Algorithmique

Luc Pellissier

2020 → 2021

Sommaire

Première partie

Cours

1

Programmer

On introduit les notions de programmes et d'algorithmes,

1.1 Ce qui calcule, ce qu'on calcule

On s'intéresse à l'automatisation de calculs. Pour cela, on commence par réfléchir à l'automate qui réalise ces calculs. Dans le cadre de ce cours, on va prendre (volontairement) une définition extrêmement large de machine.

Définition 1. Une *machine* est un dispositif (physique ou imaginaire) évoluant au cours du temps, et transformant ses entrées en sorties.

Autrement dit, on se permettra de parler de machine dès qu'on peut distinguer trois éléments :
- une évolution temporelle, fut-elle instantannée ;
- un sous-système dont on peut manipuler librement la position d'une manière qui affectera l'évolution temporelle. On le considérera comme l'entrée de la machine ;
- un sous-système dont on peut observer librement la position. On le considérera comme la sortie.

Par exemple, un piano est une machine :
- frapper le piano à différents endroits, de différentes manières, provoque des sons, qui durent un certain temps, puis s'arrêtent ;
- on peut considérer que les touches et les pédales sont des entrées : on peut librement appuyer dessus ;
- le son émis par le piano en est la sortie.

On voit qu'on pourrait prendre une structure de machine différente pour le même piano, en considérant par exemple qu'on le frapper à d'autres endroits que sur les touches et les pédales (ce qui produirait d'autres sons) — donc, en changeant l'entrée — ou même en considérant que la sortie n'est pas purement auditive, mais aussi visuelle (par exemple, en y mettant le feu[1]).

La machine en elle-même ne dit pas comment l'utiliser. Pour rester sur l'exemple du piano, est-ce qu'on a plusieurs manière d'appuyer sur une touche (plus ou moins violemment) ? Est-ce qu'on peut appuyer sur plusieurs touches en même temps (avec plusieurs doigts, ou même en jouant un cluster avec le poing ou encore l'avant-bras) ?

On doit donc spécifier plus précisément ce qu'on a le droit de faire avec la machine : quelles *actions* sont autorisées. D'une certaine manière, il s'agit de se fixer un vocabulaire primitif, et pour chaque terme de ce vocabulaire, la manière dont il faut agir sur la machine. Dans le cas de notre piano, on peut donc s'autoriser à appuyer sur la première touche, sur la deuxième,...

Cependant, si on veut être un peu plus général, les actions élémentaires ne suffisent pas. Considérons un cas de calcul, la répartition des sièges après les élections dans des universités. L'article D. 719-20 du *Code de l'Éducation* dispose que :

1. Évidemment, on pense à « I am yours » de The Makemake, représentant l'Autriche à l'Eurovision 2015.

> « Les membres des conseils sont élus au scrutin de liste à un tour à la représentation proportionnelle avec répartition des sièges restant à pourvoir selon la règle du plus fort reste, sans panachage. Pour l'élection des représentants des enseignants-chercheurs et des personnels assimilés au conseil d'administration de l'université, il est attribué dans chacun des collèges deux sièges à la liste qui a obtenu le plus de voix. Les autres sièges sont répartis entre toutes les listes à la représentation proportionnelle au plus fort reste. Toutefois, les listes qui n'ont pas obtenu un nombre de suffrages au moins égal à 10 \admises à la répartition des sièges [...] »

Et l'article D. 719-21 du même code :

> « Le nombre de voix attribuées à chaque liste est égal au nombre de bulletins recueillis par chacune d'elles.
> Le nombre de suffrages exprimés est égal au total des voix recueillies par l'ensemble des listes.
> Le quotient électoral est égal au nombre total de suffrages exprimés divisé par le nombre de sièges à pourvoir. Pour l'élection des représentants des usagers, le quotient électoral est égal au nombre total de suffrages exprimés divisé par le nombre de sièges de membres titulaires à pourvoir.
> Sous réserve des dispositions prévues au deuxième alinéa de l'article D. 719-20, il est attribué à chaque liste autant de sièges que le nombre de ses suffrages contient de fois le quotient électoral.
> Pour l'élection des représentants des usagers, chaque liste a droit à autant de sièges de membres titulaires que le nombre de voix recueillies par elle contient de fois le quotient électoral. Un suppléant est élu avec chaque membre titulaire élu.
> Les sièges non répartis par application des dispositions précédentes sont attribués successivement aux listes qui comportent les plus forts restes. Lorsqu'une liste a obtenu un nombre de voix inférieur au quotient électoral, ce nombre de voix tient lieu de reste.
> Si plusieurs listes ont le même reste pour l'attribution du dernier siège, celui-ci revient à la liste qui a obtenu le plus grand nombre de suffrages. En cas d'égalité de suffrages, le siège est attribué au plus jeune des candidats susceptibles d'être proclamés élus. [...] »

Si l'on décrit bien un calcul, on voit qu'il ne s'agit pas que d'une suite d'action. Il faut pouvoir prendre des décisions conditionnelles (« Si..., alors »), ainsi que pouvoir stocker de l'information (par exemple pour la comparer à autre chose).

Exercice 1. Lister toutes les opérations élémentaires nécessaires pour programmer les deux extraits du Code de l'Éducation ci-dessus.

Sont-elles vraiment toutes élémentaires ?

Définition 2. Un *langage* est la donnée d'opérations élémentaires, ainsi que d'une manière de les composer en un ensemble plus grand, qu'on appellera un *programme*.

Une *sémantique* d'un langage décrit comment les opérations élémentaires et la composition doivent être interprétées par une machine donnée.

Une même machine peut donc interpréter plusieurs langages, selon la complexité de ce qu'on s'autorise. Pour reprendre l'exemple du piano :

— le langage disant sur quelle touche appuyer, avec quelle intensité et pendant quelle durée ;
— le langage pour lequel le programme est une partition simple, où on « joue un do bémol double croche ».

Le second langage est plus *haut niveau* : la traduction entre le langage et les actions effectivement prises sur la machine est plus lourde, mais il permet d'être plus proche des intentions de la personne qui programme — à condition que ce qu'elle veut programmer puisse être exprimé.

De même, un même langage peut être interprété sur différentes machines, avec des sémantiques différentes : un *do bémol double croche* ne va pas être joué de la même manière, ni avoir le même son, sur un piano ou un accordéon.

Chaque langage de programmation propose donc un **vocabulaire restreint et parfaitement spécifié**[2] permettant d'exprimer des calculs. Programmer dans un langage de programmation, c'est arriver à s'exprimer dans un vocabulaire restreint. Évidemment, il y a plein de manières de s'y plier :

— la plus prosaïque peut être de s'y appliquer ;
— on peut aussi enrichir le vocabulaire (voire la grammaire), définir de nouveaux termes ;
— voire créer un nouveau langage dont on va donner la sémantique grâce à un ancien[3].

1.2 Fonctions et données

On apprend depuis l'école primaire à faire des opérations assez complexes, nécessitant de combiner des opérations plus simples. Un exemple assez spectaculaire en est la multiplication.

La *multiplication* peut être définie de plusieurs manières ; elle est souvent introduite comme une *addition itérée*, c'est-à-dire en définissant :

$$n \times m := \underbrace{n + \cdots + n}_{m \text{ fois}}.$$

Cette définition suppose que l'on peut additionner, et compter le nombre de fois que l'on fait une certaine opération. On peut s'en servir pour calculer la multiplication de n'importe quels nombres (entiers positifs).

On reviendra plus tard sur une notion de complexité permettant de mesurer cela formellement, mais on n'utilise pas cette définition parce qu'elle serait bien trop lente. Dès le ce2 (en France en tout cas), on apprend une autre technique de multiplication. En réalité, elle ne s'applique pas sur des nombres. En effet, elle nécessite :

2. En théorie… hélas, pour des langages de programmation réalistes, on en est encore loin.
3. La Direction Générale des Finances Publiques a ainsi pu écrire des langages de programmation spécialisés dans le calcul de l'impôt.

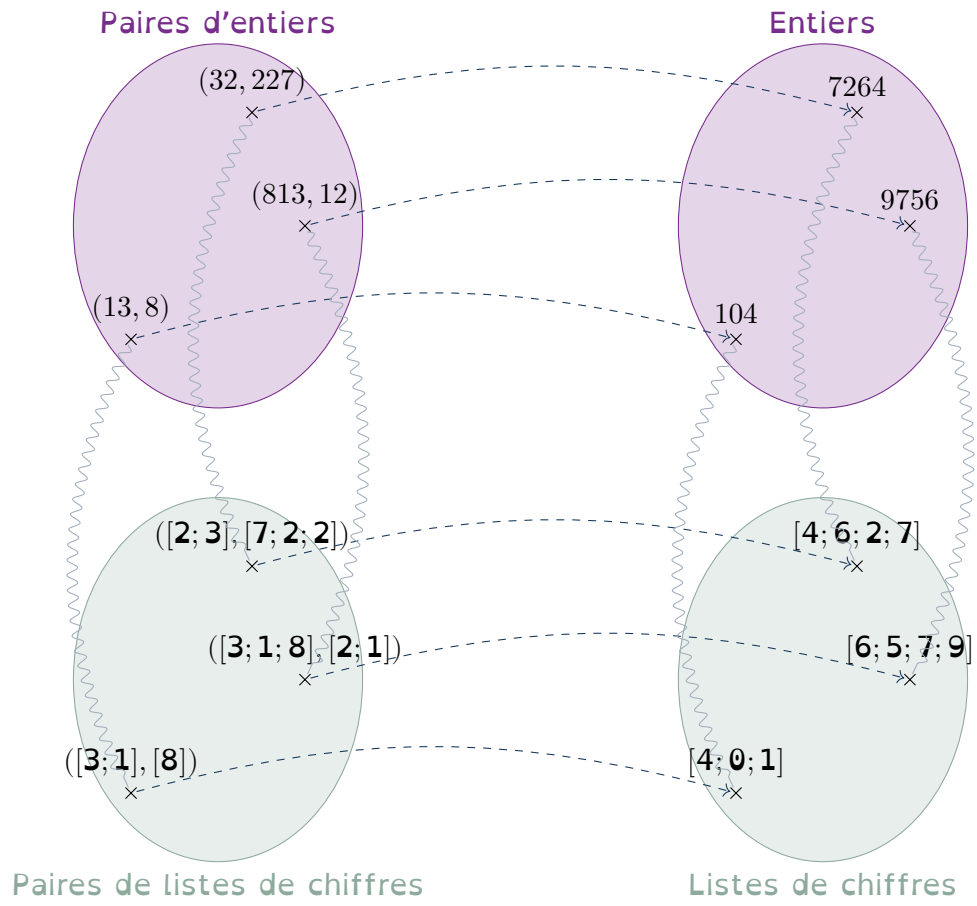


Figure 1.1 – Diagramme sagittal de la fonction produit et du programme de la multiplication

— que les nombres soient écrits en notation positionelle en base 10 ;
— que ces écritures soient placées l'une en dessous de l'autre.

Autrement dit, on ne travaille pas sur les nombres (des concepts) mais sur des *représentations* finies.

Il faut donc toujours distinguer deux niveaux :
— celui conceptuel, abstrait ;
— celui du calcul, sur des représentations.

Selon la représentation, le même calcul peut-être très facile ou au contraire, rédhibitoire. On parlera de *fonction calculable* pour la notion abstraite et de programme pour celle calculatoire — comme représenté Figure 1.1.

1.3 Algorithmes

Gesetze sind wie Würste, man sollte besser nicht dabei sein, wenn sie gemacht werden.
— Otto von Bismarck (1815→1898)
duc de Lauenburg et prince de Bismarck
père de l'unité allemande.

La notion de programme est très efficace pour décrire le calcul ; néanmoins,

elle est dépendante d'un modèle de calcul qui est assez accessoire : si un langage écrit l'addition + et un autre plus, on va avoir deux programmes différents mais qui ont pourtant le même sens : qui ne sont que des traductions les uns des autres. C'est cette idée qui est capturée par la notion d'algorithme : celle d'un programme indépendamment d'un modèle d'un calcul et de ses détails d'implémentation. Hélas, on n'a pas encore de définition mathématique de ce qu'est un algorithme, on doit se contenter d'une définition informelle.

Définition 3 (algorithme). Un *algorithme* est la description d'une série d'opérations non-ambiguës visant à obtenir un résultat.

Le cas échéant, on dira qu'un programme *implémente* un algorithme, qui lui-même *réalise* une fonction.

Il peut y avoir plusieurs algorithmes réalisant une même fonction ; par contre, il y a toujours plusieurs programmes implémentant un algorithme. Certains programmes n'implémentent pas un algorithme (voir Figure 1.2).

De cette définition un peu floue, on peut tirer quelques caractéristiques d'un algorithme :

finitude un algorithme doit aboutir à un résultat, ce qui présuppose qu'il aboutisse quelque part. On pourrait être plus exclusif et déclarer qu'aboutir à un résultat en un temps fini mais très long (plus long que la durée de vie de l'univers...) n'est pas aboutir à un résultat. En tout cas, en première approximation, on demande à ce qu'un algorithme soit fini — c'est-à-dire termine.
En conséquence, tous les programmes n'implémentent pas des algorithmes. Par exemple, un jeu vidéo peut tourner arbitrairement longtemps, et n'aboutit pas à un résultat (la victoire ou la défaite peuvent être considérés comme des résultats, mais ce serait passer à côté de la question), ainsi que tous les programmes dont nous voulons qu'ils soient toujours en service — comme un serveur mail ou un système d'exploitation.

définition chaque étape doit être définie le plus précisément possible. C'est très difficile à faire en français[4], et les langages de programmation ont été inventés pour être explicite. Néanmoins, on essaiera de l'être.

entrée les entrées doivent être spécifiées. La correction ou la terminaison d'un algorithme peuvent dépendre des entrées considérées.

sortie de même, préciser ce que l'algorithme produit (un résultat, une vérification que quelque chose est vrai...) est indispensable

effectivité chaque étape, doit, non-contente d'être parfaitement définie, être effectivement calculable. Par exemple, le nombre d'atomes dans l'univers est tout aussi défini qu'il est incalculable.

L'analogie la plus courante compare un algorithme à une recette de cuisine : une succession d'étape dans le but d'arriver à un résultat, dont chaque étape et à la fois précise, mais pouvant être plus ou moins explicite.

4. Ce n'est pas à des juristes que je vais faire semblant d'apprendre que des formulations peuvent se révéler ambiguës parfois longtemps après leur écriture.

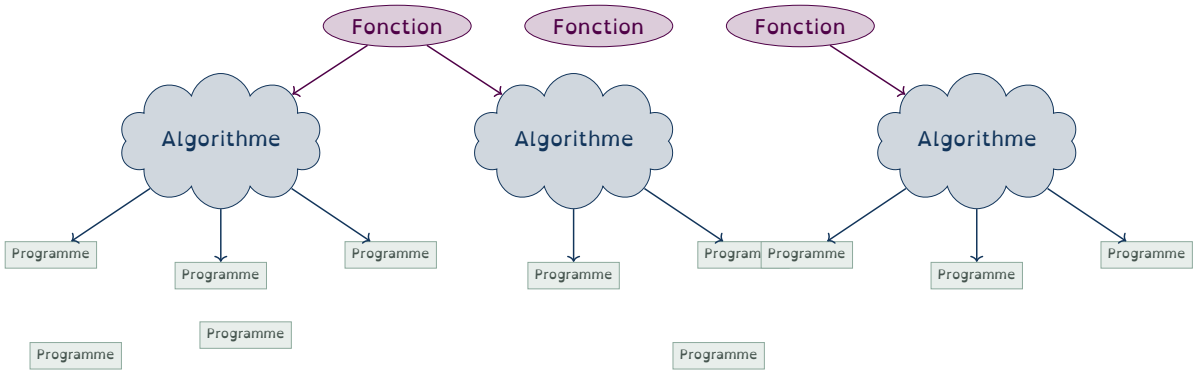


Figure 1.2 – Fonctions, algorithmes, programmes

On a vu sur l'exemple de la multiplication que la manière dont les données sont présentées est cruciale. En effet, l'algorithme de la multiplication tel qu'appris à l'école primaire suppose que les nombres sont écrits en base 10 ; on pourrait l'écrire en base 2, ou donner un autre algorithme pour les nombres écrits en chiffres romains. Un algorithme est donc intrinsèquement lié à la manière dont les données sont représentées, et doit être adapté, quand par exemple, on change de base.

On voit donc qu'un algorithme n'agit pas sur des nombres abstraits, mais sur des représentations. Dans certains cas, faire un pré-traitement des données, pour les organiser d'une manière agréable, peut être nécessaire.

Remarque 1. On compare souvent un algorithme à une recette de cuisine. On va ici utiliser une autre métaphore, inspirée de la citation de Bismark sur la politique et les saucisses. On peut obtenir des saucisses de bien des manières différentes (avec des viandes différentes, du seitan, des champignons, des haricots,...). Toujours est-il qu'une fois qu'on a fixé une composition, une usine à saucisse va avoir pour composantes :

des entrées des protéines, de la graisse, des épices, de quoi faire le boyau ;

des sorties des saucisses ;

un processus fini transformant les entrées en sorties ;

d'autres ressources de l'eau (pour nettoyer), de l'énergie, du terrain, du personnel...

L'usine transforme ses entrées en sorties, à l'aide de ces ressources, selon un certain processus. Le processus va faire appel à des abstractions (par exemple, le contenu d'un chariot, qui va contenir de la graisse hachée), qui ne sont pas pour autant des entrées (la graisse non-hachée oui, mais pas celle hachée).

On distingue donc les entrées et les sorties, qui sont la manière dont l'usine-algorithme dialogue avec l'extérieur du monde ; de ce qui compose son état interne (le fait qu'un hachoir soit plein,...) ; des ressources qu'il utilise, qu'on peut par ailleurs chercher à quantifier.

1.4 python

Un langage de programmation est un objet socio-technique. De ce fait, ses qualités et ses défauts sont toujours liés à son histoire, et pas uniquement

à ses caractéristiques techniques. En règle générale, étant donné un langage, on peut se demander :

— qui l'utilise ?

Est-il utilisé seulement par un groupe (une personne ? une entreprise ? une administration ? une industrie entière ?) ?

— qui en est responsable ?

Certains langages sont abandonnés et n'évoluent plus, d'autres sont gérés par une seule personne, par une entreprise, par un organisme de standardisation,...

Comment la responsabilité est assurée (plus ou moins formellement) ?

— est-ce que ce groupe-là a un intérêt fort pour la documentation ?

— y-a-t'il beaucoup de ressources dessus ? Disponibles sous quelle forme ?

— ...

en plus des propriétés plus techniques.

Dans le cas de python, il s'agit d'un langage créé en 1991[5] par Guido van Rossum, informaticien néerlandais qui en a assuré la direction jusqu'en 2019 avant de passer la main à comité de cinq personnes, le *Python Steering Council*, élu par les membres de l'équipe de développement, elleux-mêmes coopté·es.

Le langage est défini par une documentation disponible en ligne[6] mais surtout par une *implémentation de référence*, CPython. En effet, un langage de programmation peut être défini par :

— un texte écrit le plus souvent en anglais, décrivant les différents concepts du langage, les mots-clefs,...

Si c'est une méthode accessible, il est très difficile d'être complet et non-ambigü.

— une description mathématique.

C'est une méthode complète, mais peu accessible, et qu'on ne sait pas vraiment faire.

— un programme qui sache exécuter les programmes dans ce langage.

C'est une méthode complète (par définition, on décide que le comportement de l'implémentation de référence est le bon), mais qui nécessite de pouvoir exécuter ce programme pour en connaître le comportement.

Dans le cas de python, CPython est un programme qui simule une machine, la *Python Virtual Machine*, et qui transforme le code en entrée en une série d'instruction pour cette machine.

Remarque 2. On a donc plusieurs machines, plusieurs programmes, plusieurs langages. Essayons d'y voir plus clair.

L'ordinateur sur lequel python tourne est une première machine, physique. Il a un langage, constitué d'instructions qui ont un effet physique.

La *Python Virtual Machine* est un programme, écrit dans le langage de l'ordinateur sur lequel tourne python ; et c'est aussi une machine, c'est-à-dire que ce programme a une entrée, une sortie et une évolution temporelle. Cette machine a pour langage le *bitcode* python.

---

5. C'est à garder à l'esprit quand on dit que c'est un langage récent. « Récent » en informatique n'a pas forcément le sens que l'on attend.

6. <https://docs.python.org/3/>

CPython commence par traduire un programme python en bitcode python, puis le fait exécuter par la *Python Virtual Machine*.

Il existe d'autres types d'implémentations de langage de programmation, qui traduisent (on utilise plutôt le verbe *compiler*) directement un programme écrit en python en une série d'instruction pour la machine sous-jacente.
Il existe d'autres implémentations de python, mais seule CPython fait autorité.

Remarque 3. Vous faites tourner en permanence de nombreuses machines virtuelles sur vos machines (à minima, la *Java Virtual Machine*, la *Dalvik Virtual Machine* si vous utilisez Android, une machine Lisp si vous utilisez Emacs, une machine virtuelle Javascript (sûrement *V8*, *JavaScriptCore* ou *SpiderMonkey*)),...

python est un langage qui a énormément de défauts, mais une qualité rédemptrice : il est à la mode. De ce fait, il y a :
— beaucoup de ressources (livres, tutoriels, vidéos,...) ;
— beaucoup de bibliothèques (c'est-à-dire des boites-à-outils de composants logiciels permettant de faire des choses assez compliquées) déjà faites par quelqu'un d'autre sur à peu près tous les sujets ;
— beaucoup d'outils (des éditeurs qui en connaissent déjà la syntaxe,...).
Enfin (et sans doute pour les mêmes raisons que celles pourquoi il est à la mode), les erreurs de python sont graves, mais concernent des concepts que la majorité des programmeur·ses ne maîtrisent pas (le *scoping* par exemple), et ses défauts des fonctions que la majorité des programmeur·ses n'utilisent pas (la métaprogrammation par exemple). Bref, les défauts de python eux aussi suivent la mode.
Il est beaucoup utilisé comme « langage de glu », c'est-à-dire pour assembler des composants logiciels et les faire communiquer.
Par ailleurs, la communauté est assez mal gérée : la transition entre python 2 et python 3 est un exemple de ce qu'il ne faut pas faire.

Logiciel libre

Enfin, CPython est un *logiciel libre*, ce qui est un argument, en soi, pour l'utiliser.
Le mouvement du logiciel libre peut être compris comme un questionnement sur la notion de propriété. Peut-on posséder un logiciel ?
— si on considère un programme comme une structure mathématique, c'est évident que non : on ne considère pas qu'il y ait de propriété liée aux mathématiques ;
— si on considère que c'est une œuvre de l'esprit, alors son auteurice a des droits (moraux et patrimoniaux) dessus.
Plus généralement, quel est le sens de la propriété de quelque chose qui est, par essence, abondant (on peut copier un logiciel sans en affecter les propriétés) ?
Après beaucoup de tâtonnements, on a atterri dans un système où les différents régimes sont :

la vente pure et simple un éditeur de logiciel vend tous ses droits à quelqu'un.
Ça n'existe que pour des logiciels extrêmement spécialisés ;

la licence un éditeur de logiciel concède un contrat de licence permettant une utilisation **dans certains cas**, par exemple, en interdisant de s'en servir à usage commercial, ou pour faire fonctionner une centrale nucléaire ;

le domaine public l'auteurice abandonne tous ses droits, l'œuvre tombe dans le domaine public. Ce n'est pas vraiment autorisé en droit français. L'exemple le plus connu est TEX.

