

Algorithmique

Luc Pellissier

2020 → 2022

Avertissement

Ce cours s'adresse aux étudiants du master « Droit du numérique » de l'Université Paris-Est Créteil Val-de-Marne. Il cherche à introduire à des étudiant·es n'ayant pas étudié des mathématiques depuis plus d'une dizaine d'années des notions d'algorithmique, et voir comment les algorithmes s'insèrent dans la société et la pensée algorithmique porte un éclairage nouveau sur des sujets connus.

On écrira donc des algorithmes, mais on les exécutera aussi à la main, on prouvera leur correction, les programmera et les exécutera sur machine, et enfin nous nous interrogerons sur leur complexité. La correction et la complexité nous amènera à nous interroger sur ce que nous désirons que ces algorithmes fassent: on devra donc étudier les spécifications que l'on peut désirer. On verra en particulier que si pour certains problèmes on a des spécifications satisfaisantes, et des algorithmes les satisfaisant avec un usage de ressource raisonnable, ce n'est pas toujours le cas — soit que les spécifications soient élusives, soit qu'il n'y ait pas de manière de raisonnablement les satisfaire.

L'algorithmique étant la science des algorithmes, c'est-à-dire ce qu'on peut écrire sous forme de programme sans pour autant le programmer, ce cours est très relié avec le cours de PROGRAMMATION (on peut penser à la programmation comme à de la grammaire, et l'algorithmique plus comme de la narration). Il s'appuie aussi sur le cours de LOGIQUE, aussi bien parce qu'on va prouver des théorèmes que par l'affinité entre les notions de calcul et de démonstration. Enfin, il ouvre sur le cours de FONDEMENTS DE L'INFORMATIQUE qui traite plus spécifiquement de calculabilité et de complexité.

Ce cours peut être cité avec :

```
1 @report{2020:Pellissier:Algorithmique,
2   title =      {Algorithmique},
3   author =     {Pellissier, Luc},
4   langid =     {fr},
5   type =       {Notes de cours}
6 }
```

Il est placé sous licence Creative Commons Attribution-Partage à l'Identique (CC BY-SA)¹.

REMERCIEMENTS

Merci à Océane Gerriet qui a trouvé une erreur dans l'Algorithme 25.

Merci à Claire Leroy et Lovina Guibe pour des coquilles.

Merci à Florian Pesce et Fatih Yilmaz qui ont trouvé des erreurs dans des corrigés.

Toute remarque ou erreur est à signaler à luc.pellissier@lac1.fr

1. C'est-à-dire qu'il est possible de reproduire et faire des modifications arbitraires, tant que l'auteur original est cité.

SOMMAIRE

I Fondamentaux	3
1 Algorithmes, programmes, fonctions, données	5
2 Structures de données 1	25
3 Tris	65
4 Faire des choix	115
5 La pensée algorithmique	149
II Annexes	159
A Compléments de logique	161
B Compléments sur le langage C	163
C Graphes de données expérimentales	167
D Examens	169
Bibliographie	195
Index	197
Liste des algorithmes	199
Table des matières	200

Première partie

Fondamentaux

Algorithmes, programmes, fonctions, données

On étudie un peu la notion de calcul sur quelques exemples – notamment celui de la multiplication, constate qu'elle est problématique. On donne une définition d'algorithme et voit le genre de questions qui vont nous occuper.

1.1	Fonctions	6
1.2	Programmes	7
1.3	Multiplication	8
	Multiplication par addition itérée	8
	Multiplication posée	10
1.4	Algorithmes	12
1.5	Notations	15
1.6	Exécution	17
1.7	Résumé	17
1.A	L'algorithme d'Euclide	19
1.B	PGCD	21
	La division euclidienne	22
	Exécution	23
	Terminaison	23
	Correction	23
	Programmation	23

L'*'algorithmique*¹ (déformation assez lointaine du nom du poète persan Abū 'Abdallāh Muhammad ibn Mūsā al-Khwārizmī) est la science répondant à la question « comment calculer efficacement ? ». Pour comprendre cette question — et donc peut-être ses réponses, commençons par expliciter ces termes.

1.1 FONCTIONS

Intuitivement, un calcul part de certaines données pour produire un résultat. Une première tentative pour expliciter la notion de calcul peut venir de la notion de fonction. Une *fonction* est une machine qui associe, à un élément (*son argument*), un autre élément, *son résultat* : ainsi, la notion de fonction est intimement liée à celle de résultat.

Une fonction peut n'être définie que pour certains éléments.

Exemple 1.

1. Considérons la fonction *naissance_g*, qui associe, à chaque individu, sa date de naissance dans le calendrier grégorien. Par exemple, elle va associer la date du 26 mai 1986 à la personne « Gandhi Djuna ».

C'est une fonction bien définie pour chaque humain né (vivant ou mort, d'ailleurs) : ce n'est pas parce que la date de naissance de certaines personnes est inconnue qu'elle n'existe pas — et n'est pas unique.

2. Inversement, la fonction *mort* associant à chaque individu la date de sa mort n'est bien définie que sur les individus morts.

3. La fonction *naissance_j*, qui associe à chaque individu sa date de naissance dans le calendrier julien² est aussi une fonction ; elle n'est pas égale à la fonction *naissance_g* : à la personne « Gandhi Djuna », elle va associer la date du 13 mai 1986.

Le fait que beaucoup d'opérations soient identiques (par exemple, le calcul de l'âge d'une personne à un moment donné ne dépend pas du calendrier dans lequel on se place) si on change de calendrier doit nous faire nous interroger sur le fait que les calendriers ne sont pas fondamentaux, et qu'il doit pouvoir être définie une notion abstraite de date que l'on présenterait au moyen de calendriers.

4. L'opération qui à chaque individu associe son âge n'est pas une fonction. En effet, l'âge ne dépend pas uniquement de l'individu, mais aussi de la date à laquelle l'âge est calculé. Par contre, on peut considérer que l'âge est une fonction de l'individu et de la date courante.

5. L'opération qui, à tous les habitant·es d'une région associe le·a premier·e n'est pas une fonction. En effet, si les individus sont listés, par exemple, par ordre alphabétique, date de naissance décroissante, ordre protocolaire,...la notion aurait du sens, mais l'ordre est une propriété extrinsèque de l'ensemble des habitant·es, et dépend de la présentation.

Ces exemples nous font prendre conscience de la propriété fondamentale d'une fonction : elle est définie de façon univoque pour chaque élément de son domaine de définition, et sa valeur ne dépend ni d'éléments extérieurs, ni de la présentation d'un élément dans le domaine de définition.

Définition 1 (fonction). Une *fonction* est la donnée de :

- un ensemble de départ D, ou *domaine de définition* ;
- un ensemble d'arrivée E ;

1. On trouve parfois la variante *algorithmie*.

2. On rappelle que, si le calendrier grégorien est le calendrier officiel en France depuis le 20 décembre 1582, il a remplacé un autre calendrier, le calendrier julien (la principale différence est le nombre d'années bissextiles : par exemple, 1900 n'est pas bissextile pour le calendrier grégorien, mais l'est pour le calendrier julien). Il est encore utilisé par la majorité des églises orthodoxes, la république monastique du Mont-Athos et par les Imazighen.

— une correspondance entre les éléments de D et ceux de E telle que chaque élément de D entre exactement une fois en correspondance avec un élément de E.

Si f est une telle fonction, on notera, pour un élément x de D, $f(x)$ l'unique élément lui correspondant, et on l'appellera *l'image de x par f*.

On notera souvent une telle fonction

$$\begin{aligned} f: D &\rightarrow E \\ x &\mapsto f(x) \end{aligned}$$

Ainsi la multiplication est une **fonction** de l'ensemble des couples de nombres entiers (que l'on note en général \mathbf{N}^2) dans l'ensemble des nombres entiers (que l'on note en général \mathbf{N}), qui à chaque couple de nombres (m, n) associe leur produit $m \times n$, ou, en symboles :

$$\begin{aligned} \times: \mathbf{N}^2 &\rightarrow \mathbf{N} \\ (n, m) &\mapsto n \times m \end{aligned}$$

Exercice 1. Pour chacun des points de l'Exemple 1, donner leurs ensembles de départ et d'arrivée, et les écrire sous forme symbolique.

Il est très utile en mathématiques de considérer qu'il n'y a aucune condition particulière sur la correspondance entre les arguments et les résultats, tant qu'elle est univoque. Cela permet d'avoir des théorèmes de régularité très pratiques. Ainsi, on formalise souvent la correspondance comme un ensemble de couples (x, y) d'un élément de l'ensemble de départ et un de celui d'arrivée.

On voit donc que des fonctions avec des correspondances arbitrairement compliquées, pas forcément calculables peuvent exister. Par exemple, la fonction de l'ensemble $\{1, 2, \dots, 365\}$ dans les nombres entiers qui à chaque jour de l'année 2030 associe le nombre de personnes ayant dépassé la limite autorisée sur les routes de France est parfaitement définie, mais pour autant, on serait bien en peine de la calculer.

Les **fonctions** ne sont donc pas suffisantes pour approcher la notion de calcul.

1.2 PROGRAMMES

À partir du début du xx^{ème} siècle, après la formalisation de la notion de fonction, on a cherché à trouver une restriction aux fonctions *calculables*. De ces travaux, auxquels on associe les noms de Turing, Gödel, Ackerman, Church et Post, est surgit un consensus sur la notion de fonction calculable. Nous n'allons pas entrer dans les détails : en effet, s'il y a consensus sur la notion, il n'y en n'a pas sur sa définition, et toutes les définitions *raisonnables* – on mesure le flou du terme... – définissent la même classe de fonctions.

On procède de la sorte : on définit d'abord un **modèle de calcul**, c'est à dire un ensemble d'opérations que l'on s'autorise à faire, des manières dont on s'autorise de les composer, et une procédure pour entrer des données et récupérer des résultats une fois le calcul terminé. Dans chaque tel modèle, on peut définir des **programmes** (c'est-à-dire des listes concrètes d'opérations autorisées par le modèles, composées telles que spécifiées par le modèle et dont les données sont rentrées et récupérées selon les règles). Certains programmes calculent effectivement des fonctions — on appelle une fonction telle qu'il existe un programme la calculant une **fonction calculable** dans ce modèle.

Vous étudiez un de ces modèles en PROGRAMMATION STRUCTURÉE : le langage C — un langage de programmation. Il est de bon aloi de s'entraîner à programmer chaque fonction que l'on pense calculable.

Parmi les modèles de calcul définis au cours des années 1930, on note la machine de Turing, les fonctions récursives et le λ -calcul. La preuve que ces trois modèles, au formalisme extrêmement différents, définissaient la même classe de fonctions calculables des entiers dans les entiers a amené Church à postuler :

Postulat 1 (thèse de Church). *Les fonctions des entiers dans les entiers calculables par un dispositif physique ou action consciente d'un humain sont exactement les fonctions récursives (qui sont aussi celles λ -calculables, ou calculables par une machine de Turing, ou encore programmable en C).*

Ce postulat a donc un statut réellement métaphysique — c'est-à-dire qu'il porte au-delà de la connaissance qu'on a du monde. Il a des conséquences nombreuses en physique ou sur la possibilité d'une théorie de la connaissance, mais nous ne les explorerons pas. Disons simplement qu'il justifie le choix arbitraire de n'importe quel modèle de calcul (encore une fois, tant qu'il est raisonnable, c'est-à-dire équivalent aux fonctions récursives en terme d'expressivité).

1.3 MULTIPLICATION

Comme premier exemple de programme, intéressons-nous à la multiplication.

Multiplication par addition itérée

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.

Une *variable* est un nom que l'on donne, dont la valeur peut changer (varier...) au cours du temps. On fait une 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

$$x \leftarrow e$$

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.^b

On va noter, si e_1 et e_2 sont deux expressions

$$e_1 \equiv e_2$$

le *test*, qui va calculer chacune des deux expressions, et valoir vrai (que l'on notera v) si leurs valeurs sont égales, et faux (f) sinon.^c

^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.

^b. En C, cela se note `x = e;`

^c. En C, cela se note `e1 == e2;`

Exercice 2. Considérons la suite d'assignments et de tests suivantes :

Donner, sous forme d'un tableau, ligne à ligne, ce que valent les différentes variables. On appellera ce genre de tableau une *trace d'exécution* et nous en ferons un grand usage.

```

1 x ← 1
2 y ← 'Bonjour'
3 y ← 1
4 x ≡ y
5 x ← 2
6 x ≡ y
7 z ← (x ≡ y)
8 y ← y
9 t ← (y ≡ y)
10 t ≡ z
```

On peut donc procéder à des additions successives et les compter en faisant quelque chose comme :

```

1 résultat ← résultat + n
2 compteur ← compteur + 1
```

Exercice 3. Écrire un programme (en pseudo-code) tel que, à la fin de son exécution, la variable résultat contienne le produit de la valeur de la variable *n* et de 5.

Écrire le même programme en C.

Exercice 4. Donner une trace d'exécution de ce programme, en fixant *n* := 3, et une en fixant *n* := 8.

Le problème est que l'on souhaite répéter cette opération *m* fois, c'est-à-dire un nombre variable de fois, dépendant des nombres que l'on souhaite multiplier : on ne va pas écrire un programme différent pour chaque nombre.

Pour cela, on va se donner une autre construction : la construction de *boucle*, qui permet de répéter une même action.

```

1 pour i allant de a à b faire
2   |
3   ...
```

assigne à la variable *i* la valeur de *a*, exécute le *corps de la boucle* (les pointillés), incrémente *i*, et, si la nouvelle valeur de *i* est strictement inférieure à la valeur de *b*, continue.

On peut maintenant facilement donner un programme calculant une telle addition itérée. Par exemple en C :

```

1 int multiplication_naive(int n, int m){
2   int nm = 0;
3   if (m < 0)
4     return 0;
5   else
6   {
7     for(int i = 0; i < m; i++) {
8       nm = nm + n;
9     }
10    return nm;
11  }
12 }
```

Remarque 1. En C, les nombres entiers de type `int` peuvent être positifs ou négatifs : avec un nombre négatif, cette boucle continue à l'infini, on teste donc la positivité de `m` avant de commencer.

Ce qui, écrit sous une forme de pseudo-code donne :

Entrées: deux nombres entiers positifs a et b .

```

1 Multiplication( $a, b$ )
2   résultat  $\leftarrow 0$ 
3   pour  $i$  allant de 0 à  $a$  faire
4     |   résultat  $\leftarrow$  résultat +  $b$ 
5   fin
6   retourner résultat
```

Sorties: un nombre entier positif

Algorithme 1 : Multiplication par addition itérée

On a noté le programme sous forme d'une *procédure*, c'est-à-dire qu'on a spécifié — donné un nom et des informations sur leur type — les entrées (ce dont le programme a besoin pour tourner) et les sorties (le résultat que l'on cherche à obtenir).

On le fera systématiquement.

Exercice 5. Quelles sont les entrées, les sorties et les variables de l'Algorithme 1 ?

Exercice 6. Faire une trace d'exécution pour `Multiplication(5, 8)`, puis en faire une seconde pour `Multiplication(8, 5)`. Que remarquez-vous ?

Exercice 7. Faire une trace d'exécution de `Multiplication(5, 0)`.

Peut-on en faire une trace d'exécution de `Multiplication(-5, 0)` ?

Déjà, on peut se poser un certain nombre de questions sur ce programme :

- est-ce qu'il calcule bien quelque chose (sans erreur) pour tous les couples d'entiers positifs ?
- est-ce qu'il est rapide ?

De plus, si on prend ce programme comme la définition de la multiplication, on peut se demander si vraiment les trois écritures (une en notation mathématique, une en C, et la dernière en pseudo-code) sont bien la même définition.

Exercice 8. Considérons l'algorithme suivant. Quelles sont ses variables ? Ses entrées ? Faire une trace d'exécution pour `Mystère(3, 5, 2)`.

Multiplication posée

Néanmoins, même sans notion de *complexité*, on se rend bien compte que c'est très lent, et dès le CE2³, on apprend une autre technique de *multiplication*. Regardons-la dans un manuel :

1. On place les deux nombres l'un en dessous de l'autre.
2. On multiplie le chiffre des unités du nombre du bas par le nombre du haut. Pour ce faire :
 - a) On multiplie le chiffre des unités du nombre du bas avec le chiffre des unités du chiffre du haut, résultat que l'on a appris par cœur.
3. En France en tout cas, mais dans l'enseignement primaire un peu partout.

Entrées: deux nombres entiers positifs a , b et c .

```

1 Mystère( $a$ ,  $b$ ,  $c$ )
2   inter  $\leftarrow$  0
3   pour  $i$  allant de 0 à a faire
4     |   inter  $\leftarrow$  inter +  $b$ 
5   fin
6   resultat  $\leftarrow$  0
7   pour  $j$  allant de 0 à c faire
8     |   resultat  $\leftarrow$  resultat + inter
9   fin
10  retourner resultat

```

Sorties: un nombre entier positif

\times	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

FIGURE 1.1 – Table de multiplication en base 10

- b) Si le résultat est plus petit que 10, on l'écrit en chiffre des unités; sinon, on écrit le chiffre des unités et on note les dizaines comme retenues.
- c) On effectue la multiplication suivante en additionnant les retenues. S'il n'y a pas de multiplication suivante, écrire juste la retenue comme résultat.

Tout ceci est écrit sur une ligne.

3. Procéder de même pour chaque autre chiffre du nombre du bas, en décalant à chaque fois d'un nombre.
4. Procéder à l'addition de tous les nombres obtenus.

On voit que cette procédure est tout de même assez compliquée: elle nécessite de savoir noter des informations (les retenues, les résultats intermédiaires, le résultat final), de faire des additions, et aussi de connaître par cœur les produits des nombres à un chiffre.

Le dernier point nécessite de s'y attarder: si l'on veut programmer cela, on a deux possibilités:

- soit on écrit explicitement dans notre programme les tables de multiplication des nombres à un chiffre, c'est-à-dire tout le contenu de la Figure 1.1 ;
- soit, avant de calculer des multiplications concrètes, on commence par calculer les résultats des multiplications des nombres à un chiffre (par exemple par additions itérées).

Enfin, elle nécessite aussi d'être capable de placer les nombres l'un en dessous de l'autre; ce qui est légèrement choquant: un nombre étant a priori un concept, il ne peut pas à proprement parler être placé.

On touche ici du doigt une deuxième difficulté: en vérité, l'algorithme de la multiplication que l'on apprend en primaire n'opère pas sur des nombres, mais sur des suites de chiffres: il est dépendant de la manière dont sont présentées les données — ici, en notation positionnelle en base 10. Pour s'en convaincre, il suffit d'essayer de faire une multiplication avec un nombre écrit en chiffres romains. À ce sujet, (IFRAH 1981) rapporte une anecdote du xv^{ème} siècle sur un expert conseillant un marchand sur comment éduquer son fils:

Si vous voulez vous contenter de lui faire apprendre la pratique des additions ou des soustractions, n'importe quelle université allemande ou française fera l'affaire. En revanche, si vous tenez à pousser son instruction jusqu'à la multiplication ou la division (si tant est qu'il en soit capable!), alors il vous faudra l'envoyer dans les écoles italiennes.

Quelle que soit la véracité de l'anecdote (non-sourcée, et qui ne me semble pas correspondre à la vérité des calculs de l'époque), elle est un cas particulier d'une vérité frappante: selon la représentation des données, certains calculs vont être très efficaces, ou au contraire, très complexes.

Pour en revenir au cas de la [multiplication](#), on peut raffiner notre image du rapport entre les fonctions et les programmes: si les fonctions peuvent être définies à n'importe quel niveau d'abstraction, les programmes prennent en compte la manière dont les données sont présentées. Comme un algorithme est une idéalisation d'un programme, c'est aussi son cas. Ici, donc, la fonction produit est une fonction des paires d'entiers dans les entiers; tandis que l'algorithme de la multiplication est un algorithme des paires de listes de chiffres dans les listes de chiffres. On dispose de plus de deux correspondances, assurant le codage et le décodage, permettant de passer d'un nombre (abstrait) à sa représentation (concrète). Ainsi, on est dans la situation du diagramme de la Figure 1.2: l'algorithme du produit travaille sur des données structurées représentant les informations.

Quand on exécute à la main la technique de la [multiplication](#), on voit que les opérations élémentaires se font sur les chiffres, et que les nombres se font décomposer en unités, dizaines, centaines,... c'est-à-dire en une liste de chiffres. De plus, comme toutes les opérations se font en commençant par les unités, puis peu à peu, se déplaçant vers les chiffres de poids fort⁴, on choisit de stocker les chiffres en partant de celui des unités, puis celui des dizaines, etc. En d'autres termes, nous sommes petit-boutien⁵ (comme certains habitants de Lilliput (SWIFT 1726)).

Pour écrire le programme en entier, on aura besoin de pouvoir manipuler des données plus complexes que des nombres (c'est-à-dire, ici, des listes de chiffres). On fera ça en ??.

1.4 ALGORITHMES

Gesetze sind wie Würste, man sollte besser nicht dabei sein, wenn sie gemacht werden.

— Otto von Bismarck (1815–1898), 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. Par exemple, dans le cas de la multiplication, on voit bien qu'on peut écrire *la même chose* dans plusieurs modèles et que, aux détails d'implémentation près, les idées derrière ces trois programmes sont les mêmes et 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.

4. Dans un nombre, les chiffres de *poids fort* sont ceux qui contiennent le plus d'information sur le nombre: le chiffre des dizaines a un poids plus fort que celui des unités,...

5. Cette question est loin d'être académique! Les architectures des différents processeurs sont au choix, petit- ou gros-boutiens.

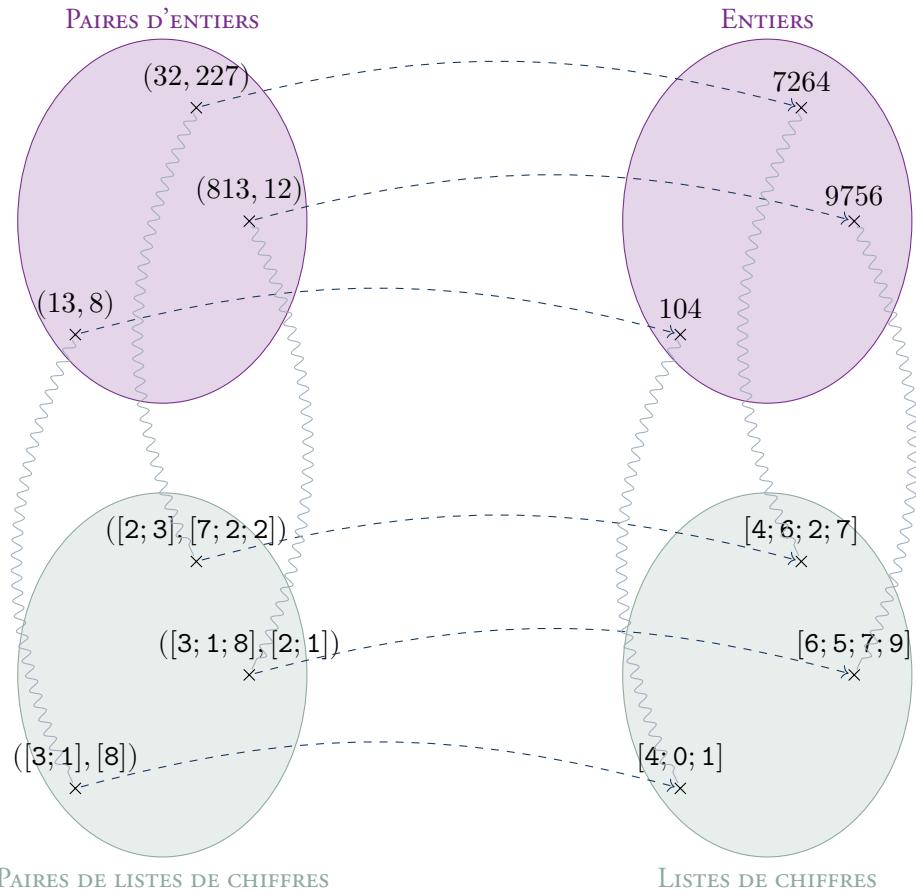


FIGURE 1.2 – Diagramme sagittal de la fonction produit et l'algorithme de la multiplication

Définition 2 (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 1.3).

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 terminé.

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 français⁶, et les langages de programmation ont été inventés pour être explicite. Néanmoins, on

6. Ce n'est pas à des juristes que je vais faire semblant d'apprendre que des formulations peuvent se révéler ambiguës

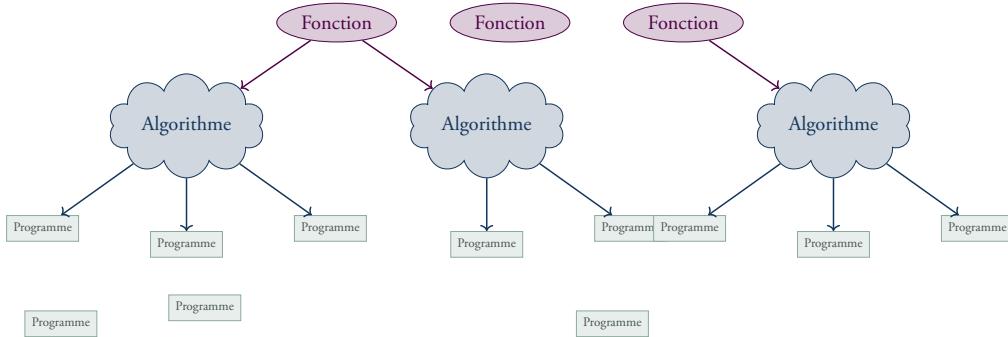


FIGURE 1.3 – Fonctions, algorithmes, programmes

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.

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 2. On compare souvent un algorithme à une recette de cuisine. On va ici utiliser une autre métaphore, inspirée de la citation de Bismarck 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.

parfois longtemps après leur écriture.

1.5 NOTATIONS

Il n'y a pas de manière complètement standard de présenter un algorithme (vu qu'il n'y en a pas de bonne définition). La seule bonne manière d'en présenter un serait de donner un programme, ce qui suppose de s'intéresser à des détails fastidieux d'un modèle de calcul particulier. Certains livres d'algorithmique font ce choix, certains inventent même un langage ! Néanmoins, la convention usuelle est plutôt de présenter les algorithmes sous la forme de *pseudo-code*, c'est-à-dire un programme pour un modèle de calcul qui n'existe pas, suffisamment souple pour accepter ce que l'on veut — tant qu'on est parfaitement précis.

La liste des conventions que l'on va utiliser ne sert qu'à ça : être parfaitement précis et se forcer à n'écrire que des choses que l'on sait pouvoir expliciter jusqu'au bout. En particulier, pour chaque instruction, on se doit de décrire, même sommairement, comme l'exécuter.

entrées On commence chaque algorithme par un ligne décrivant toutes les entrées, leurs noms, leur type (des nombres, des vecteurs,...); ainsi que les contraintes, s'il y en a, sur ces entrées. Par exemple, si l'algorithme suppose, pour fonctionner, qu'une de ses entrées est inférieure à une autre, il faut l'écrire.

sorties de même, on écrit le type et les contraintes sur les sorties. Quand on doit chaîner des algorithmes, c'est une information indispensable.

nom on donne un nom à la procédure, pour pouvoir s'en servir dans le reste de la procédure, mais aussi dans le texte, et dans les autres procédures. J'écris le nom en une police à **chasse fixe**, avec la première lettre en capitale et tout le reste en bas-de-casse, suivi entre parenthèse, des entrées.

numéro de ligne pour pouvoir parler d'un algorithme, ou l'exécuter, c'est plus pratique d'avoir des références pour chaque ligne.

variables On s'autorise deux types de variables : les variables individuelles, qui ne peuvent contenir qu'un élément (qui peut-être un nombre, une lettre, une paire de nombres,...) et qu'on écrit en *italique*; et les tableaux. Les tableaux ont une dimension, qui représente le nombre d'éléments qu'ils peuvent contenir. Leur dimension est une liste de nombres entiers, écrit avec des \times entre chaque.

Une **variable** est un nom pour un élément pouvant varier. On peut donc le concevoir comme une case de la mémoire (ou une ligne d'une feuille) avec un nom particulier et pouvant contenir un élément. Par exemple, si la variable x contient la valeur 4, on peut le concevoir comme :

4
x

Un tableau TAB de dimension 4 peut contenir 4 éléments, que l'on note :

$$\text{TAB}[0] \quad \text{TAB}[1] \quad \text{TAB}[2] \quad \text{TAB}[3].$$

ou, pour noter aussi la valeur des cases du tableau :

6	-8	35	42
$\text{TAB}[0]$	$\text{TAB}[1]$	$\text{TAB}[2]$	$\text{TAB}[3]$

De même, un tableau TAB' de dimension 3×4 peut contenir 12 éléments, que l'on note :

$$\begin{array}{cccc} \text{TAB}'[0][0] & \text{TAB}'[0][1] & \text{TAB}'[0][2] & \text{TAB}'[0][3] \\ \text{TAB}'[1][0] & \text{TAB}'[1][1] & \text{TAB}'[1][2] & \text{TAB}'[1][3] \\ \text{TAB}'[2][0] & \text{TAB}'[2][1] & \text{TAB}'[2][2] & \text{TAB}'[2][3] \end{array}$$

On peut voir un tableau comme un groupe de variables qui ont le même nom, et dont on peut accéder à chaque élément en fixant la valeur d'autres variables. On note un élément d'un tableau par un groupe de lettres en CAPITALES À CHASSE FIXE, le nom du tableau, suivi de crochet contenant l'indice de la case accédée, qui peut être une expression arbitrairement compliquée. Ainsi, TAB[2] et TAB[1 + 1] désignent la même case du tableau, ainsi que TAB[i − 1], si la variable i vaut 3.

Quand on veut désigner collectivement tout le tableau (en insistant sur ces dimensions), on le note entre parenthèses, avec en indice, les dimensions comme borne des variables. Ainsi, le tableau TAB pourra être noté $(\text{TAB}[i])_{0 \leq i < 4}$ tandis que TAB' sera $(\text{TAB}'[i][j])_{\substack{0 \leq i < 3 \\ 0 \leq j < 4}}$.

instructions chaque ligne contient une instruction qui doit pouvoir être exécutée. On distingue les initialisations

nouveau x

qui déclare qu'une nouvelle variable x est dorénavant disponible. Une initialisation crée une nouvelle variable, ou un nouveau tableau. Ainsi, quand on exécute l'algorithme, on rajoute une colonne pour chaque nouvelle variable créée, puis on passe à la ligne suivante.

On note aussi les assignations

$$x \leftarrow \text{expression}$$

qui modifient la valeur d'une variable x en la remplaçant par la valeur de l'*expression* qui la suit. Cette expression peut-être arbitrairement complexe, mais on préférera les garder simples : notre but est, là encore, de s'assurer que l'algorithme que nous proposons est effectivement calculable, et avoir une idée du temps de calcul.

Ainsi, quand on exécute l'algorithme, on calcule la valeur de l'*expression* puis on note cette valeur dans la colonne de la variable modifiée.

conditionnelles une conditionnelle **si alors sinon** évalue l'*expression* après le **si**. Si cette expression est vraie, on saute à la ligne après le **alors**; si elle est fausse, on saute à la ligne après le **sinon**. Quand on l'exécute, on évalue la valeur de l'*expression*, et saute à la ligne pertinente.

Dans certains cas, on ne veut rien faire dans le cas **sinon**. On le passe alors.

boucles il y a deux types de boucles, les boucles **pour** et les boucles **tant que**. Dans les deux cas, elles contiennent une portion de programme enfermée entre les mots-clefs **faire** et **fin**.

— la boucle **pour** contient une expression de la forme

$$i \text{ allant de } a \text{ à } b$$

Quand on l'exécute, on assigne i à la valeur a , puis on passe à la ligne suivante, disons ℓ .

Quand on atteint le mot-clef **fin**, on augmente i de 1, et si ce nouveau i est strictement inférieur à b , on ré-exécute la ligne ℓ , sinon, on passe à la ligne juste après **fin**.

— la boucle **tant que** contient une expression *exp* pouvant être vraie ou fausse. Si l'*expression* est vraie, on passe à la ligne suivante, disons ℓ , sinon, on saute à la ligne juste après **fin**.

Quand on atteint la ligne **fin**, on réévalue l'*expression* *exp*. Si elle est vraie, on recommence la ligne ℓ , sinon, on continue.

retour une instruction particulière permet de sortir de l'algorithme et de préciser ce qui est renvoyé :

$$\text{retourner } x$$

fait de x la sortie du programme.

Il est important de remarquer que dès qu'on retourne, on quitte l'algorithme ; il peut y avoir plusieurs **retourner** dans un même algorithme, qui seront dans des branches d'exécutions différentes.

Parfois on se permet aussi de retourner des erreurs, si l'algorithme proposé fonctionne sur des entrées dont on ne peut pas spécifier facilement qu'elles sont valides.

On s'aide de la typographie pour être plus explicite, ainsi, on joue sur les polices (je ne vous demande pas de reprendre mes conventions), et surtout sur les indentations (et là, je vous demande impérativement de les suivre) : chaque entrée dans une boucle ou une conditionnelle indente ce qui est à l'intérieur.

1.6 EXÉCUTION

On présente une exécution d'un algorithme sous la forme d'une *trace d'exécution* du programme sous-jacent dans une machine séquentielle. Plus concrètement, on écrit un tableau dont chaque ligne représente une étape d'exécution, et chaque colonne une variable, avec deux variables spéciales, l'une représentant la ligne en cours, l'autre la prochaine ligne.

Exemple 2. Prenons l'algorithme 1, p. 10. Il est présenté avec seulement deux variables : i et résultat. Aussi le tableau de la trace d'exécution va contenir quatre colonnes.

