sommet(dist, visite)
        if x == None:
            break
        visite[x] = True
        for rho, y in g[x]:
            delta = dist[x] + rho
            if dist[y] == None or delta < dist[y]:
                dist[y] = delta
    return dist
```

Algorithme A*

L'algorithme de Dijkstra permet de déterminer la distance d'une source s à l'ensemble des sommets accessibles depuis s. Si l'on s'intéresse uniquement à la distance entre la source et un but b, il est possible d'arrêter l'algorithme dès que le sommet b est marqué comme visité. Sur la figure ci-dessous, on a réalisé une recherche du meilleur chemin dans la ville d'Oldenburg, en Allemagne, en partant d'une source située en centre-ville et pour aller vers un but situé en périphérie, à l'ouest de la ville. Le chemin optimal trouvé est en rouge.



Nous avons colorié en vert l'ensemble des sommets « visités » par l'algorithme de Dijkstra. La zone couverte par l'algorithme est formée de tous les points dont la distance à la source est inférieure à la distance entre s et b. Cependant, notre intuition nous dit qu'il n'est surement pas utile d'aller traiter des sommets qui se trouvent tout à l'est de la ville alors que notre but est à l'ouest. Cette intuition se fonde sur le fait que la distance entre deux sommets x et y du graphe est supérieure à la distance à vol d'oiseau entre ces deux sommets.

Pour exploiter cette idée, nous allons définir la fonction $h : S \to \mathbb{R}$, appelée *heuristique*, par

$$\forall x \in S, \quad h(x) \coloneqq \|x - b\|$$

et définir une nouvelle distance ρ' sur A en définissant, pour tout arc $x \to y$

$$\rho'(x \to y) = \rho(x \to y) + h(y) - h(x).$$

Remarquons tout d'abord que ρ' est bien à valeurs positives puisque si $x \to y$ est un arc

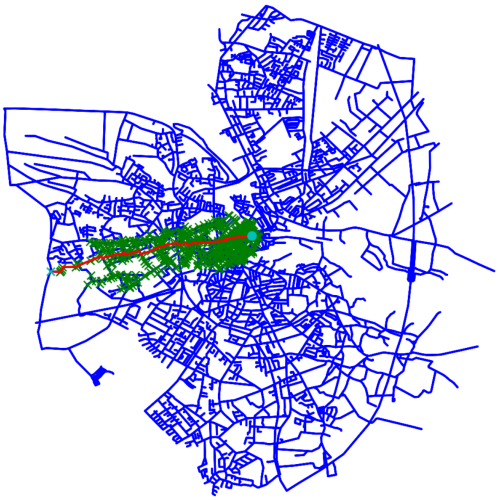
$$\rho(x \to y) \geqslant \|x - y\| \geqslant |\|x - b\| - \|y - b\|| \geqslant \|x - b\| - \|y - b\| = h(x) - h(y),$$

donc $\rho'(x \to y) = \rho(x \to y) + h(y) - h(x) \geqslant 0$. Remarquons enfin que, quel que soit le chemin $z_0 \to z_1 \to \cdots \to z_n$, on a $\rho'(z_0 \to z_1 \to \cdots \to z_n) = \rho(z_0 \to z_1 \to \cdots \to z_n) + h(z_n) - h(z_0)$. En particulier

$$\rho'(s \to z_1 \to \cdots \to z_{n-1} \to b) = \rho(s \to z_1 \to \cdots \to z_{n-1} \to b) + h(b) - h(s).$$

Puisque $h(b) - h(s)$ est indépendant du chemin allant de s à b, tout chemin entre ces deux sommets de distance minimale pour ρ' est minimal pour ρ. L'algorithme de Dijkstra appliqué sur le même graphe avec la distance ρ' au lieu de la distance ρ donnera donc un même chemin minimal entre s et b. Remarquons que cette nouvelle distance va privilégier les sommets se situant en direction du but. En effet, dans le cas extrême où il existe une ligne droite entre la source et le but et plusieurs sommets du graphe sont alignés, la distance entre ces sommets pour la nouvelle distance ρ' va être nulle et ce sont bien ces sommets qui vont être traités en premier.

En reprenant la recherche du meilleur chemin entre le centre et un point de la périphérie d'Oldenburg, on voit que l'algorithme de Dijkstra appliqué à nouvelle distance traite beaucoup moins de sommets avant d'arriver sur le but, tout en garantissant le fait que le chemin trouvé a une distance minimale pour la distance d'origine.



Chapitre 9

Langage Python

Cette annexe liste limitativement les éléments du langage Python (version 3 ou supérieure) dont la connaissance est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.

Aucune connaissance sur un module particulier n'est exigible des étudiants.

Toute utilisation d'autres éléments du langage que ceux que liste cette annexe, ou d'une fonction d'un module, doit obligatoirement être accompagnée de la documentation utile, sans que puisse être attendue une quelconque maîtrise par les étudiants de ces éléments.

Traits généraux

- Typage dynamique : l'interpréteur détermine le type à la volée lors de l'exécution du code.
- Principe d'indentation.
- Portée lexicale : lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.
- Appel de fonction par valeur : l'exécution de `f(x)` évalue d'abord x puis exécute f avec la valeur calculée.

Types de base

- Opérations sur les entiers (`int`) : `+`, `-`, `*`, `//`, `**`, `%` avec des opérandes positifs.
- Opérations sur les flottants (`float`) : `+`, `-`, `*`, `/`, `**`.
- Opérations sur les booléens (`bool`) : `not`, `or`, `and` (et leur caractère paresseux).
- Comparaisons `==`, `!=`, `<`, `>`, `<=`, `>=`.

Types structurés

- Structures indicées immuables (chaînes, tuples) : `len`, accès par indice positif valide, concaténation `+`, répétition `*`, tranche.
- Listes : création par compréhension `[e for x in s]`, par `[e] * n`, par `append` successifs ; `len`, accès par indice positif valide ; concaténation `+`, extraction de tranche, copie (y compris son caractère superficiel) ; `pop` en dernière position.
- Dictionnaires : création `{c_1 : v_1, ..., c_n : v_n}`, accès, insertion, présence d'une clé `k in d`, `len`, `copy`.

Structures de contrôle

- Instruction d'affectation avec `=`. Dépaquetage de tuples.
- Instruction conditionnelle : `if`, `elif`, `else`.
- Boucle `while` (sans `else`). `break`, `return` dans un corps de boucle.
- Boucle `for` (sans `else`) et itération sur `range(a, b)`, une chaîne, un tuple, une liste, un dictionnaire au travers des méthodes `keys` et `items`.
- Définition d'une fonction `def f(p_1, ..., p_n)`, `return`.

Divers

- Introduction d'un commentaire avec `#`.
- Utilisation simple de `print`, sans paramètre facultatif.
- Importation de modules avec `import module`, `import module as alias`, `from module import f, g, ...`

— Manipulation de fichiers texte (la documentation utile de ces fonctions doit être rappelée ; tout problème relatif aux encodages est éludé) : `open`, `read`, `readline`, `readlines`, `split`, `write`, `close`.
— Assertion : `assert` (sans message d'erreur).