Certains contrats de licence peuvent servir à donner plus de droits à l'utilisateurice. Citons-en deux :

les licences de type BSD l'auteurice fournit le code-source du logiciel. L'utilisateurice a le droit d'en faire ce qu'iel veut. Si iel le modifie et le redistribue, iel doit citer l'auteurice original·e.

les licences de type GPL l'auteurice fournit le code-source du logiciel. L'utilisateurice a le droit d'en faire ce qu'iel veut. Si iel le modifie et le redistribue, iel doit :
— citer l'auteurice original·e ;
— garder toutes ses modifications avec la même licence.
De fait, cela signifie que tous les dérivés du logiciel de base devront rester sous la même licence.

Dans les deux cas, on considère que ce sont des logiciels libres, car ils donnent :
— la liberté d'utiliser
— la liberté de reproduire
— la liberté d'étudier
— la liberté de modifier
Il n'est pas clair qu'on puisse raisonnablement prétendre *posséder* un logiciel si on n'a pas ces quatre libertés. Il est donc plus que raisonnable d'en utiliser autant que possible, si on cherche à s'émanciper, individuellement ou collectivement.

Accessoirement, c'est la loi ! Ainsi,

> « Les administrations mentionnées au premier alinéa de l'article L. 300-2 du code des relations entre le public et l'administration veillent à préserver la maîtrise, la pérennité et l'indépendance de leurs systèmes d'information.
> Elles encouragent l'utilisation des logiciels libres et des formats ouverts lors du développement, de l'achat ou de l'utilisation, de tout ou partie, de ces systèmes d'information. [...] »
>
> — *Article 16, loi n° 2016-1321 du 7 octobre 2016 pour une République numérique*

ou encore

> « Le service public de l'enseignement supérieur met à disposition de ses usagers des services et des ressources pédagogiques numériques.
> Les logiciels libres sont utilisés en priorité. »
>
> — *Article L. 123-4-1, Code de l'éducation*

1.5 Expressions et instructions

On peut tout d'abord se servir de python comme d'une calculatrice. Dans ce cas, on entre des *expressions*. Pour rester à un niveau conceptuel, une expression en python a, en plus de son texte, deux caractéristiques :

— un type, c'est-à-dire une catégorie qui détermine les opérations pouvant être faites avec cette expression (on ne fait pas la même chose avec un nombre, une lettre,...).

Étant donné une expression, on peut en connaître le type avec la fonction `type()`.

— une *valeur*, qui est ce que vaut l'expression.

Certaines expressions n'ont pas de type, par exemple parce qu'elles sont mal formées. Quand on appuie sur entrée, python va :

1. vérifier que l'expression est bien formée (par exemple `)5*+` est mal formée) ;
2. trouver son type ;
3. calculer sa valeur.

Calculer une valeur peut prendre du temps, c'est pourquoi ce n'est fait qu'en dernier.

Si on se souvient des exemples venus de textes de loi, il fallait pouvoir comparer des valeurs et prendre des décisions en fonction. De ce fait, python n'a pas que des *expressions* (qui valent quelque chose), mais aussi une deuxième catégorie : des *instructions* (qui font quelque chose)

Une variable est un nom que l'on donne, dont la valeur peut changer (varier...) au cours du temps. On fait un usage très courant de variables, y compris hors du domaine formel :

- considérons l'article 20 du *Code de procédure civile* : « Le juge peut toujours entendre les parties elles-mêmes. » Le « juge », pas plus que « les parties », ne désigne pas une personne déterminée, mais une personne variable, selon la procédure ;
- de manière plus spectaculaire, considérons le cri « Le roi est mort, vive le roi ! »[a] : le « roi » mentionné dans chacune des deux parties de la phrase n'est pas le même !

On va donc se donner une manière de modifier la valeur d'une variable, ainsi qu'une manière pour agir en fonction de cette valeur.
On va noter, si x est une variable et e une expression

$$x = e$$

l'assignation à x de la valeur de e, c'est-à-dire qu'on calcule la valeur de e, et dorénavant, on décide que x vaut ce résultat.
On va noter, si e_1 et e_2 sont deux expressions

$$e_1 == e_2$$

le test, qui va calculer chacune des deux expressions, et valoir vrai (que l'on note **True** en python) si leurs valeurs sont égales, et faux (**False**) sinon.

a. Cette déclaration était traditionnellement faite par le duc d'Uzès, premier pair du royaume, dès que le cercueil du roi défunt était descendu dans la crypte de la basilique de Saint-Denis.

On va progressivement voir un certain nombre de instructions. La première est donc l'assignation :

```
1 x = 3+5
```

n'a pas de valeur[7], mais a une action. La variable x change de valeur après exécution de la instruction.

Remarque 4. On note qu'au contraire, le test est, lui, bien une expression, qui vaut soit **True** soit **False**, et qui est du type bool des valeurs booléennes

On a une expression particulière, l'affichage :

```
1 print(x)
```

affiche la valeur de x (ici, une variable, mais qui pourrait être n'importe quelle expression).

Remarque 5. Il faut bien distinguer une expression, qui a une certaine valeur, de l'affichage de cette expression, qui affiche la valeur. Ce sera plus clair par la suite.

1.6 Structures de contrôle

Pour pouvoir programmer nos exemples (loi électorale, impôts, multiplication), il nous manque :

7. Et n'a pas non plus de type.

— la possibilité de refaire des choses un certain nombre de fois ;
— la possibilité de faire des choses différentes selon les valeurs des variables.

Pour cela, on a besoin d'autres instructions : les structures de contrôle.

Boucle for

La première chose qu'on peut vouloir faire est d'*itérer* : opérer la même opération pour chaque élément d'une collection (par exemple, calculer les impôts de chaque personne,...) ordonnée (on a besoin de savoir dans quel ordre on itère, a priori, ce n'est pas pareil de le faire dans un ordre ou dans l'autre).

Pour cela, python fournit la boucle **for**, qui se compose de trois éléments :
— une variable qui va être itérée : prendre une à une chaque valeur de la collection itérable ;
— un *itérable*, c'est-à-dire une collection dont on peut prendre un à un les éléments. On en verra plus tard d'autres exemples, mais l'itérable le plus simple est obtenu grâce à la fonction range() : ainsi range(1,6) contient les nombres de 1 (inclu) à 6 (exclu) ;
— un bloc qui va faire quelque chose et qui peut faire référence à la variable de boucle. Ce bloc peut-être arbitrairement long, mais doit être *indenté*, c'est-à-dire décalé d'une tabulation.

La syntaxe générale est :

```
for ⟨variable⟩in ⟨iterateur⟩:
    ⟨bloc⟩
```

Par exemple, si on sait que la fonction print() affiche l'élément qu'on lui passe en argument, on peut écrire une fonction affichant tous les nombres entre 1 (inclu) et 6 (exclu).

```
1 for i in range(1,6):
2     print(i)
```

C'est la première fois qu'on *affiche* quelque chose. Il faut distinguer avoir une valeur, qui est une propriété d'une expression — en particulier, une boucle **for** n'a pas de valeur, vu que c'est une instruction — et un affichage, qui est déclenché par la fonction print(). En général, dans le cadre de ce cours, on affichera peu.

Boucle while

Dans certains cas, on doit itérer sans savoir combien de fois on itère. Pour cela, on a une autre boucle, la boucle **while**, qui exécute le bloc *tant qu'une condition est vraie*. Sa syntaxe générale est donc :

```
while ⟨condition⟩:
    ⟨bloc⟩
```

```
1 while i < 16:
2     print(i)
3     i = i + 3
```

Conditionnelle *if*

Enfin, il faut pouvoir prendre des décisions et, selon les valeurs, ne pas exécuter les mêmes instructions.

```
    if ⟨condition⟩:
        ⟨bloc⟩
(   elif ⟨condition⟩:   )
        ⟨bloc⟩
(   else:               )
        ⟨bloc⟩
```

```
x = 5
if x < 10:
    print("un chiffre")
else:
    print("deux chiffres")
```

None

1.7 Fonctions

Une fois qu'on a programmé un certain calcul, on peut vouloir enrichir notre langage avec, et donc, qu'au lieu de systématiquement retaper la même suite d'instructions, on puisse juste utiliser le mot que l'on vient d'introduire.

Supposons par exemple qu'on ait programmé la séquence suivante

```
if x < 0 or x > 20:
    print("Ceci n'est pas une note")
elif x < 8:
    print("Redoublement")
elif x < 9.5:
    print("Rattrapage")
elif x < 10:
    print("Points du jury")
elif x < 12:
    print("Passable")
elif x < 14:
    print("Assez Bien")
elif x < 16:
    print("Bien")
elif x < 18:
    print("Très Bien")
elif x <= 20:
    print("Très Bien — Félicitations du Jury")
```

qui, selon la valeur de la variable x affiche un texte indiquant si un·e étudiant·e a réussi son année et avec quelle mention. On veut pouvoir l'appliquer à la chaîne à chaque étudiant·e. On peut donc en faire une *fonction* qui prend en entrée la note.

```
def validation(note:float) -> str:
    """Affiche si une personne valide ou pas et avec quelle mention"""
```

```
    if note < 0 or note > 20:
        print("Ceci n'est pas une note")
    elif note < 8:
        print("Redoublement")
    elif note < 9.5:
        print("Rattrapage")
    elif note < 10:
        print("Points du jury")
    elif note < 12:
        print("Passable")
    elif note < 14:
        print("Assez Bien")
    elif note < 16:
        print("Bien")
    elif note < 18:
        print("Très Bien")
    elif note <= 20:
        print("Très Bien — Félicitations du Jury")
```

Une fonction a un nom, ici validation. Ce nom est arbitraire, et pourrait être n'importe quoi. Elle prend un certain nombre d'*arguments* en entrée : ce sont des variables, dont la valeur n'est pas définie, mais qui ont juste un nom (arbitraire, lui aussi) et un type. Ici, la fonction ne prend qu'un argument : note, de type float (car c'est un nombre à virgule).
Ainsi, la première ligne peut être lue comme : si on fournit un nombre à virgule à validation, le corps de la fonction va être exécuté, où note aura la valeur que l'on aura fournit, et le tout va produire une chaîne de caractères.

On va pouvoir exécuter validation(3) comme validation(18).

Cette fonction a un défaut : elle affiche, ce qui est pratique pour un humain, mais ne permet pas de continuer le calcul. Par exemple, si on veut garder toutes les mentions d'une promotion (pour pouvoir les compter par exemple), on ne peut pas le faire. Une meilleur solution est donc de décider que le *résultat* de validation(3) est "Redoublement", et que si on veut afficher ce résultat, on doit faire print(validation(3)).

Pour cela, on a besoin de l'instruction **return** qui arrête un calcul en cours et fixe à la fonction la valeur de l'expression.

```
def validation(note:float) -> str:
    """Affiche si une personne valide ou pas et avec quelle mention"""
    if note < 0 or note > 20:
        return "Ceci n'est pas une note"
    elif note < 8:
        return "Redoublement"
    elif note < 9.5:
        return "Rattrapage"
    elif note < 10:
        return "Points du jury"
    elif note < 12:
```

```
12        return "Passable"
13    elif note < 14:
14        return "Assez Bien"
15    elif note < 16:
16        return "Bien"
17    elif note < 18:
18        return "Très Bien"
19    elif note <= 20:
20        return "Très Bien — Félicitations du Jury"
```

Ainsi, la fonction validation() prend en entrée un nombre à virgule (float) et renvoie une chaîne de caractères (str). On notera systématiquement le type des entrées dans les arguments des fonctions (même si ce n'est pas nécessaire), et le type de sortie, à gauche d'une flèche. Ainsi, pour nous, la syntaxe d'une fonction en python est :

```
def ⟨nom⟩(⟨variable⟩:⟨type⟩,...) -> ⟨type⟩:
    """ ⟨description de la fonction⟩ """
    ⟨corps de la fonction⟩
```

On peut accéder à la description d'une fonction en tapant help(⟨nom⟩). Ainsi, pour obtenir la description de validation, on peut taper help(validation).

Il y a trois syntaxes différentes pour appeler une fonction en python :

— la notation *infixe*, où la fonction est placée au milieu, ainsi `3+5` ;
— la notation *fonctionnelle*, où la fonction est au début, et les arguments entre parenthèses, ainsi validation(12) ;
— la notation *méthode*, où la fonction est considérée comme agir sur un objet, qui est placé en premier, ainsi maliste.append(3).

1.8 Types

On va se limiter à quelques types :

le type des nombres entiers int. On peut les additionner, les multiplier, les comparer,...

le type des nombres à virgule float[8]. On peut aussi les multiplier, les comparer,... et se servir de nombres entiers comme s'ils étaient des nombres à virgule.

le type des chaînes de caractères str. On peut les afficher, les concaténer,...

le type des booléens bool, qui valent soit **True** (vrai), soit **False** (faux).

1.9 Structures de données

Pour finir notre tour d'horizon du langage python, on doit introduire les *structures de données*, qui permettent de stocker et de manipuler des informations de manière uniforme. Pour reprendre notre exemple sur la validation d'année par des étudiant·es, si on veut appliquer le même traitement à chaque étudiant·e, il faut avoir stocké chaque note de manière à ce qu'on puisse facilement savoir quel est le nom associé, et quelle est la prochaine note (par exemple).

8. Ainsi nommé car la virgule est *flottante* (elle peut être n'importe où dans le nombre) et pas *fixée* à une position.

Tuples

La structure de donnée la plus simple est celle de *tuple* (n-uplet en français plus mathématique, mais en programmation, la convention s'est perdue et anglicisée), qui contient un nombre fixe d'éléments dans un ordre fixé. Par exemple :

```
("Gandhi", "Djuna", 1986)
```

est un tuple dont le zéroième élément est la chaîne de caractère "Gandhi", le premier est la chaîne de caractère "Djuna" et le deuxième le nombre entier 1986.

> En python, on numérote à partir de 0 et pas de 1.
> On peut remarquer que c'est une convention beaucoup plus raisonnable que celle que l'on prend usuellement :
> — par exemple, quand on compte les années, il n'y a pas d'an 0, ni de siècle 0. Résultat, le xxi$^{\text{ème}}$ siècle commence en 2001, ce qui est désagréable. Pire, si on veut savoir combien il y a d'années entre (par exemple) 2022 et 1991, il suffit de calculer la différence $2022 - 1991 = 31$.
> Cela ne marche pas du tout dès qu'on traverse la date d'origine : il n'y a pas deux ans entre l'an -1 et l'an 1 ;
> — c'est par contre ce que l'on fait pour les bâtiments : il y a un étage -1, un rez-de-chaussée (nom compliqué pour un étage 0), puis un étage 1, et d'un coup tout marche.
> Bref, numérotons à partir de 0, c'est mieux (et les Mayas le faisaient déjà).

Le tuple ci-dessus est donc de type

```
tuple[str,str,int]
```

Étant donné un tuple, on peut récupérer un élément avec l'opérateur [] : Ainsi,

```
("Gandhi", "Djuna", 1986)[1]
```

```
Djuna
```

Listes

Dans de nombreuses situations, on a à stocker une suite de données toutes de même type, représentant des éléments différents. Par exemple, si on veut représenter un tas de cartes ou une liste d'étudiant·es dans une promotion : dans les deux cas, on peut abstraire les éléments en ne retenant que les informations pertinentes (le numéro et l'enseigne pour les cartes, les noms, prénoms et options pour les étudiant·es) et les représenter d'une manière permettant de récupérer ces informations facilement. On peut représenter chacune de ces informations par un élément d'un tuple, mais on veut ensuite pouvoir parler du fait que l'on ait plusieurs instances du même genre d'information. Pour cela, on peut utiliser des listes.

On peut définir des listes de plusieurs manières en python. On va d'abord s'intéresser à la façon explicite, en écrivant les éléments. Ainsi,

```
[]
```

est une liste vide, qui ne contient aucun élément, et

```
[8, 24, 2*2, 36]
["ours", "chat", "porc"]
[True, False]
```

sont des listes, respectivement d'entiers, de chaînes de caractères et de booléens.

On dispose :
— de la fonction len(l:List[T]) -> int qui calcule la longueur d'une liste ;
— de l'opérateur d'égalité == et d'inégalité != ;
— de l'opérateur de concaténation + ;
— de l'opérateur d'accès [].

On suppose qu'on a une liste de N éléments et que k est un entier plus petit que N ($0 \leqslant k < \mathrm{N}$). On peut vouloir :

1. accéder au k-ème élément de la liste ;
2. modifier le k-ème élément de la liste ;
3. ajouter un élément juste après le k-ème élément de la liste ;
4. ajouter un élément juste avant le k-ème élément de la liste ;
5. supprimer le k-ème élément de la liste ;
6. fusionner deux listes ;
7. découper une liste en deux ;
8. copier une liste ;
9. déterminer le nombre d'éléments d'une liste ;
10. trier la liste ;
11. chercher un élément dans la liste.

Exercice 2. Pour chaque opération, donner un exemple où il est naturel de vouloir la faire.

Les programmer en python.

Dictionnaires

1.10 Structure d'un fichier python, vérification et exécution

Devoir systématiquement retaper la séquence d'instructions est assez fastidieux (mais nos ancêtres l'ont fait autrefois quand le stockage était cher ou lent).

On préfère stocker cela dans des fichiers.

Licence et copyright

On commence systématiquement un fichier par quelques lignes précisant qui en est l'auteurice, la date, de quel projet le fichier fait partie, et la licence d'utilisation, en commentaire. Par exemple, mes fichiers peuvent commencer par :

```
# Algorithmique et programmation — M1 Droit et informatique
# Luc Pellissier 2022
# Sous licence EUPL 1.2
```

Importations

Un fichier python commence en général par des *importations* qui permettent d'utiliser des fonctions, structures de données, etc... définies par quelqu'un d'autre dans votre programme.

Une *librairie* contient de nombreuses déclarations, et python permet d'en importer seulement quelques unes.

```
from ⟨librairie⟩ import ⟨déclaration⟩
```

Dans notre cas, on importera systématiquement les annotations de type.

Variables

On va systématiquement déclarer les variables et leurs types, éventuellement en leur donnant une valeur :

```
⟨variable⟩ : ⟨type⟩
```

ou bien

```
⟨variable⟩ : ⟨type⟩ = ⟨valeur⟩
```

Commentaires

Un commentaire, en python est n'importe quelle fin de ligne contenant un #. On commentera systématique nos programmes.

L'objectif des commentaires est de rendre compréhensible la structure du programme à quelqu'un le lisant. Imaginez-vous en train de relire le programme dans six mois, et imaginez les questions que vous pouvez vous poser.

Assertions

Il est possible de rajouter des assertions permettant de tester des programmes. Par exemple, si j'ai écrit une fonction réalisant la multiplication, je peux vouloir écrire des tests assurant

```
assert ⟨expression⟩ ⟨opérateur⟩ ⟨expression⟩
```

où l'⟨opérateur⟩ peut être, par exemple, ==, !=,...

L'exécution du programme va s'arrêter si une assertion est fausse, et continuer silencieusement si elle est vraie.

Exécutions

On va systématiquement vérifier que les types sont corrects avec un deuxième programme, mypy, de la manière suivante :

```
1 mypy --strict fichier.py
```

Puis, on exécutera nos fichiers par :

```
1 python3 fichier.py
```

ou

```
python3 -i fichier.py
```

si on veut avoir l'interpréteur python après l'exécution du fichier (si par exemple le fichier ne contient que des définitions).

```
from typing import Tuple, List



# Ce message va être écrit à chaque personne
salutation : str = "Bonjour"

personnes : List[str] = ["Devaki", "Vasudeva", "Balarāma",
                         "K□ṣṇa", "Subhadrā", "Rukmini"]

# Les personnes sont la famille proche de K□ṣṇa.

i : int
j : int = 0


for i in range(0,6):
    print("i", salutation, personnes[i], "!", sep=" ")
    j = j + i

assert j == 15
```

2

Tris

On définit le problème du tri. Pour cela, on axiomatise la notion d'ordre, et on donne une définition des objets sur lesquels une fonction de tri est définie. On donne le modèle de coût avec lequel on évaluera les algorithmes. Enfin, on étudie plusieurs algorithmes de tri, en particulier leur correction et leur complexité.

Prendre des éléments et les placer selon un ordre est une opération courante. Sans doute parce qu'*ordonner* et *ordonnancer* avaient déjà des sens informatique, cette opération a pris le nom de *tri*, tout impropre qu'il soit. Commençons, pour formaliser le problème, par décrire la fonction sous-jacente à un tri.

2.1 Le problème du tri

Les tris peuvent s'appliquer à de nombreuses situations, pas forcément identiques. Considérons :

le tri de nombres entiers on a une liste de nombres, on veut les mettre dans l'ordre (disons croissant).

le tri de mots on a une liste de mots, on veut les mettre dans l'ordre. La question de l'ordre est déjà assez compliqué : l'ordre *lexicographique* (l'ordre du dictionnaire) est une invention récente — il ne s'est généralisé en Europe qu'après la publication du *Catholicon* de Jean de Gênes en 1286 —, et dépend de règles assez compliquées et changeant d'un pays à l'autre (ainsi, certains digraphes sont traités comme des lettres à part entière : ij en néerlendais, ß en allemand,...).

le tri de cartes on a une liste de cartes, et on veut les placer dans l'ordre. La question de savoir ce qu'est cet ordre est déjà complexe : on peut choisir de les classer d'abord par enseigne (cœur, pique, trèfle, carreau) puis par ordre croissant dans l'enseigne, ou au contraire, par ordre croissant, en ignorant l'enseigne. Dans le deuxième cas, on a aucun moyen de décider qui est plus grand du trois de carreau ou du trois de trèfle.

le tri de candidat·es on vient de procéder à une élection et chaque candidat·e a reçu un certain nombre de voix. On veut les trier dans l'ordre décroissant du nombre de voix reçues.

Inversement, il y a des situations où on sait qu'on n'arrivera pas à trier. C'est le cas du jeu de papier-caillou-ciseaux.

Relation d'ordre

On voit qu'on doit d'abord formaliser la notion d'ordre. Pour formaliser, on va utiliser la méthode axiomatique, c'est-à-dire qu'on va donner des propriétés que l'on veut voir vérifiées par tout ordre, et prendre cette liste de propriétés comme la définition de l'ordre. En particulier, on voudra que cette définition contienne tous les exemples donnés ci-dessus.

Exemple 1 (papier, caillou, ciseau)**.** Au jeu de papier-caillou-ciseaux, on a trois éléments : ✂️, 🪨, 🧻 ; tels que :

— les ciseaux battent le papier :

✂️ > 🧻 ;

— le papier bat le caillou :

🧻 > 🪨 ;

— le caillou bat les ciseaux :

🪨 > ✂️ ;

— aucun élément ne se bat lui-même.

Supposons que l'on a un ordre $\leqslant$, que l'on lit « inférieur ou égal ». Informellement, listons des propriétés que l'on peut demander de lui[1] — comme exercice, regardez quelles propriétés sont vérifiées par les exemples ci-dessus :

totalité étant donnés deux éléments x et y, on peut demander qu'une des deux assertions soient vérifiées :

$$x \leqslant y$$

ou

$$y \leqslant x.$$

C'est-à-dire qu'étant donnés deux éléments, ils soient toujours comparables : on puisse toujours dire de l'un des deux qu'il est plus grand que l'autre.

antisymétrie étant donnés deux éléments x et y, on peut demander que s'ils sont à la fois inférieur ou égaux l'un à l'autre, alors ils soient égaux.

calculabilité il peut être raisonnable de demander qu'il existe un programme qui, étant donnés deux éléments x et y, détermine si $x \leqslant y$, $y \leqslant x$ ou aucun des deux.

complexité constante il peut être raisonnable de demander qu'il existe un programme en temps constant, qui, étant donnés deux éléments x et y, détermine si $x \leqslant y$ — c'est-à-dire que cet algorithme ne dépend pas de la longueur de x ou de y.

reflexivité un élément x est toujours inférieur ou égal à lui-même :

$$x \leqslant x.$$

transitivité étant donnés trois éléments x, y et z, si x est inférieur ou égal à y et y inférieur ou égal à z, alors x est inférieur ou égal à y.

bornitude un élément est plus grand que tous les autres ; et de même, un élément est plus petit.