On va exécuter `Multiplication(5, 8)`. L'exécution commence donc à la ligne 2, et les valeurs de a et de b sont fixées.

LIGNE	SUIVANTE	i	résultat
2	3		0
3	4	0	0
4	5	0	8
5	3	1	8
3	4	1	8
4	5	1	16
5	3	2	16
3	4	2	16
4	5	2	24
5	3	3	24
3	4	3	24
4	5	3	32
5	3	4	32
3	4	4	32
4	5	4	40
5	3	5	40
3	6		40
6	renvoyer 40		

1.7 RÉSUMÉ

On voit qu'on a un certain nombre de questions à traiter, que l'on se posera systématiquement par la suite :

- comment calculer la solution à un problème donné?
- comment s'assurer que l'algorithme proposé est correct?
- comment s'assurer que l'algorithme proposé termine?
- l'algorithme est-il efficace?
- comment présenter les données pour permettre des algorithmes efficaces?

- quels compromis faire entre lisibilité, longueur du programme, temps d'exécution, utilisation de la mémoire, consommation d'énergie,...

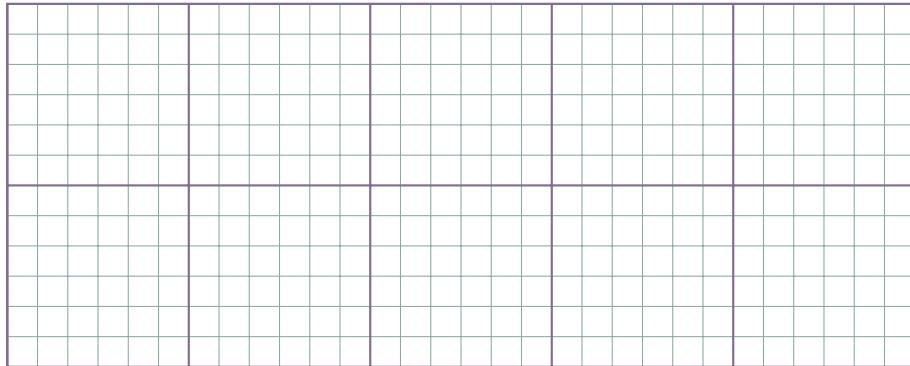


FIGURE 1.4 – Le pgcd de 12 et 30

1.A L'ALGORITHME D'EUCLIDE

L'algorithme qui a sans doute été étudié depuis le plus longtemps dans la tradition occidentale (au point qu'« algorithme » sans précision a pu le désigner) est connu sous le nom d'*algorithme d'Euclide*. Dans une formulation moderne, il permet, étant donnés deux nombres entiers a et b , de calculer le pgcd de a et de b , c'est-à-dire le plus grand nombre qui divise à la fois a et b .

Exemple 3.

- le pgcd de 5 et de 15 est 5.
- le pgcd de 30 et de 12 est 2.

Une application géométrique (présente chez Euclide) vise à, étant donné un rectangle de longueur a et de largeur b , trouver le côté des plus grands carrés permettant de pavir le rectangle (comme représenté Figure 1.4). Tel que décrit par Euclide (livre VII, proposition 2)⁷:

β'. Δύο ἀριθμῶν δοθέντων μὴ πρώτων πρὸς ἀλλήλους τὸ μέγιστον αὐτῶν κοινὸν μέτρον εὑρεῖν.

Ἐστωσαν οἱ δοθέντες δύο ἀριθμοὶ μὴ πρῶτοι πρὸς ἀλλήλους οἱ ΑΒ, ΓΔ. δεῖ δὴ τῶν ΑΒ, ΓΔ τὸ μέγιστον κοινὸν μέτρον εὑρεῖν.

Εἰ μὲν οὖν ὁ ΓΔ τὸν ΑΒ μετρεῖ, μετρεῖ δὲ καὶ ἔαυτόν, ὁ ΓΔ ἄρα τῶν ΓΔ, ΑΒ κοινὸν μέτρον ἐστίν. καὶ φανερόν, ὅτι καὶ μέγιστον· οὐδεὶς γὰρ μείζων τοῦ ΓΔ τὸν ΓΔ μετρήσει.

Εἰ δὲ οὐ μετρεῖ ὁ ΓΔ τὸν ΑΒ, τῶν ΑΒ, ΓΔ ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ μείζονος λειφθήσεται τις ἀριθμός, ὃς μετρήσει τὸν πρὸ ἔαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται· εἰ δὲ μῆ, ἔσονται οἱ ΑΒ, ΓΔ πρῶτοι πρὸς ἀλλήλους· ὅπερ οὐχ ὑπόκειται. λειφθήσεται τις ἄρα ἀριθμός, ὃς μετρήσει τὸν πρὸ ἔαυτοῦ. καὶ ὁ μὲν ΓΔ τὸν ΒΕ μετρῶν λειπέτω ἔαυτοῦ ἐλάσσονα τὸν ΕΑ, ὁ δὲ ΕΑ τὸν ΔΖ μετρῶν λειπέτω ἔαυτοῦ ἐλάσσονα τὸν ΖΓ, ὁ δὲ ΓΖ τὸν ΑΕ μετρείτω. ἐπεὶ οὖν ὁ ΓΖ τὸν ΑΕ μετρεῖ, ὁ δὲ ΑΕ τὸν ΔΖ μετρεῖ, καὶ ὁ ΓΖ ἄρα τὸν ΔΖ μετρήσει· μετρεῖ δὲ καὶ ἔαυτόν· καὶ ὅλον ἄρα τὸν ΓΔ μετρήσει. ὁ δὲ ΓΔ τὸν ΒΕ μετρεῖ· καὶ ὁ ΓΖ ἄρα τὸν ΒΕ μετρεῖ· μετρεῖ δὲ καὶ τὸν ΕΑ· καὶ ὅλον ἄρα τὸν ΒΑ μετρήσει· μετρεῖ δὲ καὶ τὸν ΓΔ· ὁ ΓΖ ἄρα τοὺς ΑΒ, ΓΔ μετρεῖ. ὁ ΓΖ ἄρα τὸν ΑΒ, ΓΔ κοινὸν μέτρον ἐστίν. λέγω δή, ὅτι καὶ μέγιστον. εἰ γὰρ μῆ ἐστιν ὁ ΓΖ τῶν ΑΒ, ΓΔ μέγιστον κοινὸν μέτρον, μετρήσει τις τοὺς ΑΒ, ΓΔ ἀριθμοὺς ἀριθμὸς μείζων ὃν τοῦ ΓΖ. μετρείτω, καὶ ἔστω ὁ Η. καὶ ἐπεὶ ὁ Η τὸν ΓΔ μετρεῖ, ὁ δὲ ΓΔ τὸν ΒΕ μετρεῖ, καὶ ὁ Η ἄρα τὸν ΒΕ μετρεῖ· μετρεῖ δὲ καὶ ὅλον τὸν ΒΑ· καὶ λοιπὸν ἄρα τὸν ΑΕ μετρήσει. ὁ δὲ ΑΕ τὸν ΔΖ μετρεῖ· καὶ ὁ Η ἄρα τὸν

7. Les citations d'Euclide sont celles de l'édition (ACERBI 2007).

ΔZ μετρήσει· μετρεῖ δὲ καὶ ὅλον τὸν $\Delta\Gamma$ · καὶ λοιπὸν ἄρα τὸν ΓZ μετρήσει ὁ μείζων τὸν ἐλάσσονα· ὅπερ ἔστιν ἀδύνατον· οὐκ ἄρα τοὺς AB , $\Gamma\Delta$ ἀριθμοὺς ἀριθμός τις μετρήσει μείζων ὃν τὸν ΓZ · ὁ ΓZ ἄρα τῶν AB , $\Gamma\Delta$ μέγιστον ἔστι κοινὸν μέτρον· [ὅπερ ἔδει δεῖξαι].

Πόρισμα

'Εκ δὴ τούτου φανερόν, ὅτι ἐὰν ἀριθμὸς δύο ἀριθμοὺς μετρῇ, καὶ τὸ μέγιστον αὐτῶν κοινὸν μέτρον μετρήσει· ὅπερ ἔδει δεῖξαι.

En langage plus moderne (et en français !), l'algorithme consiste en la suite d'opérations suivantes : étant donnés deux entiers positifs, on appelle a le plus grand des deux et b l'autre (s'ils sont égaux, on renvoie a — qui est d'ailleurs égal à b). Si $b == 0$, alors l'algorithme est terminé et on renvoie a ; sinon, on calcule r , le reste de la division euclidienne de a par b , et on recommence la procédure avec b et r . Si c'est bien une transcription assez fidèle des propos d'Euclide, un certain nombre de questions se posent :

- Euclide n'a évidemment pas pu écrire son algorithme en faisant référence à un modèle de calcul.
Est-ce que ce texte définit vraiment des programmes calculables ?
- est-ce qu'il est bien défini ? Pour tous les couples de nombres entiers ? (il parle, en particulier, de terminaison. Mais le calcul termine-t-il toujours ?)
- est-ce qu'il calcule bien le pgcd ?
- est-ce qu'il le calcule vite ?

Ce sont le genre de questions qu'on va se poser dans ce cours. Pour la première, la meilleure manière de s'en convaincre est encore de le traduire comme un programme dans un modèle de calcul (particulier, mais comme ils sont tous équivalents d'après la thèse de Church, il suffit de l'écrire dans un d'entre eux).

Par exemple, on peut l'écrire dans le langage Python :

```

1 def pgcd(a,b):
2     if a < b:
3         (b,a) = (a,b)
4     if b == 0:
5         return a
6     else:
7         return (gcd(b, a%b))

```

Ou même en C :

```

1 int pgcd(int a, int b){
2     if (a < b) {
3         return pgcd(b,a);
4     } else {
5         if (b == 0) {
6             return a;
7         } else {
8             return pgcd(b, a%b);
9         }
10    }
11 }

```

Ces deux programmes ont la même structure (à une différence notable près!). L'[algorithme d'Euclide](#) est, d'une certaine manière, ce qu'il y a de commun à ces deux programmes, ou pour le dire autrement, ces deux programmes *aux détails d'implémentation près*, c'est-à-dire aux détails dépendant explicitement du modèle de calcul. On peut essayer de l'écrire d'une manière qui ne fasse pas référence à ces détails :

Exercice 9. 1. Regarder les deux programmes en C et en Python, ainsi que l'algorithme 2. Noter toutes les différences.

Entrées: deux nombres entiers positifs a et b .

```

1 Euclide( $a,b$ )
2   si  $a < b$  alors
3     retourner Euclide( $b,a$ )
4   sinon
5     si  $b == 0$  alors
6       retourner  $a$ 
7     sinon
8        $r \leftarrow$  reste de la division euclidienne de  $a$  par  $b$ 
9       retourner Euclide( $b, r$ )
10    fin
11 fin

```

Sorties: un nombre entier positif

Algorithm 2: Algorithme d'Euclide

Début

FIGURE 1.5 – L'algorithme d'Euclide comme un organigramme

2. Faire de même en comparant l'algorithme 2 et sa description en toutes lettres en français.
3. Faire de même avec la description en grec.

On voit qu'on écrit un algorithme dans un français extrêmement rigide, et s'appuyant sur la typographie (des polices, des graisses, des alignements). Il n'y a pas une seule manière de noter des algorithmes, mais on essaye d'être le plus explicite possible: un texte, du **pseudo-code** comme ici, ou encore un organigramme de programmation (terminologie un peu datée) peuvent faire l'affaire.

Un algorithme ne se comprend pas en se lisant, mais en s'exécutant. Cet algorithme n'est pas, tel quel, exécutable. En effet, la ligne « $r \leftarrow$ reste de la division euclidienne de a par b » nécessite de savoir calculer la division euclidienne de a par b . On verra ça plus tard en Section 1.B.

1.B PGCD

Le pgcd intervient naturellement dans plusieurs situations. La plus courante est la *simplification de fractions*. Une *fraction* est déterminée par la donnée de deux nombres entiers a et b (où $b \neq 0$) que l'on écrit

$$\frac{a}{b}.$$

On dit que deux fractions $\frac{a}{b}$ et $\frac{a'}{b'}$ sont *égales* si on peut passer de l'une à l'autre en multipliant le numérateur et le dénominateur par le même nombre entier. Autrement dit, s'il y a un entier $c \neq 0$ tel que

$$\begin{cases} ac == a' \\ bc == b' \end{cases}$$

Ainsi, par exemple $\frac{1}{5}$ et $\frac{6}{30}$ sont égales. Une des deux formes est néanmoins préférable, celle utilisant des nombres plus simples, c'est-à-dire plus petit. Autrement dit, étant donnée une fraction $\frac{a}{b}$, on cherche à trouver a' , b' et c le plus grand possible tel que

$$\begin{cases} a == a'c \\ b == b'c \end{cases}$$

Ce nombre c est le plus grand nombre qui est un multiple commun à la fois à a et à b , et on le nomme le pgcd de a et de b . On a donné sommairement un algorithme pour le calculer en Section 1.A. On va le détailler et l'étudier. Redonnons l'algorithme :

Entrées: deux nombres entiers positifs a et b .
1 Euclide (a,b)
2 si $a < b$ alors
3 retourner Euclide(b,a)
4 sinon
5 si $b == 0$ alors
6 retourner a
7 sinon
8 $r \leftarrow$ reste de la division euclidienne de a par b
9 retourner Euclide(b, r)
10 fin
11 fin
Sorties: un nombre entier positif

La division euclidienne

À la ligne 8, on voit qu'il faudrait savoir calculer le reste de la division euclidienne pour pouvoir calculer un pgcd. Commençons donc par cela.

La division euclidienne d'un entier a par un autre entier b est définie par le système :

$$\begin{aligned} a &== bq + r \\ 0 &\leqslant r < b \end{aligned}$$

On appelle q le *quotient* et r le *reste*.

Exercice 10. Donner le quotient et le reste de la division euclidienne de :

- 24 par 4
- 25 par 4
- 298 par 17

Une méthode pour calculer la division euclidienne consiste à faire grossir progressivement q et calculer $a - bq$ jusqu'à ce que ce dernier nombre soit plus petit que b .

Exercice 11. Donner un algorithme `DivisionEuclidienne`(a, b) implémentant la division euclidienne de a par b . En particulier, on veut pouvoir ré-écrire l'algorithme d'Euclide en remplaçant la ligne 8 de l'algorithme d'Euclide par :

$$(q, r) \leftarrow \text{DivisionEuclidienne}(a, b)$$

Exercice 12. Exécuter l'algorithme `DivisionEuclidienne` sur les valeurs de l'Exercice 10.

Exercice 13. Calculez le nombre d'étapes (où une étape est un changement de ligne dans la trace d'exécution) pour calculer la division euclidienne de a par b .

Exécution

Si on veut exécuter l'algorithme ci-dessus, on tombe sur une difficulté qu'on rencontre pour la première fois : la procédure `Euclide` fait appel à une autre procédure (`DivisionEuclidienne`). Pour représenter ceci dans une trace d'exécution, on entame un deuxième tableau (pour la nouvelle procédure), exécute la nouvelle procédure, rapporte le résultat dans le premier tableau en indiquant très clairement où est la justification.

La procédure appelée peut être la même que celle appelant.

Exercice 14. Donner les valeurs de a et b pour lesquelles `Euclide(a, b)` retourne sans appeler une autre procédure.

Pour les autres, donner quelle sera la prochaine procédure appelée et avec quels paramètres.

Exercice 15. Exécuter `Euclide(35, 5)` et `Euclide(12, 30)`.

Terminaison

L'exécution de `Euclide(a, b)` définit une suite de paires de nombres : en effet, `Euclide(a, b)` provoque l'appel de `Euclide(b, r)`, et ainsi de suite.

On considère donc la suite

$$\begin{aligned} & (a, b) \\ & (b, r) \\ & \vdots \end{aligned}$$

des arguments de la procédure exécutée sur (a, b) .

Exercice 16. Soient a et b deux entiers positifs tels que $a > b$.

Montrer que la suite des arguments de la procédure exécutée sur (a, b) est strictement décroissante au sens où, si (a, b) est suivie par (c, d) alors

$$\begin{aligned} a &> c \\ b &> d \end{aligned}$$

Exercice 17. Soient a et b deux entiers positifs tels que $a \geq b$.

Montrer que l'exécution de `Euclide(a, b)` termine.

Exercice 18. Soient a et b deux entiers positifs.

Montrer que l'exécution de `Euclide(a, b)` termine.

Correction

Exercice 19. Montrer que pour tous nombres entiers positifs a et b , le pgcd de a et de b est égal au pgcd de b et de a .

Exercice 20. Soient a et b deux nombres entiers positifs. Soient q et r le quotient et le reste de a par b .

Montrer que si un nombre divise a et b , alors il divise aussi r .

Réciproquement, montrer que si un nombre divise b et r , alors il divise aussi a .

En conclure que le pgcd de a et de b est égal au pgcd de b et de r .

Exercice 21. Déduire de ce qui précède que le résultat de `Euclide(a, b)` est bien le pgcd de a et de b .

Programmation

Exercice 22 (bonus). Programmer en C les algorithmes proposés.

Structures de données 1

On voit les premières structures de données: tableaux et listes chaînées, ainsi que les opérations que l'on peut faire sur ces structures (accès, modification, ajout, suppression, fusion, longueur). On voit dans quelles situations elles sont adaptées. On introduit quelques notations et concepts pour mesurer la complexité d'un algorithme; on présente aussi la recursivité.

2.1	Éléments de complexité algorithmique	26
2.2	Tableaux	29
	Récupération & modification	31
	Adjonction & suppression	31
	Fusion	33
	Nombre d'éléments	34
	Résumé	34
2.3	Listes chaînées	34
	Récupération & modification	36
	Adjonction & suppression	41
	Fusion	43
	Longueur	44
	Résumé	44
2.4	La récursion	45
2.5	Piles	45
2.6	Résumé	46
2.A	Inversion Correction	48
2.B	D'un tableau à une liste et vice-versa	52
2.C	La multiplication en base 10	55
	Représentation des nombres	55
	Petite multiplication	62
	Grande multiplication	63

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. Comme on l'a vu pour la multiplication, la manière dont les données sont présentées influe sur les opérations qui sont possibles sur elles ; aussi, on va commencer par se demander quelles opérations on a à faire couramment.

On suppose qu'on a une liste de N éléments et que k est un entier plus petit que N ($0 \leq k < N$). 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 23. Pour chaque opération, donner un exemple où il est naturel de vouloir la faire.

On va proposer différentes structures pour stocker des éléments, pour chacune de ces structures, donner les algorithmes permettant de réaliser certaines des opérations listées ci-dessus (en particulier, on ne s'intéressera ni au tri ni à la recherche, ce sera un sujet pour plus tard — de même, on n'entrera pas dans le découpage d'une liste au deux, qui est au choix, une recopie de la fusion, ou quelque chose de beaucoup plus compliqué). Ensuite, on cherchera à évaluer le temps que prennent ces opérations. Pour cela, on commence par faire un point sur la complexité.

2.1 ÉLÉMENTS DE COMPLEXITÉ ALGORITHMIQUE

Supposons qu'on ait un programme dans un modèle de calcul donné : par exemple, un programme, écrit en C, 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 spéieux 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 nombre¹.

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 en pseudo-code* et en comptant 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 le 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.

Multipier 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.

Multipier 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 positions² (par exemple en faisant une première course contre la montre, compétitrice par compétitrice). 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.

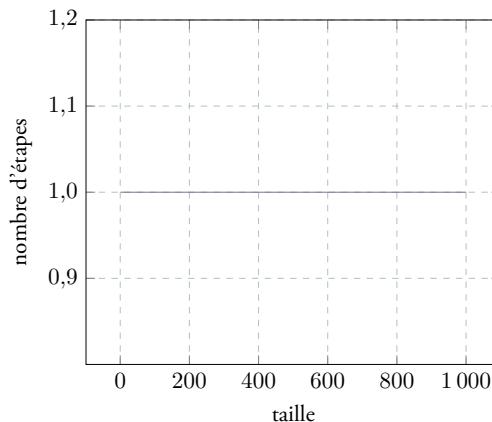
1. De la même manière qu'un algorithme est une idéalisation d'un programme, qui nous oblige à traiter des notions plus floues, le temps d'exécution n'a pas vraiment de sens pour un algorithme, et on parle d'une notion plus floue, son ordre de grandeur.

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

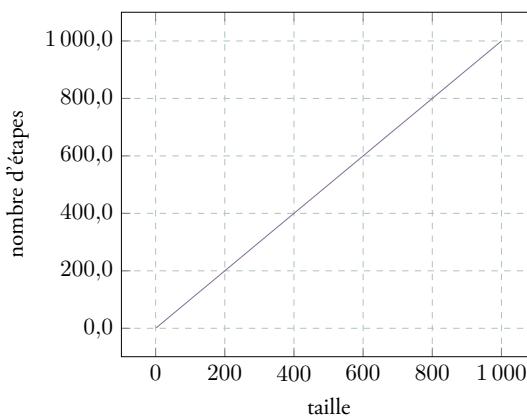
Rajouter un·e seul·e compétitrice oblige à organiser beaucoup de courses en plus (on peut calculer que s'il en fallait k à n compétitrices, il en faut $n \times k$ pour $n + 1$).

Pour évoquer ces situations, on va reprendre les notations de Landau (KNUTH 1976a). 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*:

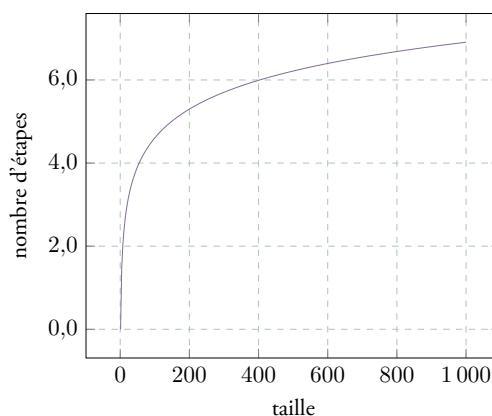
constante si le nombre d'étapes ne dépend pas de n . On le note $O(1)$. Si on représente le temps d'exécution en fonction de la taille de l'entrée, on obtient une figure de la forme:



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 $O(n)$;

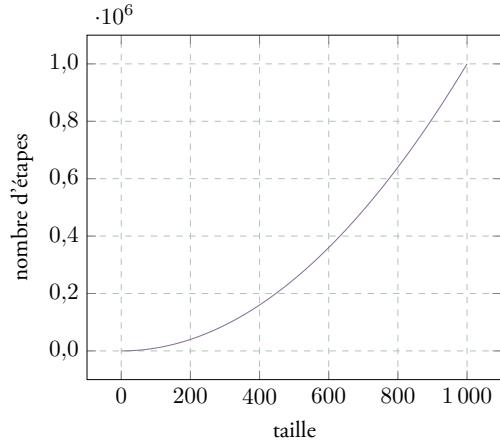


logarithmique si le nombre d'étapes croît **logarithmiquement**, c'est-à-dire augmente d'une constante quand n double. On le note $O(\log n)$;

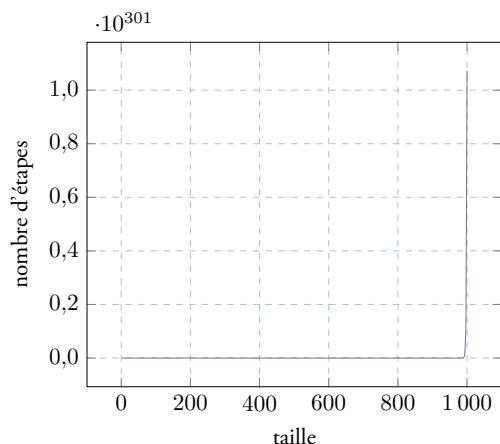


On voit ici que quand la taille double (quand elle passe de 200 à 400, ou de 400 à 800), le nombre d'étape augmente de la même quantité à chaque fois.

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



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



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.

2.2 TABLEAUX

Une variable peut être vue comme une case nommée dans la mémoire, pouvant prendre n'importe quelle valeur. Un *tableau* (ou parfois *vecteur*) doit être vu comme une série contiguë de cases dans la mémoire, dont chacune a un identifiant numérique. Ainsi, un tableau a une *longueur* et ses éléments sont indexés par les entiers plus petits que la longueur. On représentera le tableau $(\text{TAB}[i])_{0 \leq i < 10}$ défini

par:

```

TAB[0] := 6
TAB[1] := -8
TAB[2] := 35
TAB[3] := 42
TAB[4] := -90
TAB[5] := 27
TAB[6] := 12
TAB[7] := 0
TAB[8] := 0
TAB[9] := 87

```

comme ceci: dans chaque case, le contenu, et en dessous le nom de la case.

6	-8	35	42	-90	27	12	0	0	87
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]	TAB[6]	TAB[7]	TAB[8]	TAB[9]

voire juste, si on laisse implicite le nom du tableau:

6	-8	35	42	-90	27	12	0	0	87
0	1	2	3	4	5	6	7	8	9

On doit bien distinguer:

- $(\text{TAB}[i])_{0 \leq i < N}$ qui est le tableau, vu dans son intégralité;
- pour un $0 \leq i < N$ donné, $\text{TAB}[i]$ qui est une variable comme une autre.

On remarque aussi que, dans un tableau de longueur N, le premier élément est numéroté $\text{TAB}[0]$, et le dernier $\text{TAB}[N - 1]$: en particulier, il n'y a pas d'élément noté $\text{TAB}[N]$.

Exercice 24. Considérons le tableau

6	-8	35	42	-90	27	12	0	0	87
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]	TAB[6]	TAB[7]	TAB[8]	TAB[9]

Pour chacune des expressions, dire si elle a du sens, et si oui, quel type d'objet elle désigne:

- $\text{TAB}[0]$
- $\text{TAB}[10]$
- $(\text{TAB}[i])_{0 \leq i < 10}$
- $(\text{TAB}[i])_{0 < i < 10}$
- $\text{TAB}[9]$
- $\text{TAB}[8] \leftarrow 3$
- $(\text{TAB}[i])_{0 \leq i < 10} \leftarrow 0$

Récupération & modification

Étant donné un tableau et une position dans le tableau, on peut accéder immédiatement à la valeur de cet élément, et aussi la modifier :

Entrées:

- un tableau d'éléments $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N ;
- un entier k tel que $0 \leq k < N$.

1 Accès $((\text{TAB}[i])_{0 \leq i < N}, k)$
 2 | retourner $\text{TAB}[k]$

Sorties: un élément

Algorithme 3: Tableau — accès

Entrées:

- un tableau d'éléments $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N ;
- un entier k tel que $0 \leq k < N$;
- un élément a .

1 Modification $((\text{TAB}[i])_{0 \leq i < N}, k, a)$
 2 | $\text{TAB}[k] \leftarrow a$
 3 | retourner $(\text{TAB}[i])_{0 \leq i < N}$

Sorties: un tableau d'éléments, de longueur N

Algorithme 4: Tableau — modification

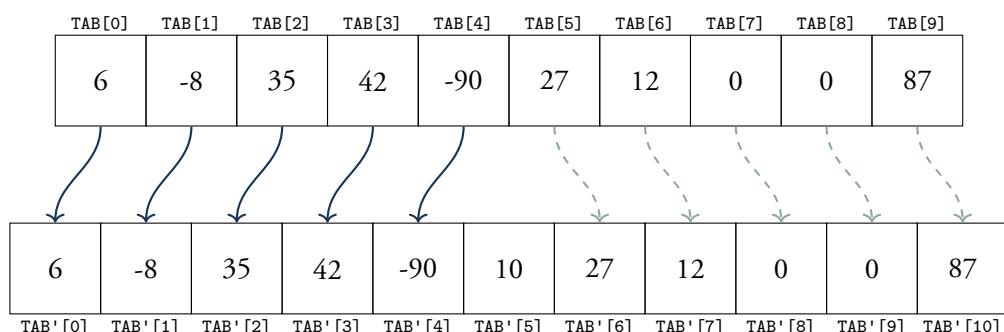
Dans les deux cas, ces deux algorithmes prennent un nombre d'étapes qui ne dépend pas de l'entrée : accéder à un élément dans un tableau et le modifier se fait de la même manière quelle que soit la longueur du tableau. La complexité est constante, en $O(1)$.

Adjonction & suppression

Ajouter un élément est plus compliqué : en effet, dans notre exemple, si on veut rajouter un élément au milieu du tableau, il faudrait pouvoir décaler tout ce qui suit, mais le tableau ainsi obtenu ne ferait plus la même taille ! Or, les tableaux ne sont que des cases contigües, et on ne s'est pas donné la possibilité d'en rajouter une à une position arbitraire.

La seule possibilité qu'on a est donc de créer un nouveau tableau, plus grand que l'ancien, et recopier l'intégralité de l'ancien tableau. L'Algorithme 5 ajoute un élément à la position d'indice donné.

Des deux boucles **pour**, l'une copie les éléments avant cet indice, l'autre les éléments après. Ainsi, en reprenant le tableau $(\text{TAB}[i])_{0 \leq i < 10}$ de l'exemple, et en insérant 10 à la position 5, on peut se représenter l'exécution ainsi :



Entrées:

- un tableau d'éléments $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N ;
- un entier k tel que $0 \leq k \leq N$;
- un élément a .

```

1 Ajout(( $\text{TAB}[i]$ ) $_{0 \leq i < N}$ ,  $k, a$ )
2   nouveau ( $\text{TAB}'[i]$ ) $_{0 \leq i < N+1}$ 
3   pour  $i$  allant de  $0$  à  $k$  faire
4     |  $\text{TAB}'[i] \leftarrow \text{TAB}[i]$ 
5   fin
6    $\text{TAB}'[k] \leftarrow a$ 
7   pour  $i$  allant de  $k$  à  $N$  faire
8     |  $\text{TAB}'[i + 1] \leftarrow \text{TAB}[i]$ 
9   fin
10  retourner ( $\text{TAB}'[i]$ ) $_{0 \leq i < N+1}$ 
```

Sorties: un tableau de longueur $N + 1$ **Algorithme 5:** Tableau — ajout d'un élément

Où l'on a dessiné en bleu les assignations faites par la première boucle, et en gris pointillé les assignations faites par la deuxième.

On voit que rajouter un élément à un tableau de N éléments prend de l'ordre de N opérations.

De même pour la suppression, on doit créer un nouveau tableau légèrement moins grand, recopier tous les éléments sauf celui qu'on veut supprimer. Comme on préfère ne pas se demander ce qu'est un tableau à zéro éléments, on décide que l'algorithme ne prend en entrée que des tableaux d'au moins deux éléments.

Entrées:

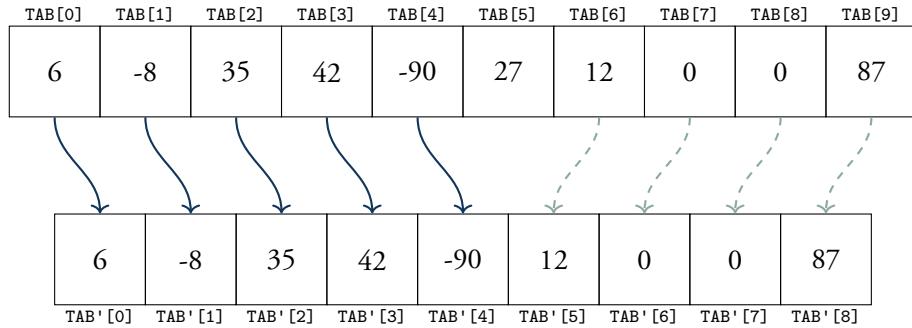
- un tableau d'éléments $(\text{TAB}[i])_{0 \leq i < N}$ de longueur $N > 1$;
- un entier k tel que $0 \leq k < N$.

```

1 Suppression(( $\text{TAB}[i]$ ) $_{0 \leq i < N}$ ,  $k$ )
2   nouveau ( $\text{TAB}'[i]$ ) $_{0 \leq i < N-1}$ 
3   pour  $i$  allant de  $0$  à  $k$  faire
4     |  $\text{TAB}'[i] \leftarrow \text{TAB}[i]$ 
5   fin
6   pour  $i$  allant de  $k + 1$  à  $N$  faire
7     |  $\text{TAB}'[i - 1] \leftarrow \text{TAB}[i]$ 
8   fin
9   retourner ( $\text{TAB}'[i]$ ) $_{0 \leq i < N-1}$ 
```

Sorties: un tableau de longueur $N - 1$ **Algorithme 6:** Tableau — suppression d'un élément

En reprenant le tableau $(\text{TAB}[i])_{0 \leq i < 10}$ de l'exemple, et en supprimant la position 5, on peut se représenter l'exécution ainsi :



Exercice 25. Montrer que la suppression est l'inverse à gauche de l'ajout, c'est à dire que, pour tout tableau $(\text{TAB}[i])_{0 \leq i < N}$, tout indice $0 \leq k \leq N$ et tout élément a ,

$$\text{Suppression}(\text{Ajout}((\text{TAB}[i])_{0 \leq i < N}, k, a), k) == (\text{TAB}[i])_{0 \leq i < N}.$$

Ces deux opérations nécessitent de recopier intégralement le tableau d'entrée, elles sont donc en temps linéaire.

Fusion

La fusion de deux tableaux (de longueur respective N et M) se fait de même, en créant un nouveau tableau de longueur $N + M$, puis en recopiant d'abord les éléments du premier tableau, puis ceux du second. Cela utilise deux boucles **pour**, et est donc en complexité $O(N + M)$.

Entrées:

- un tableau d'éléments $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N ;
- un tableau d'éléments $(\text{TAB}'[j])_{0 \leq j < M}$ de longueur M .