On se rend compte que le cas des cartes (où l'on ignore les enseignes) pose problème : la totalité n'y est pas vérifiée, de même que l'antisymétrie — sauf à considérer que toutes les cartes ayant le même numéro sont égales. Néanmoins, on voit bien que ces propriétés sont vraies sur les numéros des cartes. C'est aussi le cas pour les candidat·es dans une élection, d'ailleurs (deux candidat·es peuvent avoir le même nombre de voix, et donc ne pas pouvoir être départagé·es, mais les nombres de voix possibles sont toujours totalement ordonnés).

On voit qu'on doit donc distinguer, dans ce qu'on veut trier, une certaine propriété (le numéro des cartes, le nombre de voix, l'orthographe du nom) qui est celle qui va nous servir à ordonner. Dans ce cas, on peut donner la définition :

Définition 4. Soit E un ensemble. Un *ordre* $\leqslant$ sur E est une relation vérifiant les trois axiomes :

1. On préfère donner les propriétés avec $\leqslant$ qu'avec $<$ car elles sont plus simples à écrire.

transitivité si $x \leqslant y$ et $y \leqslant z$ alors $x \leqslant z$;

reflexivité $x \leqslant x$;

antisymétrie si $x \leqslant y$ et $y \leqslant x$ alors $x == y$.

On dit de plus qu'un ordre est *total* si

totalité $x \leqslant y$ ou $y \leqslant x$.

On dit qu'un ordre est *borné* si

bornitude il existe deux éléments $-\infty, +\infty$ tels que pour tout élément x, $-\infty \leqslant x \leqslant +\infty$.

Enfin, on dit qu'il est calculable s'il existe un programme qui, prenant comme entrées deux éléments de E, répond **True** si le premier est inférieur ou égal au deuxième et **False** sinon.

Exemple 2. L'ordre croissant sur les nombres entiers est total, calculable, mais n'est pas borné. La fonction `>(int,int) -> bool` en est une réalisation.

L'ordre sur les cartes — où l'on considère que toutes les cartes d'un numéro sont incomparables — est un ordre qui n'est pas total, mais borné.

La relation de papier, caillou, ciseau, n'est pas un ordre.

Enregistrements, clefs et tris

On va appeler *enregistrement* un élément que l'on voudra trier. Chaque enregistrement est associé à une *clef*, que l'on va prendre dans un ensemble ordonné, dont l'ordre est total et calculable. On peut exprimer le problème du tri.

Exemple 3. On peut considérer que des personnes sont des enregistrements. L'ordre lexicographique sur les noms de famille est total et calculable. Donc, on peut considérer que les noms de famille sont des clefs pour ces enregistrements.

De même pour les dates de naissance ; ou encore pour le nombre de voix reçues pour une élection.

Définition 5. Soit E un ensemble muni d'un ordre total calculable $\leqslant$. On note $\mathcal{F}$ l'ensemble des listes finies d'enregistrements à clefs dans E.

Une *fonction de tri* sur $\mathcal{F}$ est une fonction définie sur $\mathcal{F}$, à valeurs dans $\mathcal{F}$ telle que :

- pour chaque liste d'enregistrement, son image par la fonction de tri contienne les mêmes enregistrements que la liste initiale, éventuellement dans un autre ordre ;
- pour chaque liste d'enregistrement, son image par la fonction de tri a ses clefs ordonnées.

Exercice 3. On considère comme enregistrement les cartes à jouer, et comme clefs les valeurs numériques, ordonnées avec l'ordre croissant usuel.

Donner deux fonctions de tri différentes sur ces cartes.

Correction :

1. Une première fonction de tri trie non-seulement selon l'ordre des clefs, mais en plus, en plus, ordonne les cartes selon l'ordre ♠ < ♥ < ♦ < ♣. Ainsi, le 3 de cœur sera toujours placé avant le 3 de trèfle.
2. Une seconde laisse les cartes ayant la même clef dans l'ordre où elles étaient dans la liste initiale. Ainsi, si le 3 de trèfle apparaît avant le 3 de cœur, il sera dans cet ordre-là dans la liste triée.

☺

On va chercher à donner des programmes réalisant des fonctions de tri. De manière intéressante, les programmes ne dépendent pas des enregistrements, ni de l'ordre, mais seulement du fait que l'on peut ordonner.

Exercice 4. Écrire un programme de tri.

2.2 Éléments de complexité algorithmique

Supposons qu'on ait un programme dans un modèle de calcul donné : par exemple, un programme, écrit en python, compilé par un compilateur fixé dans un environnement logiciel lui aussi fixé et sur une machine fixée. On peut dans ce cas, mesurer (avec une horloge) le temps qu'il passe à s'exécuter ; on peut aussi mesurer la mémoire qu'il prend. On va pouvoir voir que deux programmes résolvant le même problème ne le font pas aussi rapidement l'un que l'autre, ou que, si l'un va plus vite, il consomme plus de mémoire en échange.

Cependant ces chiffres vont être très dépendant de détails peu pertinents (par exemple, le modèle précis de la mémoire utilisée), et en tout cas, ne permettront de ne parler que du programme dans le modèle de calcul, et pas de l'algorithme. On a néanmoins depuis le début de ce cours, écrit des algorithmes en pseudo-code, et donné un modèle d'exécution : on pourrait se dire qu'il suffit de compter les lignes dans une trace d'exécution pour obtenir une mesure de la complexité de l'algorithme. Ce n'est pas satisfaisant pour au moins deux raisons : déjà, ce qui constitue une ligne est assez arbitraire, et on peut prendre des conventions différentes pour arriver au résultat. Il est un peu spécieux de considérer qu'une ligne de redirection à la **fin** d'une boucle à la même complexité qu'une ligne faisant une comparaison ou un appel à une autre procédure... Aussi, on ne va pas compter un nombre d'étapes, mais un ordre de grandeur de ce nombre[2].

Pour donner un contenu à cette notion, commençons par remarquer que la longueur d'exécution d'un programme dépend de son entrée : il n'y a rien de choquant à ce que faire une opération sur des grandes données soit plus long que sur des petites, sans que cela dise quoi que ce soit de l'algorithme.

Autant, mesurer la complexité d'un programme semble facile : dans un modèle de calcul donné, il consomme des ressources (par exemple, du temps), et il suffit de mesurer.

Un algorithme étant un objet plus élusif, donner une définition correcte de complexité n'est pas vraiment possible — aussi on va toujours mesurer la complexité d'un algorithme écrit comme un programme python et en comptant

2. De la même manière qu'un algorithme est une idéalisation d'un programme, qui nous oblige à traiter des notions plus floues, le temps d'exécution n'a pas vraiment de sens pour un algorithme, et on parle d'une notion plus floue, son ordre de grandeur.

le nombre d'étapes dans le modèle des traces d'exécutions. On se doute bien que toutes les étapes n'ont pas le même poids : certaines effectuent un calcul ou une comparaison, d'autres ne font que sauter à une autre ligne… C'est pourquoi on ne compte jamais le nombre d'étapes, mais un ordre de grandeur de ce nombre. Plus précisément, on va se demander comment croît le nombre d'étapes quand les entrées croissent. Ainsi, ce qui va nous intéresser vraiment est de savoir si, quand on multiplie par deux la taille des entrées, le nombre d'étapes est :

— multiplié par deux ;
— augmenté d'un nombre constant d'étapes ;
— multiplié par quatre ;
— …

Donnons des exemples de ces différents cas. Pour déterminer qui est la meilleure équipe dans une compétition sportive (on suppose que ça a un sens…), on a plusieurs possibilités :

— soit, après le premier match, on considère que lea gagnant·e est champion·ne temporaire, et que chaque match qui vient après est une remise en jeu de son titre (c'est le cas de la coupe de l'America).

  Dans ce cas, multiplier par deux le nombre d'équipes participantes multiplie par deux le nombre de matchs à jouer, et donc le nombre de jours de compétition.

— soit on organise un tournoi : après chaque match, l'équipe qui a gagné est qualifiée pour continuer. À chaque tour du tournoi, il y a deux fois moins d'équipes en lice.

  Multiplier le nombre d'équipes par deux n'augmente que d'un le nombre de jours de compétition : il suffit de rajouter un tour.

— soit on organise une pool : chaque équipe affronte toutes les autres équipes, et on classe le nombre de victoires.

  Multiplier par deux le nombre d'équipes multiplie par quatre le nombre de matchs, et donc de jours de compétition.

— enfin, on peut imaginer une modalité qui n'existe à ma connaissance dans aucune compétition sportive. Dans un certain nombre de courses, certaines positions sont avantageuses (la *pole position* en Formule 1) ou au contraire, gênantes (l'*outsider* dans les courses hippiques) : de fait, certain·{es} ont moins à courir que d'autres, soit que leur placement de départ était devant, soit plus au centre de la boucle.

  Dans les vraies compétitions, on utilise le niveau supposé pour placer les plus rapides dans les meilleures positions[3] (par exemple en faisant une première course contre la montre, compétiteurice par compétiteurice). On peut imaginer qu'on fasse plutôt la course plusieurs fois, une fois pour chaque ordre possible.

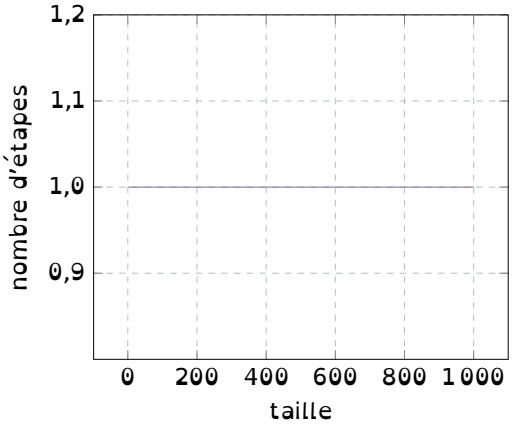
  Ainsi, s'il y a deux participant·es, on fera deux courses (une où le coureur 1 part dans la meilleure position, une où c'est la coureuse 2). S'il y en a trois, on en fera six. Et ainsi de suite.

  Rajouter un·e seul·e compétiteurice oblige à organiser beaucoup de courses en plus (on peut calculer que s'il en fallait k à n compétiteurices, il en faut $n \times k$ pour $n+1$).

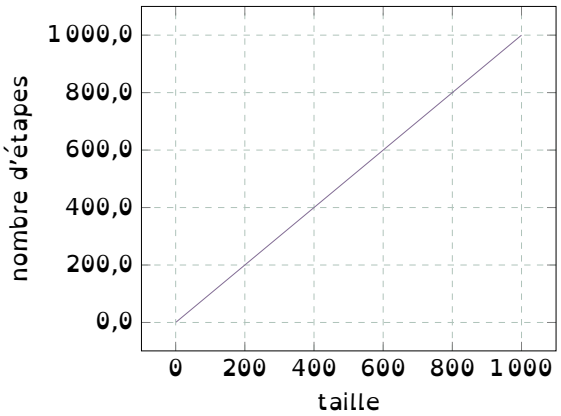
3. Dans certains cas, ça peut se justifier pour des raisons de sécurité.

Pour évoquer ces situations, on va reprendre les notations de Landau (Knuth 1976). On va noter systématiquement n la taille de l'entrée de l'algorithme (si l'algorithme a plusieurs entrées, on prend la somme des tailles). Ainsi, un algorithme opérant sur un tableau de k cases sera de taille k. On dit qu'un algorithme est de *complexité* :

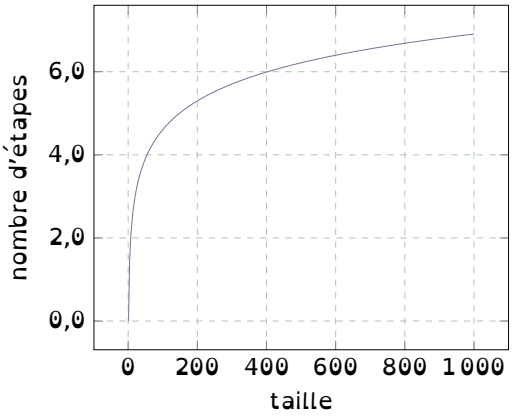
constante si le nombre d'étapes ne dépend pas de n. On le note $\mathrm{O}(1)$. Si on représente le temps d'exécution en fonction de la taille de l'entrée, on obtient une figure de la forme :



linéaire si le nombre d'étapes croît comme n croît : il augmente d'une constante quand la taille augmente d'une constante, et double quand la taille double. On le note $\mathrm{O}(n)$;

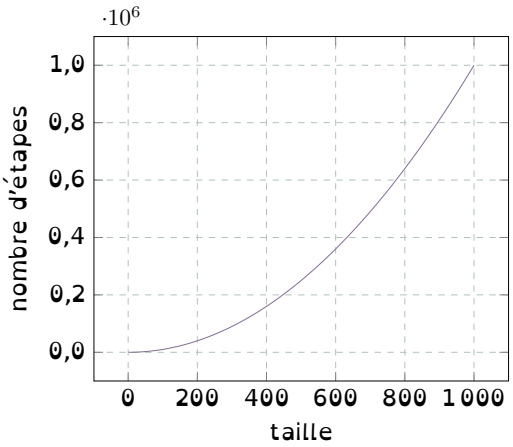


logarithmique si le nombre d'étapes croît logarithmiquement, c'est-à-dire augmente d'une constante quand n double. On le note $\mathrm{O}(\log n)$;

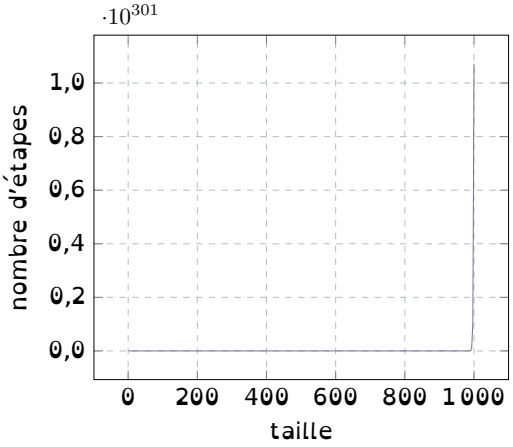


On voit ici que quand la taille double (quand elle passe de 200 à 400, ou de 400 à 800), le nombre d'étape augmente de la même quantité à chaque fois.

quadratique si le nombre d'étapes croît comme le carré de n, c'est-à-dire quadruple à chaque fois que le taille double. On le note $\mathrm{O}(n^2)$;



exponentielle si le nombre d'étapes double quand n croît d'une constante. On le note $\mathrm{O}(2^n)$.



complexité	1 000	40 000	40 millions	3 milliards
constante	1 s	1 s	1 s	1 s
logarithmique	1 s	5 s	16 s	22 s
linéaire	1 s	40 s	11 h	34 j
$\mathrm{O}(n\log(n))$	1 s	4 min	7 j	2 an
quadratique	1 s	26 min	51 an	285 193 an
exponentielle	1 s	34 842 an	🤯🤯🤯	☠️☠️☠️

Figure 2.1 – Durée d'exécution de programmes de complexité différente, selon la taille de l'entrée

Évidemment, on préfère systématiquement les complexités les plus faibles à celles les plus fortes. En particulier, on considère qu'un algorithme exponentiel n'est pas utilisable : s'il prend 1 seconde à tourner sur des données de taille 10, et 2 secondes sur des données de taille 30 (par exemple), alors il prendra 17 minutes sur des données de taille 200, 12 jours sur des données de taille 400 et 34 ans sur des données de taille 600.

Un problème lié est de trouver la complexité intrinsèque d'un problème, c'est-à-dire prouver qu'il n'existe pas d'algorithme résolvant un certain problème avec une complexité faible. Ici, on ne s'y intéressera pas, et travaillera toujours à algorithme donné.

Pour se fixer les idées, on peut considérer des programmes qui ont différentes complexités, et qui prennent tous une seconde pour résoudre une instance d'un problème de taille 1 000. Il y a de l'ordre de 40 000 étudiant·es à l'UPEC, de 40 millions de foyers fiscaux en France, et de 3 milliards d'inscrit·es sur les réseaux sociaux de Meta.

On voit très bien que certaines complexités ne sont pas raisonnables si l'on est l'UPEC, l'administration fiscale française, ou Meta, et que même des ordinateurs dix ou cent fois plus rapides ne changeraient rien au problème.

2.3 Tri par insertion

Une première idée peut être d'avancer progressivement : on va d'abord trier deux cases du tableau, puis trois, puis… Comme ça, une fois qu'on a déjà trié n cases, pour que les $n+1$ premières cases soient triées, il suffit de placer le $n+1$-ème au bon endroit. Pour faire cela, on va profiter du fait que les premières cases sont triées, et le faire reculer à sa position. Cela peut donner :

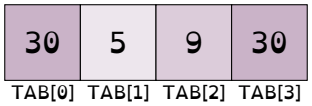
```python
# Tri par insertion — Cours d'Algorithmique et Programmation
# Luc Pellissier 2022
# CC0

from typing import List

# On ne va trier que des listes de nombres entiers. En tout état de cause, on ne
# peut pas trier des listes contenant n'importe quoi, car il faut une manière de
# trier
```

```
10
11 def triInsertion(elements:list[int]) -> list[int]:
12     """Réalise un tri par insertion de nombres entiers"""
13
14     N = len(elements)          # On calcule la longueur de la liste
15
16     for i in range(0,N):       # Pour chaque position de la liste
17         j = i
18
19         # On fait reculer l'élément à la position i jusqu'à ce qu'il soit à sa place
20
21         while j > 0 and elements[j-1] > elements[j]:
22             # Les trois lignes suivantes inversent elements[j-1] et elements[j]
23             # en utilisant une variable temporaire tmp
24             tmp = elements[j-1]
25             elements[j-1] = elements[j]
26             elements[j] = tmp
27
28             j = j-1
29
30     return elements
31
32 assert triInsertion([]) == []
33 assert triInsertion([9,30,5,30]) == [5,9,30,30]
```

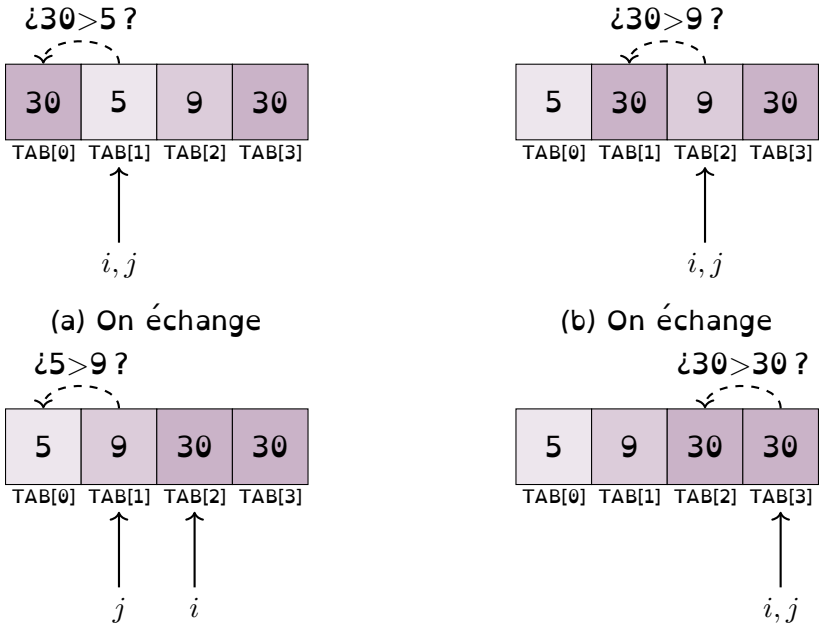
On peut rapidement l'exécuter sur un tableau d'exemple. On ne va pas le présenter comme un tableau d'exécution mais en présentant la danse des éléments du tableau, et en les coloriant selon leur grandeur, du plus clair pour le plus petit au plus foncé pour le plus grand, comme ceci :



On obtient l'exécution résumée Figure 2.2.

Complexité

Dans ce cas, la complexité en pire et en meilleur cas est très différente : en effet, on voit bien que la longueur de la boucle **tant que** est variable : dans le meilleur des cas, elle n'est pas exécutée du tout dans la i-ème boucle **pour** car la condition d'entrée dans la boucle (que la clef soit plus petite que sa voisine de gauche) n'est jamais vérifiée. Autrement dit, dans le meilleur des cas, si cette propriété est vraie pour chaque $1 \leqslant i < \mathrm{N}$, on entrera dans aucune boucle **tant que** et on aura donc une complexité linéaire. Cette propriété signifie en fait que le tableau est déjà trié : et en effet, avec un tableau déjà trié, on ne fait que vérifier qu'il l'est en le parcourant une seule fois[2]. Inversement, la boucle **tant que** peut être exécuté un nombre maximal de fois au i-ème passage dans la boucle **pour** si l'élément en i a une clef plus petite que toutes celles situées avant : en effet, dans ce cas, on devrait l'échanger avec chaque



(a) On échange (b) On échange

(c) On n'échange pas (d) On n'échange pas

Figure 2.2 – La danse des enregistrements

élément juste avant. Autrement dit, dans le cas où le tableau est exactement à l'envers, le temps d'exécution est maximal. Pour la i-ème boucle **pour**, on passe i fois dans la boucle **tant que**. On a déjà fait le calcul pour le tri par énumération : quand le tableau est trié dans l'ordre inverse de celui dans lequel on trie, on est en $\mathrm{O}(n^2)$, en complexité quadratique.

Ainsi, dans le pire cas on est quadratique, dans le meilleur linéaire. On ne vas pas essayer de calculer la complexité en moyenne, mais expérimentalement, on voit sur la Figure 2.4 que, en moyenne et dans le pire des cas, on est bien quadratique, tandis que dans le meilleur cas, on est linéaire.

Ainsi, cet algorithme est asymptotiquement aussi complexe en temps (dans le pire cas), mais meilleur en meilleur cas, car il est capable de détecter les séquences qui sont déjà dans l'ordre. De plus, il a aussi une bonne complexité en espace : contrairement au tri par énumération qui nécessitait de maintenir un tableau de taille linéaire de compteurs, ici, les seules variables sont les deux variables de boucle.

Résumé

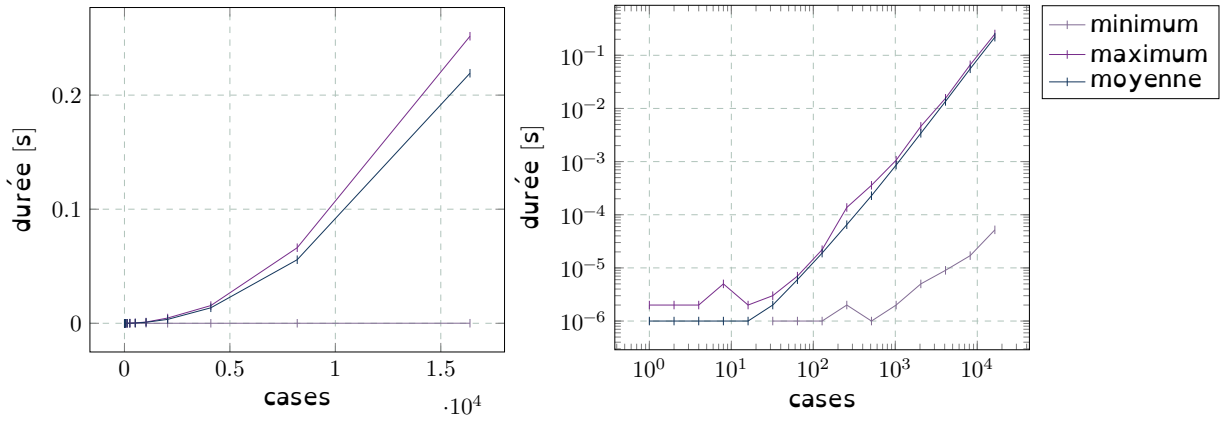
Le tri par insertion ordonne partiellement les éléments. Il ne nécessite que peu de mémoire, mais son temps d'exécution dans le pire cas est assez mauvais : dans le pire cas, il compare tous les éléments à tous les autres ; tandis que dans le meilleur, il profite de la transitivité pour faire moins de comparaisons.

2.4 Une application du tri : la recherche dichotomique

Avant de donner des algorithmes de tri, commençons par en voir une application, qui nous permettra d'introduire un paradigme algorithmique que nous réutiliserons

ligne	suivante	elements	N	i	j	tmp
11	14	[9, 30, 5, 30, 1]				
14	16	[9, 30, 5, 30, 1]	5			
16	17	[9, 30, 5, 30, 1]	5	0		
17	21	[9, 30, 5, 30, 1]	5	0	0	
21	16	[9, 30, 5, 30, 1]	5	0	0	
16	17	[9, 30, 5, 30, 1]	5	1		
17	21	[9, 30, 5, 30, 1]	5	1	1	
21	16	[9, 30, 5, 30, 1]	5	1	1	
16	17	[9, 30, 5, 30, 1]	5	2		
17	21	[9, 30, 5, 30, 1]	5	2	2	
21	24	[9, 30, 5, 30, 1]	5	2	2	
24	25	[9, 30, 5, 30, 1]	5	2	2	30
25	26	[9, 5, 5, 30, 1]	5	2	2	30
26	28	[9, 5, 30, 30, 1]	5	2	2	30
28	21	[9, 5, 30, 30, 1]	5	2	1	30
21	24	[9, 5, 30, 30, 1]	5	2	1	
24	25	[9, 5, 30, 30, 1]	5	2	1	9
25	26	[5, 5, 30, 30, 1]	5	2	1	9
26	28	[5, 9, 30, 30, 1]	5	2	1	9
28	21	[5, 9, 30, 30, 1]	5	2	0	9
21	16	[5, 9, 30, 30, 1]	5	2	0	9
16	17	[5, 9, 30, 30, 1]	5	3		
17	21	[5, 9, 30, 30, 1]	5	3	3	
21	16	[5, 9, 30, 30, 1]	5	3	3	
16	17	[5, 9, 30, 30, 1]	5	4		
17	21	[5, 9, 30, 30, 1]	5	4	4	
21	24	[5, 9, 30, 30, 1]	5	4	4	
24	25	[5, 9, 30, 30, 1]	5	4	4	30
25	26	[5, 9, 30, 1, 1]	5	4	4	30
26	28	[5, 9, 30, 1, 30]	5	4	4	30
28	21	[5, 9, 30, 1, 30]	5	4	3	30
21	24	[5, 9, 30, 1, 30]	5	4	3	
24	25	[5, 9, 30, 1, 30]	5	4	3	30
25	26	[5, 9, 1, 1, 30]	5	4	3	30
26	28	[5, 9, 1, 30, 30]	5	4	3	30
28	21	[5, 9, 1, 30, 30]	5	4	2	30
21	24	[5, 9, 1, 30, 30]	5	4	2	
24	25	[5, 9, 1, 30, 30]	5	4	2	9
25	26	[5, 1, 1, 30, 30]	5	4	2	9
26	28	[5, 1, 9, 30, 30]	5	4	2	9
28	21	[5, 1, 9, 30, 30]	5	4	1	9
21	24	[5, 1, 9, 30, 30]	5	4	1	
24	25	[5, 1, 9, 30, 30]	5	4	1	5
25	26	[1, 1, 9, 30, 30]	5	4	1	5
26	28	[1, 5, 9, 30, 30]	5	4	1	5
28	21	[1, 5, 9, 30, 30]	5	4	0	5
21	16	[1, 5, 9, 30, 30]	5	4	0	
16	31	[1, 5, 9, 30, 30]	5	5		
16	**retourner**	[1, 5, 9, 30, 30]				

Figure 2.3 – Trace d'exécution de triInsertion([9,30,5,30,1])



(a) Échelle linéaire (b) Échelle logarithmique

Figure 2.4 – Temps d'exécution du tri par insertion sur des tableaux de taille variable

plus tard. Supposons que nous ayons une liste d'enregistrements (présentée sous la forme de liste chaînée ou de tableaux) et que nous souhaitions trouver, dans cette liste, un enregistrement ayant une clef précise.

Si nous ne savons rien de la liste, le seul algorithme que nous pouvons appliquer va consister à parcourir toute la liste, et comparer la clef de chaque enregistrement avec la clef que l'on cherche. On va donc, dans le pire des cas, devoir comparer la clef que l'on cherche une fois avec la clef de chaque enregistrement, c'est-à-dire faire un nombre linéaire de comparaisons. On est en $\mathrm{O}(n)$.

Néanmoins, si la liste est présentée sous forme d'un tableau trié par ordre de clef croissante (ce qui suppose qu'il y ait un ordre sur les clefs...), on peut faire beaucoup mieux, et c'est d'ailleurs ce que l'on fait spontanément (si vous cherchez un mot dans un dictionnaire, vous ne commencez pas par regarder tous les mots de la première page, puis tous ceux de la deuxième,... mais vous essayez de sauter efficacement à des pages bien choisies) : en effet, si on constate que la clef qu'on cherche est supérieure à la clef de la case d'indice i, on n'a pas besoin de considérer les cases d'indice inférieur à i, elles ne contiendront que des éléments de clef inférieure à celle qu'on cherche ! On peut donc, en une seule comparaison, diviser par deux le champ des enregistrements à considérer : en comparant la clef à chercher avec celle de l'enregistrement au milieu du tableau.

On peut continuer ainsi : une fois qu'on a exclut une moitié du tableau, on peut en exclure un quart en comparant avec la clef située au milieu du champ des enregistrements encore possibles, et ainsi de suite. On appelle cet algorithme la *recherche dichotomique* car elle divise en deux (si on divisait en trois, ce serait trichotomique, et ainsi de suite). Cet algorithme s'écrit très naturellement sous forme récursive, ce qui donne le programme suivant.

```
1 # Recherche dichotomique — Cours d'Algorithmique et Programmation
2 # Luc Pellissier 2022
3 # CC0
4
```

```
5  from typing import List
6
7  def dichotomie(tab:list[int], clef:int) -> int:
8      """recherche la clef dans le tableau et renvoie la position"""
9
10     debut = 0
11     fin = len(tab)-1
12
13     # On va chercher clef entre debut et fin. Comme le tableau est trie, on va
14     # limiter la zone d'exploration après chaque comparaison en changeant debut
15     # ou fin.
16
17     while fin > debut: # Tant qu'il y a une case dans le fragment du tableau
18                        # considéré
19         milieu = (debut + fin)/2
20         if tab[milieu] == clef:
21             # Si on a trouvé la clef
22             return milieu
23         elif tab[milieu] < clef:
24             # Si la clef est plus grande que l'élément au milieu, on cherche à droite
25             debut = milieu
26         else:
27             # Sinon on cherche à gauche
28             fin = milieu
29
30     # On retourne -1 quand on n'a pas trouvé l'élément qui nous intéresse
31     return -1
32
```

Cet algorithme est beaucoup plus rapide que celui de recherche linéaire : en effet, supposons qu'on doive chercher dans un tableau ayant 1024 éléments. Dans le pire des cas, quand le tableau n'est pas trié, il faut effectuer 1024 comparaisons de clefs. Quand on peut appliquer la recherche dichotomique, voyons ce qui se passe : en une comparaison de clef, soit on a trouvé (car la clef qu'on cherchait était pile dans l'enregistrement du milieu), soit on sait qu'on ne doit plus chercher que dans une des deux moitiés du tableau ; autrement dit, dans le pire des cas, on n'a plus qu'à chercher parmi 512 enregistrements. Avec une deuxième comparaison, on n'a plus qu'à chercher — toujours dans le pire des cas — parmi 256 éléments. Avec trois comparaisons on limite à 128, avec quatre à 64, avec cinq à 32, six à 16, sept à 8, huit à 4, neuf à 2, et en dix comparaisons au pire, on a trouvé notre clef ou trouvé que le tableau ne la contenait pas.

On est donc passé de 1024 comparaisons à seulement 10, et ce fossé ne fait qu'accroître quand le tableau grandit : en effet, si on double la taille du tableau, il suffit d'une comparaison supplémentaire en plus à la recherche dichotomique : c'est un algorithme de complexité logarithmique dans le pire cas : en $\mathrm{O}(\log(n))$ [4].

Nous pouvons retirer deux enseignements de cet algorithme :

4. On peut dire — sans abus de langage ! — que cet algorithme est exponentiellement meilleur que celui pour un tableau non-trié.

— le premier est qu'il justifie de s'intéresser aux tris : chercher un élément vérifiant une certaine propriété est une opération tellement basique qu'on peut difficilement imaginer s'en passer dans n'importe quel programme (par exemple, il faut au minimum être capable pour un ordinateur de trouver la prochaine opération à exécuter, la bonne information à afficher,...), donc l'accélérer même un peu vaut le coût : si on trouve des algorithmes de tri un minimum efficaces, on peut imaginer trier une liste dont on se servira souvent avant de chercher ; si on trouve des algorithmes de tri très efficaces, on pourra même trier systématiquement, même une liste dans laquelle on ne cherche qu'une fois. On voit ici aussi que la manière dont les données sont présentées influe énormément sur les algorithmes possibles : la manière dont les choses sont présentées conditionne ce que l'on peut faire avec.
— le second a trait à l'algorithme lui-même. Plutôt que de traiter tout le tableau, il commence par le *diviser* en deux demi-tableaux, choisir quel demi-tableau traiter, et ne s'occuper que de celui-là. C'est une instance d'un paradigme algorithmique très puissant, que l'on nomme *diviser pour régner*[5] : en règle générale, on divise un problème en sous-problèmes, traite chacun des sous-problèmes séparément, et réunit les sous-problèmes ainsi produits. Elle marche très bien dans certains cas : par exemple, pour trouver le maximum d'un tableau, on peut le diviser en deux, chercher le maximum dans chacune des moitiés, puis comparer les deux maximums locaux, mais pas tout le temps, soit parce que le problème ne se divise pas en sous-problèmes (on ne peut pas, par exemple, construire un emploi du temps optimal en prenant ensemble l'emploi du temps optimal obtenu en considérant séparément la moitié des groupes avec la moitié des salles : avoir une vision globale du problème est indispensable), soit parce que réunir les solutions n'est pas plus simple que de résoudre le problème entier.

  En règle générale, cette stratégie suppose qu'on puisse raisonner localement (sur des sous-problèmes) et que les solutions locales s'assemblent en une solution locale.

On voit donc que la recherche dichotomique éclaire deux thèmes très importants :
— ce que l'on peut supposer de la présentation des données, qui peut rendre un problème infaisable ou au contraire très simple ;
— la dialectique entre le local et le global, qui est en règle générale, un leitmotiv en mathématiques.

Exercice 5. Considérons un autre problème et voyons si on peut appliquer le paradigme diviser pour régner. Supposons qu'on a un tableau contenant des éléments quelconques : la seule chose qu'on sait faire sur ces éléments est de tester s'ils sont égaux. On dit qu'un élément est *majoritaire* si plus de la moitié des éléments du tableau sont égaux à cet élément (par exemple, un·e candidat·e est élu·e au premier tour de la plupart des élections françaises exactement quand iel est majoritaire à ce sens-là parmi les votes exprimés).

Pour simplifier, on supposera que le tableau a pour longueur N, une puissance de 2 (donc on pourra toujours le diviser en deux parties égales sans réfléchir).

5. Par analogie avec la vieille stratégie politique.

1. Donner un algorithme calculant, pour un élément donné, son nombre d'occurrences dans le tableau.
   En déduire un algorithme permettant de déterminer si le tableau possède un élément majoritaire. Quelle est sa complexité ?
2. Appliquer le paradigme diviser pour régner à ce problème, en déduire un algorithme. Quelle est sa complexité ?
3. On va essayer d'améliorer cet algorithme. Dans un premier temps, on va donner un algorithme plus faible, mais plus rapide, et qui va satisfaire la propriété suivante :
   — soit il détecte que le tableau ne possède pas d'élément majoritaire
   — soit (c'est-à-dire dans le cas où la tableau a un élément majoritaire, et dans le cas où le tableau n'en n'a pas mais il ne l'a pas détecté) l'algorithme fournit un élement e et un entier p, supérieur à la moitié de la longueur du tableau ($p > \mathrm{N}/2$) tels que e apparaît au plus p fois dans le tableau et tout autre élément que e apparaît au plus $\mathrm{N}-p$ fois.

   Justifier qu'un tel algorithme permet de résoudre le problème de l'élément majoritaire rapidement. Donner un tel algorithme, récursif. Quelle est sa complexité ? Quelle est la complexité de l'algorithme décidant si le tableau possède un élément majoritaire ainsi obtenu.
4. Changeons complètement de point de vue. On va supposer que notre tableau est en fait une étagère, qui contient des balles colorées posées en file.
   On cherche à savoir s'il y a une couleur majoritaire parmi les balles. Supposons que les balles sont rangées de manières à ce qu'il n'y a jamais deux balles de même couleur à côté. Que sait-on sur le nombre maximal de balles de même couleur ?
5. Considérons l'algorithme suivant (décrit informellement) : on a un ensemble de balles, une étagère vide où l'on peut les ranger en file et une corbeille vide où l'on peut poser des balles et les récupérer (mais sans ordre).

   dans un premier temps on prend les balles une par une. Si la balle qu'on a en main n'est pas de la même couleur[6] que la dernière balle sur l'étagère, on la pose sur l'étagère après la dernière balle, puis on pose une balle de la corbeille (s'il y en a une) sur l'étagère après celle qu'on vient de poser. Sinon (c'est-à-dire si la balle est de même couleur que la dernière balle sur l'étagère), on met la balle qu'on a en main dans la corbeille.

   dans un second temps toutes les balles sont soit dans la corbeille, soit rangées sur l'étagère. Appelons c la couleur de la dernière balle sur l'étagère. On va retirer successivement des balles à l'étagère. Tant qu'il y a une balle sur l'étagère, on prend la dernière. Si elle est de couleur c[7], alors on jette les deux dernières balles posées sur l'étagère — sauf s'il n'en reste qu'une, auquel cas, on la met dans la corbeille. Sinon, on la jette ; dans ce cas, s'il y a une balle dans la corbeille,

6. C'est en particulier le cas s'il n'y a pas de balle sur l'étagère.
7. Cela va être le cas en particulier pour la première balle qu'on regarde.

on la jette aussi, et s'il n'y en a plus, on s'arrête en disant qu'il n'y a pas de couleur majoritaire.
Une fois l'étagère vide, on regarde la contenu de la corbeille. Si elle est vide, il n'y a pas de couleur majoritaire ; si elle contient au moins une balle alors c est la couleur majoritaire.

Exécutez cet algorithme, en supposant qu'on prend dans l'ordre des balles de couleur rouge, bleu, bleu, rouge, vert, bleu, rouge, rouge, vert, rouge, rouge, bleu. Montrer qu'à tout moment de la première phase, toutes les balles présentes dans la corbeille ont la couleur de la dernière balle sur l'étagère.
En déduire que l'algorithme est correct. Donner sa complexité en pire cas (en fonction du nombre de comparaison de couleurs).

2.5 Fusion

Le *tri fusion*[8] se base sur une idée extrêmement simple, tellement simple qu'on voit mal comment elle peut être efficace. Rappelons les conditions pour que la stratégie diviser pour régner soit applicable à un problème :

— que le problème se divise en sous-problèmes ;
— que réunir la solution de deux sous-problèmes soit simple.

C'est le cas dans le cas du tri : on peut sous-diviser le problème en triant les deux moitiés du tableau. Quant à réunir les solutions aux deux sous-problèmes, il s'agit en fait de prendre deux tableaux triés et calculer un tableau contenant les mêmes éléments, trié lui aussi. Écrivons un algorithme effectuant cela, la fonction fusion.

```
1  # Fusion de deux listes triées — Cours d'Algorithmique et Programmation
2  # Luc Pellissier 2022
3  # CC0
4
5  from typing import List
6
7  def fusion(liste1:list[int], liste2:list[int]) -> list[int]:
8      """si les deux listes liste1 et liste2 sont triées, fusionne ces deux listes
9      en une liste tirée"""
10
11     i1 = 0
12     i2 = 0
13
14     # i1 et i2 vont pointer vers des éléments respectivement de liste1 et de
15     # liste2
16
17     res = []
18
```

8. Pour l'anecdote, il s'agit sans doute du premier tri ayant été étudié en tant que tel : en 1945 margittai Neumann János Lajos, (plus connu sous son nom germano-anglicisé de John von Neumann) l'a programmé pour tester les performances de l'edvac, un des tous premiers ordinateurs électroniques programmables.

```
19      while i1 < len(liste1) and i2 < len(liste2):
20          # tant que liste1[i1] et liste2[i2] ont un sens
21
22          if liste1[i1] < liste2[i2]:
23              res = res + [liste1[i1]]
24              i1 = i1 + 1
25          else:
26              res = res + [liste2[i2]]
27              i2 = i2 + 1
28
29      while i1 < len(liste1):
30          # tant que liste1[i1] a un sens (mais liste2[i2] n'en n'a plus)
31
32          res = res + [liste1[i1]]
33          i1 = i1 + 1
34
35      while i2 < len(liste2):
36          # tant que liste2[i2] a un sens (mais liste1[i1] n'en n'a plus)
37
38          res = res + [liste2[i2]]
39          i2 = i2 + 1
40
41      return res
42
43
```

Tant qu'il y a encore des éléments à placer dans les deux tableaux, on compare les deux éléments qu'on n'a pas encore placé, et on place le plus petit d'entre eux. Si on a vidé un des deux tableaux, on vide complètement l'autre.

Exercice 6. Éxécuter la fusion sur les deux tableaux

5	9	30	30
[0]	[1]	[2]	[3]

3	10	11	12	30
[0]	[1]	[2]	[3]	[4]

Correction : C'est représenté Figure 2.5. ☺

Exercice 7. Réécrire le programme de fusion de deux tableaux avec une seule boucle **tant que**.

Cet algorithme de fusion s'exécute en temps linéaire $\mathrm{O}(\mathrm{N}+\mathrm{M})$, où N est la longueur du premier tableau, et M celle du second : en effet, après chaque comparaison, on place un élément dans sa position définitive et on ne le compare plus jamais. Comme il n'y a que $\mathrm{N}+\mathrm{M}$ éléments, on ne fait qu'autant de comparaisons.

Ainsi, on peut prendre un tableau, le découper en deux, les trier séparément, et les fusionner avec l'algorithme ci-dessus. Prenons un tableau de longueur $2\mathrm{N}$. On avait calculé que trier un tel tableau par un tri par insertion nécessitait au pire

$$\frac{2\mathrm{N}(2\mathrm{N}-1)}{2}$$

ligne	suivante	i1	i2	res
7	11			
11	12	0		
12	17	0	0	
17	19	0	0	[]
19	22	0	0	[]
22	26	0	0	[]
26	27	0	0	[3]
24	19	1	0	[3]
19	22	1	0	[3]
22	26	1	0	[3]
26	27	1	0	[3, 5]
24	19	2	0	[3, 5]
19	22	2	0	[3, 5]
22	26	2	0	[3, 5]
26	27	2	0	[3, 5, 9]
24	19	3	0	[3, 5, 9]
19	22	3	0	[3, 5, 9]
22	26	3	0	[3, 5, 9]
26	27	3	0	[3, 5, 9, 10]
24	19	4	0	[3, 5, 9, 10]
19	29	4	0	[3, 5, 9, 10]
29	35	4	0	[3, 5, 9, 10]
35	38	4	0	[3, 5, 9, 10]
38	39	4	0	[3, 5, 9, 10, 11]
39	35	4	1	[3, 5, 9, 10, 11]
35	38	4	1	[3, 5, 9, 10, 11]
38	39	4	1	[3, 5, 9, 10, 11, 12]
39	35	4	2	[3, 5, 9, 10, 11, 12]
35	38	4	2	[3, 5, 9, 10, 11, 12]
38	39	4	2	[3, 5, 9, 10, 11, 12, 30]
39	35	4	3	[3, 5, 9, 10, 11, 12, 30]
35	38	4	3	[3, 5, 9, 10, 11, 12, 30]
38	39	4	3	[3, 5, 9, 10, 11, 12, 30, 30]
39	35	4	4	[3, 5, 9, 10, 11, 12, 30, 30]
35	38	4	4	[3, 5, 9, 10, 11, 12, 30, 30]
38	39	4	4	[3, 5, 9, 10, 11, 12, 30, 30, 30]
39	35	4	5	[3, 5, 9, 10, 11, 12, 30, 30, 30]
35	41	4	5	[3, 5, 9, 10, 11, 12, 30, 30, 30]
41	**retourner**	[3, 3, 3, 3, 3, 10, 11, 12, 30]		

Figure 2.5 – Trace d'exécution de fusion([5,9,30,30],[3,10,11,12,30])

comparaisons, c'est-à-dire $N(2N-1)$ comparaisons. Trier les deux demi-tableaux ne nécessite que $\frac{N(N-1)}{2}$ chacun, ce qui fait $N(N-1)$ comparaisons en tout, et les fusionner en nécessite aussi $2N$.

Donc, découper le tableau en deux, trier les deux demi-tableaux et les fusionner ne nécessite que $2N+N(N-1) = N(N+1)$ comparaisons, ce qui est à peu près deux fois moins que les $N(2N-1)$ nécessaire en triant tout d'un coup. La stratégie de diviser pour régner est efficace.

On peut aller plus loin et itérer cette stratégie : en effet, plutôt que de trier ces demi-tableaux par un algorithme comme le tri par énumération, on peut à nouveau diviser pour régner, et les re-diviser en deux et ainsi de suite, jusqu'à arriver à des tableaux déjà triés : des tableaux de longeur 1. Le tri fusion est donc un algorithme récursif qui trie un tableau en :

— s'il est de longueur 1, le renvoyant tel quel ;
— si non, le divise en deux, applique le tri fusion sur chaque moitié, et fusionne les deux demi-tableaux triés ainsi obtenus.

On peut se demander où le tri a lieu, vu qu'on ne trie pour ainsi dire jamais : en réalité, c'est dans la phase de fusion, qui trie peu à peu des tableaux de plus en plus grands.

On peut représenter cet algorithme comme en Figure 2.6 : on représente les clefs des tableaux par des teintes de couleur. La partie inférieure représente les différentes phases de fusion, chaque losange représente un l'exécution d'une procédure récursive.

Cela nous donne l'Algorithme du programme triFusion.