1 **Fusion** $((\text{TAB}[i])_{0 \leq i < N}, (\text{TAB}'[j])_{0 \leq j < M})$

2 **nouveau** $(\text{RES}[k])_{0 \leq k < N+M}$

3 **pour** i allant de 0 à N **faire**

4 | $\text{RES}[i] \leftarrow \text{TAB}[i]$

5 **fin**

6 **pour** j allant de 0 à M **faire**

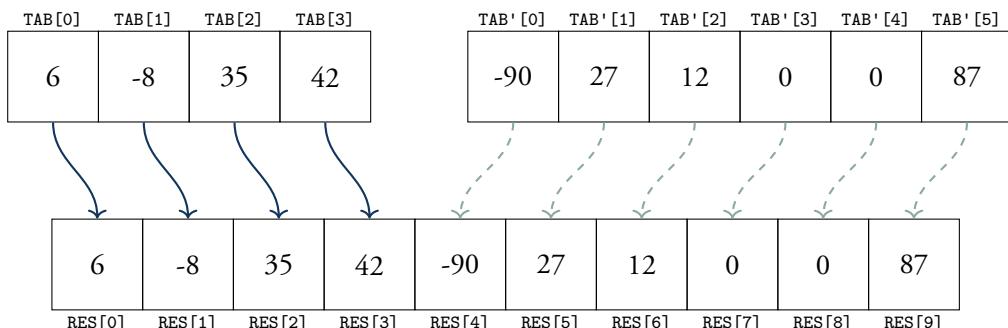
7 | $\text{RES}[N + j] \leftarrow \text{TAB}'[j]$

8 **fin**

9 **retourner** $(\text{RES}[k])_{0 \leq k < N+M}$

Sorties: un tableau de longueur $N + M$

Algorithme 7: Tableau — fusion de deux listes



Nombre d'éléments

Pour ce qui est du nombre d'éléments, c'est parfaitement trivial : il fait partie des données ! C'est donc en $O(1)$.

Résumé

On peut ainsi, sur une entrée de taille n (et éventuellement une deuxième entrée de taille m), conclure par le tableau de complexités suivant :

Accès	Modification	Ajout	Suppression	Fusion	Longueur
$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n + m)$	$O(1)$

Exercice 26 (tableaux avec éléments désactivés). On peut se dire que la suppression pourrait se faire plus efficacement si on pouvait juste désactiver un élément, en mettant un drapeau signifiant que l'élément n'est pas utilisable.

Ainsi, on peut représenter une liste d'entiers par un tableau de couples (b, n) où b est un booléen³ (c'est-à-dire une valeur pouvant valoir soit vrai, soit faux) et n un entier, et que seuls les éléments où le booléen vaut vrai sont éléments de la liste.

Réécrire tous les algorithmes dans son cas, et évaluer leur complexité.

Conclure.

Exercice 27. On a un tableau de nombres $\text{TAB}[i]_{0 \leq i < N}$. On veut récupérer le plus grand élément.

Écrire un algorithme prenant un tableau en entrée et renvoyant son plus grand élément.

Écrire un algorithme prenant un tableau en entrée et renvoyant la position dans le tableau du plus grand élément. Pour cette question, il y a plusieurs possibilités : que se passe-t-il quand plusieurs éléments sont maximaux ?

2.3 LISTES CHAÎNÉES

Si représenter des données sous forme de tableau peut-être vu comme une généralisation d'inscrire des noms sur des lignes d'une feuille de papier, alors on peut remarquer qu'en général, quand on inscrit des noms sur une feuille et qu'on veut en rajouter un au milieu, on ne recopie pas tout (ce qu'on faisait avec des tableaux) : on écrit un nom en plus, et, par des signes conventionnels (par exemple, des flèches ou un appel de note), on signale à la lecteurice à quel position ce nom va. On pourrait donc imaginer traduire ça en prenant un **tableau** et y écrire dans chaque case un couple contenant la donnée intéressante et l'indice de la prochaine donnée.

Par **couple**, on entend la donnée de deux éléments complètement hétérogènes que l'on représentera entre des parenthèses, séparés par des virgules. Par exemple :

$$(8, 3)$$

est un couple, constitué de deux entiers, de même que

$$(8, \text{'Bonjour'})$$

en est un, constitué d'un entier et d'une chaîne de caractère. On s'autorise aussi à avoir des couples de variables et ainsi :

$$(x, y) \leftarrow (3, 8)$$

3. Nommés ainsi en l'honneur de George Boole (1815–1864), mathématicien et philosophe britannique, et grand algébraïsateur de la logique ; même si la question de savoir si on peut vraiment le considérer comme ayant créé la logique booléenne est complexe (GASTALDI 2018).

assigne simultanément 3 à x et 8 à y .

Pour en revenir aux listes, le tableau

(6,4)	(-8,5)	(35,9)	(42,2)	(-90,8)	(27,6)	(12,3)	(0,•)	(0,1)	(87,7)
0	1	2	3	4	5	6	7	8	9

représente la liste $6, -90, 0, -8, 27, 12, 42, 35, 87, 0$.

Cela permet facilement de réordonner et de supprimer (**exercice**: écrire un algorithme pour ces opérations), mais pas d'ajouter des éléments (en effet, le tableau doit être assez grand pour toutes les opérations qu'on pourra vouloir faire...). Enfin, le premier élément est fixé (c'est forcément la première case du tableau) et on a dû introduire un symbole de fin de liste, noté \bullet .

On prend donc une autre structure, celle de *liste chaînée* (ou plus simplement *liste*⁴), qu'on va décrire de deux manières : l'une mathématique, l'autre en référence à la mémoire de l'ordinateur.

mathématiquement une liste est une structure *récursive*, c'est-à-dire qu'elle est définie à partir d'elle-même. En l'occurrence, une liste est soit :

- la liste vide, que l'on va noter Λ ;
- une liste composée d'un élément et d'une autre liste.

Cette définition n'a de sens que si l'on considère que la deuxième règle ne peut-être appliquée qu'un nombre fini de fois. Donc, les constructions suivantes sont des listes d'entiers :

$$\begin{aligned} \Lambda \\ (3, \Lambda) \\ (6, (7, (8, \Lambda))) \end{aligned}$$

En fait, on raisonne de la même manière qu'en disant qu'une femme est reine si, ou bien :

- sa mère l'était ;
- elle a fondé un royaume.

matériellement une liste est composée de cellules, et chaque cellule est elle-même une paire d'un élément et soit d'une adresse dans la mémoire (le prochain élément de la liste), ou un symbole particulier, signifiant qu'il n'y a pas de prochain élément dans la liste, que l'on va noter Λ . Graphiquement, on représente ça comme un tableau à deux cases, mais dont la deuxième case contient une flèche vers l'élément d'après, ou bien Λ .

Étant donnée une liste chaînée non-vide, on appelle sa *tête* son premier élément et sa *queue* le reste de la liste chaînée. Comme la tête et la queue de liste sont définies comme des éléments d'un couple, on peut récupérer la tête t et la queue Q d'une liste non-vide L par :

$$(t, Q) \leftarrow L.$$

Enfin, on fera les conventions suivantes :

- on note par des lettres minuscules les éléments et par des lettres majuscules les listes ;
- on note le constructeur de listes $::$ et une liste déjà construite entre crochets : ainsi, la liste $(6, (7, (8, \Lambda)))$ pourra être notée par $[6, 7, 8]$, et la liste vide par $[]$, et

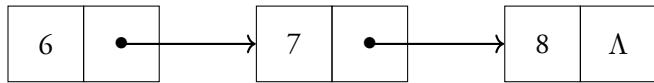
$$6 :: [7, 8] \equiv [6, 7, 8]$$

Ainsi, on pourra récupérer la tête t et la queue Q d'une liste non-vide L par :

$$t :: Q \leftarrow L.$$

On remarque bien que ceci ne sont que des notations et on peut tout faire sans les introduire.

4. Avec tout le problème qu'il y a de nommer une structure pour représenter des listes, des listes.



Exercice 28. Les expressions suivantes ont-elles un sens, et si oui, lequel ? En particulier, indiquez toutes celles qui sont des listes, et lesquelles sont des listes homogènes.

- Λ
- $(3, \Lambda)$
- (Λ, Λ)
- $((3, \Lambda), \Lambda)$
- $(2, (3, \Lambda))$
- $(\Lambda, (3, \Lambda))$
- $(\Lambda, 3)$
- $[]$
- $7 :: 8 :: []$
- $[] :: []$
- $7 :: [] :: []$
- Si $(TAB[i])_{0 \leq i < N}$ et $(TAB'[i])_{0 \leq i < M}$ sont deux tableaux, $(TAB[i])_{0 \leq i < N} :: (TAB'[i])_{0 \leq i < M}$
- Si $(TAB[i])_{0 \leq i < N}$ et $(TAB'[i])_{0 \leq i < M}$ sont deux tableaux, $(TAB[i])_{0 \leq i < N} :: (TAB'[i])_{0 \leq i < M} :: []$.
- $t :: Q \leftarrow 7 :: 8 :: []$
- Si L est une liste avec au moins un élément, $(e :: L) \leftarrow L$.

Exercice 29. Donner la représentation récursive et dans la mémoire des listes suivantes :

- 8 puis 5 puis 3

Exercice 30. Peut-on facilement connaître la longueur d'une liste ?

Récupération & modification

On va donner deux versions de l'algorithme d'accès, le premier itératif, le second récursif. En itératif, on va augmenter un compteur dans une boucle **pour** jusqu'à trouver l'élément qui nous intéresse. À chaque fois, on redéfinit la liste comme étant la queue. Cela suppose que la liste est non-vide, ce que l'on ne peut pas assurer : en effet, contrairement à un tableau, une liste chaînée ne contient pas explicitement l'information de sa propre taille : ainsi l'algorithme doit pouvoir fonctionner si on lui donne en entrée une liste et un indice quelconque, pas nécessairement plus petit que la taille de la liste.

Pour cela, on ajoute la possibilité, en sortie de renvoyer non pas un élément, mais une erreur ; et on obtient ainsi l'Algorithme 8.

On ne précisera pas ce qu'on entend par une erreur, juste que l'algorithme ne couvre pas tous les cas possibles en entrée.

On considérera donc qu'une erreur est une valeur de retour particulière : il faut systématiquement indiquer l'erreur comme une sortie possible.

Cet algorithme est satisfaisant, mais on voit qu'il a une certaine redondance : d'une certaine manière, l'itération est faite deux fois : une fois dans le compteur i qui croît ; et une fois dans le fait qu'on enlève des têtes à la liste. En effet, à chaque étape de boucle, la liste désignée par la variable L ne contient plus ses i premiers éléments. On pourrait donc chercher à accéder à l'élément $k - i$ de cette liste. Pour le représenter schématiquement, sur la liste $[4, 5, 9, 30, 18]$, pour k égal à 3, la machine exécutant

Entrées:

- une liste d'éléments L;
- un entier k.

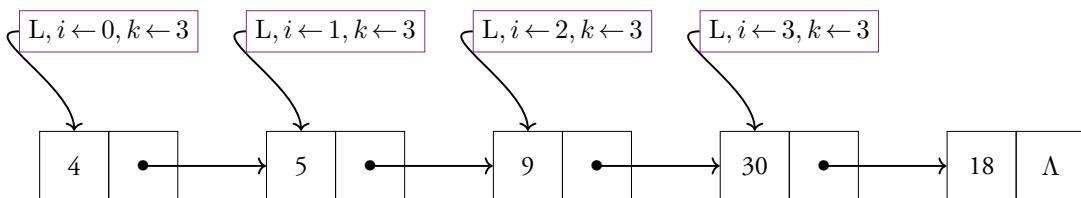
```

1 Accès(L, k)
2   pour i allant de 0 à k+1 faire
3     si L ≠ Λ alors
4       e :: L ← L
5     sinon
6       retourner erreur
7     fin
8   fin
9   retourner e

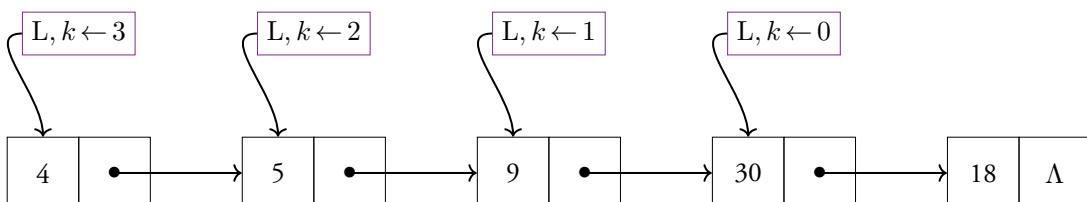
```

Sorties: un élément ou une erreur**Algorithm 8:** Liste — accès itératif

l'algorithme va passer successivement dans les états suivants (on représente l'état par une étiquette, où la variable L pointe vers la liste à laquelle elle a été assignée) :



On se dit qu'on aimerait pouvoir faire mieux, a priori, uniquement pour des raisons esthétiques et coder l'avancement de l'algorithme en un seul endroit. On aimerait bien faire quelque chose comme cela : dans lequel l'avancement est stockée dans l'endroit vers lequel pointe L et la valeur de k : en effet, si on a enlevé la tête d'une liste, chercher son $k - 1^{\text{ème}}$ élément est la même chose que chercher le $k^{\text{ème}}$ sans avoir rien enlevé. Pour se faire on utilise la récursion, c'est-à-dire qu'on va, pour calculer le résultat d'un algorithme sur une liste L, utiliser le résultat du même algorithme sur la queue de L. On obtient ainsi l'Algorithme 9.



On peut se demander comment représenter une exécution d'un tel algorithme : en effet, dans les traces d'exécutions que nous avons vu jusqu'à présent, nous ne nous sommes pas donné les moyens de représenter l'exécution de procédures imbriquées les unes dans les autres. Ici, on voit que les différents appels à Accès sont chaînés et que le résultat de la procédure appelée est le résultat de la procédure appelante. Aussi, on va faire la convention suivante : on va rajouter une colonne pour nommer la procédure en cours d'exécution et ses arguments, et à chaque fois qu'on atteindra une ligne **retourner**, on placera une étoile * comme ligne suivante, tracera une ligne horizontale, et passerons à l'exécution de la procédure appelée.

Entrées:

- une liste d'éléments L;
- un entier k.

```

1 Accès(L, k)
2   si L ≡ Λ alors
3     retourner erreur
4   sinon
5     e :: Q ← L
6     si k ≡ 0 alors
7       retourner e
8     sinon
9       retourner Accès(Q, k - 1)
10    fin
11 fin

```

Sorties: un élément ou une erreur**Algorithm 9:** Liste — accès récursif

On doit introduire une nouvelle distinction : entre algorithme et procédure. En effet, l'exécution d'un algorithme récursif entraîne l'exécution d'autres instances du même algorithme, sur d'autres entrées. On dira donc qu'on exécute une procédure ; et que l'exécution de plusieurs procédures constitue l'exécution de tout l'algorithme.

Exemple 4. Exécutons Accès([4, 5, 9, 30, 18], 3).

	LIGNE	SUIVANTE	e	Q
Accès([4, 5, 9, 30, 18], 3)	2	5		
	5	6	4	[5, 9, 30, 18]
	6	9	4	[5, 9, 30, 18]
	9	*	4	[5, 9, 30, 18]
Accès([5, 9, 30, 18], 2)	2	5		
	5	6	5	[9, 30, 18]
	6	9	5	[9, 30, 18]
	9	*	5	[9, 30, 18]
Accès([9, 30, 18], 1)	2	5		
	5	6	9	[30, 18]
	6	9	9	[30, 18]
	9	*	9	[30, 18]
Accès([30, 18], 0)	2	5		
	5	6	30	[18]
	6	7	30	[18]
	7		renvoyer 30	

On peut se demander quelle est la différence avec l'exécution d'un algorithme itératif : en effet, ici, la variation serait stockée par une colonne de plus représentant l'itérateur, tandis qu'ici, on doit représenter cette colonne en plus par la variation des entrées, que l'on doit bien représenter d'une manière ou d'une autre...

En fait, tout ce qui peut être programmé de manière récursive peut être programmée de manière itérative, et l'inverse est souvent vrai. Néanmoins, les deux méthodes sont à connaître : il s'agit de deux manières différentes de penser au même objet.

Cet algorithme peut facilement être adapté pour la modification : la seule différence est que, quand on parcours un élément, au lieu de renvoyer le résultat du calcul sur la queue de la liste, on renvoie la concaténation de l'élément considéré et du résultat sur la queue de la liste. On obtient ainsi l'Algorithm 10.

Entrées:

- une liste d'éléments L ;
- un entier k ;
- un élément a .

```

1 Modification(L, k, a)
2   si L ≡ Λ alors
3     retourner erreur
4   sinon
5     e :: Q ← L
6     si k ≡ 0 alors
7       retourner a :: Q
8     sinon
9       retourner e :: Modification(Q, k - 1, a)
10    fin
11  fin

```

Sorties: une liste ou une erreur

Algorithm 10: Liste — modification

Pour l'exécuter, on voit que la convention prise précédemment ne suffit pas : en effet, ici, après avoir récupéré un résultat calculée sur la queue, il reste du travail à faire (concaténer une nouvelle tête), on ne peut pas donc juste appeler et retourner le résultat de la procédure appelée. On dit que cet algorithme n'est pas *récursif terminal*.

Exercice 31. Donnez une version récursive terminale de cet algorithme.

Néanmoins, on veut en représenter des traces d'exécutions. Pour cela, on va utiliser une flèche pointillée allant du point appelant (l'étoile de l'exemple du dessus) à la procédure appelée, et une autre flèche, pleine, allant du résultat de la procédure appelée à l'endroit où il est utilisé.

Exemple 5. Exécutons $\text{Modification}([4, 5, 9, 30, 18], 3, 12)$. On se rend compte que e et Q ne sont pas vraiment utilisés, et juste transmis : on arrête donc de les noter. La trace d'exécution est représentée Figure 2.1.

On peut aussi tenter de schématiser ce qui se passe : la liste résultat est construite en concaténant des têtes et d'autres résultats :

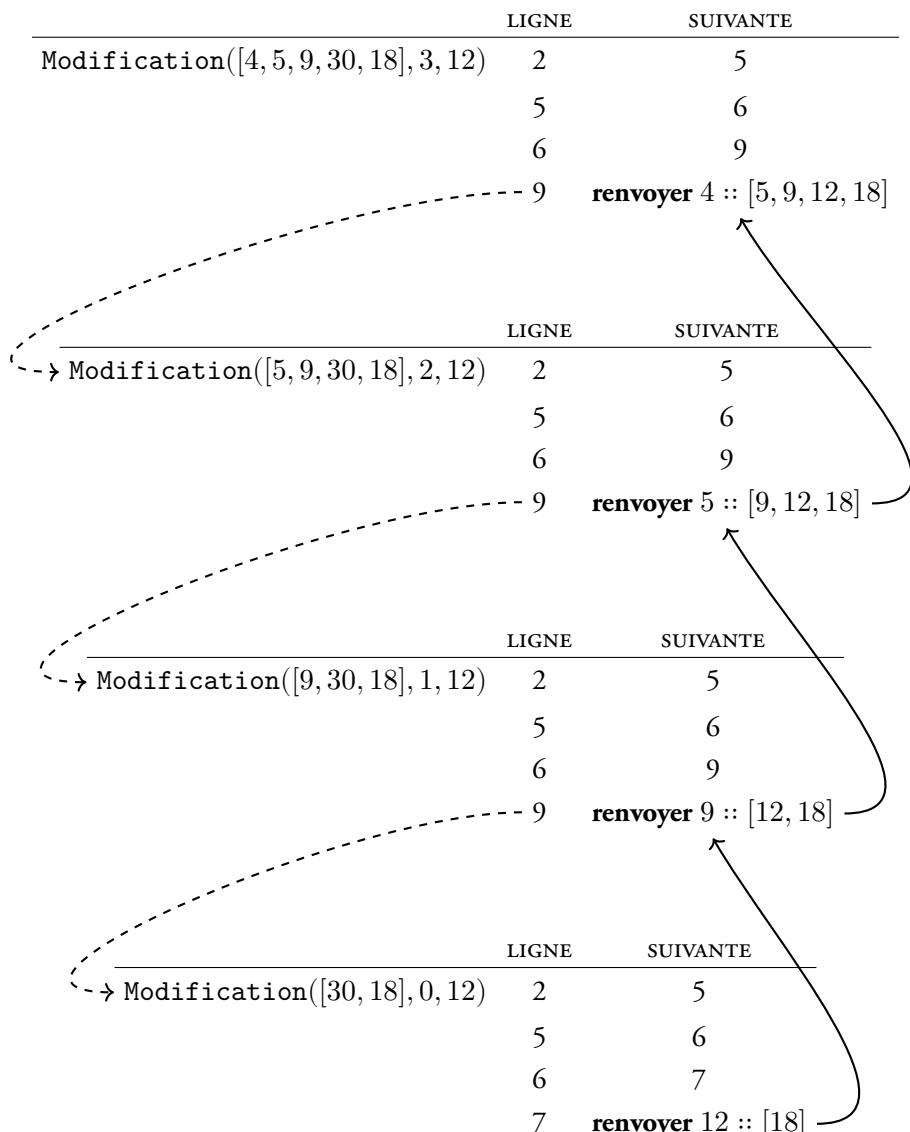
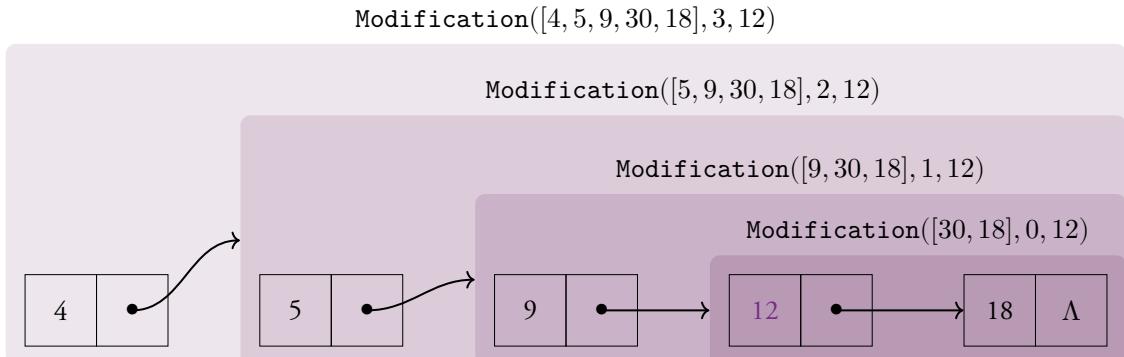


FIGURE 2.1 – Une trace d'exécution récursive



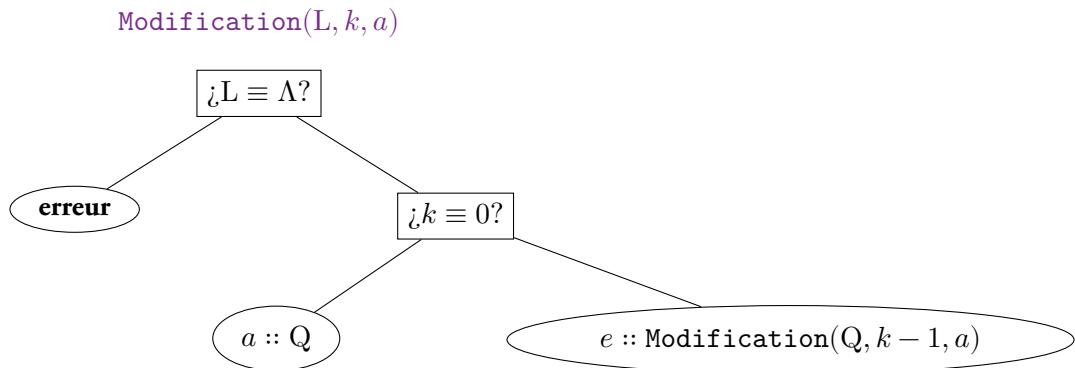
On voit que le seul point où une modification a vraiment lieu est quand l'indice à modifier vaut 0 ; le reste n'est qu'un parcours.

On peut aller encore plus loin dans la représentation de cette exécution : en fait, on a supprimé la

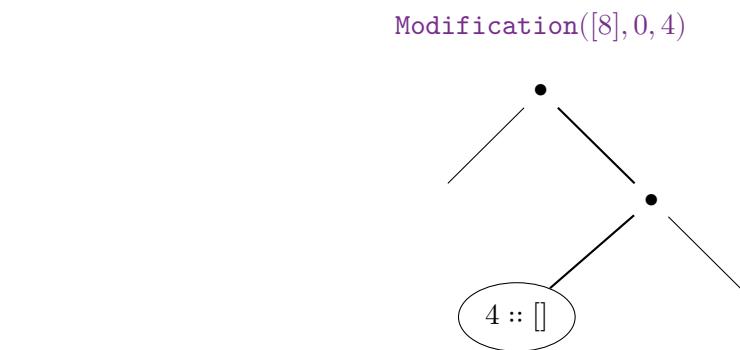
représentation des variables e et Q parce qu'elles ne varient pas. Il ne reste dans la trace d'exécution que les informations suivantes :

- le nom de la procédure;
 - la valeur de ses arguments;
 - l'embranchement choisi dans les disjonctions de cas créées par les *si*.

En effet, l'algorithme tel qu'écrit définit un arbre



et on peut intégralement représenter son exécution par le chemin pris dans cet arbre, ainsi que la valeur de retour. On pourra donc représenter l'exécution de `Modification([8], 0, 4)` par l'arbre suivant :



Donc, l'exécution complète de `Modification`([4, 5, 9, 30, 18], 3, 12) peut-être représentée, en reprenant le principe des flèches mais en remplaçant les tableaux par des arbres, par la Figure 2.2.

On constate que récupérer ou modifier un élément dans une liste chaînée nécessite potentiellement de la parcourir en entier. Ainsi, la complexité de ces deux algorithmes est linéaire.

Adjonction & suppression

Pour ce qui est de l'ajout et la suppression d'un élément, on a juste besoin d'adapter très légèrement l'algorithme pour la modification : en effet, dans le cas de la modification, une fois récupéré l'élément qui nous intéresse, ce qui a lieu quand on cherche l'élément d'indice 0, on le modifie. Ici, on va plutôt rajouter un à ce point, ou au contraire le supprimer ; c'est-à-dire, schématiquement, remplacer un pointeur représenté par une flèche continue en deux pointeurs en pointillé, et inversement. En effet, ces deux opérations sont parfaitement symétriques.

Exercice 32. Exécutez `Ajout([4, 5, 9, 30, 18], 3, 12)`.

Exercice 33. Pourquoi dans l’Algorithme 11, on teste d’abord si la position de l’ajout est nulle, est seulement alors si la liste est vide?

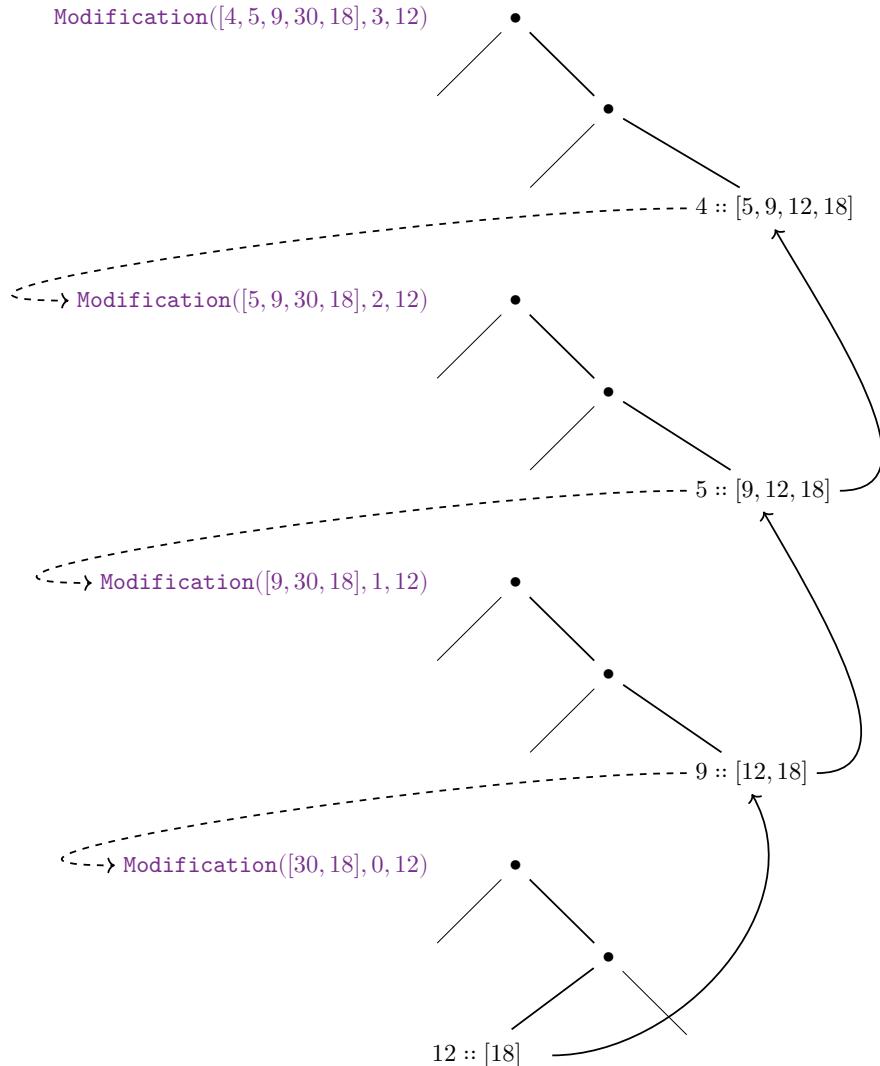
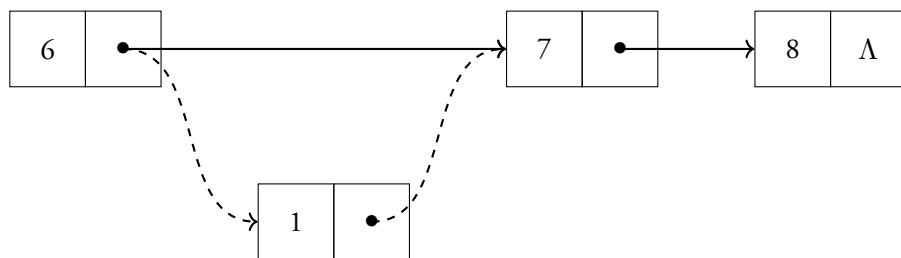


FIGURE 2.2 – La trace de la Figure 2.1, sous forme d'arbre de décision



Exercice 34. Exécutez **Suppression([4, 5, 9, 30, 18], 3)** et **Suppression([4, 5, 9, 30, 18], 6)**.

Là encore, il est nécessaire de parcourir la liste, potentiellement intégralement, pour ajouter ou supprimer un élément. La complexité de ces deux opérations est linéaire. Toutefois, les listes chaînées ont un avantage sur les tableaux : on n'a pas à recopier intégralement le contenu des listes — si on s'intéressait à la complexité en espace (c'est-à-dire à l'espace mémoire pris pas les opérations), on dirait que ces opérations sont de complexité constante en mémoire, alors que pour les tableaux, elles sont de complexité linéaire ; mais nous avons choisi de ne pas nous y intéresser.

Entrées:

- une liste d'éléments L;
- un entier k ;
- un élément a .

```

1 Ajout(L, k, a)
2   si  $k \equiv 0$  alors
3     retourner  $a :: L$ 
4   sinon
5     si  $L \equiv \Lambda$  alors
6       retourner erreur
7     sinon
8        $e :: L \leftarrow L$ 
9       retourner  $e :: \text{Ajout}(L, k - 1, a)$ 
10    fin
11 fin

```

Sorties: une liste**Algorithme 11:** Liste — ajout d'un élément**Entrées:**

- une liste d'éléments L;
- un entier k .

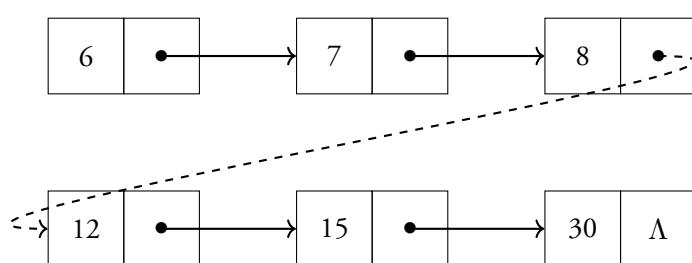
```

1 Suppression(L, k)
2   si  $L \equiv \Lambda$  alors
3     retourner erreur
4   sinon
5      $e :: Q \leftarrow L$ 
6     si  $k \equiv 0$  alors
7       retourner Q
8     sinon
9       retourner  $e :: \text{Suppression}(Q, k - 1)$ 
10    fin
11 fin

```

Sorties: une liste ou une erreur**Algorithme 12:** Liste — suppression d'un élément**Fusion**

Pour la fusion de deux listes, il suffit de parcourir la première, et de remplacer le symbole de fin de liste par un pointeur vers la deuxième liste.



Entrées: deux liste d'éléments L et L'.

```

1 Fusion(L, L')
2   | si L ≡ Λ alors
3   |   | retourner L'
4   | sinon
5   |   | e :: Q ← L
6   |   | retourner e :: Fusion(Q, L')
7   | fin

```

Sorties: une liste

Algorithme 13: Liste — fusion de deux listes

Exercice 35. Exécutez $\text{Fusion}([4, 5, 9, 30, 18], [0, 12, 3])$.