Remarque 6. La division entière en python se note `//`, ce qui fait que `3/2==1`. On remarque que dans ce cas, `N//2` et `N-N//2` ne sont pas toujours égaux.

```
# Tri Fusion — Cours d'Algorithmique et Programmation
# Luc Pellissier 2022
# CC0


def triFusion(liste:list[int]) -> list[int]:
    """tri fusion de la liste liste. On copie à chaque fois, aussi bien en
    séparant la liste qu'en la refusionnant"""

    # On cree deux demi-listes et on les remplit

    demiliste1 = []
    demiliste2 = []

    milieu = len(liste)//2

    if milieu == 0:
        # si la liste ne contenait qu'un element
        return liste

    # sinon, on copie deux demi-listes

```

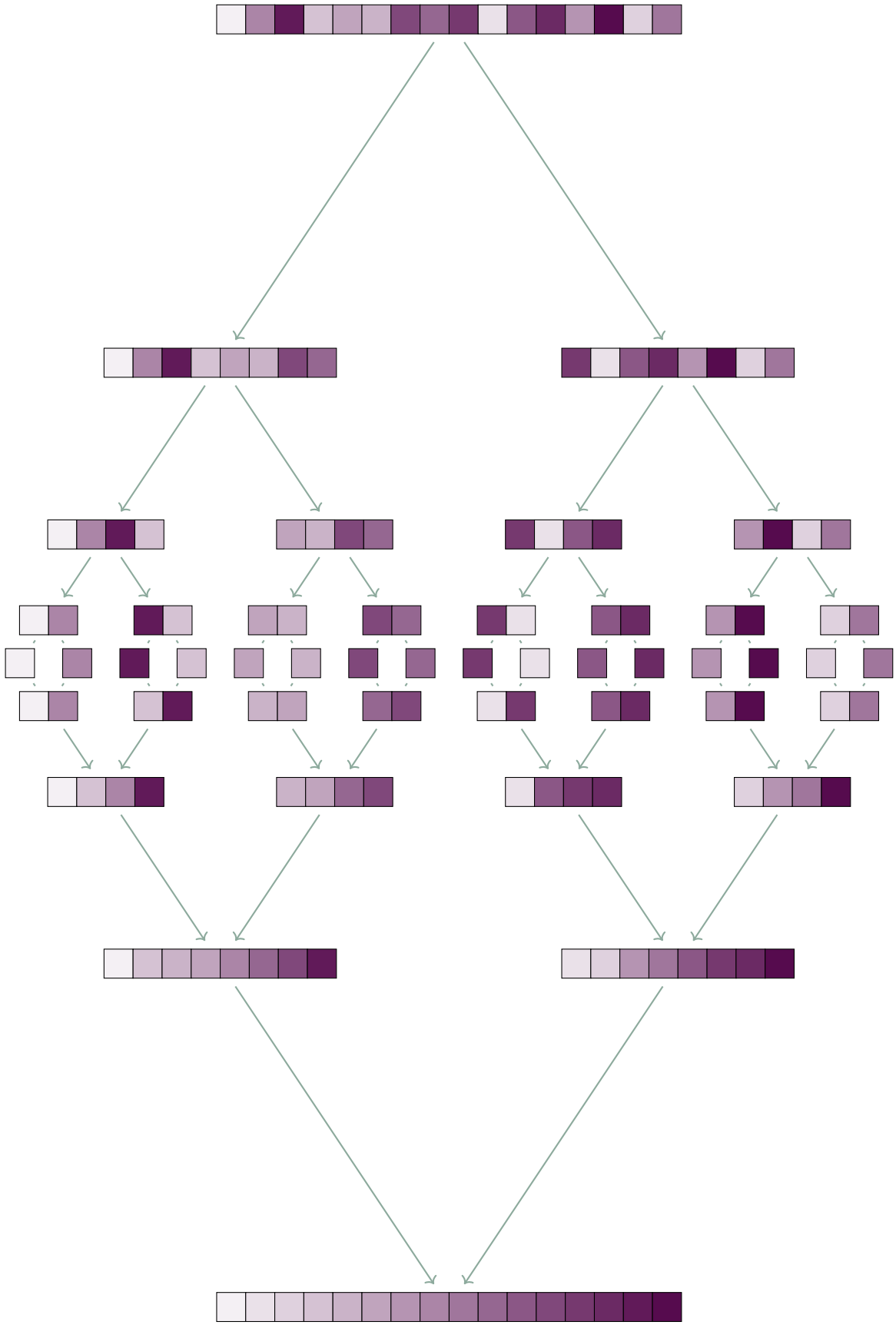
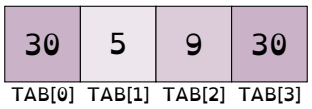


Figure 2.6 – Le tri fusion

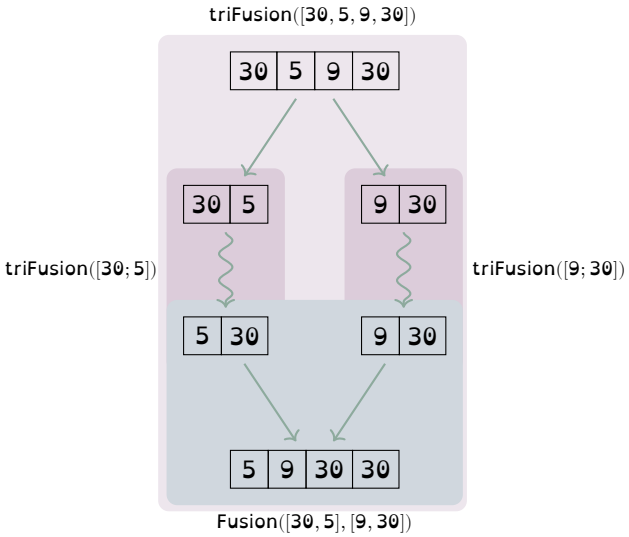
```
23      for i in range(0,milieu):
24          demiliste1 = demiliste1 + [liste[i]]
25
26      for i in range(milieu,len(liste)):
27          demiliste2 = demiliste2 + [liste[i]]
28
29      # on les trie
30
31      demiliste1 = triFusion(demiliste1)
32      demiliste2 = triFusion(demiliste2)
33
34      # on les fusionne
35
36      return fusion(demiliste1,demiliste2)
```

Tel qu'écrit, cet algorithme peut fonctionner sur des tableaux de longueur quelconque. Néanmoins, on va se limiter dans tous les exemples à des tableaux dont les longueurs sont des puissances de 2 — et donc que l'on peut toujours diviser par 2. D'une certaine manière, gérer les cas qui ne sont pas des puissances de 2 est un problème de programmation et pas d'algorithmique.

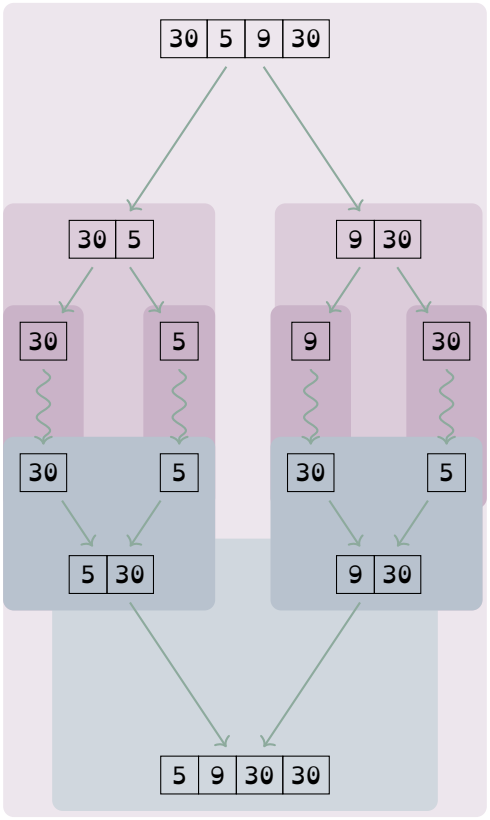
Exercice 8. Éxecuter cet algorithme sur le tableau



Correction : Plutôt que de faire un tableau, on va représenter les différentes procédures exécutées. Autrement dit, on veut exécuter triFusion([30; 5; 9; 30].



De même, l'exécution de chacun des deux triFusion entraîne des appels récursifs et un appel à fusion. Autrement dit, le schéma pertinent est plutôt celui-ci



où les différents appels à des procédures s'emboîtent les uns dans les autres comme des poupées russes. En réalité les appels ne se font pas en parallèle mais les uns après les autres, de gauche à droite (car la flèche de gauche correspond à la ligne 10 de l'algorithme, et celle de droite à la ligne 14) et d'abord en profondeur (c'est-à-dire qu'on fait tout une branche avant de passer à celle d'après). Si on déplie complètement, on retombe sur le genre de tableaux d'exécutions qu'on faisait. ☺

Correction

On veut se convaincre de la correction du tri fusion. Pour cela, on ne va pas pouvoir utiliser la méthode des invariants de boucle : en effet, le programme triFusion n'utilise pas de boucles. Néanmoins, on peut faire le raisonnement suivant : cet algorithme est correct (c'est-à-dire renvoie un tableau trié contenant les mêmes enregistrements que le tableau d'entrée) si le tableau d'entrée est de longueur 1 : il renvoie le tableau d'entrée, qui est trié car ne contenant qu'un élément. Par ailleurs, si on prouve que l'algorithme est correct sur des tableaux d'une certaine longueur N, alors, pour peu que l'algorithme Fusion soit aussi correct (c'est-à-dire qu'il renvoie un tableau trié contenant les mêmes enregistrements que ses deux tableaux en entrée), l'algorithme est correct sur deux tableaux de longueur $2\mathrm{N}$ (en effet, il ne fait que diviser le tableau de longueur $2\mathrm{N}$ en deux, s'appliquer sur les deux moitiés — et on a supposé qu'il était correct sur des tableaux de longueur N — et appliquer fusion -- dont on suppose aussi qu'il est correct — à ces tableaux). Autrement dit, si fusion est correct, alors le tri fusion l'est.

L'algorithme de fusion, lui, est construit autour de trois boucles, on peut donc en trouver des invariants de boucle. En réalité, ces trois boucles ont le

même invariant de boucle : toutes remplissent le tableau de sortie avec les enregistrements dans les deux tableaux d'entrée de manière à ce que le tableau de sortie soit trié[9]. L'invariant de boucle est donc qu'à chaque tour de boucle, les cases de 0 à $i+j$ du tableau res contiennent les enregistrements contenus dans les cases de 0 à i du tableau demiliste1 et ceux contenus dans les cases de 0 à j du tableau demiliste2, et, de plus, les cases de 0 à $(i+j$ du tableau res sont triées. On le vérifie.

Complexité

L'algorithme tel qu'on l'a écrit est assez mauvais en terme d'affectations. On se reportera au prochain paragraphe pour une analyse. On ne va donc que compter les comparaisons. Pour cela, considérons la Figure 2.6. On y a représenté une exécution sur un tableau de taille 16. Dans la partie supérieure, on ne fait que décomposer, aucune comparaison n'est faite. Les comparaisons n'interviennent que pour recombiner ; de plus, à chaque ligne, on recombine exactement autant d'éléments qu'il y en avait dans le tableau. Autrement dit, s'il y a N éléments dans le tableau initial, chaque ligne nécessite N comparaisons. Il y a un nombre logarithmique de lignes : si on multiplie par deux le nombre d'éléments dans le tableau, on va rajouter une seule ligne au début pour le couper en deux. Donc, le nombre de comparaisons est égal au nombre de lignes multiplié par le nombre de comparaisons de chaque ligne, c'est-à-dire $\mathrm{O}(\mathrm{N}\log(\mathrm{N}))$.

On a donc une complexité qui n'est pas tout à fait linéaire, mais qui est tout de même une nette amélioration par rapport à la complexité quadratique des algorithmes précédents : la fonction logarithme croit très lentement. On peut le voir sur la Figure 2.7, où la croissance semble linéaire car on ne considère que des petits tableaux (de taille plus petite que 10 000) : en effet, le nombre de lignes nécessaires pour un tableau de taille 10 est 4, de taille 100 est 7, de taille 1 000 est 10, et enfin de taille 10 000, 14.

On vérifie qu'empiriquement, le tri a bien la complexité annoncée.

Exercice 9. Écrire un algorithme de tri fusion ne créant pas deux nouveaux tableaux à chaque appel récursif, mais délimitant les tableaux par des positions.

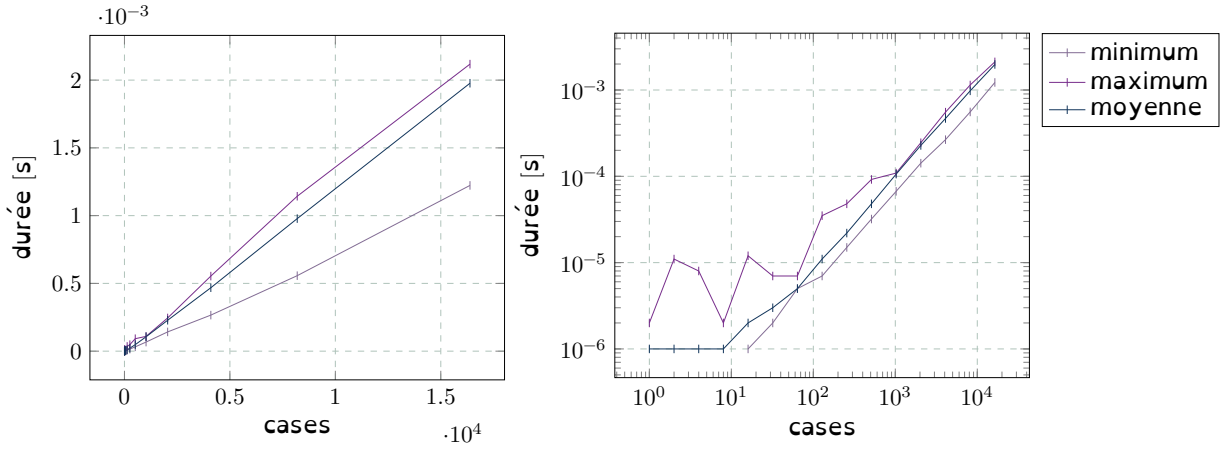
Résumé

Le tri fusion a une complexité qui dépend juste de la taille de l'entrée, en $\mathrm{O}(n\log(n))$. C'est un algorithme récursif, et un exemple de la stratégie diviser-pour-régner.

2.6 Borne inférieure de complexité des tris par comparaison

On a donc trouvé un tri de complexité $\mathrm{O}(n\log(n))$, c'est-à-dire bien meilleure que les méthodes que nous avons trouvé de prime abord, qui, au fond, nécessitaient de comparer deux à deux tous les éléments. On peut dès lors se poser deux questions :

9. D'une certaine manière, c'est pour la même raison que l'Exercice 7 est possible : ces trois boucles n'en sont secrètement qu'une.



(a) Échelle linéaire (b) Échelle logarithmique

Figure 2.7 – Temps d'exécution du tri fusion sur des tableaux de taille variable

— peut-on faire mieux ? On se doute bien qu'il y a une borne inférieure à la complexité des tris : en particulier, on ne pourra trier n éléments avec moins de n comparaisons, vu qu'on aura aucun moyen pour ne serait-ce que vérifier que les données sont déjà dans l'ordre. On peut donc peut-être trouver une borne qui serait intrinsèque à la nature du problème et qui limiterait tous les algorithmes, existant ou encore à découvrir, le résolvant.
— on a vu que le tri par insertion se comportait différemment selon son entrée, et si, dans le pire des cas, il était de complexité quadratique, dans le meilleur, il est linéaire. Ne peut-on pas imaginer des tris qui soit de la même complexité que le tri fusion dans le pire des cas, mais encore meilleurs dans certains cas particuliers ? Voire même, si on a trouvé une borne inférieure à la complexité du problème du tri, ne peut-on pas trouver un algorithme qui dans certains cas particuliers soit meilleur que cette borne ?

Lemme 1. *Il y a $n!$ tableaux différents composés de n enregistrements fixés.*

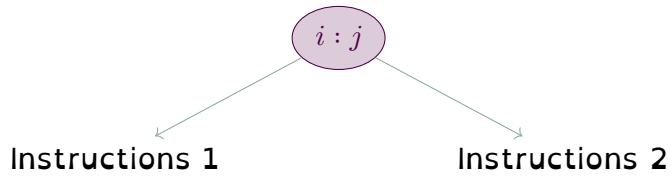
Démonstration. En effet, on a n choix pour le premier élément, seulement $n-1$ pour le deuxième, et ainsi de suite. ☺

Théorème 1. *Un algorithme (séquentiel, déterministe) qui résoud le problème du tri ne peut le résoudre en moins de $\mathrm{O}(n\log(n))$ dans le pire des cas.*

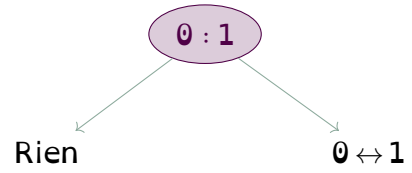
Démonstration. Considérons n'importe quel algorithme de tri. On peut le ré-écrire de manière à ce qu'il fasse d'abord toutes les comparaisons, note dans une structure de données auxiliaires les déplacements à faire, puis, à la fin, fasse toutes les insertions. Autrement, dit, on peut le ré-écrire d'une manière qui d'abord ne fasse que des comparaisons, écrive des instructions qui échangent le contenu de deux cases, et enfin exécute toutes les instructions.

Supposons qu'un tel algorithme ne fasse aucune comparaison : alors les instructions qui doivent être exécutées seront les mêmes pour tous les tableaux,

quelque soit sa taille ou son contenu. De même, si une comparaison est faite, alors on a deux séries d'instructions possibles, *selon le contenu du tableau* : en effet, selon le résultat de la comparaison, on peut appliquer des instructions différentes. Si la comparaison est vraie, alors on exécute les instructions 1, sinon, les instructions 2.

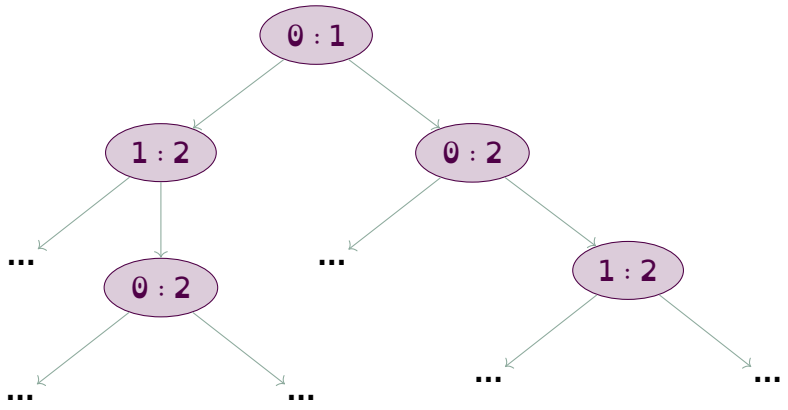


Et ainsi de suite. Par exemple, pour trier un tableau à deux éléments, tous les algorithmes vont comparer l'élément en position 0 et celui en position 1 et, si celui en position 0 est plus grand les inverser, sinon, ne rien faire, c'est-à-dire :



En règle générale, on peut dessiner un arbre représentant les comparaisons effectivement faites (qui peuvent être différentes dans chaque branche), avec, aux feuilles, les listes d'instructions devant être faites si le résultat des comparaisons est bien celui le long de la branche.

Tout algorithme de tri sur une taille donnée (basé sur des comparaisons...) peut être représenté ainsi : par l'arbre de ses choix de comparaisons, et la liste des instructions afférentes aux feuilles. Par exemple, le tri par insertion sur les tableaux de taille 3 peut être représenté par :



(**exercice** : donner explicitement toutes les instructions à chaque feuille)

Dans cette représentation, toutes les instructions sont effectuées aux feuilles. Autrement dit, si deux tableaux suivent la même branche, les mêmes instructions seront appliquées, et une fois appliquée, le tableau sera trié. On voit donc que si on fixe n enregistrements, un seul tableau peut suivre une branche donnée. Donc, on sait que l'arbre correspondant à toutes les exécutions possibles d'un

algorithme de tri sur des tableaux de n enregistrements fixés doit avoir au moins autant de feuilles qu'il y a de tableaux différents avec ces n enregistrements.

Comme le nombre de comparaisons dans le pire cas est alors la longueur de la plus longue branche, on a réduit un problème de complexité algorithmique (connaître le meilleur nombre de comparaisons dans le pire cas) à un problème purement combinatoire : quelle est la hauteur minimale d'un arbre binaire (chaque nœud intérieur a deux enfants) ayant $n!$ feuilles ?

Prenons un arbre binaire de hauteur h. Si toutes les branches ne sont pas de longueur h, on voit que les rallonger permet d'avoir plus de feuilles. Si toutes les branches sont de longueur h, on voit qu'il y a exactement 2^h feuilles. Autrement dit, la hauteur minimale d'un arbre binaire ayant $n!$ feuilles est le plus petit entier h tel que

$$2^h \geqslant n!$$

On admet (la démonstration nécessite la formule de Stirling, qui établit que $n! \sim 2^{n\log(n)}$, et que l'on ne va pas essayer d'expliquer ici) que cet entier est en $\mathrm{O}(n\log(n))$. ☺

Ce théorème mérite un peu d'attention. On a prouvé une borne inférieure intrinsèque au problème, et pas à notre imagination. Il n'y aura jamais d'algorithme de tri plus rapide que $\mathrm{O}(n\log(n))$ dans le pire cas — et qui plus est, on en connaît un.

On pourrait croire le domaine de recherche mort ; ce n'est pas du tout le cas, et pour deux raisons :

— si le tri fusion est optimal, il ne l'est qu'asymptotiquement : le nombre d'opérations varie comme il faut quand la taille des entrées augmente, mais peut-être fait-il trop d'opérations dès le début (un algorithme faisant $1000 \times n\log(n)$ opérations et un autre en faisant $1000 \times n\log(n)$ ont la même complexité asymptotique — mais pas les mêmes performances) ;

— le tri fusion est optimal dans le pire cas. Or on a vu un algorithme linéaire dans le meilleur cas. On peut imaginer qu'il en existe qui soit en moyenne bien meilleurs que $\mathrm{O}(n\log(n))$, voire qui soient bien meilleurs sur certaines données (en effet, on trie rarement des données placées dans un ordre aléatoire, on peut imaginer pouvoir en exploiter la structure).

2.7 Résumé

	Meilleur cas	Moyenne (empirique)	Pire cas
Insertion	$\mathrm{O}(n)$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^2)$
Fusion	$\mathrm{O}(n\log(n))$	$\mathrm{O}(n\log(n))$	$\mathrm{O}(n\log(n))$
Minimum théorique	?	?	$\mathrm{O}(n\log(n))$

3

Faire des choix

On s'intéresse maintenant aux problèmes qui ont plusieurs solutions, solutions pouvant être construites comme une série de choix. On va voir que dans certains cas, une solution est meilleure que les autres, et qu'elle peut ou non être obtenue en faisant le meilleur choix possible à chaque moment. Dans d'autres cas, on va devoir revenir en arrière pour prendre un autre choix ; enfin, on étudiera un problème dont la structure est telle que l'on peut faire des choix globaux.

Pour l'instant, on a vu des problèmes qui avaient une solution à peu près unique[1]. Ce n'est pas le cas en général : souvent un problème admet plusieurs solutions. Considérons, par exemple, un problème de logistique : on veut acheminer des doses de vaccin depuis les usines du fabricant jusqu'à des centres de vaccinations, pour cela, on voudrait un algorithme qui prenne en entrée la position des usines et leur capacité de production, ainsi que la position des centres de vaccination et leur capacité de vaccination, et qui, en sortie, donne le trajet de véhicules à même de transporter les doses. Il va de soi que l'algorithme suivant répond au problème : on assigne à chaque dose un véhicule, qui va aller d'une usine à un centre de vaccination. Si le véhicule pouvait contenir plus d'une dose, c'est de l'espace gâché, si deux centres étaient juste à côté (ou en tout cas, dans la même direction depuis l'usine), on a raté une possibilité de mutualiser,... Ainsi, cet algorithme de planification résout le problème, mais de manière pas du tout optimale.

Ainsi, souvent, on ne veut pas juste une solution à un problème : on veut une solution qui soit *optimale*. Il est en vérité souvent très difficile de définir ce qu'est cette optimalité : pour reprendre l'exemple, veut-on utiliser le moins de véhicules possibles, qu'ils passent le moins de temps possible sur la route (si par exemple, le vaccin ne doit pas quitter trop longtemps les super-réfrégirateurs), ou un peu des deux ? Et désirons-nous vraiment la solution optimale (une fois qu'on a défini notre critère d'optimalité), ou nous satisfaisons-nous d'une solution proche de l'optimum (par exemple, si planifier optimalement prendrait un mois de calcul en plus, on préfère une solution pas trop mauvaise à la meilleure possible) ? Cela nécessite non seulement de caractériser l'optimalité, mais aussi la distance à l'optimalité.

En résumé, dans de nombreux cas, on veut une solution qui soit optimale, pour une optimalité définie de manière floue, et qu'on ne recherche même pas complètement ; et ces considérations sont a priori complètement indépendantes de la complexité de l'algorithme produisant la solution : cela fait partie de sa spécification, et donc de la correction.

3.1 Gloutonnerie

Dans cette section, on va s'intéresser à une famille d'algorithmes, les *algorithmes gloutons* : pour les définir grossièrement, ce sont des algorithmes qui font une succession de choix pour résoudre un problème (par exemple, *choisir* d'affecter un véhicule sur un certain trajet), chaque choix est optimal, mais purement localement : on ne regarde jamais le problème dans sa globalité, on fait le meilleur choix à l'instant donné, et on ne revient jamais sur ses choix. Pour certains problèmes, les algorithmes gloutons donnent une solution globalement optimale, pour d'autre, juste une solution.

C'est donc une famille d'algorithmes rapides mais ne donnant pas toujours la solution optimale : il faut toujours se demander s'ils sont pertinents, mais savoir quand ne pas en utiliser un.

1. Ou en tout cas, il n'y avait pas de manière de classer les différentes solutions (certes, il y a plusieurs manières de trier une liste contenant plusieurs fois la même clef, et différents algorithmes donneront des résultats différents, mais les différentes listes triées sont tout autant la solution).

Sac à dos fractionné et sac à dos

Commençons par un problème de logistique simplifié : supposons que nous avons un *sac à dos* d'une capacité de 10 L. On veut transporter dans ce sac à dos des objets en essayant de maximiser leur valeur monétaire[2]. L'analyse que nous avons fait plus tôt nous fait donc dire que nous cherchons non seulement une solution (une liste d'objets tenant, ensemble, dans le sac à dos), mais une solution optimale, où l'optimalité est donnée, extérieurement à l'objet, par une valeur monétaire que l'on prend comme acquise[3]. Par exemple, supposons que l'on a les objets suivants, avec leur volume, leur prix et leur prix volumique :

	volume	prix	prix volumique
lingot d'or	6 L	5,4 M€	895 k€/L
lingot d'argent	4 L	28 k€	7 k€/L
lingot de palladium	8 L	6,0 M€	751 k€/L
lingot de platine	5 L	3,5 M€	692 k€/L
lingot de platine	5 L	3,5 M€	692 k€/L

En testant les différentes possiblités, on voit que le plus rentable est de prendre les deux lingots de platine, et ainsi, pouvoir partir avec 7,0 M€ dans le sac à dos[4]. Voyons comment modéliser ce problème. On suppose qu'on a une liste d'enregistrements, représentant chacun un objet, une fonction volume et une fonction prix permettant de récupérer, pour chaque enregistrement, respectivement le volume et le prix. On va appeler une telle liste un *inventaire*. Étant donné un inventaire et une capacité totale, on commence par trier l'inventaire par prix décroissant (ce qui se fait en $\mathrm{O}(n\log(n))$), puis, on compare le volume de chaque enregistrement avec la capacité restante : s'il est supérieur, on ne le prend pas, sinon, on le prend.

Exercice 10. Écrire cet algorithme sous forme d'une fonction python. On utilisera l'affichage pour donner des informations intéressantes.

Correction : Il faut commencer par choisir une représentation des données d'entrées. Assez naturellement, on choisit un tableau dont chaque élément est un triplet composé du nom de l'objet, de son volume et de son prix (l'ordre est totalement arbitraire, il faut juste le garder de manière constante).

On doit donc commencer par pouvoir trier l'inventaire en ordre décroissant. Cela implique de ré-écrire un des tris (de préférence le tri fusion, qui est de complexité $\mathrm{O}(n\log n)$) de manière à ce qu'il trie selon la clef qui est l'élément 2 d'un tuple.

```
1  def fusion(liste1, liste2):
2      """si les deux listes liste1 et liste2 sont des listes de tuples, triées
3      selon leur élément 2, fusionne ces deux listes en une liste triée
4      """
5
6      i1 = 0
```

2. Pour simplifier, on va supposer que les formes n'ont aucune importance et seul compte le volume.

3. Mais si l'argent est justement ce qui vaut (Simmel 1987, premier chapitre, III), construire une échelle d'optimalité revient donc à créer une échelle monétaire. Ici on la suppose donnée.

4. Il faut un dos solide, vu que 10 L de platine pèsent 214,5 kg.

```
7       i2 = 0
8
9       # i1 et i2 vont pointer vers des éléments respectivement de liste1 et de
10      # liste2
11
12      res = []
13
14      while i1 < len(liste1) and i2 < len(liste2):
15          # tant que liste1[i1] et liste2[i2] ont un sens
16
17          if liste1[i1][2] > liste2[i2][2]:
18              res = res + [liste1[i1]]
19              i1 = i1 + 1
20          else:
21              res = res + [liste2[i2]]
22              i2 = i2 + 1
23
24      while i1 < len(liste1):
25          # tant que liste1[i1] a un sens (mais liste2[i2] n'en n'a plus)
26
27          res = res + [liste1[i1]]
28          i1 = i1 + 1
29
30      while i2 < len(liste2):
31          # tant que liste2[i2] a un sens (mais liste1[i1] n'en n'a plus)
32
33          res = res + [liste2[i2]]
34          i2 = i2 + 1
35
36      return res
37
38  def triFusion(liste):
39      """tri fusion de la liste liste. On copie à chaque fois, aussi bien en
40      séparant la liste qu'en la refusionnant"""
41
42      # On cree deux demi-listes et on les remplit
43
44      demiliste1 = []
45      demiliste2 = []
46
47      milieu = len(liste)//2
48
49      if milieu == 0:
50          # si la liste ne contenait qu'un element
51          return liste
52
53      # sinon, on copie deux demi-listes
54
55      for i in range(0,milieu):
56          demiliste1 = demiliste1 + [liste[i]]
57
58      for i in range(milieu,len(liste)):
59          demiliste2 = demiliste2 + [liste[i]]
60
61      # on les trie
62
63      demiliste1 = triFusion(demiliste1)
64      demiliste2 = triFusion(demiliste2)
65
66      # on les fusionne
67
```

```
68      return fusion(demiliste1,demiliste2)
```

Puis, on a qu'à parcourir la liste, et rajouter à une liste d'objets pris chaque objet pour lequel on a encore de la place. Cela signifie qu'on va considérer :

en entrée la taille du sac-à-dos et la liste des objets (pas forcément triée !)

en sortie un triplet constitué de la liste des objets, leur volume et leur prix. Les deux derniers sont inutiles (car peuvent être recalculés facilement à partir de la liste des objets) mais pourquoi pas !

Et nous utiliserons, comme variable interne, la liste des objets déjà pris, le volume déjà occupé et le prix de ce que l'on a déjà volé.

```
1   def sacADosGlouton(objets,volume):
2       """Résoud le problème du Sac À Dos sur les objets, en suivant l'algorithme
3       glouton"""
4
5       objets = triFusion(objets)
6
7       prix = 0
8       reste = volume
9       contenuSac = []
10
11      for i in range(0,len(objets)):
12          if objets[i][1] <= reste:
13              print("On prend l'objet", objets[i][0])
14              contenuSac = contenuSac + [objets[i]]
15              reste = reste - objets[i][1]
16              prix = prix + objets[i][2]
17          else:
18              print("On ne prend pas l'objet", objets[i][0])
19
20      return (contenuSac,reste,prix)
21
```

☺

Comme il parcourt linéairement l'inventaire, l'algorithme complet (tri et SacÀDos) est lui aussi en $O(n \log n)$. C'est un algorithme glouton : localement, il fait le choix optimal, c'est-à-dire prendre l'objet le plus cher qu'il peut mettre dans son sac à dos.

Globalement, il ne choisit pas du tout l'optimum : en effet, ici, on va d'abord prendre le lingot de palladium, ce qui ne laissera que 2 L disponibles — or, ils sont insuffisants pour mettre les autres lingots. Donc, on n'emportera avec nous que ce lingot de palladium, pour une valeur de 6,0 M€.

Cet algorithme glouton nous donne donc une solution, qui est loin d'être mauvaise, mais qui n'est pas la solution optimale. En réalité, trouver la solution optimale pour le problème du sac à dos est extrêmement compliqué : on peut montrer que c'est un problème NP-complet.

Exercice 11. Écrire sous forme d'une fonction python l'algorithme qui énumère tous les choix possibles et choisit l'optimal. On utilisera l'affichage pour donner des informations intéressantes.

Néanmoins, une variante du problème existe pour lequel un algorithme glouton fonctionne ! On l'appelle le *problème du sac à dos fractionné*, dans lequel on peut prendre une fraction des objets (par exemple, si ce sont des épices). Il suffit alors de trier l'inventaire par prix volumique décroissant.

Figure 3.1 – Les pays membres de l'Union Européenne

Exercice 12. Écrire un algorithme glouton trouvant la solution optimale pour le problème du sac à dos fractionné.

Montrer que cet algorithme donne la solution optimale.

Coloriage de graphes

Considérons maintenant un problème qui n'a rien à voir : supposons que nous cherchions à réaliser une carte politique d'une certaine région. En règle générale, on essaye de colorier chaque unité politique (un pays, un département,...) avec une couleur, de manière à ce que deux unités voisines ne soient jamais de la même couleur. Par exemple, sur le site du Conseil de l'Union Européenne, on trouve la carte de la Figure 3.1, où certains pays ne partageant pas de frontières ont des teintes quasi-identiques[5] (comme par exemple, la France et la Grèce).

On peut vouloir un algorithme résolvant ce problème : colorier chaque pays d'une carte avec une couleur, de manière à ce que deux pays ne se touchant pas n'aient pas la même couleur. On a une solution évidente : colorier chaque pays par une couleur différente. Néanmoins, on peut vouloir éloigner les couleurs de pays voisins, pour diminuer les confusions (quitte ensuite, à les différencier légèrement — c'est ce qui a été fait pour la carte de l'Union Européenne). Parmi toutes les solutions, on a un critère d'optimalité assez simple : on peut chercher un coloriage qui utilise le moins possible de couleurs.

5. Ici, le choix a été fait que chaque pays soit colorié différemment ; cela ne change rien au problème, comme on verra.

Avant de traiter le problème, on doit commencer par le modéliser. Qu'entend-on par un pays ? Une couleur ? Une frontière ? Et comment représenter tout ça dans une structure de données ?

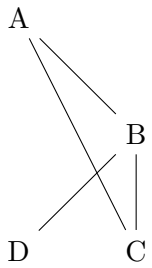
On se rend compte assez vite que la forme des pays et le tracé exact des frontières importe peu, seule compte l'information de savoir quel pays a une frontière avec quel autre (cela nécessite sans doute de définir ce que partager une frontière signifie : en particulier, il faut décider de si on compte les frontières maritimes [6], les zones disputées, l'outre-mer [7]... mais ce n'est pas notre problème). Autrement dit, on peut modéliser le problème par un graphe représentant la présence ou non d'une frontière.

Un *graphe* est un outil essentiel de modélisation. Dans sa version la plus simple [8], on peut le voir comme :

— un ensemble fini de *sommets*

— un ensemble fini d'*arêtes* reliant les sommets du graphe entre eux.

On peut très facilement dessiner un graphe. Par exemple,



est la représentation d'un graphe G, dont les sommets sont A, B, C, et D ; et dont les arêtes, qui relient A et B, B et C, B et D, et A et C, sont représentées par des segments.

Ainsi, on va représenter la carte de la Figure 3.1 par le graphe de la Figure 3.2.

et le problème revient à trouver un coloriage des sommets tel qu'aucune arête ne connecte deux sommets de la même couleur. On voit qu'on a rendu le problème plus abstrait (au lieu de devoir colorier une carte, on colorie les sommets d'un graphe. En particulier, plusieurs cartes peuvent avoir le même graphe, par exemple, les triplets de pays France/Espagne/Portugal et Roumanie/Bulgarie/ sont identiques) permettant de n'en garder que les éléments qui nous intéressent ; on l'appelle le *problème du coloriage de graphes*.

Avant de continuer, assurons-nous que nous savons représenter un graphe par les structures de données que nous connaissons déjà. Comme pour les listes, on va commencer par se demander quelles opérations on peut faire sur un graphe. Pour rester simple, disons que l'on veut au minimum pouvoir :

— ajouter un sommet ;

— ajouter une arête ;

— enlever un sommet ;

6. Auquel cas, l'Italie et l'Espagne partagent une frontière au large de la Sardaigne et des Baléares.

7. Auquel cas, la France et les Pays-Bas partagent une frontière terrestre à Saint-Martin.

8. Que, dans des cadres plus généraux, on va appeler un *graphe simple fini non-orienté*.

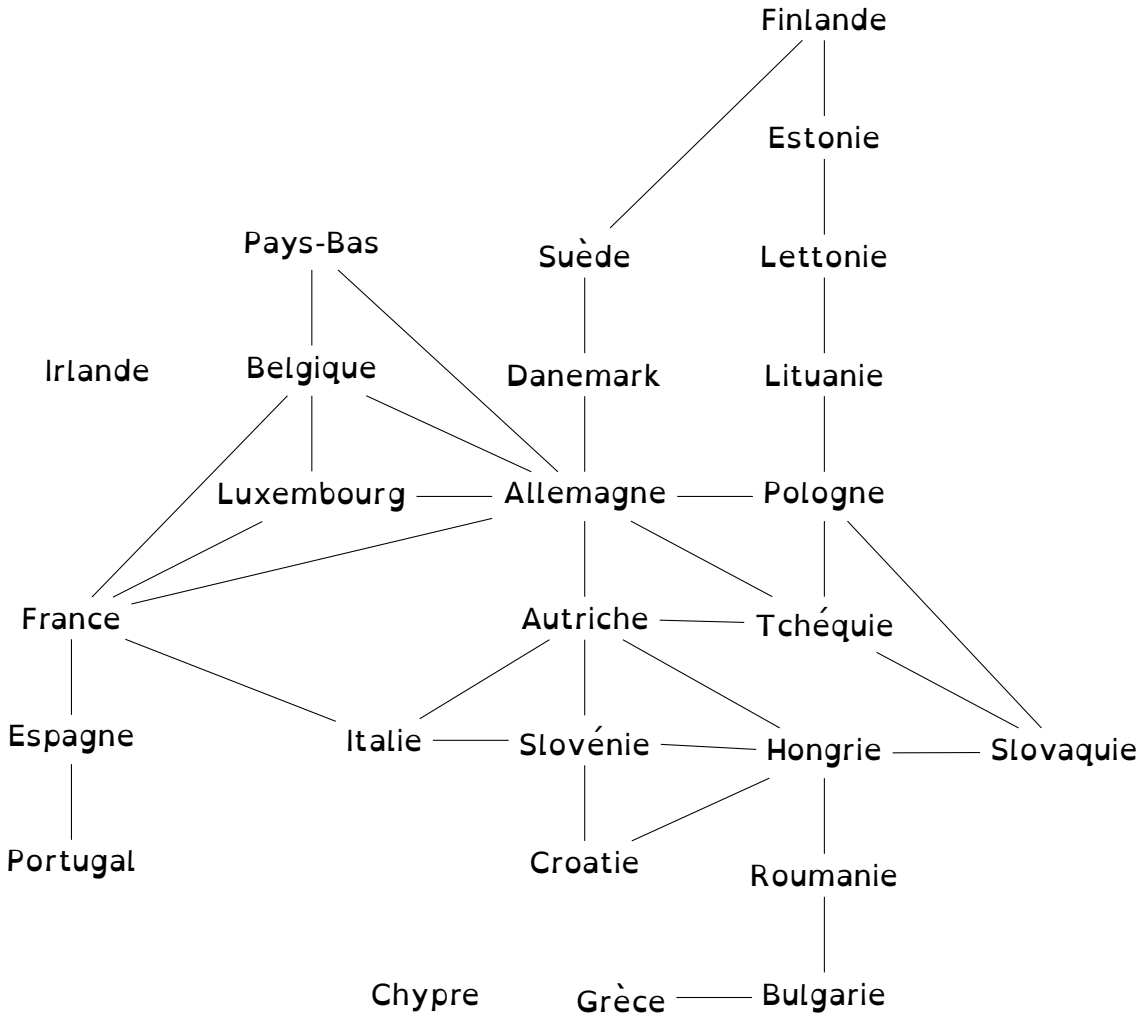


Figure 3.2 – Le graphe des frontières terrestres en Europe des pays membres de l'Union Européenne

— enlever une arête ;
— tester si deux sommets ont une arête entre eux.

Il y a deux méthodes principales pour représenter des graphes :

avec des tableaux si on considère que les sommets sont numérotés $0, ..., n$, on peut représenter un graphe par un tableau à deux dimensions $(\mathsf{TAB}[i][j])_{0 \leqslant i < n, 0 \leqslant j < n}$ tel que

$$\mathsf{TAB}[i][j] == 1$$

s'il y a une arête entre le sommet i et le sommet j, et 0 sinon.
Ainsi, le graphe G de la page 59 serait représenté par un tableau de dimension 4 associant à chaque numéro le sommet de ce numéro : $[|\mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}|]$ ainsi qu'un second tableau, de dimension 4×4, que l'on va noter $(\mathsf{G}[i][j])_{0 \leqslant i < 4, 0 \leqslant j < 4}$ et que l'on peut représenter par :

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Autrement dit, vu qu'on a décidé que les sommets étaient ordonnées dans l'ordre A, puis B, puis C, puis D, $G[0][1] == 1$ signifie qu'il y a une arête entre le sommet 0 et le sommet 1 (donc entre A et B), et $G[0][3] == 0$ signifie qu'il n'y a pas d'arête entre le sommet 0 et le sommet 3 (donc entre A et D).

avec des listes on se donne une liste, dont chaque élément est la paire d'un sommet et d'une liste contenant les sommets ayant une arête en commun avec ce sommet.
Ainsi, le graphe G de la page 59 serait représenté par la liste

$$[(A, [B, C]), (B, [A, C, D]), (C, [A, B]), (D, [B])]$$

.

Exercice 13. Pour ces deux représentations, représentez le graphe des voisins de l'Union Européenne.

Exercice 14. Pour ces deux représentations, donnez des algorithmes pour les opérations listées ci-dessus. Évaluez leur complexité. Avez-vous des remarques à faire sur ces deux représentations ?

Exercice 15. Montrer que les tableaux représentant un graphe sont tous *symétriques*, c'est-à-dire symétriques par rapport à la diagonale du coin supérieur gauche au coin inférieur droit[9], c'est-à-dire que pour un tel tableau $(G[i][j])_{0 \leqslant i < N, 0 \leqslant j < N}$, pour chaque entiers $0 \leqslant i < N, 0 \leqslant j < N$,

$$G[i][j] == G[j][i].$$

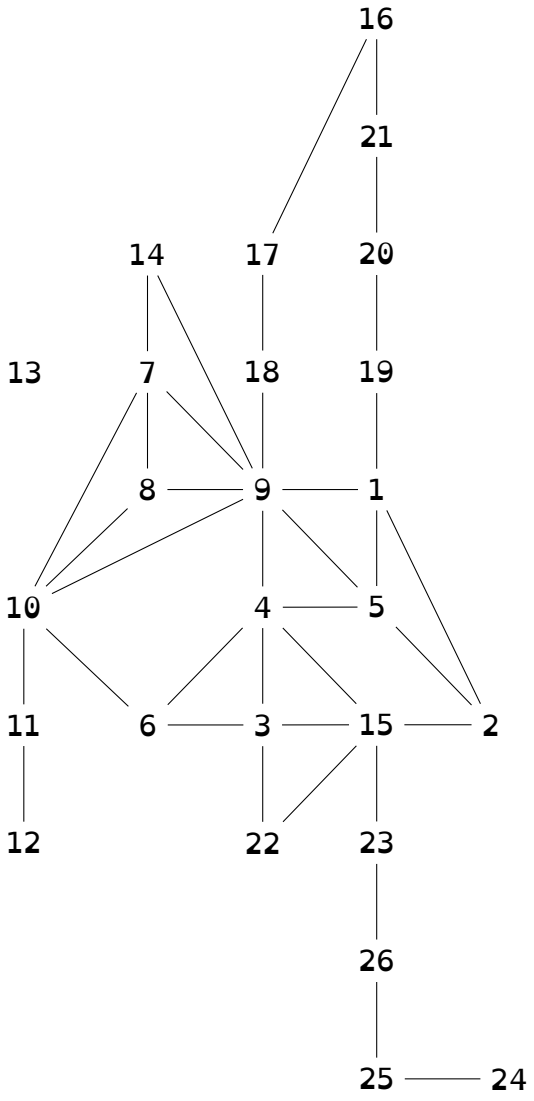
Nous pouvons revenir à notre problème : pour faire mieux que la solution triviale[10], il suffit de prendre chaque sommet, de supposer que les couleurs sont ordonnées, et d'attribuer à ce sommet la première couleur qui n'entre pas en conflit avec ces voisins. C'est bien un algorithme glouton : il fait des choix localement, et prend le moins de couleurs possibles étant données les contraintes locales, mais rien ne garantit que le choix soit le meilleur globalement.

Supposons que les pays et les couleurs sont numérotées (pour distinguer, on numérotera les pays avec des nombres et les couleurs avec des lettres) : ainsi, plutôt que de dire « la Slovaquie est coloriée en rouge », on dira (par exemple) « 8 est D ». Supposons qu'on a ordonné les pays dans l'ordre Pologne,

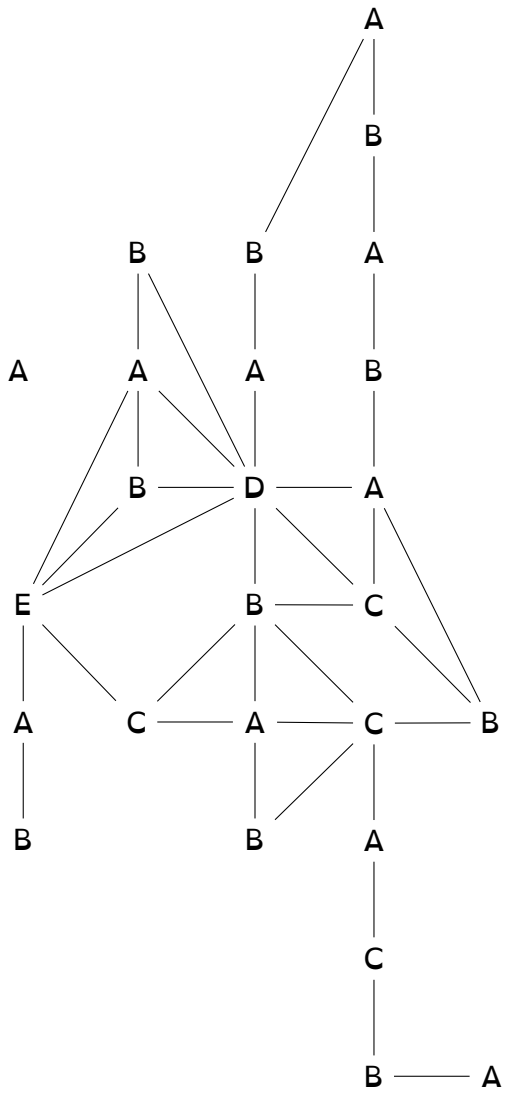
9. En algèbre, ces tableaux sont appelés des *matrices* et la diagonale en question a un rôle si important — vu qu'elle connecte un élément avec lui-même — qu'on l'appelle *la* diagonale.