On voit que cet algorithme est dissymétrique: on parcours intégralement la première liste, et pas du tout la seconde. Ainsi, il est linéaire en la longueur de la première liste, et constant en celle de la seconde; c'est-à-dire, si on note n la longueur de la première liste et m celle de la seconde, en $O(n)$.

Longueur

Autant calculer la longueur d'un tableau était immédiat (sa longueur était une des entrées), autant ce n'est pas le cas pour une liste. Il n'y a guère d'autre possibilité que de parcourir une liste chaînée en entier pour récupérer sa longueur, d'où l'Algorithme 14.

Entrées: une liste d'élément L.

```

1 Longueur(L)
2   | si L ≡ Λ alors
3   |   | retourner 0
4   | sinon
5   |   | e :: Q ← L
6   |   | retourner 1 + Longueur(Q)
7   | fin

```

Sorties: un nombre entier positif

Algorithme 14: Liste — longueur

Exercice 36. Exécutez $\text{Longueur}([4, 5, 9, 30, 18])$.

Cet algorithme a donc une complexité linéaire.

Résumé

	Accès	Modification	Ajout	Suppression	Fusion	Longueur
Liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Exercice 37. Soit L, une liste chaînée de nombres. On veut récupérer son plus grand élément. Écrire un algorithme prenant une liste en entrée et renvoyant son plus grand élément.

2.4 LA RÉCURSION

Une liste chaînée est un exemple de structure *récursive*, c'est-à-dire de structure faisant appel à elle-même dans sa propre définition. Nous allons voir trois types d'objets récursifs : des algorithmes, des propriétés et des structures.

Définition 3 (structures récursives). Une *structure récursive* est donnée par un ensemble fini de *constructeurs* de deux types :

constructeurs récursifs qui construisent une instance de la structure à partir de plusieurs objets, dont d'autres instances de la structure ;

constructeurs de base qui construisent une instance de la structure à partir de plusieurs objets, mais aucune instance de la structure.

Une *instance* d'une structure récursive est donnée par l'application d'un nombre fini de constructeurs.

Exemple 6. Les nombres entiers naturels sont une structure récursive : en effet, il y a deux constructeurs :

- 0, qui construit un nombre entier à partir de rien ;
- S, le successeur, qui construit un nombre entier à partir d'un autre nombre entier (en lui rajoutant un).

Ceci est une manière de reformuler que chaque nombre entier est soit égal à zéro, soit un autre entier plus un.

Ainsi, 5 est un entier naturel, car $5 = S(S(S(S(0))))$.

Exemple 7. Les *listes chaînées* sont une structure récursive : en effet, il y a deux constructeurs :

- Λ , qui construit une liste (la liste vide) à partir de rien ;
- (\cdot, \cdot) qui construit une liste à partir d'un élément et d'une liste.

Définition 4 (algorithmes récursifs). Un *algorithme récursif* est donné par une disjonction de cas parmi lesquels les cas de bases, qui ne font pas appel à des résultats de l'algorithme sur d'autres valeurs, et les cas récursifs, qui font appels aux résultats de l'algorithme sur d'autres valeurs pour calculer le résultat demandé.

Un algorithme récursif ne termine que s'il y a moyen d'ordonner les entrées de manière à ce que les entrées sur lesquels l'algorithme soient de plus en plus petites au cours de l'exécution : c'est le cas des algorithmes sur les listes qu'on a écrit : à chaque fois, ils s'appellent sur la *queue* des listes considérées avant, qui sont bien plus petites que la liste entière.

Postulat 2 (référence structurelle). *Si une propriété est :*

- *vraie sur toutes les instances obtenues par application des constructeurs de base* ;
- *telle que si elle est vraie sur toutes les instances nécessaires aux constructeurs récursifs, elle est aussi vraie sur l'instance construite*

alors elle est vraie sur toutes les instances d'une structure récursive.

Ce postulat peut se prouver dans différents systèmes d'axiomes. Néanmoins, il est bien quelque chose de fondamental, que l'on sait vrai du fait de notre expérience de la finitude : les systèmes d'axiomes sont construits pour prouver ce postulat.

2.5 PILES

De ce qu'on a vu des listes, elles ont l'air brutalement inefficaces ; ce n'est pas tout à fait le cas. En fait, avec une *liste chaînée*, on ne peut agir facilement que sur la tête de la liste : la *récursivité* nous permet d'agir ailleurs, mais au prix d'un parcours, nécessairement en temps linéaire. Ainsi, dans certaines

situations, on ne s'intéresse pas à l'ordre des éléments, juste que l'on puisse facilement en rajouter et en récupérer pour les traiter. Dans ce cas, on peut se servir d'une *liste chaînée* comme d'une *pile*, c'est-à-dire une structure de données sur laquelle on ne peut agir qu'au niveau de la *tête*. On se retrouve comme ça avec des algorithmes simplifiés (les Algorithmes 15 à 18), et tous en temps constant ! Évidemment, on peut faire moins de choses, mais plus efficacement.

Entrées: une liste d'éléments L.

```

1 Accès(L)
2   | si L == Λ alors
3   |   | retourner erreur
4   | sinon
5   |   | (e, Q) ← L
6   |   | retourner e
7 fin

```

Sorties: un élément ou une erreur

Algorithm 15: Pile — accès à la tête

Entrées:

- une liste d'éléments L;
- un élément a.

```

1 Modification(L, a)
2   | si L == Λ alors
3   |   | retourner erreur
4   | sinon
5   |   | (e, Q) ← L
6   |   | retourner (a, Q)
7 fin

```

Sorties: une liste ou une erreur

Algorithm 16: Pile — modification de la tête

Entrées:

- une liste d'éléments L;
- un élément a.

```

1 Ajout(L, a)
2   | retourner (a, L)

```

Sorties: une liste

Algorithm 17: Pile — ajout d'une tête

On remarque que la modification peut être vue comme la suppression suivie de l'ajout.

2.6 RÉSUMÉ

Un tableau gagne à être utilisé quand on ne modifie pas le nombre d'éléments ; une liste gagne à être utilisée sous forme de pile, c'est-à-dire en n'agissant que sur sa tête.

Entrées: une liste d'éléments L.

```

1 Suppression(L)
2   | si L == Λ alors
3   |   | retourner erreur
4   | sinon
5   |   | (e, Q) ← L
6   |   | retourner Q
7   | fin

```

Sorties: une liste ou une erreur

Algorithme 18: Pile — suppression d'une tête

	Accès	Modification	Ajout	Suppression	Fusion	Longueur
Tableau	O(1)	O(1)	O(n)	O(n)	O(n + m)	O(1)
Liste	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
Pile	O(1)	O(1)	O(1)	O(1)	—	—

2.A INVERSION CORRECTION

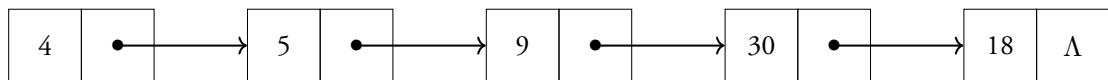
On veut inverser l'ordre dans lequel les éléments d'une liste ou d'un tableau sont écrits. Par exemple, si on avait un tableau

6	-8	35	42	-90	27	12	0	0	87
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]	TAB[6]	TAB[7]	TAB[8]	TAB[9]

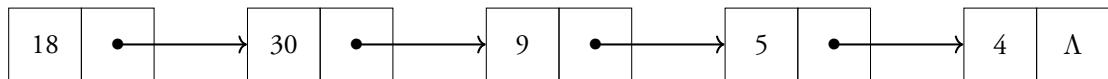
On veut le remplacer par le tableau

87	0	0	12	27	-90	42	35	-8	6
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]	TAB[6]	TAB[7]	TAB[8]	TAB[9]

Et de même, pour la liste



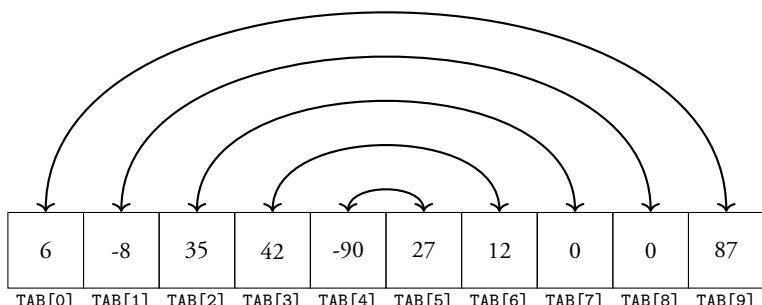
On veut passer à la liste



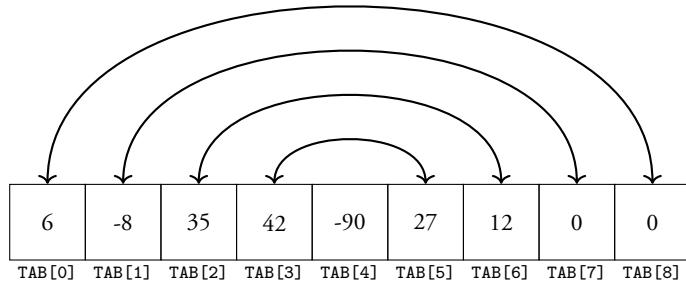
Exercice 38. Donner un algorithme Inversion qui inverse un tableau.

Indication: Vous pourrez considérer que la division est exacte (c'est à dire que $3/2 == \frac{3}{2}$) ou entière (c'est à dire que $3/2 == 1$).

Correction: On va parcourir la première moitié du tableau et échanger chaque élément avec son symétrique de l'autre côté. Ainsi, sur l'exemple, on va donc échanger comme ceci :



En notant bien que si le tableau était de longueur impaire, il faudrait laisser inchangé l'élément du milieu :



Pour échanger la valeur de deux variables x et y , le plus simple est de considérer une nouvelle variable a , de déposer la valeur de la première variable x dans cette nouvelle variable a de manière temporaire, d'assigner la valeur de y à x , et enfin, la valeur de a à y . De cette manière, on s'évite d'écraser une valeur.

On va donc vouloir faire une boucle, allant de 0 jusqu'à la moitié de la longueur du tableau. On doit s'interroger sur la situation au milieu du tableau. On a deux possibilités, tout aussi légitimes : soit on va échanger l'élément du milieu avec lui-même, soit le laisser inchangé, dans tous les cas, le résultat sera le même. On obtient un résultat ou l'autre selon la condition de fin de la boucle : si on fait boucler jusqu'à la moitié incluse, ou exclue.

Si on considère, comme le dit l'indication, que la division est exacte, c'est-à-dire que $3/2$ est égal à la fraction $\frac{3}{2}$ (et pas à 1 comme en C), alors une boucle allant jusqu'à $3/2$ va s'exécuter pour la variable de boucle égale à 1, et donc échange l'élément du milieu.

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

1 **Inversion** $((\text{TAB}[i])_{0 \leq i < N})$

2 **nouveau** a

3 **nouveau** ℓ

4 $\ell \leftarrow N/2$

5 **pour** i allant de 0 à ℓ **faire**

6 $a \leftarrow \text{TAB}[i]$

7 $\text{TAB}[i] \leftarrow \text{TAB}[N - 1 - i]$

8 $\text{TAB}[N - 1 - i] \leftarrow a$

9 **fin**

10 **retourner** $(\text{TAB}[i])_{1 \leq i < N}$

Sorties: un tableau de longueur N

Algorithme 19: Inversion d'un tableau

Une autre possibilité est d'utiliser un deuxième tableau. Ce n'est pas celle que j'avais en tête, mais elle est correcte.

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

1 **Inversion** $((\text{TAB}[i])_{0 \leq i < N})$

2 **nouveau** $(\text{TAB}'[i])_{0 \leq i < N}$

3 **pour** i allant de 0 à N **faire**

4 $\text{TAB}'[N - 1 - i] \leftarrow \text{TAB}[i]$

5 **fin**

6 **retourner** $(\text{TAB}'[i])_{1 \leq i < N}$

Sorties: un tableau de longueur N

Algorithme 20: Inversion d'un tableau

⊕

Remarques:

- A. Si vous **retournez**, l'algorithme s'arrête : en particulier, il n'est pas possible de retourner plusieurs valeurs. Donc retourner $\text{TAB}[i]$ retourne un élément, et ne peut pas être utilisé pour retourner un tableau.

- B. J'ai choisi de faire l'inversion *en place*, c'est à dire sur le même tableau. Vous pouvez tout à fait créer un deuxième tableau et le remplir.
- C. N'utilisez que des variables que vous avez affecté ou qui sont des entrées.
- D. Soyez cohérent sur vos entrées : si vous avez noté une entrée, elle doit apparaître dans la liste des arguments et vice-versa.
- E. Vos variables de travail ne sont pas des entrées.
- F. Vous ne pouvez pas créer un tableau d'une longueur non-spécifiée.
- G. Attention aux indices ($N - i, \dots$).

Exercice 39. Exécuter cet algorithme sur le tableau

4	5	9	30	18
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]

Correction : L'algorithme utilise comme variable les différents éléments du tableau, ainsi que a et ℓ , et i ! N est fixé à 5.

LIGNE	SUIVANTE	a	ℓ	i	TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]
1	2				4	5	9	30	18
2	3				4	5	9	30	18
3	4				4	5	9	30	18
4	5				4	5	9	30	18
5	6				4	5	9	30	18
6	7	4	5	0	4	5	9	30	18
7	8	4	5	0	18	5	9	30	18
8	9	4	5	0	18	5	9	30	4
9	5	4	5	0	18	5	9	30	4
5	6	4	5	1	18	5	9	30	4
6	7	5	5	1	18	5	9	30	4
7	8	5	5	1	18	30	9	30	4
8	9	5	5	1	18	30	9	5	4
9	5	5	5	1	18	30	9	5	4
5	6	5	5	2	18	30	9	5	4
6	7	9	5	2	18	30	9	5	4
7	8	9	5	2	18	30	9	5	4
8	9	9	5	2	18	30	9	5	4
9	10	9	5	2	18	30	9	5	4
10									

Renvoyer le tableau



Considérons l'algorithme

Entrées: deux listes d'éléments L et L' .

```

1 InversionAprès(L,L')
2   si L == [] alors
3     retourner L'
4   sinon
5     (e,Q) ← L
6     retourner InversionAprès(Q,(e,L'))
7   fin

```

Sorties: une liste d'éléments

Remarques:

- A. Dans un tableau d'exécution, on doit avoir une colonne pour la ligne, la ligne suivante, et chaque variable. pas de colonne pour un « résultat » qui n'a pas été introduit dans l'algorithme.

	LIGNE	SUIVANTE	e	Q
<code>InversionAprès((4, (5, (9, Λ))), (30, (18, Λ)))</code>	1	2		
	2	5		
	5	6	4	(5, (9, Λ))
	6	*	4	(5, (9, Λ))
<code>InversionAprès((5, (9, Λ)), (4, (30, (18, Λ))))</code>	1	2		
	2	5		
	5	6	5	(9, Λ)
	6	*	5	(9, Λ)
<code>InversionAprès((9, Λ), (5, (4, (30, (18, Λ))))</code>	1	2		
	2	5		
	5	6	9	Λ
	6	*	9	Λ
<code>InversionAprès(Λ, (9, (5, (4, (30, (18, Λ))))))</code>	1	2		
	2	3		
	3	Renvoyer (9, (5, (4, (30, (18, Λ)))))		

FIGURE 2.3 – Exécution d’InversionAprès

B. Soyez cohérents dans vos traces d’exécution : si vous passez une fois par la ligne de fin d’une boucle, passez-y à chaque fois.

Exercice 40. Exécuter `InversionAprès((4, (5, (9, Λ))), (30, (18, Λ)))`.

Correction: Cet algorithme est récursif terminal, aussi, on peut tout représenter avec un seul tableau d’exécution, où on précisera les arguments d’appel. Les variables sont e et Q . L’exécution est représentée Figure 2.3. ☺

Remarques:

- A. Si vous introduisez des notations (des flèches,...), introduisez-les.
- B. Ne confondez pas $Λ$ (la liste vide) et $(Λ)$ (je ne sais même pas ce que c’est). De même, $(5, 9, 8, Λ)$ n’est pas une liste, c’est un quadruplet d’éléments. $(5, (9, (8, Λ)))$ est une liste.
- C. Le premier appel de `InversionAprès` ne retourne pas $(5, (9, Λ)), (4, (30, (18, Λ)))$ (et encore moins $Q, (e, L)$) mais le résultat de `InversionAprès((5, (9, Λ)), (4, (30, (18, Λ))))`.

Exercice 41. Que fait `InversionAprès`?

Correction: `InversionAprès` prend en entrée deux listes : elle inverse l’ordre de la première puis y concatène, à sa suite, la seconde. ☺

Exercice 42. Écrire un algorithme d’inversion de liste faisant appel à `InversionAprès`.

Correction: ☺

Entrées: une liste d’éléments L .

- 1 `Inversion(L)`
 - 2 | `retourner InversionAprès(L, Λ)`
- Sorties:** une liste d’éléments

Dans une pile, on rajoute les éléments qui arrivent en haut, et on retire les éléments en haut de la pile aussi. On dit que la politique est FILO (*first in, last out*). Ainsi, si on considère qu’un programme un système devant traiter des dossiers (par exemple une administration), ça ne semble pas une structure de

données adaptée : un dossier peut prendre un temps arbitrairement long pour être traité, si des dossiers continuent d'arriver après.

On appelle une *file* une structure de donnée permettant d'appliquer une politique FIFO (*first in, first out*).

Exercice 43. Expliquer comment on peut réaliser une file avec deux piles.

Évaluer la complexité des opérations.

Correction: Il suffit de considérer deux piles D et F :

- D contiendra le début de la file, dans l'ordre ;
- F contiendra la fin de la file, dans l'ordre inverse.

On rajoutera les éléments en tête de D, et on les supprimera en tête de F.

Ces deux opérations sont donc en O(1), comme pour des piles. Néanmoins, il arrive que la pile F soit vide. Dans ce cas, on supprime un à un les éléments de D, dans l'ordre (chaque suppression se fait en O(1) donc), et on les insère dans F, ce qui est aussi en O(1). Cela inverse donc l'ordre des éléments. Ainsi, on a une opération de complexité linéaire à faire à chaque fois que F est vide.

Néanmoins, cette opération arrive rarement : en effet, un élément donné, ajouté dans D, se fera retourner exactement une fois dans F avant d'être supprimé.

Une file construite de cette manière fait donc l'ajout et la suppression en O(1), et permet d'accéder et modifier le premier et le dernier élément en O(1) aussi. ☺

Exercice 44 (bonus). Écrire l'algorithme de l'exercice 38 en C.

Indication : on attend une fonction dont le prototype est

```
1 void inversion(int* tab, int longueur);
```

qui prend en entrée un tableau `tab` (c'est le sens de `int*` : un tableau d'entiers, de longueur non spécifiée) et sa longueur `longueur`, et ne renvoie rien (`void`). La syntaxe pour accéder à la *i*-ème case du tableau `tab` est `tab[i]`. Comme dans le cours, les tableaux commencent à l'indice 0.

Correction:

```
1 void inversion(int* tab, int longueur){
2     int a;
3     for(int i = 0; i <= longueur/2; i++){
4         a = tab[i];
5         tab[i] = tab[longueur-1-i];
6         tab[longueur-1-i] = a;
7     }
8 }
```

On peut le tester avec :

```
1 int main(){
2     int tab[5] = {4, 5, 9, 30, 18};
3     printf("%d %d %d %d %d\n", tab[0], tab[1], tab[2], tab[3], tab[4]);
4     inversion(tab,5);
5     printf("%d %d %d %d %d", tab[0], tab[1], tab[2], tab[3], tab[4]);
6     return 0;
7 }
```

☺

Remarques:

A. Je demande ici un programme en C.

2.B D'UN TABLEAU À UNE LISTE ET VICE-VERSA

Parfois, on peut vouloir changer de structure de données, par exemple si on se retrouve à devoir faire des opérations qui étaient rares (on peut par exemple supposer que pendant une année universitaire, il n'y a pas d'inscriptions, mais des changements de groupe, et utiliser un tableau, mais à la rentrée, préférer utiliser une liste).

Exercice 45. Écrire un algorithme **ListeDe** prenant en entrée un tableau et retournant une liste contenant les mêmes éléments dans le même ordre.

Correction : On a envie de commencer par parcourir les éléments du tableau (avec une boucle **pour**) et mettre chaque élément en tête de liste. Le problème est qu'en faisant ainsi, on inverse l'ordre des éléments du tableau. On peut donc:

- inverser le tableau avant la boucle;
- inverser la liste après la boucle;
- lire le tableau à l'envers, en commençant par son dernier élément.

On va plutôt préférer la troisième solution, plus économique (on ne lit chaque élément qu'une fois, au lieu de deux). On n'a pas donné de syntaxe particulière pour faire des boucles décroissantes, plutôt que d'en inventer, on va lire l'élément $N - 1 - i$ du tableau au i -ème passage dans la boucle.

Entrées: un tableau $(TAB[i])_{0 \leq i < N}$ de longueur N .

```

1 ListeDe((TAB[i])0 ≤ i < N)
2   nouveau L
3   L ← Λ
4   pour i allant de 0 à N faire
5     |   L ← (TAB[N - 1 - i], L)
6   fin
7   retourner L
Sorties: une liste
```

On aurait pu faire autrement. Parmi les possibilités, on pouvait faire un algorithme récursif qui, en plus du tableau, prenne en argument la liste en train d'être construite ainsi que l'indice du tableau que l'on est en train de copier. On voit particulièrement sur cet exemple à quel point les boucles et les appels récursifs ont le même rôle: ici, c'est peu naturel, on se retrouve à gérer à la main l'incrémentation et la condition d'arrêt de la boucle. Cela donne un premier algorithme: que l'on

Entrées:

- un tableau $(TAB[i])_{0 \leq i < N}$ de longueur N ;
- un entier i ;
- une liste L .

```

1 ListeDeRec((TAB[i])0 ≤ i < N, i, L)
2   si i ≠ N alors
3     |   retourner ListeDeRec((TAB[i])0 ≤ i < N, i + 1, (TAB[N - 1 - i], L))
4   sinon
5     |   retourner L
6   fin
Sorties: une liste
```

appelle au sein d'un deuxième:



Entrées: un tableau $(TAB[i])_{0 \leq i < N}$ de longueur N

```

1 ListeDe((TAB[i])0 ≤ i < N)
2   retourner ListeDeRec((TAB[i])0 ≤ i < N, 0, Λ)
Sorties: une liste
```

Exercice 46. L'exécuter sur le tableau

4	5	9	30	18
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]

Correction: On a, comme variables, i et L . D'où la trace d'exécution, où l'on représente en une seule étape les lignes 1, 2 et 3 ; ainsi que la fin et le début de la boucle :

LIGNE	SUIVANTE	i	L
1-2-3	4	0	Λ
4	5	0	Λ
5	6	0	(18, Λ)
6-4	5	1	(18, Λ)
5	6	1	(30, (18, Λ))
6-4	5	2	(30, (18, Λ))
5	6	2	(9, (30, (18, Λ)))
6-4	5	3	(9, (30, (18, Λ)))
5	6	3	(5, (9, (30, (18, Λ))))
6-4	5	4	(5, (9, (30, (18, Λ))))
5	6	4	(4, (5, (9, (30, (18, Λ))))
6	7	0	(4, (5, (9, (30, (18, Λ))))
7	Retourner L		

⊕

Exercice 47. Écrire un algorithme TableauDe prenant en entrée une liste et retournant un tableau contenant les mêmes éléments dans le même ordre.

Indication: n'hésitez pas à commencer par calculer la longueur de la liste.

Correction: On commence par calculer la longueur de la liste, en faisant appel à l'Algorithme 14.

On remarque que dans la boucle, on n'a pas besoin de vérifier que la liste L soit vide : vu qu'on itère sur sa longueur, tout doit bien se passer (et on peut d'ailleurs le prouver. En fait, il s'agit d'un **invariant de boucle**). De même ici, on a utilisé une

Entrées: une liste L .

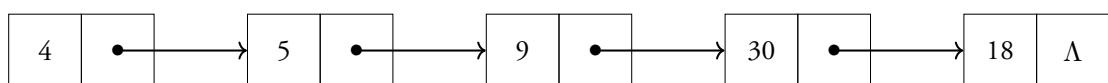
```

1 TableauDe(L)
2   nouveau N
3   nouveau TAB[i]_0 ≤ i < N
4   N ← Longueur(L)
5   pour i allant de 0 à N faire
6     (e, L) ← L
7     TAB[i] ← e
8   fin
9   retourner TAB[i]_0 ≤ i < N
Sorties: un tableau
```

boucle, on pouvait faire à la place des appels récursifs.

⊕

Exercice 48. L'exécuter sur la liste



Correction: Les variables ici sont i , L , N et e , ainsi que chacune des cases du tableau TAB.

LIGNE	SUIVANTE	L	N	e	i	TAB				
						[0]	[1]	[2]	[3]	[4]
1-2-3	4	(4, (5, (9, (30, (18, Λ))))))	5		0					
4	5	(4, (5, (9, (30, (18, Λ))))))	5	4	0					
5	6	(4, (5, (9, (30, (18, Λ))))))	5	4	0	4				
6	7	(5, (9, (30, (18, Λ))))	5	5	1	4				
7	8	(5, (9, (30, (18, Λ))))	5	5	1	4	5			
8-5	6	(5, (9, (30, (18, Λ))))	5	5	2	4	5			
6	7	(9, (30, (18, Λ)))	5	9	2	4	5			
7	8	(9, (30, (18, Λ)))	5	9	2	4	5	9		
8-5	6	(9, (30, (18, Λ)))	5	9	3	4	5	9		
6	7	(30, (18, Λ))	5	30	3	4	5	9		
7	8	(30, (18, Λ))	5	30	3	4	5	9	30	
8-5	6	(30, (18, Λ))	5	30	4	4	5	9	30	
6	7	(18, Λ)	5	18	4	4	5	9	30	
7	8	(18, Λ)	5	18	4	4	5	9	30	18
8	9	(18, Λ)	5	18	4	4	5	9	30	18
9	Retourner le tableau									

⊕

2.C LA MULTIPLICATION EN BASE 10

Représentation des nombres

Pour exécuter l'algorithme de la multiplication posée, en base 10, on ne travaille pas sur des nombres, mais sur des listes de chiffres : en effet, on va multiplier séparément le chiffre des unités, puis celui des dizaines, et ainsi de suite. De plus, on va le faire de manière *petit-boutienne*, c'est-à-dire qu'on commence avec les chiffres les moins importants.

On va représenter les nombres par des tableaux d'entiers, chaque entier étant entre 0 et 9 (inclus). Par exemple, le nombre 873 sera représenté par le tableau

3	7	8
0	1	2

Néanmoins, certains autres tableaux peuvent représenter le même nombre, ainsi

13	6	8	0
0	1	2	3

peut aussi être vu comme représentant 873.

On dira qu'un tableau d'entier est *sous forme standard* si tous ses éléments sont entre 0 et 9 et qu'il ne termine pas par un 0. La forme standard d'un tableau est le tableau sous forme standard représentant le même nombre en base 10.

Exercice 49. Donner une forme standard pour les tableaux suivants :

13	5	8	0
0	1	2	3

2	25	8	0	0
0	1	2	3	4

0	0
0	1

2	5	328
0	1	2

Correction: Ces tableaux représentent respectivement les nombres 863, 1052, 0 et 32825. Leurs formes standard sont donc, respectivement :

3	6	8
0	1	2

2	5	0	1
0	1	2	3

0
0

2	5	8	2	3
0	1	2	3	4

⊕

Barème :

— $\frac{1}{4}$ par tableau juste.

Remarques :

1. Un nombre n'est pas un tableau, ainsi, si on demande la forme normale d'un tableau, c'est un tableau qu'il faut rendre.

Exercice 50. Écrire un algorithme prenant en entrée un tableau et renvoyant un autre tableau, dont les premières cases sont identiques, mais sans ses éventuels 0 finals.

Correction: On va parcourir le tableau en commençant par sa dernière case, et compter le nombre de cases qui sont des 0, en s'arrêtant avant la zéroième case (pour faire cela, on utilise une boucle **tant que**). On va pouvoir se servir de ce nombre pour créer un tableau de la bonne taille, et recopier chacune des cases intiales du tableau de départ dedans.

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

- 1 **ZérosFinals** $((\text{TAB}[i])_{0 \leq i < N})$
- 2 $\ell \leftarrow N$
- 3 **tant que** $\ell > 1$ **et** $\text{TAB}[\ell - 1] \equiv 0$ **faire**
- 4 $\ell \leftarrow \ell - 1$
- 5 **fin**
- 6 **nouveau** $(\text{RES}[j])_{0 \leq j < \ell}$
- 7 **pour** j **allant de** 0 **à** ℓ **faire**
- 8 $\text{RES}[j] \leftarrow \text{TAB}[j]$
- 9 **fin**
- 10 **retourner** $(\text{RES}[j])_{0 \leq j < \ell}$

Sorties: un tableau de longueur ℓ

La boucle **tant que** ne regarde pas la zéroième case du tableau. ⊕

Barème:

- $\frac{1}{6}$ pour les entrées
- $\frac{1}{6}$ pour les sorties
- $\frac{1}{6}$ pour le nom, les noms des entrées, la valeur de retour, cohérents avec les entrées et les sorties
- $\frac{1}{6}$ pour un tableau de la bonne longueur
- $\frac{1}{6}$ pour la présence d'une boucle
- $\frac{1}{6}$ pour une boucle correcte

Remarques:

1. Un algorithme correct est correct sur toutes les entrées possibles, pas seulement sur les exemples.

Exercice 51. Exécuter cet algorithme sur le deuxième et le troisième tableau.

Correction: Les variables sont ℓ, j , ainsi que toutes les cases du tableau $(\text{RES}[j])_{0 \leq j < \ell}$. Ce tableau a une longueur qui va dépendre de la valeur de ℓ , aussi, ça n'a pas vraiment de sens de les marquer dès le début de la trace d'exécution.

On va appeler $(\text{TAB}_2[i])_{0 \leq i < 5}$ et $(\text{TAB}_3[i])_{0 \leq i < 2}$ les deux tableaux à traiter.

ZérosFinals $((\text{TAB}_2[i])_{0 \leq i < 5})$			
LIGNE	ℓ	j	
1-2	5		
3	5		
4	4		
5	4		
3	4		
4	3		
5	3		
6	3		$\text{RES}[0] \quad \text{RES}[1] \quad \text{RES}[2]$
7	3	0	
8	3	0	2
9	3	0	2
7	3	1	2
8	3	1	2
9	3	1	2
7	3	2	2
8	3	2	2
9	3	2	2
7	3	3	2
10	Retourner $\begin{bmatrix} 2 & 25 & 8 \\ 0 & 1 & 2 \end{bmatrix}$		

<u>ZérosFinals((TAB₃[i])_{0 ≤ i < 2})</u>		
LIGNE	ℓ	j
1-2	2	
3	2	
4	1	
5	1	
6	1	<u>RES[0]</u>
7	3	0
8	3	0
9	3	0
7	3	1
10	Retourner	<u>0</u>

☺

Barème:

- $\frac{1}{4}$ pour les variables (tout ce qui varie doit être présent)
- des points perdus à chaque fois que vous n'exécutez pas votre algorithme.

Exercice 52. Écrire un algorithme prenant en entrée un tableau d'entiers et renvoyant un tableau de même taille représentant le même nombre, mais tel que tous les éléments — sauf éventuellement le dernier — sont entre 0 et 9.

Correction: On parcourt le tableau en commençant par le début, pour chaque case contenant un nombre à deux chiffres, on

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

```

1 ChiffresDansCases((TAB[i])0 ≤ i < N)
2   pour  $i$  allant de 0 à  $N - 1$  faire
3     tant que  $\text{TAB}[i] > 9$  faire
4        $\text{TAB}[i] \leftarrow \text{TAB}[i] - 10$ 
5        $\text{TAB}[i + 1] \leftarrow \text{TAB}[i + 1] + 1$ 
6     fin
7   fin
8   retourner  $(\text{TAB}[i])_{0 \leq i < N}$ 
```

Sorties: un tableau de longueur N

lui soustrait 10, et augmente de 1 la case d'après.

☺

Barème:

- $\frac{1}{6}$ pour les entrées
- $\frac{1}{6}$ pour les sorties
- $\frac{1}{6}$ pour le nom, les noms des entrées, la valeur de retour, cohérents avec les entrées et les sorties
- $\frac{1}{4}$ pour la boucle **pour** correcte
- $\frac{1}{4}$ pour la boucle **tant que** correcte et son contenu

Exercice 53. Exécuter cet algorithme sur le premier et le deuxième tableau.