10. Trivial, en mathématiques, a un sens assez différent du sens courant, et signifie « là où rien n'a été fait ». Ici, la solution triviale du coloriage consiste à donner une couleur différente à chaque nœud. On n'a rien fait, la solution ne dépend pas du graphe.

Slovaquie, Slovénie, Autriche, Tchéquie, Italie, Belgique, Luxembourg, Allemagne, France, Espagne, Portugal, Irlande, Pays-Bas, Hongrie, Finlande, Suède, Danemark, Lituanie, Lettonie, Estonie, Croatie, Roumanie, Chypre, Grèce, Bulgarie. Une fois que l'on a donné cet ordre, le nom des pays importe peu, autrement dit, on cherche tout aussi bien à colorier le graphe :



Exécutons notre algorithme sur cet ordre. On commence par le sommet numéro 1. Aucun de ses voisins n'a été colorié : on n'a aucune contrainte sur sa couleur, on peut donc prendre la première possible. Donc, on le colorie avec la couleur A. Le sommet 2 a un voisin de couleur A, aussi, on ne peut pas le colorier aussi avec : on le colorie avec B. Le sommet 3 n'a pas de voisin de colorié, on peut donc le colorier à nouveau avec A. Et ainsi de suite. Une fois colorié, on a les couleurs :



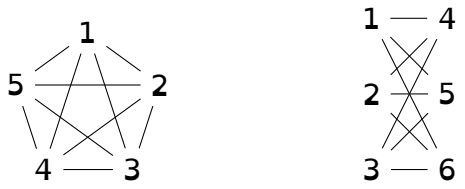
On a utilisé cinq couleurs dans ce coloriage. Ce n'est pas optimal (on peut en trouver un avec quatre couleurs), et en fait, le coloriage dépend de l'ordre que l'on a choisi pour les sommets.

Exercice 16. Trouver un ordre pour les sommets coloriant ce graphe avec seulement quatre couleurs.

Exercice 17. Écrire l'algorithme pour chacune des deux représentations des graphes.

Remarque 7 (théorème des quatre couleurs)**.** On dit qu'un graphe est *planaire* s'il peut être dessiné sur un plan sans qu'aucune arrête ne se croise. Par exemple, le graphe des frontières des pays de l'Union Européenne est planaire, tandis

que les deux graphes :



ne le sont pas (on les appelle K_5 et $\mathrm{K}_{3,3}$ et on peut montrer que, à un certain sens, ce sont les seuls, c'est-à-dire qu'ils sont présents dans tous les graphes non-planaires (Kuratowski 1930 ; Wagner 1937)).

En 1852, Francis Guthrie, mathématicien et botaniste sud-africain, a conjecturé que tous les graphes planaires peuvent être coloriés avec seulement quatre couleurs (cela a pour conséquence que toutes les cartes d'unités politiques sans enclaves ni exclaves peuvent être coloriées avec seulement quatre couleurs). Cela n'a été prouvé qu'en 1976 (voire en 2008, selon vos standards de ce qu'est une preuve (Gonthier 2008)), sous le nom de *théorème des quatre couleurs*.

On peut le prouver : pour chaque graphe planaire, une coloration avec seulement quatre couleurs existe[11].

Exercice 18. Représenter $\mathrm{K}_{3,3}$ et K_5 comme des tableaux et comme des listes.

Exercice 19. On cherche à construire un emploi du temps pour une formation. Certains cours sont obligatoires, d'autres non. On attend la fin des inscriptions pédagogiques pour procéder à l'établissement de l'emploi du temps, ainsi, on connaît toutes les incompatibilités entre cours.

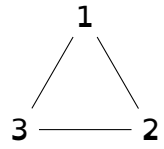
- le cours d'*Histoire de l'art du Moyen-Âge* est obligatoire pour toute la promotion ;
- chaque étudiant a du choisir exactement un cours entre *Esthétique et Philosophie de l'Art* et *Archéologie du monde rural*
- tout le monde a pris une langue morte (*Latin* ou *Grec*), mais cette année, personne n'a choisi à la fois *Grec* et *Esthétique et Philosophie de l'Art*
- les personnes ayant pris *Archéologie du monde rural* ont la possiblité de prendre une mineure *Droit de l'archéologie*.
- tout le monde a pris une langue vivante (*Anglais*, *Allemand*, *Espagnol* ou *Italien*), et on trouve des personnes ayant pris chaque langue vivante avec chaque autre option.

On veut donc construire un emploi du temps qui utilise des créneaux différents sans conflits. Écrire ce problème sous forme d'un problème de coloriage de graphe.

Comment peut-on faire s'il y a des contraintes en plus ? (par exemple, certains cours sont trop petits pour certaines salles, certains cours sont enseignés par la même personne,...)

Exercice 20. Écrire comme une formule en logique propositionnelle que le graphe

11. Attention, on ne connaît aucune preuve raisonnablement courte de ce résultat — se référer à l'article de Gonthier pour plus de détails.



n'est pas coloriable avec deux couleurs.

Rendu de monnaie

Exercice 21. On veut payer une certaine somme en pièces. Pour cela, on cherche un algorithme prenant en entrée les pièces possibles (on suppose avoir autant de pièces qu'il le faut de chaque valeur faciale) et la somme que l'on veut payer, et donne en sortie le nombre de chaque pièce qu'il faut.

Donner un algorithme trivial, et un algorithme glouton pour résoudre ce problème.

Correction : On va supposer que les pièces possibles nous sont données comme une liste chaînée, et que l'on va renvoyer une liste chaînée de couple dont le premier élément est le nombre de pièces, le second la valeur.

Une idée d'algorithme trivial peut sélectionner la plus petite pièce (en supposant que tous les prix possibles sont des multiples de cette pièce — si ce n'est pas le cas, on renvoie une erreur) et rendre la monnaie uniquement avec cette pièce-là. On va supposer que la liste des pièces en entrée est donnée en ordre croissant.

Entrées :
- un entier S ;
- une liste d'entiers L, triée par ordre croissant.

```
1  Rendu(S, L)
2  |  (e, L) ← L
3  |  n ← 0
4  |  tant que S > 0 faire
5  |  |  S ← S − e
6  |  |  n ← n + 1
7  |  fin
8  |  si S == 0 alors
9  |  |  retourner ((n, e), Λ)
10 |  sinon
11 |  |  retourner une erreur
12 |  fin
```

Sorties : une liste d'entiers ou une erreur

Pour ce qui est de l'algorithme glouton, il va fonctionner comme pour le problème du sac à dos : on commence par utiliser autant qu'on peut la plus grosse pièce, puis, dès que la plus grosse pièce est plus grosse que la somme qu'il reste à rendre, on passe à celle d'après,... On suppose donc, dans l'Algorithme 1 que la liste des pièces possibles est triée par ordre décroissant. ☺

Exercice 22. Avant 1971, la livre sterling, monnaie britannique avait des subdivisions assez baroque. En effet, une livre (£) se divisait en vingt shilling (s.), et un shilling se subdivisait lui-même en 12 pence (d.). Autrement dit :

$$1\,£ \equiv 20\,\mathrm{s.} \equiv 240\,\mathrm{d.}$$

Entrées :
— un entier S ;
— une liste d'entiers L, triée par ordre décroissant.

```
1  Rendu(S, L)
2  |  R ← Λ
3  |  tant que L ≠ Λ faire
4  |  |  (e, L) ← L
5  |  |  n ← 0
6  |  |  tant que S ⩾ e faire
7  |  |  |  S ← S − e
8  |  |  |  n ← n + 1
9  |  |  fin
10 |  |  R ← ((n, e), R)
11 |  fin
12 |  si S == 0 alors
13 |  |  retourner R
14 |  sinon
15 |  |  retourner une erreur
16 |  fin
```

Sorties : une liste d'entiers ou une erreur

Algorithme 1 : Rendu de monnaie — Algorithme glouton

De plus, circulaient (en Angleterre et au Pays-de-Galles, on ne va pas considérer l'Écosse, l'Irlande, les dépendances de la couronne ni les colonies) des pièces au noms et valeurs variées :

five pounds	1 200 d.
double sovereign	480 d.
sovereign	240 d.
crown	60 d.
half crown	30 d.
florin	24 d.
shilling	12 d.
sixpence	6 d.
groat	4 d.
threepence	3 d.
penny	1 d.
halfpenny	$\frac{1}{2}$ d.
farthing	$\frac{1}{4}$ d.

Montrer que l'algorithme glouton ne produit pas un résultat optimal dans ce cas-là.

Correction : Si on cherche à rendre 8 d., l'algorithme glouton nous fait rendre un *sixpence* et deux *penny*. Or, on peut réaliser 8 d. avec deux *groats*. ☺

Exercice 23. Prouver que l'algorithme glouton est optimal pour le rendu en euros.

Correction : Commençons par une première remarque : l'algorithme glouton ne produit pas un résultat optimal pour le système anglais d'avant la décimalisation, donc la démonstration va utiliser de manière importante des propriétés du système monétaire en euro.

Considérons une somme S et une solution optimale

$$[(n_{200}, 200); (n_{100}, 100); (n_{50}, 50); (n_{20}, 20); (n_{10}, 10); (n_5, 5); (n_2, 2); (n_1, 1)].$$

On a que :

- $n_{100}, n_{50}, n_{10}, n_5, n_1 < 2$, car sinon, on pourrait les remplacer par leur double et obtenir une solution avec moins de pièces utilisées ;
- $n_{20} + n_{10} < 3$, car sinon, on pourrait remplacer ces pièces d'une meilleure manière (en effet, si on a trois pièces de vingt, on peut les remplacer par une de cinquante et une de dix, et si on a deux de vingt et une de dix, on peut les remplacer par une de cinquante), et de même $n_2 + n_1 < 3$.

On voit qu'en réalité, on a très peu de latitude. Considérons maintenant la solution fournie par l'algorithme glouton

$$[(g_{200}, 200); (g_{100}, 100); (g_{50}, 50); (g_{20}, 20); (g_{10}, 10); (g_5, 5); (g_2, 2); (g_1, 1)].$$

On a que :

- $g_{100}, g_{50}, g_{10}, g_5, g_1 < 2$, en effet, la somme est strictement inférieure au double de ces valeurs quand on les considère ;
- $g_{20} + g_{10} < 3$, et de même $g_2 + g_1 < 3$. En effet, la somme est strictement inférieure à 50 ou à 5 quand on considère ces pièces-là.

Par ailleurs, les pièces plus grandes que celles de cinq centimes ne changent pas le chiffre des unités. Donc, on a

$$n_5 \times 5 + n_2 \times 2 + n_1 \times 1 == g_5 \times 5 + g_2 \times 2 + g_1 \times 1.$$

Par ailleurs, on remarque que $n_2 \times 2 + n_1 \times 1 < 5$ et $g_2 \times 2 + g_1 \times 1 < 5$. Si n_5 vaut 0, alors c'est aussi le cas de g_5, et vice-versa. De même, si n_5 vaut 1, alors c'est aussi le cas de g_5, et vice-versa. Donc

$$n_5 == g_5.$$

Comme la parité de S est aussi celle de $n_5 \times 5 + n_1$, la parité de $\mathrm{S} - n_5 \times 5$ est celle de n_1, et aussi celle de g_1. Donc

$$n_1 == g_1.$$

L'équation ci-dessus implique donc aussi que $n_2 == g_2$.

On peut reproduire le raisonnement fait sur les unités sur les dizaines, puis sur la parité des centaines.

Ainsi, l'algorithme glouton fournit une solution optimale, mais cela dépend fortement des pièces existantes. ☺

Deuxième partie

Annexes

A

Feuilles de TD

A.1 Prise en main du REPL

Une expression en python (c'est-à-dire du texte qu'on tape) a deux caractéristiques :
— un type, c'est-à-dire une catégorie qui détermine les opérations pouvant être faites avec cette expression (on ne fait pas la même chose avec un nombre, une lettre,...).

Étant donné une expression, on peut en connaître le type avec la fonction type().
— une valeur, qui est ce que vaut l'expression.

Exercice 24. Pour chacune des expressions suivantes, écrivez ce que vous pensez être son type et sa valeur, puis testez :

```
3+5
(3+5)
7/2
7//2
7%2
((3+5)*2+1)/(2*2)
((3+5)*2+1)//(2*2)
'a'
'a' + 'b'
'a' + 3
) *+ 3
3 ** 3
1000000000 ** 100000000000 ** 1000000000000
type(3)
'ab' == 'a' + 'b'
'a' == 'a' + 'b'
```

Que connaissez vous comme type en python, et que pouvez-vous dire des différentes opérations ci-dessus ?

On peut faire autre chose que s'en servir comme d'une grosse calculatrice, et pour ça, on a besoin de la notion de variable : une variable est une case qui a un nom, et une valeur, qui peut changer.

On va noter, si x est une variable et e une expression

$$x = e$$

l'*assignation* à x de la valeur de e, c'est-à-dire qu'on calcule la valeur de e, et dorénavant, on décide que x vaut ce résultat.

On va noter, si e_1 et e_2 sont deux expressions

$$e_1 == e_2$$

le *test*, qui va calculer chacune des deux expressions, et valoir vrai (que l'on note **True** en python) si leurs valeurs sont égales, et faux (**False**) sinon. #+end$_{\text{aparte}}$

Exercice 25. Présenter sous forme d'un tableau (dont les colonnes sont les variables, et les lignes suivent les lignes du programme suivant) l'état successif des variables

```
x = 1
y = "Bonjour"
x == y
x = 2
z = x + 1
x == y
x = 1
z
z = (x == y)
y = y
t = (y == y)
t = z
```

Exercice 26. Faire une trace d'exécution de :

```
resultat = 0
for i in range(1,6):
    resultat = resultat + 3
resultat
```

Que vaut resultat ?

Exercice 27. On suppose que deux listes ont concouru pour une élection à *la proportionnelle au plus fort reste* où 5 sièges sont en lice. En supposant que le nombre de voix reçues par la première soit dans une variable voix_a et celle de la deuxième dans une variable voix_b, écrire un programme calculant le nombre de sièges reçus dans une variable siège_a pour la première liste et siege_b pour la seconde.

Impressions

La fonction print() qu'on a vu permet d'afficher une valeur.

Exercice 28. Afficher une chaîne de caractère, un nombre, puis une variable contenant cette chaîne de caractère et ce nombre.

Exercice 29. Utiliser une boucle **for** pour compter de 0 à 50.
Utiliser une boucle **while** pour compter de 0 à 50.
Le refaire pour compter à rebours.
On veut une phrase, par exemple « Il reste 50 secondes ». Comment faire ?

Exercice 30. On veut afficher un certain nombre d'astérisques (par exemple, pour séparer deux parties dans un livre).
Écrire une fonction qui prend en entrée un entier n et affiche n astérisques.

Corriger

On veut dire bonjour de manière très polie : on a une liste de personnes à qui dire bonjour, et on veut, pour chaque personne, dire bonjour une fois de plus qu'à la personne d'avant.
Quelqu'un a écrit ce code :

```python
from typing import Tuple, List

salutation : str = "Bonjour"

personnes : List[str] = ["Devaki", "Vasudeva", "Balarāma",
                         "K□ṣṇa", "Subhadrā", "Rukmini"]
j = 0


for i in range(0,6):
    print("i", salutation, personnes[i], "!", sep=" ")
    j = j + i
```

Exercice 31. Écrire une fonction dis-bonjour(personne:str) -> str qui prend en entrée le nom d'une personne et lui dise bonjour une fois.

Exercice 32. Écrire des tests pour cette fonction.

Exercice 33. En déduire une réécriture de ce programme, sous la forme d'une fonction

qui prenne en entrée une liste de personne et renvoie le nombre de fois total qu'on a salué.

Écrire des tests pour cette fonction.

Exercice 34. Rajouter toutes les annotations de type et commenter.

A.2 Listes

Exercice 35. Écrire une fonction prenant en entrée une liste et renvoyant une liste ne contenant qu'un élément sur deux, celui en position 0, puis en position 2,...

Correction : On va écrire ce programme en ajoutant des éléments à une liste vide.

```python
# Algorithmique et Programmation — Master Droit & Informatique — UPEC
# Luc Pellissier 2022
# CC0

from typing import List, Any

def decimer(l:list[Any]) -> list[Any]:
    """On remplit une liste vide d'un élément sur deux de la liste l"""
    resultat : list[Any] = []
    compteur = 0

    # resultat est vide au début
    # compteur compte le nombre d'éléments déjà observés dans l

    for elt in l:                    # pour chaque élement elt de la liste l
        if compteur % 2 == 0:        # si compteur est pair
            resultat = resultat + [elt] # on rajoute elt à la fin de la liste resultat
        compteur = compteur + 1      # dans tous les cas, on incrémente compteur
    return resultat

assert decimer(["honi", "soit", "qui", "mal", "y", "pense"]) == ["honi", "qui", "y"]
assert decimer([]) == []
```

(**exercice :** faire de même en supprimant des éléments à la liste). ☺

Exercice 36. En faire une trace d'exécution sur la liste

```
1 ["honi", "soit", "qui", "mal", "y", "pense"]
```

Correction : L'exécution se déclenche en exécutant

```
1 decimer(["honi", "soit", "qui", "mal", "y", "pense"])
```

ce qui donne à l la valeur ["honi", "soit", "qui", "mal", "y", "pense"].
On peut présenter ça comme le tableau suivant :

ligne	suivante	resultat	compteur	elt
8	9	[]		
9	14	[]	0	
14	15	[]	0	'honi'
15	16	[]	0	'honi'
16	17	['honi']	0	'honi'
17	14	['honi']	1	'honi'
14	15	['honi']	1	'soit'
15	17	['honi']	1	'soit'
17	14	['honi']	2	'soit'
14	15	['honi']	2	'qui'
15	16	['honi']	2	'qui'
16	17	['honi', 'qui']	2	'qui'
17	14	['honi', 'qui']	3	'qui'
14	15	['honi', 'qui']	3	'mal'
15	17	['honi', 'qui']	3	'mal'
17	14	['honi', 'qui']	4	'mal'
14	15	['honi', 'qui']	4	'y'
15	16	['honi', 'qui']	4	'y'
16	17	['honi', 'qui', 'y']	4	'y'
17	14	['honi', 'qui', 'y']	5	'y'
14	15	['honi', 'qui', 'y']	5	'pense'
15	17	['honi', 'qui', 'y']	5	'pense'
17	14	['honi', 'qui', 'y']	6	'pense'
14	18	['honi', 'qui', 'y']	6	
18	**retourner** ['honi', 'qui', 'y']			

☺

Exercice 37. Écrire une fonction prenant en entrée une liste de nombres entiers et renvoyant la liste contenant tous les entiers pairs de cette liste.

Exercice 38. En faire une trace d'exécution sur la liste

```
1 [12, 3, 3, 2*2, 6, 8]
```

Exercice 39. Écrire une fonction prenant en entrée d'une liste, en affiche chaque élément séparé par "et puis", et renvoie le nombre d'éléments de la liste.

Exercice 40. En faire une trace d'exécution sur la liste

```
1 [12, 3, 3, 2*2, 6, 8]
```

Exercice 41. Écrire vous-même la fonction qui calcule la longueur d'une liste.

Exercice 42. En faire une trace d'exécution sur la liste

```
1 [12, 3, 3, 2*2, 6, 8]
```

Exercice 43. On s'intéresse aux transactions d'une personne, qui reçoit et dépense de l'argent. On veut être sûr que cette personne n'a jamais été dans le rouge : la somme de l'argent qu'iel possède doit toujours être positive.

On va représenter la liste des transactions par une liste d'entiers.

Écrire une fonction qui prenne en entrée une telle liste et en fasse les sommes partielles : c'est-à-dire qu'on veut une nouvelle liste, représentant le solde après les premières transactions.

Écrire une deuxième fonction qui prenne en entrée une liste et qui vérifie qu'elle ne contient pas d'éléments négatifs.

En déduire une solution au problème.

A.3 Assertions

Exercice 44. Donner des assertions pour tester chacune des fonctions précédentes.

Exercice 45. Faites-en de beaux fichiers !

A.4 Chaînes de caractères et listes

Exercice 46. Évaluer les expressions suivantes dans python :

```
1 'Bonjour'[0]
2 len('Bonjour')
3 len('')
4 for c in 'Bonjour':
5     print(c)
6 for i in range(0,len('Bonjour')):
7     print('Bonjour'[i])
```

Qu'en déduisez-vous ?

Exercice 47. Écrire une fonction prenant en entrée une chaîne de caractère et la renvoyant à l'envers.

Exercice 48. En faire une trace d'exécution sur 'Bonjour'.

Exercice 49. Écrire une fonction prenant en entrée une chaîne de caractère et la séparant en une liste d'une chaîne de caractères découpée selon ses espaces. Ainsi, on veut que

```
1 "Salut les masterant·es"
```

soit découpé en la liste de chaînes de caractères

```
1 ["Salut", "les", "masterant·es"]
```

Exercice 50. En faire une trace d'exécution sur "Salut les masterant·es".

Exercice 51. Écrire une fonction prenant en entrée une chaîne de caractère composée d'une liste de personnes (où chaque personne est sous la forme Nom, Prénom et un passage à la ligne entre chaque personne), et renvoyant une liste de paires de chaînes de caractères (ainsi, chaque personne sera représentée par un élément de la liste, et sous la forme (Prénom, Nom)).

Indication : la fin de ligne se note '\n'.

A.5 Boucle tant que

La boucle **tant que** est une variante de la boucle **pour** : tant qu'une condition est vérifiée, on exécute le corps de la boucle, quand elle ne l'est plus, on en sort.

Exercice 52. Effectuer une trace d'exécution de mystere("Ouest")[1] et mystere("Lisbonne

```
def mystere(texte:str) -> int:
    i = 0
    n = len(texte)

    while i < n and texte[i] != 'n':
        i = i + 1

    if i == n:
        print("Pas trouvé")
        return (-i)
    else:
        print("Trouvé")
        return i
```

Que fait la fonction mystere(srt) -> int ?
Que se passe-t-il si on inverse l'ordre des lignes print et **return** ?
Comment la modifier pour qu'elle trouve aussi les capitales ?

Exercice 53. Écrire une fonction cherchant le plus petit élément d'une liste de nombres.

Exercice 54. Écrire une fonction cherchant, dans une liste de chaînes de caractères, celle qui a un 'a' ou un 'A' le plus tôt.

A.6 Analyse de complexité

Exercice 55. Considérons le programme suivant :

```
# Algorithmique et Programmation — Master Droit & Informatique — UPEC
# Luc Pellissier 2022
# CC0

from typing import List

def maximum(l:list[int]) -> list[int]:
    """Renvoie les positions du plus grand élément de la liste"""
    maxi = l[0]
    longueur = len(l)

    for i in range(1,longueur):
        # Comme on a initialisé maxi avec la valeur de l[0], on n'a pas besoin
        # de considérer l[0] ici, donc la boucle commence en 1
```

1. *The Wild Wild West*, Michael Garrison, 1965 → 1969.
2. *Mistérios de Lisboa*, Raúl Ruiz, 2010.

```
15
16          if l[i] > maxi:
17              maxi = l[i]
18
19      res = []
20
21      for i in range(0,longueur):
22          # On rajoute à une liste toutes les positions d'éléments égaux à maxi
23
24          if l[i] == maxi:
25              res = res + [i]
26
27      return res
```

Quelle est sa complexité ?

Exercice 56.

```
1  # Algorithmique et Programmation — Master Droit & Informatique — UPEC
2  # Luc Pellissier 2022
3  # CC0
4
5  from typing import List
6
7  def maximum(l:list[int]) -> list[int]:
8      maxi = l[0]
9      longueur = len(l)
10     res = [0]
11
12     for i in range(1,longueur):
13         if l[i] > maxi:
14             maxi = l[i]
15             res = []
16         elif l[i] = maxi:
17             res = res + [i]
18
19     return res
```

Que fait ce programme ? Quelle est sa complexité ?

A.7 Récursivité

On veut écrire la fonction de recherche dichotomique comme une fonction récursive.

Exercice 57. Écrire une fonction récursive prenant en entrée trois arguments : un tableau, une position de début et une position de fin et qui fasse une recherche dichotomique entre le début et la fin.

Exercice 58. En déduire une fonction récursive réalisant la recherche dichotomique.

On représente une famille de la manière suivante : une personne est représentée par un tuple, dont le premier élément est son nom, et le deuxième est la liste

de ses enfants. On peut comme ça représenter un arbre[3] généalogique descendant.

Par exemple,

```
("Elizabeth",
 [
    ("Charles",
     [
        ("William",[("George",[]),("Charlotte",[]),("Louis",[])]),
        ("Henry",[("Archie",[]),("Lilibet",[])])
     ]
     ),
    ("Anne",
     [
        ("Peter",[("Savanaah",[]),("Isla",[])]),
        ("Zara",[("Mia",[]),("Lena",[]),("Lucas",[])])
     ]
     ),
    ("Andrew",
     [
        ("Beatrice",[("Sienna",[])]),
        ("Eugenie",[("August",[]),("Ernest",[])])
     ]
     ),
    ("Edward",
     [
        ("Louise",[]),
        ("James",)
     ]
     )
 ]
 )
```

Exercice 59. Écrire une fonction récursive prenant en entrée un tel arbre généalogique et affichant (dans n'importe quel ordre) les membres de la famille.

A.8 Tours de Hanoï

Le problème « des tours de Hanoï » est le suivant : on a trois piquets. Sur chaque piquet, il y a des anneaux plus ou moins gros, tous de taille différente. On ne peut déplacer qu'un anneau à la fois. Sur chaque piquet, un anneau ne peut être posé que sur un anneau plus gros.

Exercice 60. On part de la situation où tous les anneaux sont placés sur le premier piquet. On veut les placer sur le dernier.

Écrire une fonction :

— prenant en entrée le nombre d'anneaux ;
— affichant la suite de mouvements qu'il faut faire ;
— renvoyant en sortie le nombre de mouvements conseillés.

3. S'il n'y a jamais de recouvrement entre les descendances...

Par exemple, on veut, s'il n'y a que deux anneaux, afficher quelque chose comme :

```
Déplacer un anneau du piquet 1 au piquet 2
Déplacer un anneau du piquet 1 au piquet 3
Déplacer un anneau du piquet 2 au piquet 3
```

A.9 Affichage

Exercice 61. Écrire une fonction qui prenne en entrée une taille t et un caractère c et qui affiche une case de taille t de ca caractère. Par exemple, une case de taille 3 de '-' est :

```
---
---
---
```

Exercice 62. En déduire une fonction qui prenne quatre paramètres et affiche un damier de cases de deux caractères en alternance. Par exemple, un damier de taille 4 de cases de taille 3 des deux caractères '-' et '*' donnerait :

```
---***---***
---***---***
---***---***
***---***---
***---***---
***---***---
---***---***
---***---***
---***---***
***---***---
***---***---
***---***---
```

Exercice 63. Écrire une fonction qui, étant donnée en entrée trois tableaux :
- un contenant les jours de la semaine
- un contenant les mois de l'année
- un contenant le nombre de jours de chaque mois

affiche chaque jour de l'année 2024 (qui commence par un lundi ! et est bisextile).

Ainsi, le résultat devra commencer par les lignes :

```
lundi 1 janvier
mardi 2 janvier
mercredi 3 janvier
```

Exercice 64. Modifier le programme précédent pour qu'il prenne une entrée en plus représentant le jour par lequel commence l'année.

Le modifier encore pour qu'il renvoie une liste des jours, plutôt que de les afficher.

A.10 Sac à dos

On va fixer que l'inventaire des objets sera sous la forme d'un tableau de triplets, contenant (dans l'ordre) le nom, le volume et le prix de chaque objet.

Dans un fichier python (c'est-à-dire un fichier texte avec extension .py), recopier le programme suivant, qui permettra de tester vos algorithmes.

```
1  import random
2
3  def sacADosAleatoire(objets):
4      """crée un inventaire avec des prix et des volumes aléatoires"""
5
6      inventaire = []
7
8      for i in range(0,objets):
9          inventaire = inventaire + [("objet " + str(i),randrange(0,50),randrange(10,1000))]
10
11     return inventaire
```

Exercice 65. Que fait-il ?

Jouer avec les paramètres.

Exercice 66. Écrire une fonction de tri (en $\mathrm{O}(n \log n)$) qui trie selon la composante 2 d'un triplet.

Exercice 67. Écrire un algorithme glouton pour le problème du sac-à-dos sous forme d'une fonction python. On utilisera l'affichage pour donner des informations intéressantes.

Exercice 68. En faire une trace d'exécution sur un petit exemple, intéressant, que vous choisirez.

Exercice 69. Écrire une fonction python donnant la solution optimale du problème du sac-à-dos. On utilisera l'affichage pour donner des informations intéressantes.

Exercice 70. En faire une trace d'exécution sur un petit exemple, intéressant, que vous choisirez.

Exercice 71. Écrire un algorithme glouton pour le problème du sac-à-dos fractionné sous forme d'une fonction python. On utilisera l'affichage pour donner des informations intéressantes.

Exercice 72. En faire une trace d'exécution sur un petit exemple, intéressant, que vous choisirez.

Exercice 73. Montrer que cet algorithme produit la solution optimale.

B

Examens

B.1 2022 → 2023 — Premier

Trace

Considérons la fonction suivante :

```python
from typing import List

def mystere(l:list[int]) -> int:
    a = l[0]

    for x in l:
        if x > a:
            a = x

    return a
```

Exercice 74. Faire une trace d'éxecution de mystere([10, 3, 2, 3, 11, 4, 11]).

Correction :

ligne	suivante	l	a	x
3	4	[10, 3, 2, 3, 11, 4, 11]		
4	6	[10, 3, 2, 3, 11, 4, 11]	10	
6	7	[10, 3, 2, 3, 11, 4, 11]	10	10
7	6	[10, 3, 2, 3, 11, 4, 11]	10	10
6	7	[10, 3, 2, 3, 11, 4, 11]	10	3
7	6	[10, 3, 2, 3, 11, 4, 11]	10	3
6	7	[10, 3, 2, 3, 11, 4, 11]	10	2
7	6	[10, 3, 2, 3, 11, 4, 11]	10	2
6	7	[10, 3, 2, 3, 11, 4, 11]	10	3
7	6	[10, 3, 2, 3, 11, 4, 11]	10	3
6	7	[10, 3, 2, 3, 11, 4, 11]	10	11
7	8	[10, 3, 2, 3, 11, 4, 11]	10	11
8	6	[10, 3, 2, 3, 11, 4, 11]	11	11
6	7	[10, 3, 2, 3, 11, 4, 11]	11	4
7	6	[10, 3, 2, 3, 11, 4, 11]	11	4
6	7	[10, 3, 2, 3, 11, 4, 11]	11	11
7	6	[10, 3, 2, 3, 11, 4, 11]	11	11
6	10	[10, 3, 2, 3, 11, 4, 11]	11	
10	**retourner** 11			

☺

Exercice 75. Que fait cette fonction ? Justifier.

Correction : Cette fonction calcule et renvoie le maximum du tableau l. En effet, on stocke le plus grand élément déjà vu dans a, et on itère x sur chaque élément de l. Si x est plus grand que a, on change la valeur, sinon on la laisse inchangée. ☺

Exercice 76. Ré-écrire cette fonction en utilisant range.

Correction :

```python
# Algorithmique et Programmation — Master Droit & Informatique — UPEC
# Luc Pellissier 2022
# CC0

from typing import List
```

```python

def maximum(l:list[int]) -> int:
    """Renvoie le plus grand élément de la liste"""
    maxi = l[0]
    longueur = len(l)

    for i in range(1,longueur):
        # Comme on a initialisé maxi avec la valeur de l[0], on n'a pas besoin
        # de considérer l[0] ici, donc la boucle commence en 1

        if l[i] > maxi:
            maxi = l[i]

    return maxi
```

☺

Programmation

Supposons avoir deux listes de même longueur :
— dans la première, on a des noms ;
— dans la deuxième, des nombres : la taille (en centimètres) des personnes correspondantes.

Par exemple, les deux listes suivantes rentrent dans ce cadre :

```python
noms = ["Benny Andersson", "Agnetha Fältskog",
        "Anni-Frid Lyngstad", "Björn Ulvaeus"]
tailles = [177, 172, 172, 175]
```

Exercice 77. Écrire une fonction qui prenne en entrée deux telles listes et renvoie en sortie le nom de la personne la plus grande.

Que remarquez-vous ?

Correction :

```python
# Algorithmique et Programmation — Master Droit & Informatique — UPEC
# Luc Pellissier 2022
# CC0

from typing import List

def plusGrand(noms:list[str], tailles:list[int]) -> str:
    """Renvoie le nom de la personne ayant la plus grande taille"""
    maxi = tailles[0]
    personne = noms[0]
    longueur = len(noms)

    for i in range(1,longueur):
        if tailles[i] > maxi:
            maxi = tailles[i]
            personne = nom[i]

    return personne
```

Comme on nous a garanti dans l'énoncé que les deux listes faisaient la même taille, on ne le vérifie pas.

Par contre, on remarque que la fonction qu'on a écrit ne satisfait pas l'énoncé : s'il y a plusieurs personnes faisant la même taille et étant collectivement les plus grandes du groupe, on ne renvoie que le nom de la première. En vérité, l'énoncé est absurde : il nous demande de renvoyer un seul nom, mais celui-ci est mal défini.

Sinon, on peut aussi remarquer que les différences de taille entre les membres du groupe ABBA sont faibles. ☺

Exercice 78. En faire une trace d'exécution sur l'exemple ci-dessus.

Correction :

ligne	suivante	maxi	personne	longueur	i
7	9				
9	10	177			
10	11	177	'Benny Andersson'		
11	13	177	'Benny Andersson'	4	
13	14	177	'Benny Andersson'	4	1
14	13	177	'Benny Andersson'	4	1
13	14	177	'Benny Andersson'	4	2
14	13	177	'Benny Andersson'	4	2
13	14	177	'Benny Andersson'	4	3
14	13	177	'Benny Andersson'	4	3
13	18	177	'Benny Andersson'	4	
18	**retourner** "Benny Andersson"				

☺

Couples

En python, il est possible de stocker un *couple* de valeurs dans la même variable. Ainsi, si j'écris :

```
1 a : tuple[str,int] = ("Benny Andersson", 177)
```

la variable a contiendra deux parties, auxquelles on accède par l'opérateur []. Ainsi, a[0] vaudra "Benny Andersson" et a[1] vaudra 177. Elle a le type tuple[str,int], signifiant que sa première partie est un str et sa seconde un int.

Exercice 79. Écrire une fonction qui prenne en entrée une liste contenant des couples dont le premier élément est une chaîne de caractère et le second est un nombre, liste que l'on comprend comme contenant les noms et tailles de personnes, et qui renvoie un couple constitué d'un nombre (la plus petite taille) et d'une liste de chaînes de caractères (la liste des noms des plus petites personnes).

Ainsi, on voudra que, exécutée sur

```
1 [("Benny Andersson",177), ("Agnetha Fältskog",172),
2  ("Anni-Frid Lyngstad",172), ("Björn Ulvaeus",175)]
```

le résultat de la fonction puisse être

```
1 (172, ["Agnetha Fältskog", "Anni-Frid Lyngstad"])
```

Exercice 80. En faire une trace d'exécution sur la liste ci-dessus.

Tri

On va considérer qu'on veut trier des enregistrements constitués de couples, et que la clef (un nombre entier) est située en deuxième position du couple, tandis que la première est une chaîne de caractères.

Exercice 81. Donner le type d'un programme de tri triant dans ce cas.

Correction : Le programme prend en entrée un list[tuple[str,int]] et renvoie la même chose en sortie. ☺

Exercice 82. Le programmer.

B.2 2022 → 2023 — Deuxième

Tri

On se dit qu'on peut tenter la stratégie suivante pour trier un tableau à n éléments :

- on cherche dans un tableau, entre l'élément 0 et n, le plus petit élément ;
- on l'échange avec l'élément en position 0 ;
- on recommence, en cherchant le plus petit élément entre l'élément 1 et n ;
- et ainsi de suite.

Exercice 83. Écrire une fonction trouvant le plus petit élément d'un tableau entre deux bornes.

Correction : La fonction doit prendre en entrée trois paramètres : le tableau lui-même, la borne de début et la borne de fin. On va supposer que le tableau contient des entiers positifs. Si on donne des bornes absurdes, on renvoie -1. Donc :

```
1  def petitEntreBornes(tableau: list[int], debut: int, fin: int) -> int:
2
3      """Trouve le plus petit element entre debut et fin, renvoie -1 si les bornes
4      sont absurdes"""
5
6      n = len(tableau)
7
8      if (debut < 0) or (fin > n) or (fin < debut):
9          return -1
10
11     tmp = tableau[debut]
12
13     for i in range(debut, fin):
14         if tableau[i] < tmp:
15             tmp = tableau[i]
16
17     return tmp
```

☺

Exercice 84. Écrire une fonction échangeant deux éléments d'un tableau.

Correction : La fonction doit prendre en entrée un tableau et deux positions et les échanger. Elle ne fait rien si les positions sont absurdes.

```
1  def echange(tableau: list[Any], pos1: int, pos2: int) -> list[Any]:
2      """échange tableau[pos1] et tableau[pos2]"""
3
4      if pos1 < 0 or pos2 < 0 or pos1 > len(tableau) or pos2 > len(tableau):
5          return tableau
6
7      tmp = tableau[pos1]
8      tableau[pos1] = tableau[pos2]
9      tableau[pos2] = tmp
10
11     return tableau
```

☺

Exercice 85. Écrire la fonction de tri.

Correction :

```
1  def triParRecherche(tableau: list[int]) -> list[int]:
2
3      """trie en échangeant le plus petit élément et celui en 0-ème position et
4      ainsi de suite"""
5
6      N = len(tableau)
7      debut = 0
8
9
10     while debut < N:
11         tmp = tableau[debut]
12
13         for i in range(debut, fin):
14             if tableau[i] < tmp:
15                 tmp = tableau[i]
16                 j = i
17
18         tableau[j] = tableau[debut]
19         tableau[debut] = tmp
20         debut = debut + 1
21
22
23     return tableau
```

☺

Exercice 86. Donner une trace de l'exécution du tout sur

30	5	9	30
[0]	[1]	[2]	[3]

Exercice 87. Donner la complexité de cet algorithme de tri. Qu'en concluez-vous ?

Transposition

Remarque : tout ce qui suit est une version extrêmement simplifiée de la théorie musicale.

Les notes de la gamme diatonique sont, dans l'ordre **do**, **ré**, **mi**, **fa**, **sol**, **la**, **si**. Si on ne prend pas en compte la hauteur, on peut considérer que ces notes bouclent, et donc qu'au dessus du si, il y a de nouveau un do.

Une mélodie écrite dans cette gamme peut donc être représentée comme une suite de ces notes avec une durée, dès qu'on fixe un intervalle arbitraire. Par exemple,

va être représenté par :

— do pour une durée de 1
— mi pour une durée de 3
— mi pour une durée de 2
— mi pour une durée de 1
— la pour une durée de 3
— la pour une durée de 3
— mi pour une durée de 1
— do pour une durée de 1
— la pour une durée de 1
— mi pour une durée de 1
— re pour une durée de 12

Exercice 88. En ayant lu la suite des exercices, proposer une structure de données pour représenter une telle mélodie et une fonction prenant une mélodie et l'affichant sous le format ci-dessus.

Exercice 89. Un problème qu'on a souvent en composition est de *transposer*, c'est-à-dire de jouer en partant d'une autre note, mais en respectant les mêmes intervalles : c'est-à-dire que si l'intervalle entre la première et la deuxième note est de deux degrés avant transposition, il doit aussi être de deux degrés après, la transposition se fait uniformément sur toute la partition.

Écrire une fonction qui prenne en entrée trois notes, chacune représentée comme vous voulez, et qui transpose la troisième en transformant la première en la deuxième.

Écrire une fonction qui prenne en entrée un morceau (tel que représenté au-dessus) et une note N, et qui renvoie le morceau transposé où la première note est remplacée par N.

B.3 2023 → 2024 — DM 1

Consignes : Ce devoir est à rendre pour le 19 janvier 2023 à 24:00 (heure de Paris). Vous pouvez travailler à deux, à condition que votre rédaction soit différente, et que ce soit clairement indiqué et cohérent. La solution n'est jamais d'inventer de la syntaxe nouvelle. Chercher la solution sur Internet ou en la demandant à quelqu'un d'autre n'a aucun intérêt. Amusez-vous bien !

Indication syntaxique : Si l et l' sont deux listes, l+l' est une nouvelle liste, contenant les éléments de l suivis des éléments de l'.

Citius, Altius, Fortius

Exercice 90. Écrire une fonction qui renvoie la position du plus grand élément d'un tableau (on suppose que cet élément existe et est unique).

On veut en particulier avoir les entrées et les sorties de cette fonction.

Correction : Ce programme a en entrée une liste et en sortie un nombre entier.

```
1  def position_max(l):
2      """Retourne la position du plus grand élément de l"""
3
4      a = l[0]
5      position = 0
6
```

```
7     for i in range(1,len(l)):
8         if l[i] > a:
9             a = l[i]
10            position = i
11
12    return position
```

☺

Exercice 91. L'exécuter sur le tableau [5,3,8,5,9].

Exercice 92. En déduire une fonction donnant la valeur du plus grand élement d'un tableau.

Correction : Il y a plusieurs possibilités, la plus simple est de réutiliser la fonction ci-dessus. On obtient une fonction prenant en entrée une liste (d'éléménts pouvant être comparés par >) et renvoyant un tel élément.

```
1  def max(l):
2      """Retourne le plus grand élément de l"""
3
4      return l[position_max(l)]
5
```

☺

Exercice 93. On veut ne pas avoir besoin de supposer qu'il y a un unique plus grand élement. Écrire une fonction qui renvoie les positions de toutes les occurrences du pland élément du tableau.

Correction : Cette fois cette fonction prend en entrée une liste et renvoie une liste d'entiers.

```
1  def positions_max(l):
2      """Retourne la liste des positions des plus grands éléments"""
3
4      positions = [0]
5      a = l[0]
6
7      for i in range(1,len(l)):
8          if l[i] = a:
9              positions = positions + [i]
10         elif l[i] > a:
11             a = l[i]
12             positions = [i]
13
14     return positions
```

☺

Exercice 94. L'exécuter sur le tableau [5,3,8,5,8].

Exercice 95. On suppose maintenant que le tableau est trié. Comment peut-on simplifier ces trois fonctions ? Les ré-écrire dans ce cadre.

Correction :

```
1  def position_max(l):
2      """Retourne la position du plus grand élément d'une liste triée l"""
3
4      return len(l)-1
5
6  def max(l):
7      """Retourne le plus grand élément d'une liste triée l"""
```

```
8
9       return l[len(l)-1]
10
11  def positions_max(l):
12      """Retourne la liste des positions des plus grands éléments d'une liste triée"""
13
14      positions = [len(l)-1]
15      a = l[len(l)-1]
16
17      for i in range(2,len(l)):
18          if l[len(l)-i] = a:
19              positions = positions + [i]
20          else:
21              return positions
```

☺

Retourner

Exercice 96. Écrire une fonction prenant en entrée un tableau et retournant le tableau contenant les mêmes élements, mais dans l'ordre inverse (évidemment, sans servir d'une éventuelle fonction faisant déjà ça...).

Correction : Cette fonction prend en entrée une liste et renvoie une liste, qui contient le même type d'éléments.

```
1   def renverse(l):
2       """Renverse la liste l"""
3
4       resultat = []
5
6       for i in range(0,len(l)):
7           resultat = [l[i]] + resultat
8
9       return resultat
```

☺

Exercice 97. L'exécuter sur le tableau [5,3,8,5,9].

B.4 2023 → 2024 — Devoir 1

Pour chaque fonction, on attendra en particulier ses entrées et sortie. On s'attend des fonctions qu'elles aient une description, des commentaires,...

Échanger

Exercice 98. Écrire une fonction echange qui prenne en entrée un tableau et deux positions et qui retourne le tableau dont les deux éléments des deux positions sont échangés.

Exercice 99. L'exécuter sur le tableau ["d","c","b","a"], et les positions 0 et 2.

L'exécuter sur le tableau ["d","c","b","a"], et les positions 2 et 0.

Exercice 100. Que se passe-t-il si on l'exécute avec des entrées n'ayant pas de sens ? Soyez précis·es.

Bulles

On s'intéresse à un autre tri, le tri à bulles.

Exercice 101. Écrire une fonction remonteBulles prenant en entrée un tableau qui compare chaque paire d'éléments consécutifs du tableau : si deux éléments consécutifs ne sont pas dans l'ordre croissant, on les échange. Par exemple, si on rencontre 7 suivi de 6, on les échange.

Exercice 102. L'exécuter sur le tableau [9,6,30,1,30,7].
Indication : si votre fonction fait appel à la fonction echange ci-dessus, on considèrera que chaque appel à cette fonction se fait en une étape.

Exercice 103. Sans faire de trace d'exécution, donner les résultats de
- remonteBulle([9,6,30,1,30,7])
- remonteBulle(remonteBulle([9,6,30,1,30,7]))
- remonteBulle(remonteBulle(remonteBulle([9,6,30,1,30,7])))
- remonteBulle(remonteBulle(remonteBulle([9,6,30,1,30,7])))
- remonteBulle(remonteBulle(remonteBulle(remonteBulle(remonteBulle([9,6,30,1,30,7
- remonteBulle(remonteBulle(remonteBulle(remonteBulle(remonteBulle(remonteBulle

En déduire un algorithme de tri utilisant remonteBulle.

Exercice 104. Quelle est la complexité de l'algorithme ainsi programmé ?

Cocktail

Considérons le programme suivant :

```
1  def cocktail(t):
2      drapeau = True
3      debut = 0
4      fin = len(t)-1
5      while drapeau==True:
6
7          drapeau = False
8
9          for i in range(debut, fin):
10             if t[i] > t[i+1]:
11                 echange(t,i,i+1)
12
13         fin = fin-1
14
15         for i in range(0,fin-debut+1):
16             if t[fin-i] > t[fin-i+1]:
17                 echange(t,fin-i,fin-i+1)
18                 drapeau = True
19
20         debut = debut+1
21
22     return t
```

Exercice 105. L'exécuter sur [3,0,28,2,1,3].

Exercice 106. Que fait ce programme ? Comment fonctionne-t-il ? Quelle est sa complexité ?

Bibliographie

Georges Gonthier (2008). « Formal Proof—The Four-Color Theorem ». In : *Notices of the AMS* **55** (11), p. 1382-1393.

Donald E. Knuth (avr. 1976). « Big Omicron and Big Omega and Big Theta ». In : *SIGACT News* **8.2**, p. 18-24.

Kazimierz Kuratowski (1930). « Sur le problème des courbes gauches en Topologie ». In : *Fundamenta Mathematicæ* **15** (1), p. 271-283.

Georg Simmel (1900). *Philosophie des Geldes*.

— (1987). *Philosophie de l'argent*. Trad. par Sabine Cornille et Philippe Ivernel. Original : (Simmel 1900).

Klaus Wagner (1937). « Über eine Eigenschaft der ebenen Komplexe ». In : *Mathematische Annalen* **114** (1), p. 570-590.

Index

Liste des Algorithmes

Table des matières