Correction: En appelant $(\text{TAB}_1[i])_{0 \leq i < 4}$ et $(\text{TAB}_2[i])_{0 \leq i < 5}$ les deux tableaux :

ChiffresDansCases($(\text{TAB}_1[i])_{0 \leq i < 4}$)

LIGNE	<i>i</i>	TAB			
		[0]	[1]	[2]	[3]
1	13	5	8	0	
2	0 13	5	8	0	
3	0 13	5	8	0	
4	0 3	5	8	0	
5	0 3	6	8	0	
6	0 3	6	8	0	
3	0 3	6	8	0	
7	0 3	6	8	0	
2	1 3	6	8	0	
3	1 3	6	8	0	
7	1 3	6	8	0	
2	2 3	6	8	0	
3	2 3	6	8	0	
7	2 3	6	8	0	
2	3 3	6	8	0	
8	Retourner	3	6	8	0
		0	1	2	3

ChiffresDansCases((TAB ₂ [<i>i</i>]) _{0 ≤ i < 5})						
LIGNE	<i>i</i>	TAB				
		[0]	[1]	[2]	[3]	[4]
1	2	25	8	0	0	
2	0 2	25	8	0	0	
3	0 2	25	8	0	0	
6	0 2	25	8	0	0	
2	1 2	25	8	0	0	
3	1 2	25	8	0	0	
4	1 2	15	8	0	0	
5	1 2	15	9	0	0	
6	1 2	15	9	0	0	
3	1 2	15	9	0	0	
4	1 2	5	9	0	0	
5	1 2	5	10	0	0	
6	1 2	5	10	0	0	
3	1 2	5	10	0	0	
7	1 2	5	10	0	0	
2	2 2	5	10	0	0	
3	2 2	5	10	0	0	
4	2 2	5	0	0	0	
5	2 2	5	0	1	0	
6	2 2	5	0	1	0	
3	2 2	5	0	1	0	
7	2 2	5	0	1	0	
2	3 2	5	0	1	0	
3	3 2	5	0	1	0	
7	3 2	5	0	1	0	
2	4 2	5	0	1	0	
8	Retourner	2	5	0	1	0
		0	1	2	3	4

⊕

Barème :— $\frac{1}{4}$ pour les variables

Exercice 54. Écrire un algorithme prenant en entrée un tableau d'entiers et renvoyant un tableau d'entier (potentiellement de longueur différente) représentant le même nombre, dont toutes les premières cases (sauf la dernière) sont identiques, et dont les dernières sont entre 0 et 9.

Correction: On a juste à observer la dernière case. On ne sait pas, a priori, de combien de cases nous allons avoir besoin pour mettre le tableau sous forme standard. Ainsi, nous allons placer les chiffres que nous allons calculer dans une liste. ☺

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

```

1 DernièreCase((\text{TAB}[i])_{0 \leq i < N})
2   L ← Λ
3   u ← TAB[N - 1]
4   d ← 0
5   modification ← v
6   longueur ← -1
7   tant que modification ≡ v faire
8     modification ← f
9     tant que u > 9 faire
10    u ← u - 10
11    d ← d + 1
12    modification ← v
13  fin
14  L ← u :: L
15  u ← d
16  d ← 0
17  longueur ← longueur + 1
18 fin
19 nouveau (RES[i])_{0 \leq i < N + longueur}
20 pour i allant de 0 à N - 1 faire
21   | RES[i] ← TAB[i]
22 fin
23 pour i allant de 0 à longueur + 1 faire
24   | e :: L ← L
25   | RES[N + longueur - 1 - i] ← e
26 fin
27 retourner (RES[i])_{0 \leq i < N + longueur}
```

Sorties: un tableau de longueur plus que N ($N + \text{longueur}$)

Barème:

- $\frac{1}{6}$ pour les entrées
- $\frac{1}{6}$ pour les sorties
- $\frac{1}{6}$ pour le nom, les noms des entrées, la valeur de retour, cohérents avec les entrées et les sorties
- $\frac{1}{6}$ pour le calcul de la longueur du nouveau tableau
- $\frac{1}{4}$ pour son contenu

Exercice 55. Exécuter cet algorithme sur le quatrième tableau.

Correction: La trace d'exécution va être un peu longue... De ce fait, on ne va représenter la boucle allant des lignes 9 à 13 que la première fois, puis l'abrévier en une seule étape. De plus, on va découper la trace en deux parties : avant l'exécution de

DernièreCase(((\text{TAB}[i])_{0 \leq i < N})

LIGNE	L	u	d	modification	longueur
1-2	Λ				
3	Λ	328			
4	Λ	328	0		
5	Λ	328	0	v	
6	Λ	328	0	v	-1
7	Λ	328	0	v	-1
8	Λ	328	0	F	-1
9	Λ	328	0	F	-1
10	Λ	318	0	F	-1
11	Λ	318	1	F	-1
12	Λ	318	1	v	-1
13	Λ	318	1	v	-1
9-10-11-12-13	Λ	308	2	v	-1
9-10-11-12-13	Λ	298	3	v	-1
9-10-11-12-13	Λ	288	4	v	-1
9-10-11-12-13	Λ	278	5	v	-1
9-10-11-12-13	Λ	268	6	v	-1
9-10-11-12-13	Λ	258	7	v	-1
9-10-11-12-13	Λ	248	8	v	-1
9-10-11-12-13	Λ	238	9	v	-1
9-10-11-12-13	Λ	228	10	v	-1
9-10-11-12-13	Λ	218	11	v	-1
9-10-11-12-13	Λ	208	12	v	-1
9-10-11-12-13	Λ	198	13	v	-1
9-10-11-12-13	Λ	188	14	v	-1
9-10-11-12-13	Λ	178	15	v	-1
9-10-11-12-13	Λ	168	16	v	-1
9-10-11-12-13	Λ	158	17	v	-1
9-10-11-12-13	Λ	148	18	v	-1
9-10-11-12-13	Λ	138	19	v	-1
9-10-11-12-13	Λ	128	20	v	-1
9-10-11-12-13	Λ	118	21	v	-1
9-10-11-12-13	Λ	108	22	v	-1
9-10-11-12-13	Λ	98	23	v	-1
9-10-11-12-13	Λ	88	24	v	-1
9-10-11-12-13	Λ	78	25	v	-1
9-10-11-12-13	Λ	68	26	v	-1
9-10-11-12-13	Λ	58	27	v	-1
9-10-11-12-13	Λ	48	28	v	-1
9-10-11-12-13	Λ	38	29	v	-1
9-10-11-12-13	Λ	28	30	v	-1
9-10-11-12-13	Λ	18	31	v	-1
9-10-11-12-13	Λ	8	32	v	-1
9	Λ	8	32	v	-1
14	[8]	8	32	v	-1
15	[8]	32	32	v	-1
16	[8]	32	0	v	-1
17	[8]	32	0	v	0
18	[8]	32	0	v	0
7	[8]	32	0	v	0
8	[8]	32	0	F	0
9-10-11-12-13	[8]	2	3	v	0
9	[8]	2	3	v	0
14	[2, 8]	2	3	v	0
15	[2, 8]	3	3	v	0
16	[2, 8]	3	0	v	1
17	[2, 8]	3	0	v	1
18	[2, 8]	3	0	v	1
7	[2, 8]	3	0	v	1
8	[2, 8]	3	0	F	1
9	[2, 8]	3	0	F	1
14	[3, 2, 8]	3	0	F	1
15	[3, 2, 8]	0	0	F	1
16	[3, 2, 8]	0	0	F	1
17	[3, 2, 8]	0	0	F	2
18	[3, 2, 8]	0	0	F	2
7	[3, 2, 8]	0	0	F	2

LIGNE	L	u	d	modification	longueur	i	RES					e										
							[0]	[1]	[2]	[3]	[4]											
19	[3, 2, 8]	0	0	F	2																	
20	[3, 2, 8]	0	0	F	2	0																
21	[3, 2, 8]	0	0	F	2	0	2															
22	[3, 2, 8]	0	0	F	2	0	2															
20	[3, 2, 8]	0	0	F	2	1	2															
21	[3, 2, 8]	0	0	F	2	1	2	5														
22	[3, 2, 8]	0	0	F	2	1	2	5														
20	[3, 2, 8]	0	0	F	2	2	2	5														
23	[3, 2, 8]	0	0	F	2	0	2	5														
24	[2, 8]	0	0	F	2	0	2	5			3											
25	[2, 8]	0	0	F	2	0	2	5		3	3											
26	[2, 8]	0	0	F	2	0	2	5		3	3											
23	[2, 8]	0	0	F	2	1	2	5		3	3											
24	[8]	0	0	F	2	1	2	5		3	2											
25	[8]	0	0	F	2	1	2	5	2	3	2											
26	[8]	0	0	F	2	1	2	5	2	3	2											
23	[8]	0	0	F	2	2	2	5	2	3	2											
24	□	0	0	F	8	2	2	5	2	3	2											
25	□	0	0	F	8	2	2	5	8	2	3	2										
26	□	0	0	F	8	2	2	5	8	2	3	2										
23	□	0	0	F	8	3	2	5	8	2	3	2										
27	Retourner				<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>8</td><td>2</td><td>3</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	2	5	8	2	3	0	1	2	3	4							
2	5	8	2	3																		
0	1	2	3	4																		

⊕

Exercice 56. En déduire un algorithme mettant un tableau sous forme standard.

Correction: On a juste à chaîner les trois algorithmes précédents (dans le bon sens!):

⊕

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N.

1 **FormeStandard** $((\text{TAB}[i])_{0 \leq i < N})$

2 | **retourner DernièreCase(ChiffresDansCases(ZérosFinal $((\text{TAB}[i])_{0 \leq i < N}))$)**

Sorties: un tableau de longueur inconnue

Petite multiplication

Pour effectuer la multiplication, on a du apprendre les *tables de multiplications*, c'est-à-dire les résultats des multiplications de tous les chiffres. On va devoir, dans notre algorithme, écrire d'une manière ou d'une autre ces tables. On peut les écrire explicitement, ou alors les recalculer, par exemple en utilisant une addition itérée, à chaque fois qu'on en a besoin.

Une autre solution peut être de les calculer une fois pour toute, et s'en servir à chaque fois qu'on en a besoin.

On va choisir cette troisième option.

Exercice 57. Écrire un algorithme ne prenant rien en entrée, et renvoyant un tableau de dimension 10×10 contenant les tables de multiplication calculées par sommes itérées.

Correction: C'est très proche de ce qu'on a fait en cours... On pourrait le faire avec trois boucles **pour** imbriquées, mais on va plutôt le faire avec seulement deux et stocker.

⊕

Barème:

- $\frac{1}{3}$ pour les entrées et les sorties
- $\frac{2}{3}$ pour le reste
- bonus de $\frac{1}{3}$ s'il n'y a que deux boucles imbriquées

Entrées:

```

1 TableMultiplications()
2   nouveau (MULT[i][j])0 ≤ i < 100 ≤ j < 10
3   pour i allant de 0 à 10 faire
4     a ← 0
5     pour j allant de 0 à 10 faire
6       a ← a + i
7       MULT[i][j] ← a
8     fin
9   fin
10  retourner (MULT[i][j])0 ≤ i < 100 ≤ j < 10

```

Sorties: un tableau de dimension 10×10 **Grande multiplication**

Exercice 58. Écrire un algorithme, qui étant donné un tableau de dimension 10×10 et deux tableaux représentant des entiers positifs sous forme standard, renvoie un tableau de nombres représentant leur produit, en appliquant la technique de la multiplication posée.

Correction:**Entrées:**

- un tableau $(\text{MULT}[i][j])_{0 \leq i < 10} \text{ de dimension } 10 \times 10;$
- deux tableaux $(A[i])_{0 \leq i < N}$ et $(B[i])_{0 \leq i < M}$ sous forme standard.

```
1 Multiplication((MULT[i][j])0 ≤ i < 10, (A[i])0 ≤ i < N, (B[i])0 ≤ i < M)
```

```

2   nouveau (RES[i])0 ≤ i < N+M-1
3   pour i allant de 0 à N + M - 1 faire
4     | RES[i] ← 0
5   fin
6   pour i allant de 0 à N faire
7     | pour j allant de 0 à M faire
8     |   | RES[i + j] ← RES[i + j] + MULT[A[i]][B[j]]
9     |   fin
10    fin
11  retourner (RES[i])0 ≤ i < N+M

```

Sorties: un tableau de dimension $N + M$

Exercice 59. L'exécuter sur le résultat de l'exercice d'avant, 63 et 37.

Correction : Le premier tableau d'entrée $(\text{MULT}[i][j])_{0 \leq i < 10}$ contient la table de multiplication, quant aux deux autres, il sont définis par :

$$\begin{aligned}
 A[0] &:= 3 \\
 A[1] &:= 6 \\
 B[0] &:= 7 \\
 B[1] &:= 3
 \end{aligned}$$

Multiplication((MULT[i][j])_{0 ≤ i < 10, 0 ≤ j < 10}, (A[i])_{0 ≤ i < 2}, (B[i])_{0 ≤ i < 2})

LIGNE	i	j	RES	[0]	[1]	[2]
1-2						
3	0					
3	0					
4	0		0			
5	0		0			
3	1		0			
4	1		0	0		
5	1		0	0		
3	2		0	0		
4	2		0	0	0	
5	2		0	0	0	
3	3		0	0	0	
4	3		0	0	0	
5	3		0	0	0	
3	4		0	0	0	
6	0		0	0	0	0
7	0	0	0	0	0	0
8	0	0	21	0	0	0
9	0	0	21	0	0	0
7	0	1	21	0	0	0
8	0	1	21	9	0	0
9	0	1	21	9	0	0
7	0	2	21	9	0	0
10	0	2	21	9	0	0
6	1		21	9	0	0
7	1	0	21	9	0	0
8	1	0	21	51	0	0
9	1	0	21	51	0	0
7	1	1	21	51	0	0
8	1	1	21	51	18	0
9	1	1	21	51	18	0
7	1	2	21	51	18	0
10	1	2	21	51	18	0
6	2		21	51	18	0
11	Retourner		21	51	18	0
			0	1	2	3

⊕

Exercice 60. En déduire un algorithme, prenant en entrée deux tableaux représentant des entiers positifs sous forme standard, qui renvoie leur produit sous forme de tableau sous forme standard.

Correction:

⊕

Entrées: deux tableaux $(A[i])_{0 \leq i < N}$ et $(B[i])_{0 \leq i < M}$ sous forme standard.

1 **MultiplicationStandard**((A[i])_{0 ≤ i < N}, (B[i])_{0 ≤ i < M})

2 **retourner**

 FormeStandard(Multiplication(TableMultiplications(), (A[i])_{0 ≤ i < N}, (B[i])_{0 ≤ i < M}))

Sorties: un tableau de dimension $N + M$

Exercice 61. Justifier que cet algorithme est meilleur que celui par addition itérée.

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.

3.1	Le problème du tri	66
	Relation d'ordre	66
	Enregistrements, clefs et tris	68
3.2	Algorithmes de tri: comparaisons et implémentations	68
3.3	Une application du tri: la recherche dichotomique	71
3.4	Énumération	74
	L'algorithme	74
	Complexité	79
	Les invariants de boucle	79
	Correction	81
	Résumé	89
3.5	Insertion	89
	Correction	91
	Complexité	92
	Résumé	93
3.6	Fusion	93
	Correction	99
	Complexité	99
	Résumé	100
3.7	Borne inférieure de complexité des tris par comparaison	100
3.8	Résumé	102
3.A	Améliorer le tri fusion	103
	Fusion de listes	103
	Rajouter le prochain élément	104
	Fusion équilibrée	106
	Partir de briques plus longues	108
3.B	Chanson sur mon drôle de tri	109
	La moitié plus un	111
	Les deux tiers	112

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.

3.1 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 néerlandais, 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 (œur, 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 8 (papier, caillou, ciseau). Au jeu de papier-caillou-ciseaux, on a trois éléments : , , ; tels que :

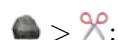
— les ciseaux battent le papier :



— 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 lui¹ — comme exercice, regardez quelles propriétés sont vérifiées par les exemples ci-dessus :

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

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

$$x \leq y$$

ou

$$y \leq x.$$

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 algorithme qui, étant donnés deux éléments x et y , détermine si $x \leq y$, $y \leq x$ ou aucun des deux.

complexité constante il peut être raisonnable de demander qu'il existe un algorithme en temps constant, qui, étant donnés deux éléments x et y , détermine si $x \leq y$ — 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 :

$$x \leq x.$$

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 5. Soit E un ensemble. Un *ordre* \leq sur E est une relation vérifiant les trois axiomes :

transitivité $\forall x \in E, \forall y \in E, \forall z \in E, (x \leq y \wedge y \leq z) \implies x \leq z$;

reflexivité $\forall x \in E, x \leq x$;

antisymétrie $\forall x \in E, \forall y \in E, (x \leq y \wedge y \leq x) \implies x = y$.

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

totalité $\forall x \in E, \forall y \in E, x \leq y \vee y \leq x$.

On dit qu'un ordre est *borné* si

bornitude $\exists -\infty \in E, \exists +\infty \in E, \forall x \in E, -\infty \leq x \leq +\infty$.

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

Exemple 9. L'ordre croissant sur les nombres entiers est total, calculable, mais n'est pas borné.

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 10. 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 6. 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 62. 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 $\spadesuit < \heartsuit < \diamondsuit < \clubsuit$. 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 algorithmes réalisant des fonctions de tri — ce qui va d'abord nécessiter de choisir sur quelles structures de données représenter les listes d'enregistrements. De manière intéressante, les algorithmes ne dépendent pas des enregistrements, ni de l'ordre, mais seulement du fait que l'on peut ordonner.

3.2 ALGORITHMES DE TRI : COMPARAISONS ET IMPLÉMENTATIONS

Il y a énormément d'algorithmes de tri. On en verra peu. On se demandera systématiquement leur complexité, leur correction, et leurs avantages. Pour mesurer leur complexité, on comptera le nombre d'opérations que les algorithmes de tri font. En règle générale, ils font deux types d'opération :

- les comparaisons entre différents éléments;
- les insertions, plaçant un élément dans la structure ordonnée.

Nous compterons chaque type d'opérations séparément : certains algorithmes réaliseront peu de comparaison mais inversement beaucoup d'insertions.

Par ailleurs, on va rapidement remarquer qu'une autre source de différence entre ces algorithmes est que certains vont avoir un comportement très variable selon les tableaux en entrée : par exemple, certains font moins d'opérations si la liste est déjà triée, d'autres non. Pour cela on doit distinguer plusieurs complexités :

la complexité dans le pire cas où l'on compte systématiquement le nombre d'opérations en supposant que tout se passe mal, que les boucles **tant que** ne s'arrête jamais avant la fin, que l'élément que l'on cherche est toujours le dernier ;

la complexité dans le meilleur des cas où l'on compte systématiquement que tout se passe bien ;

la complexité en moyenne sans doute plus pertinente, on calcule le nombre d'opération en moyenne, sur toutes les entrées possibles.

Elles ont toutes les trois leur intérêts, celles en pire et en meilleur cas car elles donnent des bornes, et la complexité en moyenne car, justement, elle est en moyenne. La complexité en moyenne est souvent difficile à calculer — à la fois pour des raisons pratiques (il faut des connaissances en probabilités), mais aussi théoriques (il faut se demander quelles sont vraiment les entrées possibles) — aussi, on se contente souvent de la complexité en pire cas.

En général, quand on parle de la complexité d'un algorithme sans autre précision, on parle de sa complexité asymptotique en pire cas, c'est-à-dire la manière dont évolue le nombre d'opérations, dans le pire cas, quand les entrées grandissent², et ce qu'on appelle **théorie de la complexité** ne traite que ce cas-là.

Dans ce chapitre, nous allons programmer chaque algorithme que nous verrons en C. Pour cela, en Appendix B, nous allons voir des compléments sur le langage. Comme dans un tri, on déplace souvent des éléments, mais on ne modifie pas la longueur de ce que l'on trie, on va plus souvent utiliser des tableaux. Dans ce cas, à chaque fois, on va écrire une fonction

```
1 int tri(size_t longueur, int avant[longueur], int apres[longueur]);  
triant le tableau avant en apres.
```

Cela va nous permettre d'adoindre, à l'étude théorique, des résultats pratiques, c'est-à-dire qu'on va constater en vrai, que les algorithmes plus efficaces théoriquement sont plus rapides dans la pratique, et même que les différences algorithmiques écrasent complètement les différences de performance de chaque machine. On va donc prendre différentes tailles de tableau et faire tourner cent fois chaque algorithme sur des tableaux d'une taille donnée remplis aléatoirement et mesurer les temps d'exécution : ainsi, nous pourrons évaluer empiriquement le temps d'exécution dans le pire cas, dans le meilleur cas et en moyenne. Vous pouvez prendre le code C qui suit tel quel, néanmoins des explications sont données. En Appendix B.2, nous voyons comment remplir aléatoirement un tableau et en Appendix B.3, nous voyons comment mesurer le temps d'exécution des programmes.

Ce qu'on va donc faire est donc, pour plusieurs tailles de tableau :

- trier cent tableaux de cette taille remplis aléatoirement ;
- mesurer le temps nécessaire au tri ;
- calculer le temps minimal, maximal et moyen pour chaque taille de tableau.

Ainsi, on pourra confirmer empiriquement les considérations de complexité théorique.

Le fragment de C suivant calcule cela pour une fonction de tri quelconque. On l'a exécuté dans le cas d'une fonction fausse et triviale, qui recopie à l'identique le tableau. On voit que le temps pris, représenté en Figure 3.1) croît **linéairement** comme on pouvait s'y attendre.

```
1 int tableau_aleatoire(const size_t longueur, int tab[longueur]) {  
2     for (int i = 0; i < longueur; i++) {  
3         tab[i] = rand()%1000;  
4     }  
5     return 0;  
6 }  
7  
8 int main(){  
9     srand(time(NULL));  
10    int* avant;  
11    int* apres;  
12    int longueur;  
13    double temps;
```

2. C'est le choix que nous fîmes dans le chapitre précédent.

```

14     clock_t t;
15     double min;
16     double max;
17     double total;
18
19     printf("cases,moyenne,min,max\n");
20
21     for(int i = 0; i < 15; i++){
22         longueur = (int) pow(2,i);
23         avant = (int*) calloc(longueur,sizeof(int));
24         apres = (int*) calloc(longueur,sizeof(int));
25         total = 0.0;
26         min = DBL_MAX;
27         max = DBL_MIN;
28
29         for(int j = 0; j < 100; j++){
30             tableau_aleatoire((size_t)longueur,avant);
31             t = clock();
32             tri(longueur,avant,apres);
33             t = clock() - t;
34             temps = ((double)t)/CLOCKS_PER_SEC;
35
36             if (temps < min)
37                 min = temps;
38             if (temps > max)
39                 max = temps;
40             total += temps;
41         }
42         t = clock();
43         tri(longueur,apres,apres);
44         t = clock() - t;
45         temps = ((double)t)/CLOCKS_PER_SEC;
46
47         if (temps < min)
48             min = temps;
49         if (temps > max)
50             max = temps;
51         total += temps;
52
53         printf("%d,",longueur);
54         printf("%f,",total/101);
55         printf("%f,",min);
56         printf("%f\n",max);
57         free(avant);
58         free(apres);
59     }
60 }
61

```

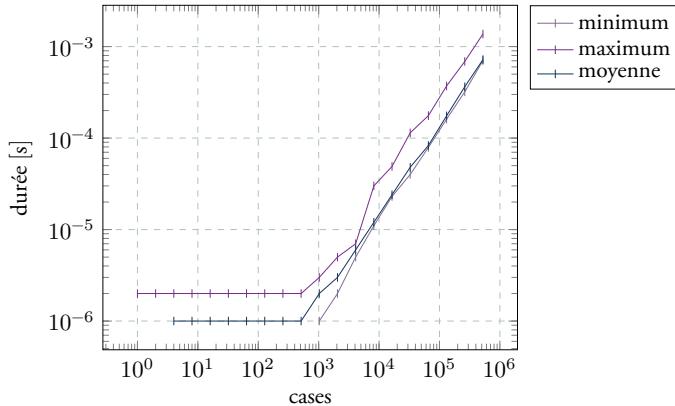


FIGURE 3.1 – Temps d'exécution de la fonction identité sur des tableaux de taille variable

3.3 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 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 $O(n)$.

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 l'Algorithme 21.

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

Entrées:

- un tableau d'enregistrements $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N , trié;
- une clef κ ;
- deux indices $0 \leq i, j < N$.

```

1 RechercheDichotomique((TAB[i])0 ≤ i < N, κ, i, j)
2   si  $i \equiv j$  alors
3     si clef(TAB[i]) ≡ κ alors
4       retourner i
5     sinon
6       retourner une erreur
7   fin
8   sinon
9     si clef( $\frac{i+j}{2}$ ) ≡ κ alors
10    retourner  $\frac{i+j}{2}$ 
11   sinon
12     si clef( $\frac{i+j}{2}$ ) < κ alors
13       retourner RechercheDichotomique((TAB[i])0 ≤ i < N, κ, i,  $\frac{i+j}{2}$ )
14     sinon
15       retourner RechercheDichotomique((TAB[i])0 ≤ i < N, κ,  $\frac{i+j}{2}$ , j)
16   fin
17 fin
18 fin

```

Sorties: un indice ou une erreur

Algorithme 21: Recherche dichotomique

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 $O(\log(n))$ ³.

Nous pouvons retirer deux enseignements de cet algorithme :

- 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*⁴: 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

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

4. Par analogie avec la vieille stratégie politique.

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 63. 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).

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 élément e et un entier p , supérieur à la moitié de la longueur du tableau ($p > N/2$) tels que e apparaît au plus p fois dans le tableau et tout autre élément que e apparaît au plus $N - p$ 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 couleur⁵ que la dernière balle sur l'étagère, on la pose sur l'étagère après la

5. C'est en particulier le cas s'il n'y a pas de balle sur l'étagère.

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 ⁶, 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, 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).

3.4 ÉNUMÉRATION

Le *tri par énumération* est un algorithme de tri particulièrement simple qui peut être résumé ainsi : pour chaque enregistrement, on compare sa clef avec toutes les autres clefs, et on compte le nombre de clefs à qui cette clef est supérieure. Soit i ce nombre. On place l'enregistrement en position i . On va voir que ce tri est peu efficace. Il va avant tout nous servir pour présenter les notations et concepts qui nous serviront par la suite.

L'algorithme

La première question à se poser est de savoir comment représenter les données. On va devoir comparer chaque clef avec toutes les autres clefs, dans n'importe quel ordre. Une *liste chaînée* ne serait pas efficace (et aucune de ses spécialisations, comme une *pile* ou une file non plus), un tableau si. Enfin, on va traiter chaque enregistrement l'un après l'autre, et ensuite le placer correctement, n'importe où. Il semble plus sage d'utiliser un deuxième tableau pour ça : en fait, on voit mal comment faire cela en place. Donc, on va prendre en entrée un *tableau d'enregistrements*, en sortie, un autre tableau d'enregistrements.

Enfin, on suppose qu'on a une fonction *clef* qui associe à chaque *enregistrement* sa *clef*.

On veut donc une boucle qui parcourt les éléments du tableaux, et à l'intérieur une deuxième, qui, pour chaque élément, le compare avec tous les autres. Écrit sous forme de pseudo-code, on obtient un premier jet :

Il y a néanmoins deux problèmes avec cette tentative : le premier est un problème de complexité, le second un problème de correction.

- On voit qu'on compare chaque paire d'éléments deux fois. Une fois quand c'est au tour d'un des deux d'être comparé à tout le reste, une deuxième fois quand c'est au tour du deuxième.
- Plus grave, l'algorithme qu'on vient d'écrire traite mal le cas où deux enregistrements ont la même clef ! En effet, s'ils ont la même clef, on cherchera à les placer au même endroit dans le tableau...

6. Cela va être le cas en particulier pour la première balle qu'on regarde.

Entrées: un tableau d'enregistrements $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

```

1 TriÉnumération( $(\text{TAB}[i])_{0 \leq i < N}$ )
2   nouveau  $(\text{RES}[i])_{0 \leq i < N}$ 
3   nouveau  $c$ 
4   pour  $i$  allant de 0 à  $N$  faire
5      $c \leftarrow 0$ 
6     pour  $j$  allant de 0 à  $N$  faire
7       si  $i \neq j$  alors
8         si clef( $\text{TAB}[i]$ ) > clef( $\text{TAB}[j]$ ) alors
9           |    $c \leftarrow c + 1$ 
10          fin
11        fin
12      fin
13       $\text{RES}[c] \leftarrow \text{TAB}[i]$ 
14    fin
15  retourner  $(\text{RES}[i])_{0 \leq i < N}$ 
```

Sorties: un tableau d'enregistrements de longueur N

On va commencer par régler le premier point, ce qui rendra plus facile de régler le second : en effet, c'est la même caractéristique de ce qu'on écrit qui provoque les deux problèmes.

Pour l'instant, on calcule séparément le nombre d'enregistrements auquel chaque enregistrement est supérieur, et on ne garde pas trace du fait que si sa clef n'est pas supérieure, alors elle est inférieure ou égale à celle de l'enregistrement auquel on l'a comparé. Il suffit de le noter à ce moment pour n'avoir à le faire qu'une fois. Ainsi, on va introduire un nouveau tableau COMPTE⁷ qui comptera, pour chaque indice, le nombre de clefs à laquelle la clef correspondante est supérieure. On pourra, en comparant deux clef, incrémenter la bonne.

Cela nous donne l'Algorithme 22. On suppose tous les tableaux numériques initialisés à 0.

On voit qu'on a aussi réglé le second problème : en effet, si les deux clefs sont égales, on incrémentera tout de même le compteur d'une des deux.

Exercice 64. Programmer l'algorithme 22 en C. Rappel : on attend une fonction de prototype

```
1 int tri(const size_t longueur, const int avant[longueur], int apres[longueur]);
```

Correction :

```

1 int tri(const size_t longueur, const int avant[longueur], int apres[longueur]){
2   size_t compte[longueur];
3   for(size_t i = 0; i < longueur; i++){
4     compte[i] = 0;
5   }
6   for(size_t i = 0; i < longueur; i++){
7     for(size_t j = i+1; j < longueur; j++){
8       if (avant[i] > avant[j]) {
9         compte[i]++;
10      } else {
11        compte[j]++;
12      }
13    }
14  }
15  for(size_t i = 0; i < longueur; i){
```

7. “STOP THE COUNT!”

Entrées: un tableau d'enregistrements $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N .

```

1 TriÉnumération((\text{TAB}[i])_{0 \leq i < N})
2   nouveau (\text{RES}[i])_{0 \leq i < N}
3   nouveau (\text{COMPTE}[i])_{0 \leq i < N}
4   pour i allant de 0 à N faire
5     pour j allant de i + 1 à N faire
6       si clef(\text{TAB}[i]) > clef(\text{TAB}[j]) alors
7         |   \text{COMPTE}[i] \leftarrow \text{COMPTE}[i] + 1
8       sinon
9         |   \text{COMPTE}[j] \leftarrow \text{COMPTE}[j] + 1
10      fin
11    fin
12  fin
13  pour i allant de 0 à N faire
14    |   \text{RES}[\text{COMPTE}[i]] \leftarrow \text{TAB}[i]
15  fin
16  retourner (\text{RES}[i])_{0 \leq i < N}
```

Sorties: un tableau d'enregistrements de longueur N

Algorithme 22: Tri par énumération

```

16     apres[compte[i]] = avant[i];
17 }
18 return 0;
19 }
```



Exercice 65. Exécuter l'algorithme sur le tableau

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

Correction: Les différentes variables sont i , j , et toutes les cases des deux tableaux RES et COMPTE. N vaut 4. La trace d'exécution est Figure 3.2.



Une manière moins formelle de se représenter l'exécution de cet algorithme est de regarder la danse des deux variables de boucle i et j , et les comparaisons faites en conséquent. On va donc représenter Figure 3.3 les deux tableaux TAB et COMPTE, les deux variables i et j , ainsi que la comparaison en train d'être faite sur l'exemple de l'Exercice 65. On n'a pas représenté la dernière étape où la variable i atteind la dernière case.

Exercice 66. Repérer, dans la trace d'exécution Figure 3.2, les lignes correspondant aux différentes sous-figures de la Figure 3.3.

On voit que, si on regarde avec qui un élément donné est comparé quand vient son tour, il est seulement comparé avec ceux à sa droite: il est comparé avec ceux à sa gauche à leur tour. Ainsi, on fait toutes les comparaisons exactement une fois, et à chaque comparaison, on incrémente exactement un compteur. C'est ainsi qu'on voit qu'on a réglé les deux problèmes de la première tentative que nous fîmes.

LIGNE	SUIVANTE	<i>i</i>	<i>j</i>	COMPTE				RES			
				[0]	[1]	[2]	[3]	[0]	[1]	[2]	[3]
1-2-3	4			0	0	0	0				
4	5	0		0	0	0	0				
5	6	0	1	0	0	0	0				
6	7	0	1	0	0	0	0				
7	11	0	1	1	0	0	0				
11-5	6	0	2	1	0	0	0				
6	7	0	2	1	0	0	0				
7	11	0	2	2	0	0	0				
11-5	6	0	3	2	0	0	0				
6	9	0	3	2	0	0	0				
9	11	0	3	2	0	0	0	1			
11	12	0		2	0	0	0	1			
12-4	5	1		2	0	0	0	1			
5	6	1	2	2	0	0	0	1			
6	9	1	2	2	0	0	0	1			
9	11	1	2	2	0	0	1	1			
11-5	6	1	3	2	0	1	1	1			
6	9	1	3	2	0	1	1	1			
9	11	1	3	2	0	1	1	2			
11	12	1		2	0	1	1	2			
12-4	5	2		2	0	1	1	2			
5	6	2	3	2	0	1	1	2			
6	9	2	3	2	0	1	1	2			
9	11	2	3	2	0	1	1	3			
11	12	2		2	0	1	1	3			
12-4	5	3		2	0	1	1	3			
5	12	3		2	0	1	1	3			
12	13			2	0	1	1	3			
13	14	0		2	0	1	1	3			
14	15	0		2	0	1	1	3	30		
15-13	14	1		2	0	1	1	3	30		
14	15	1		2	0	1	1	3	5	30	
15-13	14	2		2	0	1	1	3	5	30	
14	15	2		2	0	1	1	3	5	9	30
15-13	14	3		2	0	1	1	3	5	9	30
14	15	3		2	0	1	1	3	5	9	30
15	16			2	0	1	1	3	5	9	30
16	Retourner le tableau										

FIGURE 3.2 – La trace d'exécution de l'Exercice 65

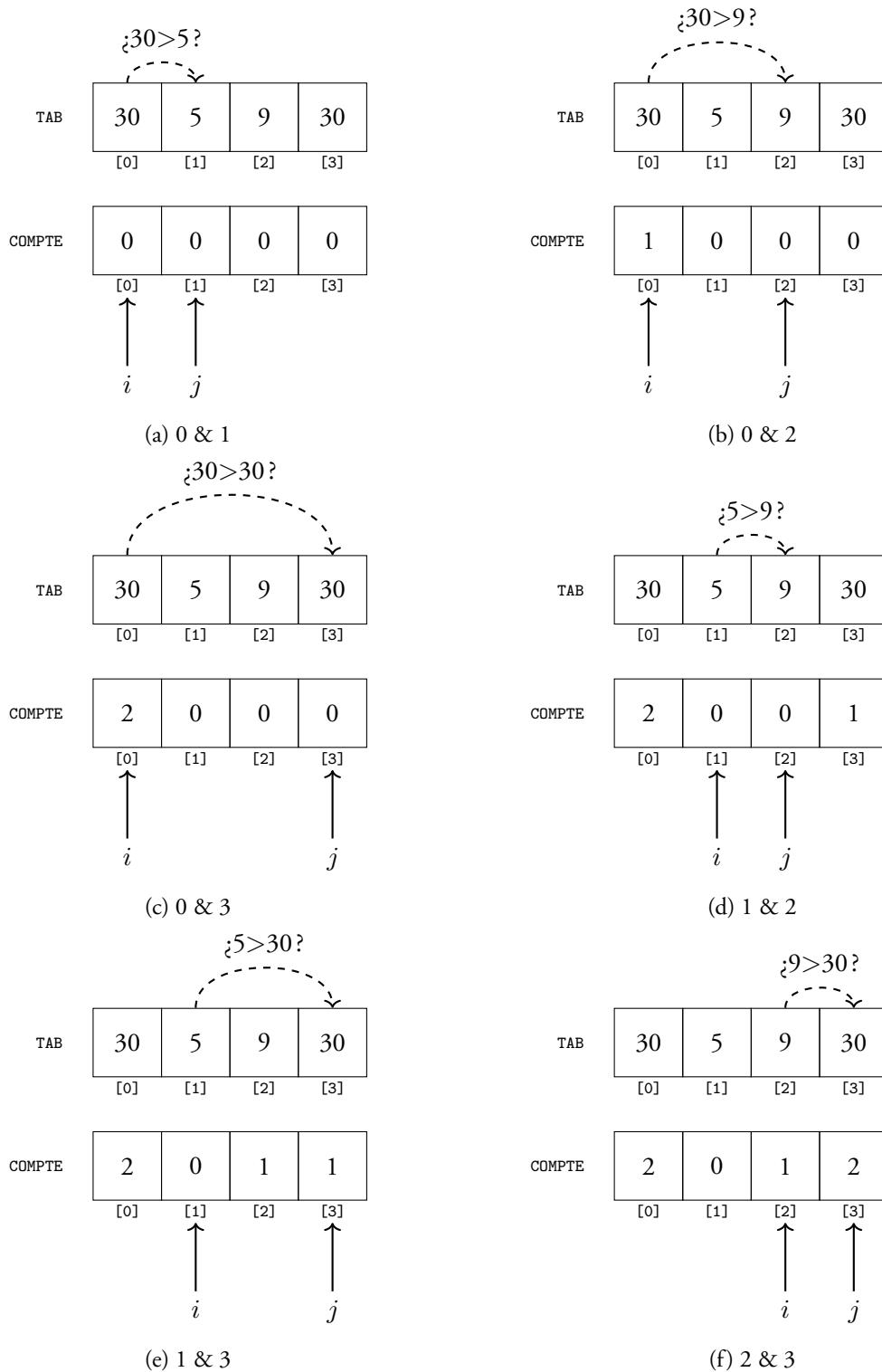


FIGURE 3.3 – La danse des variables de boucle

Complexité

La seule comparaison est faite ligne 5. Elle a lieu pour chaque passage dans la boucle intérieure. Comme on sait combien de fois la boucle extérieure est exécutée, il reste juste à calculer le nombre d'exécution de la boucle intérieure à chaque passage dans la boucle extérieure : au $i^{\text{ème}}$ passage dans la boucle extérieure, on exécute $N - (i + 1)$ fois la boucle intérieure. Ainsi, on exécute la boucle intérieure

$$(N - (0 + 1)) + (N - (1 + 1)) + (N - (2 + 1)) + \cdots + (N - (N - 1 + 1))$$

où, écrit plus symboliquement

$$\sum_{i=0}^{N-1} N - (i + 1)$$

fois. Un changement de variables nous donne $\sum_{i=0}^{N-1} i$, c'est-à-dire $\frac{N(N-1)}{2}$.

Ainsi, il y a $\frac{N(N-1)}{2}$ comparaisons. On peut représenter ce nombre en regardant, pour chaque élément, à qui on va le comparer. Dans la figure ci-dessous, on a représenté d'une couleur l'élément qu'on compare, et dans une autre, les éléments à qui on le compare. En représentant l'avancement de l'exécution de bas en haut, on voit qu'on a autant de comparaisons qu'il y a de cases au-dessus de la diagonale.

30	5	9	30
30	5	9	30
30	5	9	30
30	5	9	30

Il y a autant d'affectations que de comparaisons dans la première boucle, par contre, il y a aussi des affectations dans la seconde : exactement N , ce qui fait, en tout $\frac{N(N+1)}{2}$ affectations.

La complexité est donc quadratique, en $O(N^2)$, aussi bien en terme d'affectations que de comparaisons. On remarque que tout ce qu'on a dit est vrai aussi bien en pire qu'en meilleur cas : la complexité ne dépend pas du fichier que l'on trie.

Numériquement, on s'attend donc à tracer des paraboles, où, en échelle logarithmique, des droites de pente 2, qui se confondent. C'est bien ce qui est confirmé par la Figure 3.4. Accessoirement, on voit que cet algorithme est inutilisable : sur ma machine, récente et haut de gamme, trier vingt mille nombres compris entre 0 et 1000 prend une seconde (à titre de comparaison, j'ai déjà reçu quinze mille mails que j'ai conservé sur l'année 2020).

Exercice 67. Prouver, que pour tout nombre entier n strictement supérieur à 1, la somme des entiers entre 1 et n (exclu) est donné par :

$$\sum_{i=1}^n i = \frac{n(n-1)}{2}$$

Les invariants de boucle

On veut prouver la correction de l'Algorithme 22, c'est-à-dire montrer que l'algorithme réalise bien une [fonction de tri](#). L'algorithme est principalement composé de boucles. On va donc utiliser des invariants de boucle pour prouver cette propriété de correction.

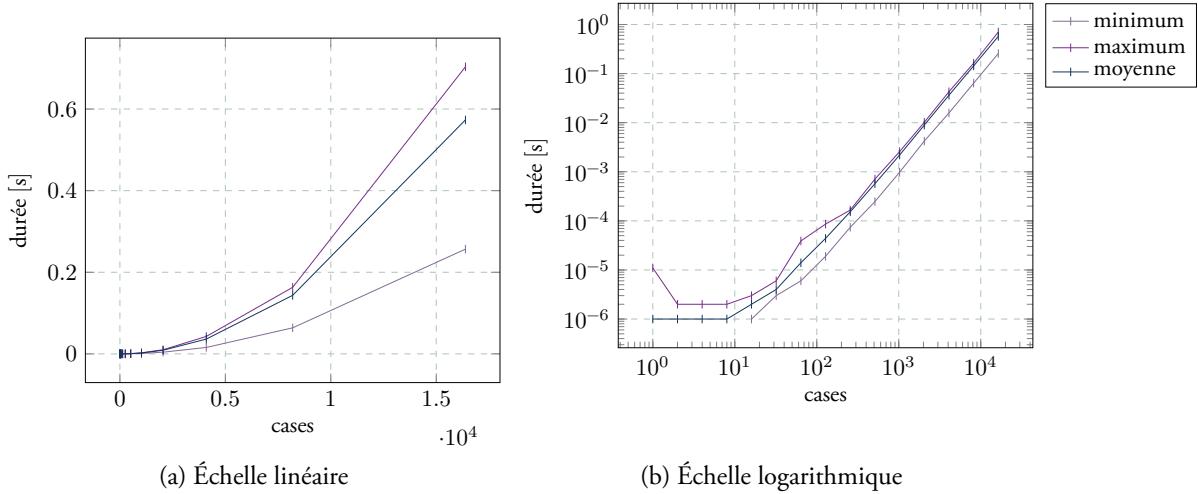


FIGURE 3.4 – Temps d'exécution du tri par énumération sur des tableaux de taille variable

Un *invariant de boucle* est une propriété des variables qui est vraie à chaque passage dans la boucle, ou, dit autrement, dont la vérité est *invariante* par le passage de la boucle. Un bon invariant de boucle a les propriétés suivantes :

1. Il est vrai quand il est instancié avec l'état des variables avant chaque passage dans la boucle ;
2. Il est vrai quand il est instancié avec l'état des variables après tous les passages dans la boucle ;
3. Il permet de déduire une propriété intéressante après tous les passages.

De ce fait, pour prouver un invariant de boucle, il suffit de montrer qu'il est vrai avant d'entrer dans la boucle, et que s'il est vrai, il est encore vrai après une exécution entière du corps de la boucle. Ensuite, on va montrer que l'invariant de boucle implique, une fois sorti de la boucle, la propriété de correction qui nous intéresse.

Exemple 11. Considérons l'Algorithm 23. Il crée un tableau et le remplit avec des zéros. Ainsi, au

Entrées:	un entier N.
1	Remplissage(N)
2	nouveau (TAB[i]) _{0 ≤ i < N}
3	pour i allant de 0 à N faire
4	TAB[i] ← 0
5	fin
6	retourner (TAB[i]) _{0 ≤ i < N}
Sorties:	un tableau de longueur N

Algorithm 23: Tableau — remplissage avec des zéros

début du i -ème passage dans la boucle, chaque case d'indice strictement inférieur à i contient un 0 ; ou dit de manière formulaïque :

$$\forall 0 \leq \ell < i, \text{TAB}[\ell] = 0.$$

Ainsi, un invariant de boucle est une propriété qui est vraie tout au long de l'exécution de la boucle. En général, on a construit la boucle en pensant à une telle propriété, mais pas forcément de manière explicite — de la même manière que faire le commentaire des combinaisons phonologiques d'un poème peut en expliciter le sens (JAKOBSON et LÉVI-STRAUSS 1962).

Trouver des invariants de boucles est une activité délicate. C'est une activité créative à part entière, au même titre que de trouver des nouveaux algorithmes.

Correction

Ici, il y a trois boucles. Il va nous falloir trouver trois invariants de boucles.

- commençons par regarder la troisième boucle, celle de la ligne 13 à 14. L'invariant de boucle semble facile à trouver : les éléments d'indice strictement inférieur à i sont rangés par ordre croissant. Ainsi, avant d'entrer dans la boucle, la propriété est trivialement vérifiée avant d'entrer dans la boucle ; et à la sortie de la boucle, est la propriété qui nous intéresse. En réalité, comme On place dans le tableau $(\text{RES}[i])_{0 \leq i < N}$ les éléments du tableau d'entrée, à l'indice indiqué par le tableau $(\text{COMPTE}[i])_{0 \leq i < N}$.

Pour que cette opération ait un sens, il faut que pour chaque $0 \leq i < N$, $\text{COMPTE}[i]$ contienne un nombre compris entre 0 et N (qui puisse donc être pris comme indice de tableau). De plus, si on place l'enregistrement numéro i en position $\text{COMPTE}[i]$, c'est parce qu'on espère qu'en $\text{COMPTE}[i]$, il y a le nombre d'éléments dont la clef est plus petite que celle de $\text{TAB}[i]$ — ou de même clef mais qu'on a choisi de mettre avant. Introduisons une notation pour décrire cela.

Pour i et j deux indices ($0 \leq i, j < N$), on va noter $i \succ j$ la relation

$$(\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])) \vee ((\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])) \wedge (i > j))$$

En fait, cette relation — qui signifie que la clef de l'enregistrement en i est strictement supérieure à celle de l'enregistrement en j , ou bien qu'elles sont égales mais que i vient après j — correspond exactement aux cas où on incrémenté $\text{COMPTE}[i]$.

Ainsi, pour que cette boucle donne le résultat escompté, il suffit que le tableau $(\text{COMPTE}[i])_{0 \leq i < N}$ contienne exactement en sa case i le nombre d'éléments plus petits que i pour la relation \succ :

$$C : \forall 0 \leq i < N, \text{COMPTE}[i] = \text{Card} \{0 \leq j < N \mid i \succ j\}$$

On note cette propriété (C).

Au début de l'étape i de la boucle, on a déjà placé i éléments : ils sont placés en ordre croissant. C'est notre invariant de boucle, que l'on va noter, pour chaque $0 \leq i < N$, $P_3(i)$

$$\begin{aligned} P_3(i) : \forall 0 \leq k, \ell < i, (\text{COMPTE}[k] > \text{COMPTE}[\ell]) \\ \implies \text{clef}(\text{RES}[\text{COMPTE}[k]]) \geq \text{clef}(\text{RES}[\text{COMPTE}[\ell]])) \end{aligned}$$

Ainsi, on a identifié deux propriétés, une devant être vraie avant d'entrer dans la troisième boucle, une devant être vérifiée à chaque étape d'exécution de la boucle.

- pour la boucle allant de la ligne 4 à 11, il nous faut trouver un invariant de boucle qui implique, une fois la boucle finie d'être déroulée, la propriété (C). Les éléments du tableau d'indice inférieur à i ne sont plus touchés après le i -ème passage dans la boucle : à chaque fois, on ne compare l'élément i qu'avec les éléments d'indice plus grand. On peut se dire que l'invariant de cette boucle serait donc une propriété semblable à (C) mais partielle, uniquement pour les indices inférieurs à i . C'est effectivement un invariant de boucle, mais insuffisant : en effet, il ne dit rien sur le contenu des cases du tableau $\text{COMPTE}[\ell]$ pour ℓ plus grand que i , qui peuvent néanmoins être modifiée.

En effet, à chaque début de i -ème boucle, on peut observer que, dans chaque case $\text{COMPTE}[\ell]$ pour $\ell \geq i$, ces cases contiennent le nombre d'éléments plus petits que $\text{TAB}[\ell]$ d'indice plus petit que i . Autrement dit, notre invariant de boucle, que l'on notera $(P_1(i))$, indiquera que les cases avant i contiennent le nombre d'enregistrement de clef plus petite qu'eux, tandis que les cases avant i contiennent le nombre d'enregistrements plus petits qu'eux d'indice plus petit que i . Autrement dit :

$$\begin{aligned} P_1(i) : \quad \forall 0 \leq \ell < i, \text{COMPTE}[\ell] = \text{Card} \{0 \leq j < N \mid \ell \succ j\} \\ \wedge \forall i \leq \ell < N, \text{COMPTE}[\ell] = \text{Card} \{0 \leq j < i \mid \ell \succ j\} \end{aligned}$$

On voit qu'effectivement, pour $i = N$, $(P_1(N))$ est bien la propriété (C).

- enfin, pour la boucle intérieure, allant de la ligne 5 à 10, l'invariant ($P_1(i)$) n'est que partiellement vraie. En effet, au j -ème passage dans cette boucle, le compte n'a pas été modifié pour les éléments d'indice supérieur à j , tandis que l'élément d'indice i n'a pas encore été comparé avec tout.

$$\begin{aligned} P_2(i, j) : \quad & \forall 0 \leq \ell < i, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ & \wedge \text{COMPTE}[i] = \text{Card} \{0 \leq k < j \mid \ell \succ k\} \\ & \wedge \forall i < \ell < j, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k \leq i \mid \ell \succ k\} \\ & \wedge \forall j \leq \ell < N, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k < i \mid \ell \succ k\} \end{aligned}$$

On a donc identifié les propriétés $P_1(i), P_2(i, j), C, P_3(i)$. Si on reprend l'Algorithme 22, en annotant, en commentaires, les positions où ces différentes propriétés doivent être vérifiées, on obtient :

```

Entrées: un tableau d'enregistrements  $(\text{TAB}[i])_{0 \leq i < N}$  de longueur N.
1 TriÉnumération( $(\text{TAB}[i])_{0 \leq i < N}$ )
2   nouveau ( $\text{RES}[i])_{0 \leq i < N}$ 
3   nouveau ( $\text{COMPTE}[i])_{0 \leq i < N}$ 
4   pour  $i$  allant de 0 à N faire
5     /* ( $P_1(i)$ ) */ *
6     pour  $j$  allant de  $i + 1$  à N faire
7       /* ( $P_2(i, j)$ ) */ *
8       si clef( $\text{TAB}[i]$ ) > clef( $\text{TAB}[j]$ ) alors
9         |  $\text{COMPTE}[i] \leftarrow \text{COMPTE}[i] + 1$ 
10        sinon
11          |  $\text{COMPTE}[j] \leftarrow \text{COMPTE}[j] + 1$ 
12        fin
13      fin
14    /* (C) */ *
15    pour  $i$  allant de 0 à N faire
16      /* ( $P_3(i)$ ) */ *
17       $\text{RES}[\text{COMPTE}[i]] \leftarrow \text{TAB}[i]$ 
18    fin
19    retourner ( $\text{RES}[i])_{0 \leq i < N}$ 
Sorties: un tableau d'enregistrements de longueur N

```

Il ne nous reste plus qu'à montrer que ces propriétés sont vraies. Avant cela, attardons-nous sur la propriété ($P_2(i, j)$), la plus complexe, mais aussi celle qui capture l'essence du fonctionnement de cet algorithme.

En effet, la propriété énonce comment, à chaque étape du calcul, le tableau $(\text{COMPTE}[i])_{0 \leq i < N}$ est rempli. En particulier, ce tableau se fait diviser en quatre zones, selon leur état d'avancement, comme représenter en Figure 3.5. On peut, de manière plus abstraite, voir cela comme des comparaisons qui ont déjà où non été faites, directement sur le tableau TAB.

Exemple 12. On peut en profiter pour relire l'exécution de l'Exercice 65 à la lumière de ces invariants de boucles. La manière la plus simple est de colorier les différentes zones de la mémoire dans la Figure 3.6 selon les couleurs de la Figure 3.5.

L'autre élément important de cette analyse est d'avoir mis en évidence la relation \succ . Elle a des propriétés importantes.

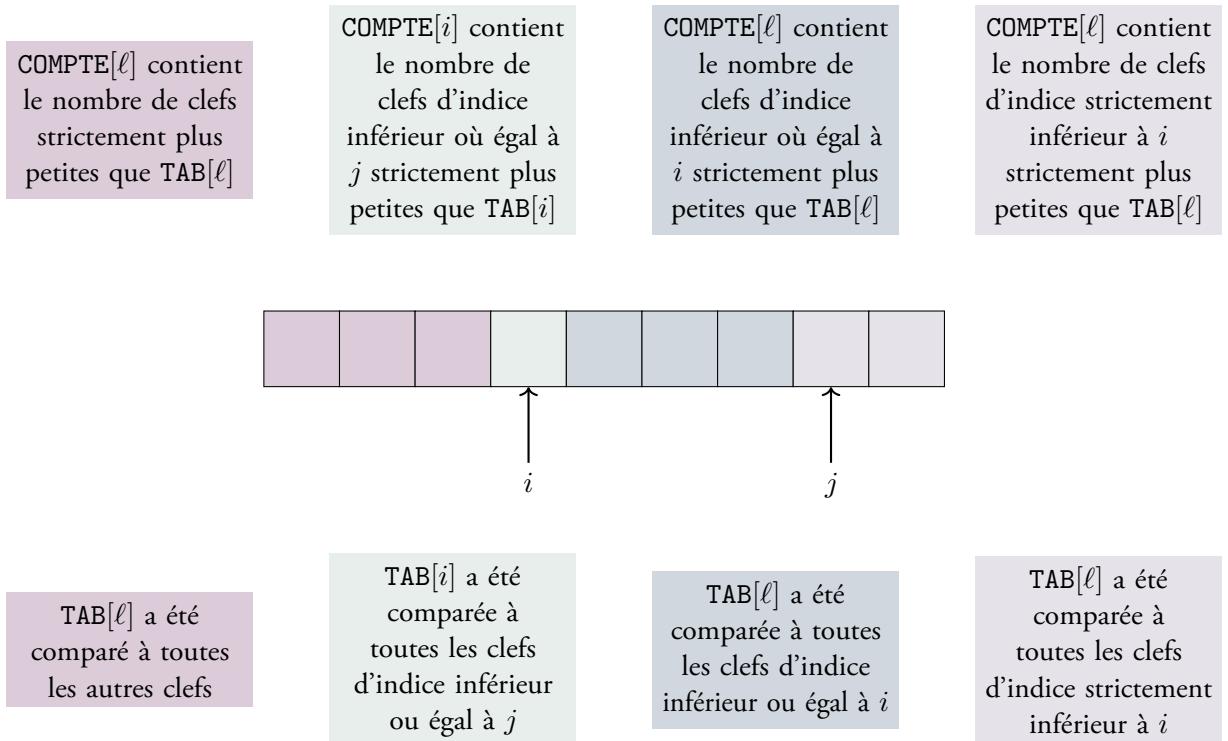


FIGURE 3.5 – Les tableau COMPTE et TAB ont des cases dans différents états d'avancement

Proposition 1. La relation \succeq (qui met en relation deux indices si ils sont égaux ou en relation pour \succ) sur les indices est un ordre total.

Démonstration. Si on l'écrit en entier, la relation \succeq est définie par $i \succeq j$ si et seulement si:

$$(i = j) \vee (\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])) \vee ((\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])) \wedge (i > j))$$

On doit montrer les trois axiomes d'un ordre, ainsi que la totalité:

réflexivité soient i et j deux indices. Par définition, si $i = j$, alors $i \succeq j$.

transitivité soient i , j et k trois indices. Supposons $i \succeq j$ et $j \succeq k$. On veut montrer que $i \succeq k$. Il y a trois possibilités faisant que $i \succeq j$, et de même, trois possibilités faisant que $j \succeq k$.

Si $i = j$, alors, $i = j \succeq k$, et de même, si $j = k$, $i \succeq j = k$. Il nous reste donc quatre cas:

1. Si $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])$ et $\text{clef}(\text{TAB}[j]) > \text{clef}(\text{TAB}[k])$.

On déduit de ces deux inégalités strictes les deux inégalités larges ; c'est-à-dire que $\text{clef}(\text{TAB}[i]) \geq \text{clef}(\text{TAB}[j])$ et $\text{clef}(\text{TAB}[j]) \geq \text{clef}(\text{TAB}[k])$. (cette étape est nécessaire car on a écrit les axiomes d'un ordre large et pas strict!).

Par transitivité de \leq , on en déduit que

$$\text{clef}(\text{TAB}[i]) \geq \text{clef}(\text{TAB}[k]).$$

De plus, si $\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[k])$, alors, par antisymétrie, $\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])$, or c'est exclu par hypothèse.

Donc $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[k])$, et donc, par définition, $i \succeq k$.

2. Si $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])$ et $(\text{clef}(\text{TAB}[j]) = \text{clef}(\text{TAB}[k])) \wedge (j > k)$.

Alors, $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j]) = \text{clef}(\text{TAB}[k])$, donc, par définition, $i \succeq k$.

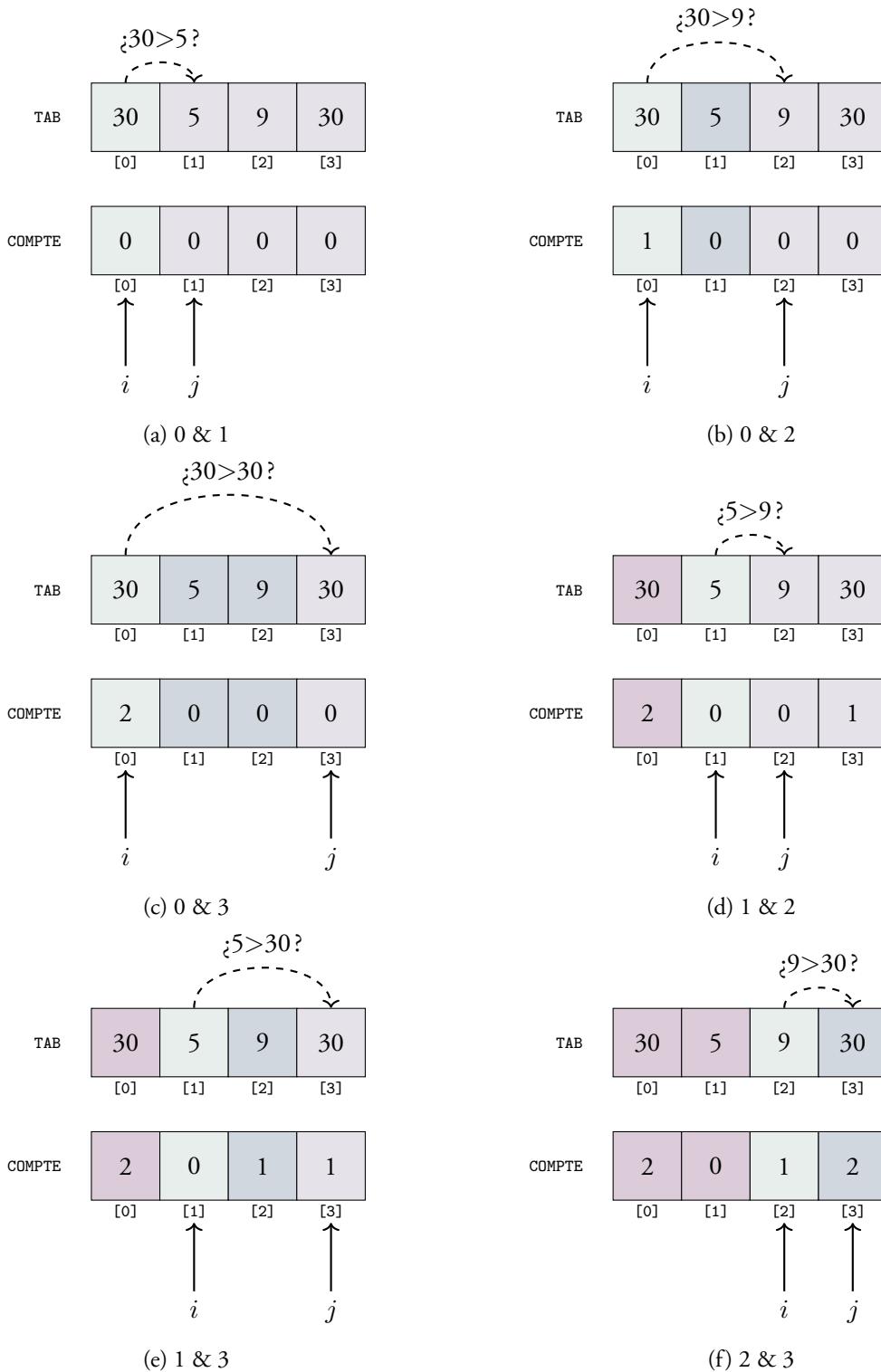


FIGURE 3.6 – La danse des variables de boucle avec les invariants

3. Si $(\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])) \wedge (i > j)$ et $\text{clef}(\text{TAB}[j]) > \text{clef}(\text{TAB}[k])$.
Alors, $\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j]) > \text{clef}(\text{TAB}[k])$, donc, par définition, $i \succeq k$.
4. Si $(\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])) \wedge (i > j)$ et $(\text{clef}(\text{TAB}[j]) = \text{clef}(\text{TAB}[k])) \wedge (j > k)$.
Alors $\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j]) = \text{clef}(\text{TAB}[k])$. Comme de plus, $i > j$ et $j > k$, par transitivité et antisymétrie de l'ordre sur les indices, on déduit que $i > k$. Donc $\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[k])$ et $i > k$, ce qui, par définition, implique $i \succeq k$.

Ainsi, dans tous les cas, $i \succeq k$.

antisymétrie soient i et j deux indices. Supposons $i \succeq j$ et $j \succeq i$. On veut prouver que $i = j$.

Comme ci-dessus, on a trois possibilités pour justifier la première inégalité, trois pour la seconde. Cela fait neuf cas, mais dans cinq d'entre eux, $i = j$ est une hypothèse (et donc on conclut en y faisant appel). Cela laisse quatre autres cas :

1. Si $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])$ et $\text{clef}(\text{TAB}[j]) > \text{clef}(\text{TAB}[i])$.
2. Si $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])$ et $(\text{clef}(\text{TAB}[j]) = \text{clef}(\text{TAB}[i])) \wedge (j > i)$.
3. Si $(\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])) \wedge (i > j)$ et $\text{clef}(\text{TAB}[j]) > \text{clef}(\text{TAB}[i])$.
4. Si $(\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])) \wedge (i > j)$ et $(\text{clef}(\text{TAB}[j]) = \text{clef}(\text{TAB}[i])) \wedge (j > i)$.

Ces quatre cas sont tous absurdes : en effet, on suppose en même temps une inégalité stricte dans les deux sens opposés dans le premier et le dernier ; et une inégalité stricte et une égalité dans les deux autres cas.

Donc, pour tous les cas où ça a un sens, on a $i = j$.

totalité soient i et j deux indices. On veut montrer qu'ils sont comparables pour l'ordre \succeq , c'est-à-dire que soit $i \succeq j$, soit $j \succeq i$. L'ordre $>$ sur les clefs est total, aussi, étant données deux clefs, soit elles sont égales, soit une des deux est strictement plus grande.

- si $\text{clef}(\text{TAB}[i]) > \text{clef}(\text{TAB}[j])$, alors $i \succeq j$.
- si $\text{clef}(\text{TAB}[i]) < \text{clef}(\text{TAB}[j])$, alors $j \succeq i$.
- si $\text{clef}(\text{TAB}[i]) = \text{clef}(\text{TAB}[j])$, alors on a trois possibilités :
 1. soit $i = j$, et $i \succeq j$ par définition.
 2. soit $i > j$, et donc $i \succeq j$ par définition.
 3. soit $j > i$, et donc $j \succeq i$ par définition.

Dans tous les cas, on a pu comparer les indices.



On voit que le fait que la relation sur les clefs est un ordre total est fondamentale pour que \succeq soit, à son tour, un ordre total. Vérifions que cette propriété est importante.

Exercice 68. Reprenons l'exemple de papier-caillou-ciseaux. On a trois symboles : , et , tels que :

$$\begin{array}{l} \text{Rock} > \text{Paper} \\ \text{Paper} > \text{Scissors} \\ \text{Scissors} > \text{Rock} \end{array}$$

Essayez d'appliquer l'algorithme au tableau suivant :

		
TAB [0]	TAB [1]	TAB [2]

Quel est le problème?

On peut maintenant prouver que les propriétés qu'on a énoncées sont vérifiées à certains moment de l'exécution.

Proposition 2. *À l'exécution 0 de la ligne 4, la propriété ($P_1(0)$) est vérifiée.*

Démonstration. La propriété ($P_1(0)$) est vérifiée : en effet, elle est la conjonction de deux propriétés, une sur les indices strictement inférieurs à zéro (il n'y en n'a pas), l'autre, instanciée en 0, donne :

$$\forall 0 \leq \ell < N, \text{COMPTE}[\ell] = \text{Card} \{0 \leq j < 0 \mid \ell \succ j\}$$

Or, l'ensemble à droite est vide, donc la condition ($P_1(0)$) énonce que le tableau COMPTE contient des zéros dans chacune de ses cases. \odot

Proposition 3. *Pour $0 \leq i < N$, si, à la i -ème exécution de la ligne 4, la propriété ($P_1(i)$) est vérifiée, alors à la prochaine exécution 0 de la ligne 5, la propriété ($P_2(i, i + 1)$) est vérifiée.*

Démonstration. La propriété ($P_2(i, i + 1)$) s'écrit :

$$\begin{aligned} \forall 0 \leq \ell < i, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ \wedge \text{COMPTE}[i] &= \text{Card} \{0 \leq k < i + 1 \mid \ell \succ k\} \\ \wedge \forall i < \ell < i + 1, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k \leq i \mid \ell \succ k\} \\ \wedge \forall i + 1 \leq \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < i \mid \ell \succ k\} \end{aligned}$$

tandis que $P_1(i)$ est :

$$\begin{aligned} \forall 0 \leq \ell < i, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq j < N \mid \ell \succ j\} \\ \wedge \forall i \leq \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq j < i \mid \ell \succ j\} \end{aligned}$$

que l'on peut ré-écrire

$$\begin{aligned} \forall 0 \leq \ell < i, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq j < N \mid \ell \succ j\} \\ \wedge \text{COMPTE}[i] &= \text{Card} \{0 \leq j < i \mid i \succ j\} \\ \wedge \forall i < \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq j < i \mid \ell \succ j\} \end{aligned}$$

On voit donc que si $\text{Card} \{0 \leq j < i \mid i \succ j\} = \text{Card} \{0 \leq k < i + 1 \mid i \succ k\}$, alors $P_1(i)$ implique $P_2(i, i + 1)$. Cette égalité est vraie, car en effet, la relation \succ est un ordre strict : il est faux que $i \succ i$. \odot

Proposition 4. *Pour $0 \leq i < N$ et $i < j < N - 1$, si au j -ème passage ligne 5, la propriété ($P_2(i, j)$) est vérifiée, alors au $j + 1$ -ème passage ligne 5, la propriété ($P_2(i, j + 1)$) l'est aussi.*

Démonstration. Supposons ($P_2(i, j)$) vérifiée au j -ème passage ligne 5. Elle s'écrit, pour rappel :

$$\begin{aligned} P_2(i, j) : \quad \forall 0 \leq \ell < i, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ \wedge \text{COMPTE}[i] &= \text{Card} \{0 \leq k < j \mid \ell \succ k\} \\ \wedge \forall i < \ell < j, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k \leq i \mid \ell \succ k\} \\ \wedge \forall j \leq \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < i \mid \ell \succ k\} \end{aligned}$$

En particulier, on a :

$$\begin{aligned}\text{COMPTE}[i] &= \text{Card} \{0 \leq k < j \mid i \succ k\} \\ \text{COMPTE}[j] &= \text{Card} \{0 \leq k < i \mid j \succ k\}\end{aligned}$$

La clef de l'enregistrement en i est ensuite comparée à celle de l'enregistrement en j . On a deux cas :

- si celle en i est strictement supérieure, alors, par définition de \succ , $i \succ j$, donc

$$\text{Card} \{0 \leq k < j + 1 \mid i \succ k\} = \text{Card} \{0 \leq k < j \mid i \succ k\} + 1.$$

Par ailleurs, on incrémente $\text{COMPTE}[i]$. Donc, après l'incrémantation,

$$\begin{aligned}\text{COMPTE}[i] &= \text{Card} \{0 \leq k < j \mid i \succ k\} + 1 \\ &= \text{Card} \{0 \leq k < j + 1 \mid i \succ k\}\end{aligned}$$

Quant à $\text{COMPTE}[j]$, comme il est faux que $j \succ i$,

$$\begin{aligned}\text{COMPTE}[j] &= \text{Card} \{0 \leq k < i \mid j \succ k\} \\ &= \text{Card} \{0 \leq k \leq i \mid j \succ k\}\end{aligned}$$

Comme aucune autre case du tableau n'est modifiée, on a bien vérifié ($P_2(i, j + 1)$).

- sinon, soit la clef dans le j enregistrement est strictement supérieure à celle dans le i -ème (auquel cas $j \succ i$), soit les clefs sont égales, mais comme $j > i$, on a aussi $j \succ i$. Donc

$$\text{Card} \{0 \leq k \leq i \mid j \succ k\} = \text{Card} \{0 \leq k \leq i \mid j \succ k\} + 1.$$

Par ailleurs, on incrémente $\text{COMPTE}[j]$. Donc, après l'incrémantation,

$$\begin{aligned}\text{COMPTE}[j] &= \text{Card} \{0 \leq k < i \mid j \succ k\} + 1 \\ &= \text{Card} \{0 \leq k \leq i \mid j \succ k\}\end{aligned}$$

Quant à $\text{COMPTE}[i]$, comme il est faux que $i \succ j$,

$$\begin{aligned}\text{COMPTE}[i] &= \text{Card} \{0 \leq k < j \mid i \succ k\} \\ &= \text{Card} \{0 \leq k < j + 1 \mid i \succ k\}\end{aligned}$$

Comme aucune autre case du tableau n'est modifiée, on a bien vérifié ($P_2(i, j + 1)$).

Donc, dans tous les cas, après passage dans la boucle, on vérifie ($P_2(i, j + 1)$). ⊕

Proposition 5. Pour $0 \leq i < N - 1$, si au $N - 1$ -ème passage ligne 5, la propriété ($P_2(i, N - 1)$) est vérifiée, alors au $i + 1$ -ème passage ligne 4, la propriété ($P_1(i + 1, i + 2)$) l'est aussi.

Démonstration. Rappelons la propriété ($P_2(i, N - 1)$).

$$\begin{aligned}P_2(i, N - 1) : \quad &\forall 0 \leq \ell < i, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ &\wedge \text{COMPTE}[i] = \text{Card} \{0 \leq k < N - 1 \mid \ell \succ k\} \\ &\wedge \forall i < \ell < N - 1, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k \leq i \mid \ell \succ k\} \\ &\wedge \text{COMPTE}[N - 1] = \text{Card} \{0 \leq k < i \mid \ell \succ k\}\end{aligned}$$

La clef de l'enregistrement en i est ensuite comparée à celle de l'enregistrement en $N - 1$. On a deux cas :

— si celle en i est strictement supérieure, alors, par définition de \succ , $i \succ N - 1$, donc

$$\text{Card} \{0 \leq k < N \mid i \succ k\} = \text{Card} \{0 \leq k < N - 1 \mid i \succ k\} + 1.$$

Par ailleurs, on incrémente $\text{COMPTE}[i]$. Donc, après l'incrémantation,

$$\begin{aligned} \text{COMPTE}[i] &= \text{Card} \{0 \leq k < N - 1 \mid i \succ k\} + 1 \\ &= \text{Card} \{0 \leq k < N \mid i \succ k\} \end{aligned}$$

On a donc $\forall 0 \leq \ell < i + 1, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k < N \mid \ell \succ k\}$. Par ailleurs, il est faux que $\ell \succ i$, donc $\text{COMPTE}[N - 1] = \text{Card} \{0 \leq k \leq i \mid \ell \succ k\}$. Comme aucune autre case du tableau n'est modifiée, après l'incrémantation, on a :

$$\begin{aligned} \forall 0 \leq \ell < i + 1, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ \wedge \forall i < \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k \leq i \mid \ell \succ k\} \end{aligned}$$

— sinon, $N - 1 \succ i$, donc

$$\text{Card} \{0 \leq k < i + 1 \mid \ell \succ k\} = \text{Card} \{0 \leq k < i \mid \ell \succ k\} + 1.$$

Par ailleurs, on incrémente $\text{COMPTE}[N - 1]$, donc après l'incrémantation, on a :

$$\begin{aligned} \text{COMPTE}[N - 1] &= \text{Card} \{0 \leq k < i \mid \ell \succ k\} + 1 \\ &= \text{Card} \{0 \leq k < i + 1 \mid \ell \succ k\} \end{aligned}$$

On a donc $\forall i < \ell < N - 1, \text{COMPTE}[\ell] = \text{Card} \{0 \leq k \leq i \mid \ell \succ k\}$. Par ailleurs, il est faux que $i \succ N - 1$, donc $\text{COMPTE}[i] = \text{Card} \{0 \leq k < N \mid \ell \succ k\}$. Comme aucune autre case du tableau n'est modifiée, après l'incrémantation, on a :

$$\begin{aligned} \forall 0 \leq \ell < i + 1, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ \wedge \forall i < \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k \leq i \mid \ell \succ k\} \end{aligned}$$

On peut remarquer que, comme il est faux que $i + 1 \succ i + 1$,

$$\text{COMPTE}[i + 1] = \text{Card} \{0 \leq k < i + 2 \mid \ell \succ k\},$$

donc cette propriété est en fait équivalente à

$$\begin{aligned} P_2(i + 1, i + 2) : \quad \forall 0 \leq \ell < i + 1, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < N \mid \ell \succ k\} \\ \wedge \text{COMPTE}[i + 1] &= \text{Card} \{0 \leq k < i + 2 \mid \ell \succ k\} \\ \wedge \forall i + 2 \leq \ell < N, \text{COMPTE}[\ell] &= \text{Card} \{0 \leq k < i + 1 \mid \ell \succ k\} \end{aligned}$$

⊕

On remarque que $(P_1(N))$, $(P_2(N - 1, N))$ et (C) sont la même propriété.

Par ailleurs, $P_3(0)$ est vraie tout le temps.

Proposition 6. *Après être rempli par les deux boucles imbriquées, chaque case du tableau COMPTE contient un entier différent. De plus, chaque entier entre 0 et N en fait partie.*

Démonstration. C'est une conséquence de ce que \succ est un ordre total. En effet, soient i et j deux indices distincts. Par totalité, quitte à les inverser, supposons $i \succ j$. Par transitivité, l'ensemble des indices strictement plus petits que j est strictement inclus dans celui des indices strictement plus petits que i :

$$\{0 \leq k < N \mid j \succ k\} \subset \{0 \leq k < N \mid i \succ k\}$$

Par conséquent, par la propriété (C) , $\text{COMPTE}[j] < \text{COMPTE}[i]$.

Comme, par la propriété (C) , les entiers dans les cases sont tous compris entre 0 et N (exclus), il y a donc N entiers différents pouvant prendre N valeurs : toutes les valeurs sont prises. ⊕

Proposition 7. Pour $0 \leq i < N$, si, à la i -ème exécution de la ligne 13, la propriété $(P_3(i))$ est vérifiée, alors à la prochaine exécution de la ligne 14, la propriété $(P_3(i+1))$ est vérifiée.

Démonstration. Rappelons

$$P_3(i) : \forall 0 \leq k, \ell < i, (\text{COMPTE}[k] > \text{COMPTE}[\ell] \implies \text{clef}(\text{RES}[\text{COMPTE}[k]]) \geq \text{clef}(\text{RES}[\text{COMPTE}[\ell]]))$$

Soit $k < i$. Supposons que $\text{COMPTE}[i] > \text{COMPTE}[k]$. Comme l'ordre \succ est total, cela signifie que $i \succ k$. En particulier, $\text{clef}(\text{TAB}[i]) \geq \text{clef}(\text{TAB}[k])$. Donc, après l'affectation, on a bien que $\text{clef}(\text{RES}[\text{COMPTE}[i]]) \geq \text{clef}(\text{RES}[\text{COMPTE}[k]])$. \odot

On déduit de cette chaîne de propositions que, en sortir de la dernière boucle,

$$\forall 0 \leq k, \ell < N, (k > \ell \implies \text{clef}(\text{RES}[k]) \geq \text{clef}(\text{RES}[\ell]))$$

En effet, tous les entiers entre 0 et N exclus sont présents dans COMPTE . D'où le:

Théorème 1. L'algorithme de tri par énumération (Algorithme 22) est correct: il implémente bien une fonction de tri.

Résumé

L'algorithme de tri par énumération est simple: il repose sur une seule idée: compter, pour chaque enregistrement, les enregistrements de clef plus petite que la sienne. C'est un *tri stable*, c'est-à-dire que deux enregistrements de même clef sont, dans le tableau final, dans l'ordre où ils étaient dans le tableau initial. Son temps d'exécution ne dépend pas du tableau qu'il trie: il est incapable de repérer des morceaux déjà triés. Ainsi, il a une complexité en $O(N^2)$ (si N est la longueur du tableau) dans tous les cas.

Bien que simple, son étude nous a nécessité d'introduire un ordre sur les indices, décrivant la manière dont les enregistrements sont rangés; et d'introduire des invariants de boucle, telle la propriété $P_2(i, j)$ représentée Figure 3.5.

3.5 INSERTION

On l'a dit, le tri par énumération est inefficace. Une raison qui saute aux yeux est que certes, on ne compare qu'une fois chaque élément avec chaque autre, mais on fait quand même trop de comparaisons: supposons qu'on a constaté que la clef de l'enregistrement en position $i + 1$ est plus petite que celle de l'enregistrement en i , puis que la clef de l'enregistrement en $i + 2$ est plus grande que celle de l'enregistrement en i , on met à jour le compteur en $i + 2$, pour indiquer qu'il y a une clef de plus petite (celle en i). Or, on sait déjà (par transitivité!) qu'il y a deux clefs de plus petites, celle en i , mais aussi celle en $i + 1$. On pourrait imaginer un système où l'on garde en mémoire le nombre d'éléments de clef plus petite croisée à chaque tour de boucle, mais ce serait assez compliqué.

Exercice 69. Adapter l'Algorithme de tri par énumération pour éviter certaines comparaisons.

Lister les difficultés.

Ainsi, on veut pouvoir ne pas reproduire des comparaisons dont, par transitivité, on connaît déjà le résultat, sans avoir à utiliser de structures de données trop compliquée pour rendre compte de cette transitivité. Une possibilité peut-être de déplacer les enregistrements au fur et à mesure qu'on les traite: quand on traite l'enregistrement en position i , tous ceux à sa gauche ont des clefs qui sont inférieures à la sienne. On va donc faire une boucle qui va traiter les éléments les uns à la suite des autres, et faire reculer cet enregistrement de manière à ce qu'il soit placé avant ceux ayant une clef supérieure qui étaient, avant, placés avant lui. Cet algorithme s'appelle le *tri par insertion*

En s'autorisant une nouvelle notation \leftrightarrow qui nous permet d'échanger deux variables (\leftrightarrow n'est qu'une abréviation pour trois instructions), cela nous donne l'Algorithme 24 :

Entrées:	un tableau d'enregistrements $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N.
1	TriInsertion $((\text{TAB}[i])_{0 \leq i < N})$
2	pour i allant de 1 à N faire
3	$j \leftarrow i$
4	tant que $j > 0$ et $\text{clef}(\text{TAB}[j - 1]) > \text{clef}(\text{TAB}[j])$ faire
5	$\text{TAB}[j - 1] \leftrightarrow \text{TAB}[j]$
6	$j \leftarrow j - 1$
7	fin
8	fin
9	retourner $(\text{TAB}[i])_{0 \leq i < N}$
Sorties: un tableau d'enregistrements de longueur N	

Algorithme 24: Tri par insertion

On fait commencer la boucle à 1 car le 0-ème élément ne peut pas être comparé avec les éléments avant lui, vu qu'il n'y en n'a pas.

Ce tri est un *tri stable*, comme le *tri par énumération*. De plus, c'est un *tri en place*, c'est-à-dire qu'il modifie le tableau qu'on lui a donné en entrée. Enfin, c'est un tri *incrémentiel*, c'est-à-dire qu'il n'y a aucune difficulté à le faire tourner sur des données partielles (les premières cases du tableau) et rajouter des éléments à chaque fois qu'une nouvelle case devient disponible. C'est pour ça que c'est le tri préféré des joueurs de bridge, qui reçoivent leurs cartes une à une.

Exercice 70. Le tri par énumération était-il en place? incrémentiel?

Exercice 71. Écrire un algorithme implémentant \leftrightarrow , c'est à dire une procédure telle qu'écrire

$$(a, b) \leftarrow \text{Echange}(a, b)$$

aie le même effet que

$$a \leftrightarrow b.$$

On peut rapidement l'exécuter sur notre tableau préféré. 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 :

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

On obtient l'exécution résumée Figure 3.7.

On voit déjà qu'on a comparé le dernier enregistrement uniquement avec un autre, et que, par transitivité, cela suffit.

Exercice 72. Programmer l'algorithme 24 en C. Rappel: comme c'est un tri en place, on attend une fonction de prototype

```
1 int tri(const size_t longueur, int tab[longueur]);
```

Correction:

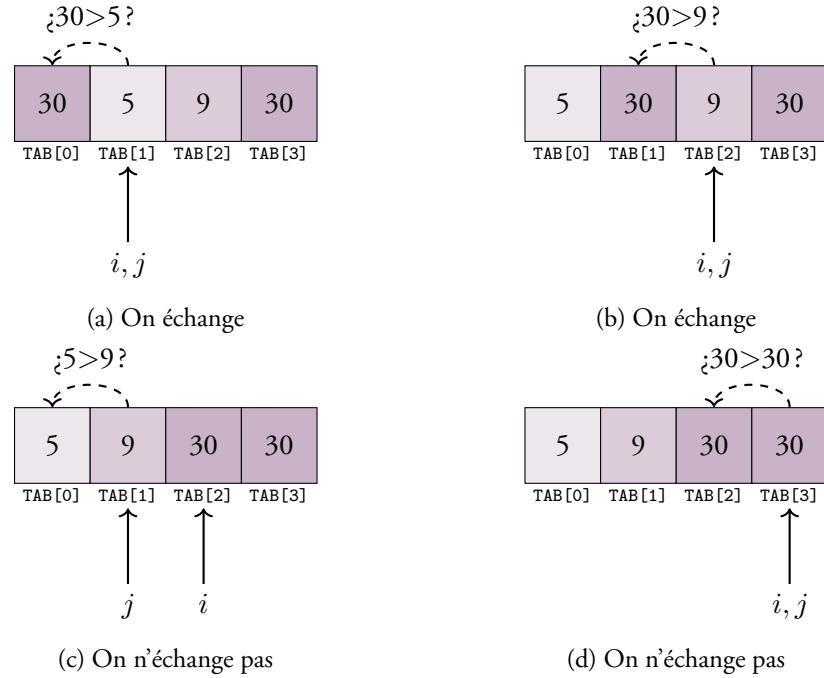


FIGURE 3.7 – La danse des enregistrements

```
1 int tri(const size_t longueur, int tab[longueur]){
2     for(size_t i = 1; i < longueur; i++){
3         size_t j = i;
4         while(j > 0 && tab[j-1] > tab[j]){
5             int a = tab[j-1];
6             tab[j-1] = tab[j];
7             tab[j] = a;
8             j--;
9         }
10    }
11    return 0;
12 }
```

Correction

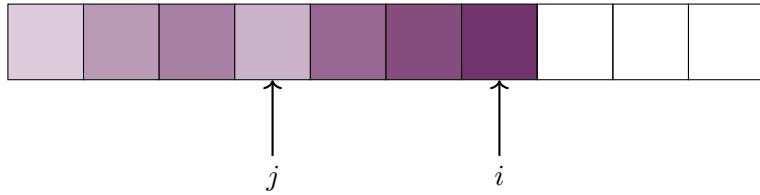
Pour évaluer la correction, il faut commencer par trouver des invariants de boucle. Comme il y a deux boucles, on veut en trouver deux. Pour la boucle **pour** extérieure, c'est assez facile: il s'agit d'un progrès partiel: tous les enregistrements à gauche de i sont classés par clef croissante (ce qui ne veut pas dire que ce sont les plus petites clefs de tout le tableau). On note, pour $1 \leq i \leq N$,

$$P_1(i) : \forall 0 \leq k, \ell < i, (k < \ell \implies \text{clef}(\text{TAB}[k]) \leq \text{clef}(\text{TAB}[\ell]))$$

Pour la boucle **tant que** intérieure, la propriété est plus complexe : il s'agit d'énoncer que les enregistrements avant i sont classés par clef croissante, sauf celui en j , dont la clef est néanmoins inférieure à celles entre j et i . On note, pour $1 \leq i \leq N$ et $0 < j \leq i$,

$$\begin{aligned} P_2(i, j) : \forall 0 \leq k, \ell < i, (k < \ell \wedge (k \neq j \vee \ell \neq j)) \implies \text{clef}(\text{TAB}[k]) \leq \text{clef}(\text{TAB}[\ell])) \\ \wedge (j < \ell \implies \text{clef}(\text{TAB}[j]) < \text{clef}(\text{TAB}[\ell])) \end{aligned}$$

Ce qu'on peut représenter ainsi (la case en j interrompt le camaïeu graduel) :



Les cases avant i sont dans l'ordre, sauf la case en j , qui recule tant qu'elle a une clef plus petite que celle en $j - 1$.

Proposition 8. *À l'exécution 0 de la ligne 2, la propriété $(P_1(0))$ est vérifiée.*

Démonstration. C'est le cas car la propriété $(P_1(0))$ est vide: elle ne porte que sur des indices plus grands et strictement plus petits que 0 — c'est-à-dire sur aucun indice. \odot

Proposition 9. *Si à la i -ème exécution de la ligne 2 la propriété $(P_1(i))$ est vérifiée, alors avant la prochaine exécution de la ligne 4, la propriété $(P_2(i, i))$ est vérifiée.*

Démonstration. La propriété $P_2(i, i)$ se ré-écrit

$$\begin{aligned} P_2(i, i) : \forall 0 \leq k, \ell < i, (k < \ell \wedge (k \neq i \vee \ell \neq i)) \implies \text{clef}(\text{TAB}[k]) \leq \text{clef}(\text{TAB}[\ell]) \\ \wedge (i < \ell \implies \text{clef}(\text{TAB}[i]) < \text{clef}(\text{TAB}[\ell])) \end{aligned}$$

mais on remarque que la prémissse de la deuxième partie est absurde, car ℓ est pris à la fois strictement inférieur et strictement supérieur à i , et les conditions $k \neq i \vee \ell \neq i$ sont aussi toujours vérifiées, donc $(P_2(i, i))$ est en fait équivalent à $(P_1(i))$. \odot

Proposition 10. *Si on entre dans la boucle **tant que** alors que la propriété $(P_2(i, j))$ est vraie, alors en fin de boucle (ligne 7), la propriété $(P_2(i, j))$ est vérifiée (par la nouvelle valeur de j).*

Démonstration. La condition qui fait que l'on entre dans la boucle est exactement celle qui permet de conclure qu'après la ligne 6 $(P_2(i, j - 1))$ est vérifiée. \odot

Proposition 11. *Si on n'entre pas dans la boucle **tant que** alors que la propriété $(P_2(i, j))$ est vraie, alors la propriété $(P_1(i + 1))$ est vérifiée.*

Démonstration. Comme on entre pas dans la boucle, c'est soit que $j == 0$, et dans ce cas $(P_1(i + 1))$ est vérifiée; soit que le tableau est dans l'ordre, et $(P_1(i + 1))$ est vérifiée. \odot

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 $\text{clef}(\text{TAB}[i - 1]) \leq \text{clef}(\text{TAB}[i])$. Autrement dit, dans le meilleur des cas, si cette propriété est vraie pour chaque $1 \leq i < N$, 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 fois⁸. 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 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

8. On se rappelle que ce n'est pas du tout le cas du tri par énumération: vérifier que le tableau est trié nécessite de comparer tous les éléments deux à deux.

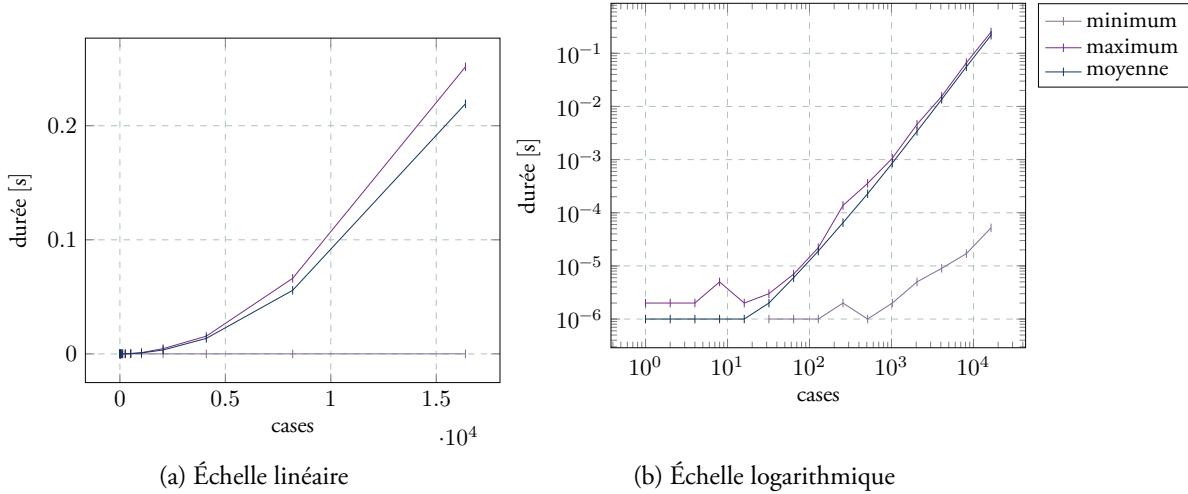


FIGURE 3.8 – Temps d'exécution du tri par insertion sur des tableaux de taille variable

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 $O(n^2)$, en complexité **quadratique**.

Ainsi, dans le pire cas on est quadratique, dans le meilleur linéaire. On ne va pas essayer de calculer la complexité en moyenne, mais expérimentalement, on voit sur la Figure 3.8 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.

3.6 FUSION

Le *tri fusion*⁹ 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, l’Algorithme 25.

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.

9. 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.

Entrées: deux tableaux d'enregistrements $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N et $(\text{TAB}'[i])_{0 \leq i < M}$ de longueur M , tous deux triés.

```

1 Fusion((\text{TAB}[i])_{0 \leq i < N}, (\text{TAB}'[i])_{0 \leq i < M})
2   \text{nouveau } (\text{RES}[i])_{0 \leq i < N+M}
3   i \leftarrow 0
4   j \leftarrow 0
5   \text{tant que } i < N \text{ et } j < M \text{ faire}
6     \text{si } \text{clef}(\text{TAB}[i]) < \text{clef}(\text{TAB}'[j]) \text{ alors}
7       \text{RES}[i + j] \leftarrow \text{TAB}[i]
8       i \leftarrow i + 1
9     \text{sinon}
10    \text{RES}[i + j] \leftarrow \text{TAB}'[j]
11    j \leftarrow j + 1
12  \text{fin}
13
14  \text{tant que } i < N \text{ faire}
15    \text{RES}[i + j] \leftarrow \text{TAB}[i]
16    i \leftarrow i + 1
17  \text{fin}
18  \text{tant que } j < M \text{ faire}
19    \text{RES}[i + j] \leftarrow \text{TAB}'[j]
20    j \leftarrow j + 1
21  \text{fin}
22  \text{retourner } (\text{RES}[i])_{0 \leq i < N+M}
```

Sorties: un tableau d'enregistrements de longueur $N + M$

Algorithme 25: Fusion de tableaux triés

Exercice 73. Exécuter la fusion sur les deux tableaux

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

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

Exercice 74. Réécrire l'Algorithme 25 avec une seule boucle **tant que**.

Cet algorithme de fusion s'exécute en temps linéaire $O(N + M)$: 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 $N + M$ 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 $2N$. On avait calculé que trier un tel tableau par

un tri par énumération¹⁰ nécessitait

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

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 longueur 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 3.9 : 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 26.

Remarque 3. On suppose que la division est entière, comme en C, ce qui fait que $3/2==1$. On remarque que dans ce cas, $N/2$ et $N - N/2$ ne sont pas toujours égaux.

Entrées: un tableau d'enregistrements $(TAB[i])_{0 \leq i < N}$ de longueur N .

```

1 TriFusion((TAB[i])0 \leq i < N)
2   | si  $N = 1$  alors
3   |   | retourner  $(TAB[i])_{0 \leq i < N}$ 
4   | sinon
5   |   | nouveau (PREMIER[i])0 \leq i < N/2
6   |   | nouveau (SECOND[i])0 \leq i < N - N/2
7   |   | pour  $i$  allant de 0 à  $N/2$  faire
8   |   |   | PREMIER[i]  $\leftarrow TAB[i]$ 
9   |   | fin
10  |   | (PREMIER[i])0 \leq i < N/2  $\leftarrow$  TriFusion((PREMIER[i])0 \leq i < N/2)
11  |   | pour  $i$  allant de 0 à  $N - N/2$  faire
12  |   |   | SECOND[i]  $\leftarrow TAB[N/2 + i]$ 
13  |   | fin
14  |   | (SECOND[i])0 \leq i < N - N/2  $\leftarrow$  TriFusion((SECOND[i])0 \leq i < N - N/2)
15  |   | retourner Fusion((PREMIER[i])0 \leq i < N/2, (SECOND[i])0 \leq i < N - N/2)
16 | fin
```

Sorties: un tableau d'enregistrements de longueur N .

Algorithme 26: Tri fusion — la procédure récursive naïve

10. On choisit un tri par énumération car le nombre de comparaisons est constant. Le raisonnement fonctionne à l'identique avec un autre tri.

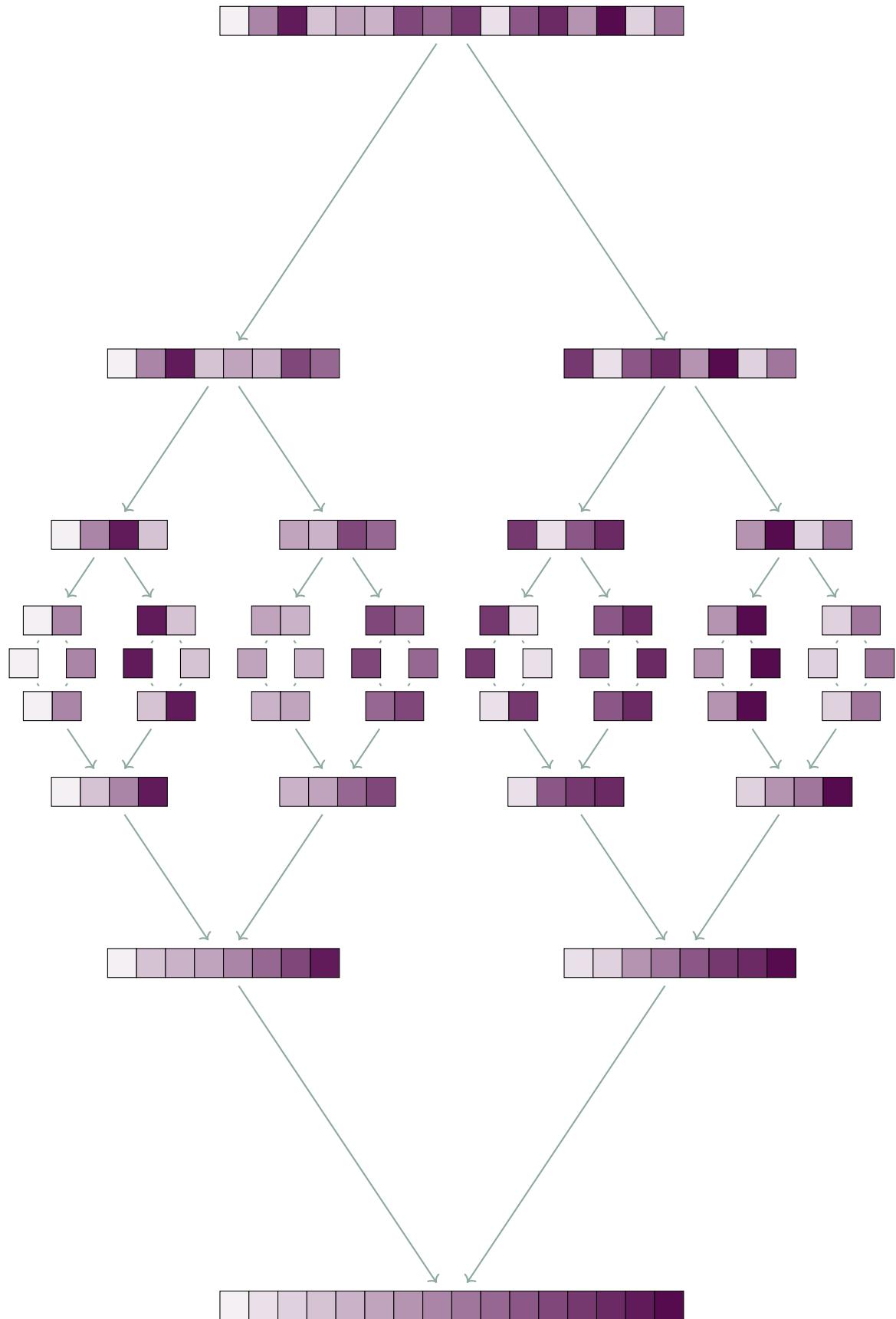


FIGURE 3.9 – Le tri fusion

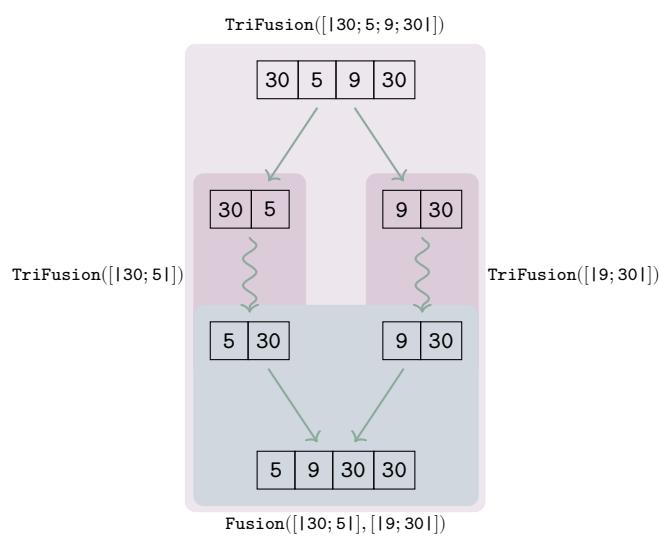
Tel qu'écrit, cet algorithme peut fonctionner (tout dépend des conventions sur les divisions entières) sur des tableaux de longueur quelconque. Néanmoins, on va se limiter dans tous les exemples à des tableaux dont les longueurs sont des puissances de 2 — et donc que l'on peut toujours diviser par 2. 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 75. Éxécuter cet algorithme sur le tableau

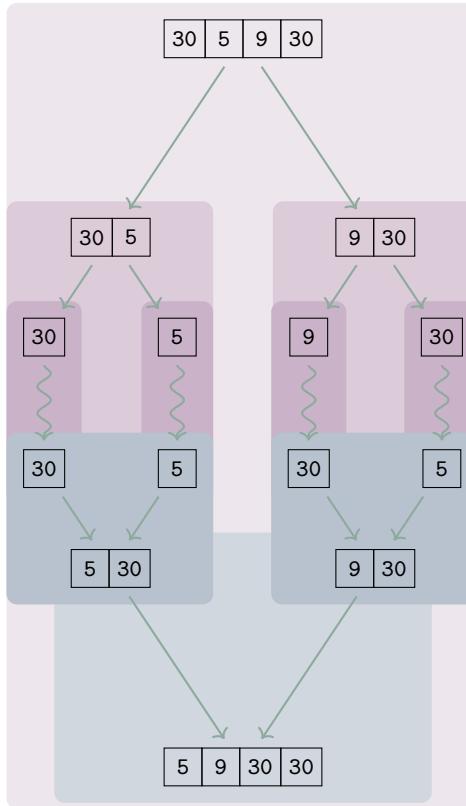
30	5	9	30
TAB[0]	TAB[1]	TAB[2]	TAB[3]

Correction: Plutôt que de faire un tableau, on va représenter les différentes procédures exécutées. On va noter $[[a; b; \dots]]$ les différents tableaux. Autrement dit, on veut exécuter $\text{TriFusion}([|30; 5; 9; 30|])$.

Pour exécuter, on voit que, ligne 10, on a besoin du résultat de l'exécution de $\text{TriFusion}([|30; 5|])$, ligne 14, celle de $\text{TriFusion}([|9; 30|])$, et enfin, ligne 15, on va appliquer Fusion sur leurs deux résultats respectifs.



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. ☺

Exercice 76. Programmer le tri fusion en C. Rappel: on attend une fonction de prototype

```
1 int tri(const size_t longueur, int avant[longueur], int apres[longueur]);
```

Correction:

Comme on n'a pas écrit de fonction de fusion de deux tableaux, on fait tout dans la même fonction.

```
1 int tri(const size_t longueur, int avant[longueur], int apres[longueur]){
2     if (longueur == 1) {
3         apres[0] = avant[0];
4     } else {
5         size_t milieu = longueur/2;
6         int premier[milieu];
7         int second[longueur-milieu];
8         for(size_t i = 0; i < milieu; i++){
9             premier[i] = avant[i];
10        }
11        for(size_t i = milieu; i < longueur; i++){
12            second[i-milieu] = avant[i];
13        }
14        tri(milieu, premier, premier);
15        tri(longueur-milieu, second, second);
16        size_t i = 0;
17        size_t j = 0;
18        while(i < milieu && j < longueur - milieu){
19            if (premier[i] < second[j]) {
20                apres[i+j] = premier[i];
21                i++;
22            }
23        }
24    }
25}
```

```

22     } else {
23         apres[i+j] = second[j];
24         j++;
25     }
26 }
27 while (i<milieu){
28     apres[i+j] = premier[i];
29     i++;
30 }
31 while (j< longueur - milieu){
32     apres[i+j] = second[j];
33     j++;
34 }
35 }
36 return 0;
37 }

```

©

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, l’Algorithme 26 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 $2N$ (en effet, il ne fait que diviser le tableau de longueur $2N$ 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 trié¹¹. L’invariant de boucle est donc qu’à chaque tour de boucle, les cases de 0 à $(i + j)$ du tableau $(\text{RES}[i])_{0 \leq i < N+M}$ contiennent les enregistrements contenus dans les cases de 0 à (i) du tableau $(\text{TAB}[i])_{0 \leq i < N}$ et ceux contenus dans les cases de 0 à (j) du tableau $(\text{TAB}'[i])_{0 \leq i < M}$, et, de plus, les cases de 0 à $(i + j)$ du tableau $(\text{RES}[i])_{0 \leq i < N+M}$ sont triées. On le vérifie.

Complexité

L’algorithme tel qu’on l’a écrit est assez mauvais en terme d’affections. On se reportera au prochain paragraphe pour une analyse. On ne va donc que compter les comparaisons. Pour cela, considérons la Figure 3.9. 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 $O(N \log(N))$.

11. D’une certaine manière, c’est pour la même raison que l’Exercice 74 est possible : ces trois boucles n’en sont secrètement qu’une.

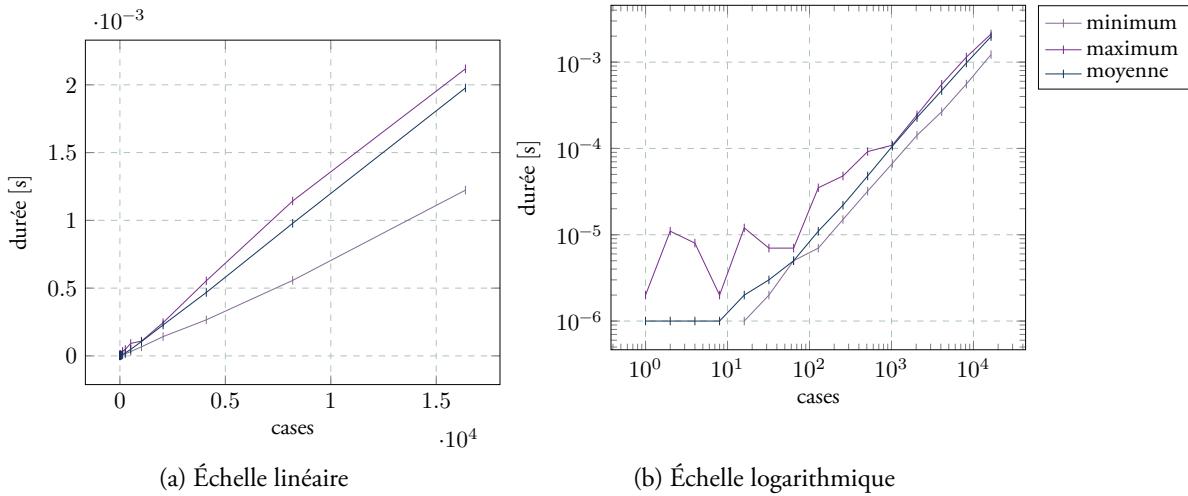


FIGURE 3.10 – Temps d'exécution du tri fusion sur des tableaux de taille variable

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 croît très lentement. On peut le voir sur la Figure 3.10, 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 77. 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 $O(n \log(n))$. C'est un algorithme récursif, et un exemple de la stratégie diviser-pour-régner.

3.7 BORNE INFÉRIEURE DE COMPLEXITÉ DES TRIS PAR COMPARAISON

On a donc trouvé un tri de complexité $O(n \log(n))$, 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 :

- 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 ? Voir 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 ?

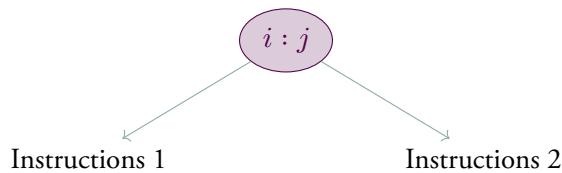
Lemme 1. Il y a $n!$ tableaux différents composés de n enregistrements fixés.

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

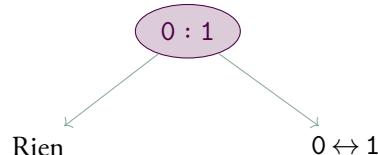
Théorème 2. *Un algorithme (séquentiel, déterministe) qui résoud le problème du tri ne peut le résoudre en moins de $O(n \log(n))$ dans le pire des cas.*

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, écrire des instructions de la forme $5 \leftrightarrow 8$ (qui échange 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.

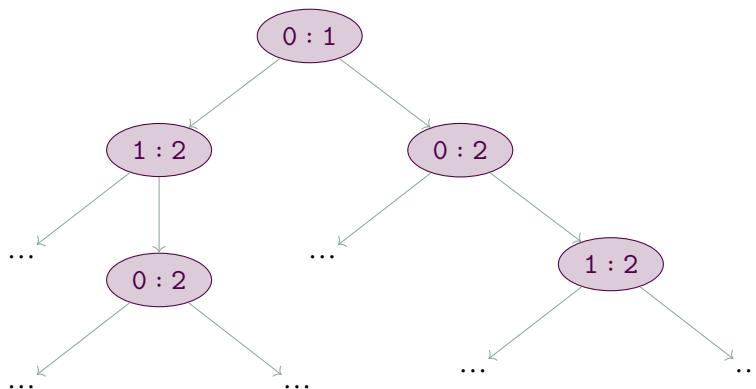


Et ainsi de suite. Par exemple, pour trier un tableau à deux éléments, tous les algorithmes vont comparer l'élément en position 0 et celui en position 1 et, si celui en position 0 est plus grand les inverser, sinon, ne rien faire, c'est-à-dire :



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 noeud 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

$$2^h \geq n!$$

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 $O(n \log(n))$. ☺

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 $O(n \log(n))$ 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 $O(n \log(n))$, 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).

3.8 RÉSUMÉ

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

3.A AMÉLIORER LE TRI FUSION

Dans ce devoir, on va s'intéresser à améliorer le tri fusion par plusieurs manières. Commençons par lister les aspects par lesquels le tri fusion (tel que nous l'avons écrit) est problématique :

espace mémoire à chaque fois que l'on sépare le tableau en deux, et de même, à chaque fois qu'on fusionne, on crée un nouveau tableau, et l'on recopie les données. Cela prend de la mémoire, une quantité qui croît en $O(n \log(n))$ (par comparaison, le tri par insertion nécessite une mémoire constante $O(1)$ et le tri par énumération une mémoire linéaire $O(n)$), et cela prend du temps de faire toutes ces copies.

absence d'incrémentalité vu qu'on veut diviser le tableau en deux, appliquer le tri fusion nécessite d'avoir toutes les données déjà disponibles. En particulier, cela signifie que l'on doit garder, avant le traitement, toutes les données¹².

comportement uniforme le tri fusion n'est pas capable d'exploiter des régularités présentes dans les données : si un morceau du tableau est déjà trié, il ne sera pas plus performant.

On va commencer par essayer de s'attaquer au premier problème. En somme, il vient de ce que la structure de tableau ne permet pas facilement d'être divisée en deux ou fusionnée. Essayons de voir ce qui se passerait avec des listes.

Rappel : la notation $[a_1, a_2, \dots, a_n]$ est une abréviation pour $(a_1, (a_2, \dots, (a_n, \Lambda)))$.

Fusion de listes

Exercice 78. Écrire un algorithme `FusionListesTriées` qui prend en entrée deux listes triées et retourne une liste triée composée des mêmes enregistrements.

Indication : on suppose avoir une fonction `clef` qui associe à chaque enregistrement sa clef.

Correction : On va supposer les listes triées par ordre croissant.

Cet algorithme va prendre les deux listes et s'appeler récursivement soit sur la queue de la première et la seconde, soit sur la queue de la seconde et la première, selon quel élément est le plus petit.

Entrées : deux listes d'enregistrements L et L' toutes deux triées.

```

1 FusionListesTriées(L, L')
2   si L == Λ alors
3     |   retourner L'
4   fin
5   si L' == Λ alors
6     |   retourner L
7   fin
8   (e, L) ← L
9   (e', L') ← L'
10  si clef(e') < clef(e) alors
11    |   retourner (e', FusionListesTriées((e, L), L'))
12  sinon
13    |   retourner (e, FusionListesTriées(L, (e', L')))
```

Sorties : une liste d'enregistrement triée

Algorithme 27 : Fusion de listes triés



12. Pensez aux questions légales : des données se trouvent qualifiées en données personnelles même si aucun humain ne les voit, par le simple fait qu'elles existent. Ici, on se retrouve à devoir stocker intégralement avant de pouvoir commencer le tri.

Exercice 79. Exécuter cet algorithme sur les deux listes $[2, 67]$ et $[3, 7, 9]$. On attend une trace d'exécution d'une fonction récursive.

Correction: On voit qu'on n'est pas récursif terminal : quand on fait un appel récursif, il reste à chaîner un élément. On a représenté une trace Figure 3.11. ⊕

Exercice 80. Donner la complexité en temps de cet algorithme.

Correction: À chaque fois qu'on fait une comparaison, on place définitivement un élément. Donc, on fait moins de comparaisons qu'il n'y a d'éléments, c'est-à-dire qu'on est en linéaire en la somme des longueurs des deux listes. ⊕

Rajouter le prochain élément

Si les listes chaînées résolvent bien le problème de la recopie mémoire pour la fusion, elles rendent la première étape (celle de division) moins claire : en effet, si on connaît la taille d'un tableau — on peut donc facilement le couper en son milieu — on ne connaît pas celle d'une liste sans la calculer, ce qui se fait en temps linéaire.

Une stratégie peut être de systématiquement rajouter à la liste déjà triée la liste constituée de l'élément suivant : on ne connaît pas la longueur totale, mais on sait toujours quoi fusionner.

Exercice 81. Écrire un algorithme `TriFusionListe` prenant en entrée une liste et appliquant la stratégie suivante : on parcourt la liste, on sépare le premier élément, on en fait une liste, on sépare le deuxième élément, on en fait une liste et on fusionne cette liste ainsi obtenue avec celle du premier, puis la liste ainsi obtenue avec le troisième élément,...

Indication : on pourra commencer par écrire une fonction `TriFusionListeAvec` qui prend en entrée deux listes L et L' , en supposant L' triée, et qui, fusionne d'abord L' avec le premier élément de L et s'appelle récursivement sur la queue de L et cette liste ; et renvoie ainsi une liste triée contenant les enregistrements de L et ceux de L' et s'appelle récursivement.

Correction: On commence par la fonction conseillée

Entrées: une liste d'enregistrements L .

```

1 TriFusionListe(L)
2   | retourner TriFusionListeAvec(L, Λ)
3
4 Sorties: une liste d'enregistrement triée
```

Entrées: deux listes d'enregistrements L et L' , L' triée.

```

1 TriFusionListeAvec(L, L')
2   | si L == Λ alors
3   |   | retourner L'
4   | sinon
5   |   | (e, L) ← L
6   |   | Q ← FusionListesTriées(L', [e])
7   |   | retourner TriFusionListeAvec(L, Q)
8   | fin
9
10 Sorties: une liste d'enregistrement triée
```

⊕

Exercice 82. Quelle est la complexité de cet algorithme dans le pire cas ? Le meilleur ? Qu'en concluez-vous ?

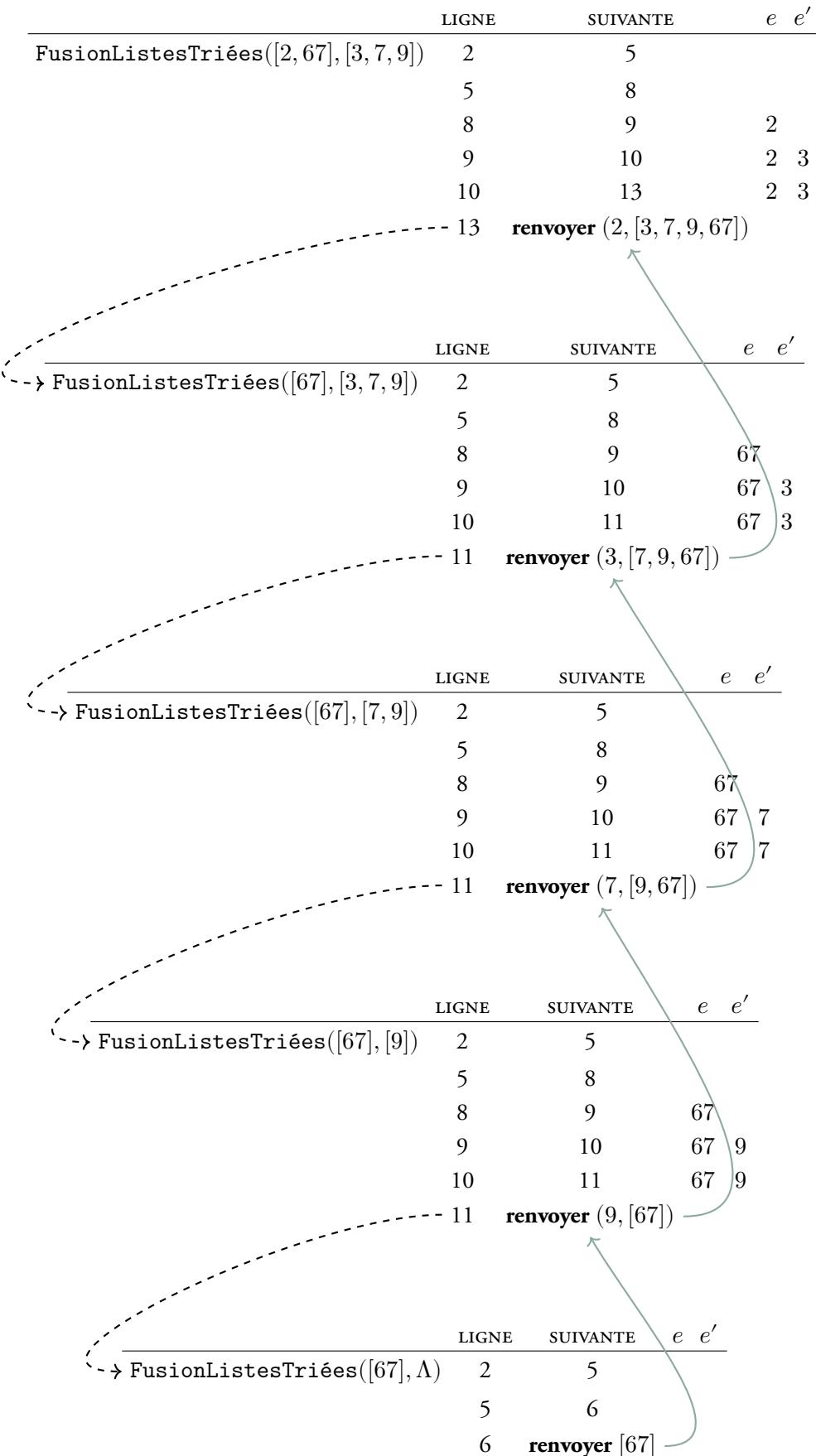


FIGURE 3.11 – Une trace d'exécution récursive

Correction: Chaque fusion se fait linéairement, au début sur deux listes de longueur totale 1, puis 2, ..., puis n . Donc, cet algorithme est de complexité quadratique dans le pire cas, linéaire dans le meilleur. En réalité, on a plutôt programmé un tri par insertion avec des listes. ☺

Fusion équilibrée

Le problème avec cette approche est que les fusions ne sont pas du tout équilibrées, contrairement avec ce qu'on faisait sur les tableaux. On va donc essayer de faire des fusions équilibrées *à l'aveugle*, sans connaître la longueur de la liste.

On va considérer qu'on a une pile de listes déjà en partie fusionnées : on part avec la pile vide, puis, à chaque étape, on rajoute, en haut de la pile, la liste contenant uniquement le premier élément de la liste que l'on souhaite trier. On compare la longueur du premier élément de la pile (qui est une liste) avec celle du second : tant que les deux premiers éléments sont de longueur égales, on les fusionne, sinon on continue. Enfin, on termine si on n'a plus de nouveaux éléments.

Pour éviter de recalculer toutes les longueurs, on va les stocker aussi dans la pile. Autrement dit, on a une pile dont chaque élément est un couple constitué d'un entier et d'une liste.

Sur la liste [8, 23, 2], on commence avec la liste vide, puis on rajoute, en haut de la pile, le couple (1, [8]). Puis, on rajoute le couple (1, [23]). Comme les deux derniers éléments sont de longueur égale, on supprime les deux premiers éléments de la pile, et on les remplace par (2, [8, 23]). Enfin, on rajoute (1, [2]) en haut de la pile. S'il restait d'autres éléments, on ne fusionnerait pas, comme il n'en reste pas, on fusionne, et on obtient (3, [2, 8, 23]).

Exercice 83. Représenter l'évolution de la pile pendant l'exécution sur la liste [8; 23; 2; 43; 9].

Correction: On va représenter la pile verticalement, avec son fond vers le bas, et le temps horizontalement.

		(1, [43])		
(1, [23])		(1, [2])	(1, [2])	(2, [2, 43])
(1, [8])	(1, [8])	(2, [8, 23])	(2, [8, 23])	(2, [8, 23])
		(4, [2, 8, 23, 43])	(4, [2, 8, 23, 43])	(5, [2, 8, 9, 23, 43])

☺

Exercice 84. Écrire un algorithme `AjoutFusionPile` prenant en entrée :

- une pile dont chaque élément est un couple constitué d'un entier n et d'une liste triée de longueur n , dont on suppose que les longueurs sont classées par ordre strictement (le fond de pile a donc la plus longue liste, etc...)
 - un couple constitué d'un entier et d'une liste triée de cette longueur
- et qui renvoie une pile ayant la même propriété, contenant les mêmes enregistrements que toutes ces listes, en fusionnant le moins de listes possibles.

Ainsi, sur la pile [(1, [2]), (2, [8, 12]), (8, [1, 2, 3, 4, 5, 6, 7, 8])] et le couple (1, [23]), on ne peut pas juste rajouter le couple en haut de la liste, car les deux premières longueurs ne sont pas strictement croissantes ; on doit fusionner les deux listes, mais ce faisant on obtient une liste de longueur 2, et on aurait pas une longueur strictement croissante, donc on doit encore fusionner. Le résultat doit donc être [(4, [2, 8, 12, 23]), (8, [1, 2, 3, 4, 5, 6, 7, 8])].

Correction:

☺

Exercice 85. Écrire un algorithme `FusionTotalePile` prenant en entrée une pile dont chaque élément est un couple constitué d'un entier n et d'une liste triée de longueur n et qui renvoie une liste qui est la fusion de toutes les listes de la pile.

Correction:

☺

Entrées:

- une pile de couples d'entiers et de listes d'enregistrement \mathbb{P} ;
- un couple d'un entier et une liste d'enregistrements \mathcal{L} .

```

1 AjoutFusionPile( $\mathbb{P}, \mathcal{L}$ )
2   | si  $\mathbb{P} == \Lambda$  alors
3   |   | retourner ( $\mathcal{L}, \Lambda$ )
4   | sinon
5   |   |  $((n', L'), \mathbb{P}) \leftarrow \mathbb{P}$ 
6   |   |  $(n, L) \leftarrow \mathcal{L}$ 
7   |   | si  $n < n'$  alors
8   |   |   | retourner  $((n, L), ((n', L'), \mathbb{P}))$ 
9   |   | sinon
10  |   |   | retourner AjoutFusionPile( $\mathbb{P}, (n + n', \text{FusionListesTriées}(L, L'))$ )
11  |   | fin
12  | fin

```

Sorties: une pile de couples d'entiers et de listes d'enregistrement

Entrées: une pile de couples d'entiers et de listes d'enregistrement \mathbb{P} .

```

1 FusionTotalePile( $\mathbb{P}$ )
2   | si  $\mathbb{P} == \Lambda$  alors
3   |   | retourner  $\Lambda$ 
4   | sinon
5   |   |  $((n, L), \mathbb{P}) \leftarrow \mathbb{P}$ 
6   |   | retourner FusionListesTriées( $L, \text{FusionTotalePile}(\mathbb{P})$ )
7   | fin

```

Sorties: une liste d'enregistrements

Exercice 86. Écrire un algorithme qui prend en entrée une liste et la trie suivant la stratégie développée dans ce paragraphe. Attention, il doit fonctionner même sur une liste qui n'a pas comme longueur une puissance de 2.

Indication: on pourra commencer par écrire un algorithme qui prenne en entrée la liste à trier et une pile, et fusionne en triant cette liste et la pile

Correction:

Entrées:

- une liste d'enregistrements L ;
- une pile de couples d'entiers et de listes d'enregistrement \mathbb{P} .

```

1 TriFusionListeAvec( $L, \mathbb{P}$ )
2   | si  $L == \Lambda$  alors
3   |   | retourner FusionTotalePile( $\mathbb{P}$ )
4   | sinon
5   |   |  $(e, L) \leftarrow L$ 
6   |   |  $\mathbb{P} \leftarrow \text{AjoutFusionPile}(\mathbb{P}, (1, [e]))$ 
7   |   | retourner TriFusionListeAvec( $L, \mathbb{P}$ )
8   | fin

```

Sorties: une liste d'enregistrement triée



Exercice 87. Évaluer la complexité en temps de cet algorithme.

Correction: Chaque élément va être ajouté sur la pile dans une liste de longueur 1, puis il sera comparé quand cette liste va être fusionnée en une liste de longueur 2,... Chaque élément subit donc $O \log(n)$ fusions, qui sont toutes linéaires.

La complexité globale est bien $O(n \log(n))$, ceci est une implémentation du tri fusion.



Partir de briques plus longues

On a produit une version fidèle du tri fusion pour des listes chaînées. Néanmoins, cela rend légèrement artificiel le fait de découper les listes jusqu'à des listes de taille 1 : on pourrait bien rajouter n'importe quelle liste déjà triée sur la pile et la fusionner avec toute liste de longueur inférieure ou égale.

Exercice 88. Écrire un algorithme **PréfixeTrié** prenant en entrée une liste L et renvoyant deux listes T et Q telles que la concaténation de T et de Q soit égale à L (T est un *préfixe* de L) et que T est triée, et le plus grand préfixe trié de L.

Indication: on pourra supposer qu'on dispose d'une fonction **Inverse** qui inverse une liste.

Correction:

Entrées: deux listes d'enregistrements L et '.

```

1 PréfixeTriéAvec(L,L')
2   si L == Λ alors
3     retourner (Inverse(L'), Λ)
4   sinon
5     (e,L) ← L
6     si L' == Λ alors
7       retourner PréfixeTriéAvec(L, [e])
8     sinon
9     (e',L') ← L'
10    si clef(e) ≥ clef(e') alors
11      retourner PréfixeTriéAvec(L, (e, (e', L')))
12    sinon
13      retourner (Inverse((e', L)), (e, L))
14    fin
15  fin
16 fin
```

Sorties: deux listes d'enregistrements

Entrées: une liste d'enregistrements L.

```

1 PréfixeTrié(L)
2   retourner PréfixeTriéAvec(L, Λ)
```

Sorties: deux listes d'enregistrements



Exercice 89. Exécuter cet algorithme sur la liste [1, 3, 2].

Exercice 90. Quelle est sa complexité en temps ?

Correction: Cet algorithme est linéaire : il parcourt deux fois le préfixe. ☺

Exercice 91. Adapter l'algorithme de la section précédente de manière à ce qu'au lieu de rajouter sur la pile des listes de longueur 1, il rajoute les sous-listes déjà dans l'ordre.

Indication: on pourra utiliser la fonction calculant la longueur d'une liste

Correction: On ne va ré-écrire que l'algorithme auxiliaire :

Entrées:

- une liste d'enregistrements L ;
- une pile de couples d'entiers et de listes d'enregistrement P.

```

1 TriFusionListeAvec(L, P)
2   | si L == Λ alors
3   |   | retourner FusionTotalePile(P)
4   | sinon
5   |   | (T, Q) ← PréfixeTrié(L)
6   |   | n ← Longueur(T)
7   |   | P ← AjoutFusionPile(P, (n, [T]))
8   |   | retourner TriFusionListeAvec(Q, P)
9   | fin
```

Sorties: une liste d'enregistrement triée



Exercice 92. Quelle est la complexité en temps dans le meilleur cas d'un tel algorithme ?

Correction: Si on l'exécute sur une liste déjà triée, on va récupérer la liste entière comme plus grand préfixe, donc on est de complexité linéaire. ☺

Exercice 93. A-t-on répondu à tous les desiderata en introduction ?

Correction: Cet algorithme est incrémental, ne recopie pas les listes et les déplace juste, et a une complexité dans le pire cas égale au minimum théorique (comme le tri fusion) et dans le meilleur cas linéaire (comme le tri par insertion).

C'est une victoire sur toute la ligne. ☺

Exercice 94. Avez-vous des idées pour faire mieux ?

Correction: Plein !

Voir (AUGER et al. 2018). ☺

3.B CHANSON SUR MON DRÔLE DE TRI

Repartons du tri fusion : c'est un tri qui applique la stratégie diviser-pour-régner en divisant le tableau que l'on souhaite trier en deux parties, triant indépendamment les deux parties, et enfin fusionnant les deux parties triées en une seule. On se dit qu'il doit y avoir d'autres manières de fusionner de manière intelligente, et qu'on peut en déduire de nouveaux tris.

Considérons le tableau suivant :

30	5	9	30	6	12
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]

Si on trie les trois premières cases du tableau, puis les trois dernières, on n'obtient pas un tableau trié et on doit donc fusionner (c'est le tri fusion).

Exercice 95. Trier les quatre premières cases du tableau, puis les quatre dernières. Qu'observez-vous ?

Donner un exemple de tableau à six cases pour lequel cette observation est fausse.

Correction: On trie les quatre premières cases :

5	9	30	30	6	12
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]

puis les quatre dernières :

5	9	6	12	30	30
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]

On voit que le tableau final n'est pas trié, mais pas loin : les deux plus grands éléments sont bien à leur place. Cette procédure peut produire des tableaux triés. C'est par exemple le cas sur :

30	5	6	30	9	12
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]

⊕

Exercice 96. On suppose qu'on a un algorithme *Tri4* qui prend en entrée un tableau à quatre cases et le renvoie trié.

En déduire un algorithme, qui, prenant en entrée un tableau à six cases, trie le tableau tout entier, en utilisant uniquement l'algorithme *Tri4*.

Correction: Comme vu plus haut sur l'exemple, en triant les quatre premières cases, puis les quatre dernières, les deux enregistrement de plus grandes clefs se sont bien retrouvées en dernière position. C'est en réalité le cas sur tous les tableaux possibles : en effet, considérons un enregistrement dont la clef est une des deux plus grandes. Soit

- il est dans une des quatre premières cases. Alors, la première passe de tri l'enverra dans la case 2 ou 3, et la seconde dans une des deux dernières cases ;
- il est dans l'une des deux dernières cases, auquel cas, la première passe de tri n'y touche pas, et la seconde le place dans une des deux dernières cases.

(ce raisonnement ne fonctionne que s'il n'y a pas plus de deux enregistrements ayant les deux plus grandes clefs. **Exercice :** l'adapter au cas général.) Donc, les deux dernières cases ne sont pas à modifier si l'on veut trier le tableau entier.

Autrement dit, il suffit de trier une nouvelle fois les quatre premières cases, d'où l'algorithme, où l'on suppose que *Tri4* prend trois arguments : le tableau à trier, ainsi que la position du début et de la fin du sous-tableau à trier, et trie sur place : ⊕

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < 6}$.

- 1 $\text{Tri6}((\text{TAB}[i])_{0 \leq i < 6})$
- 2 $\quad \text{Tri4}((\text{TAB}[i])_{0 \leq i < 6}, 0, 4)$
- 3 $\quad \text{Tri4}((\text{TAB}[i])_{0 \leq i < 6}, 2, 6)$
- 4 $\quad \text{Tri4}((\text{TAB}[i])_{0 \leq i < 6}, 0, 4)$
- 5 $\quad \text{retourner } (\text{TAB}[i])_{0 \leq i < 6}$

Sorties: un tableau

Pour en déduire un algorithme récursif, il faut qu'on décide comment généraliser le fait de prendre quatre cases pour en trier six.

La moitié plus un

Exercice 97. Écrire un algorithme récursif triant un tableau de longueur N de la manière suivante :

1. on note n le plus petit nombre entier strictement supérieur à $N/2$ (donc, si $N \equiv 12$, $n \equiv 7$ et si $N \equiv 11$, $n \equiv 6$). On va supposer que l'instruction $n \leftarrow N/2 + 1$ a cet effet¹³ ;
2. on trie d'abord les n premières cases ;
3. puis les n dernières cases ;
4. on répète les étapes 2 et 3 tant que le tableau n'est pas trié dans son entièreté.

Évidemment, pour faire un algorithme récursif, il faut aussi décider ce que l'on fait quand N est bien trop petit¹⁴ (il faut aussi décider ce que trop petit veut dire) et, par ailleurs, il faut savoir combien de fois répéter les étapes 2 et 3.

Justifier une réponse à ces deux problèmes, et écrire cet algorithme.

Correction: Quand N ne peut pas être divisé de manière satisfaisante en 2, auquel on rajoute 1, c'est-à-dire quand $N \equiv 1 + N/2$, c'est-à-dire quand $N \equiv 2$, on trie à la main (c'est-à-dire qu'on inverse les deux valeurs si elles ne sont pas dans le bon ordre, et ne fait rien sinon).

On voit qu'il y a une interface, par laquelle se fait un échange entre les deux moitiés du tableau. Il suffit donc de continuer tant que l'échange doit encore avoir lieu, c'est-à-dire tant que l'élément dans la case $N/2 - 2$ est plus grand que celui de la case $N/2 - 1$ ou que celui de la case $N/2$ est plus grand que celui de la case $N/2 + 1$. Donc: \odot

1	Entrées: un tableau $(TAB[i])_{0 \leq i < N}$.
2	TriPlusUn $((TAB[i])_{0 \leq i < N})$
3	$TriPlusUn((TAB[i])_{0 \leq i < N/2+1})$
4	$TriPlusUn((TAB[i])_{N/2-1 \leq i < N})$
5	tant que $TAB[N/2 - 2] > TAB[N/2 - 1]$ ou $TAB[N/2] > TAB[N/2 + 1]$ faire
6	$TriPlusUn((TAB[i])_{0 \leq i < N/2+1})$
7	$TriPlusUn((TAB[i])_{N/2-1 \leq i < N})$
8	fin
	retourner $(TAB[i])_{0 \leq i < N}$
	Sorties: un tableau

Exercice 98. L'exécuter sur le tableau

30	45	29	30	6	12	8	4
$TAB[0]$	$TAB[1]$	$TAB[2]$	$TAB[3]$	$TAB[4]$	$TAB[5]$	$TAB[6]$	$TAB[7]$

Exercice 99. Justifier la correction de cet algorithme (même informellement).

Correction: On a déjà fait le raisonnement plus haut: on va appeler *passe* le fait de trier les premiers éléments, puis les derniers. Au bout d'une passe, le dernier élément du tableau a bien une clef maximale, au bout de deux passes, les deux derniers, ... \odot

Exercice 100. Que pouvez-vous dire de la complexité de cet algorithme? Le raisonnement que nous avons fait pour le tri fusion est-il toujours valable?

Correction: Le raisonnement n'est pas valable: en effet, au lieu de traiter deux tableaux deux fois plus petits, on doit trier jusqu'à N tableaux deux fois plus petits. On se retrouve donc avec une complexité exponentielle. \odot

13. C'est le cas en C: $n = (N/2) + 1$; a bien ce comportement. Bah ouais.

14. Mon ami.

Exercice 101. Programmez cet algorithme en C.

On attend une fonction de prototype `int tri(size_t longueur, int tab[longueur]);` qui trie le tableau en place (c'est-à-dire en le modifiant) et retourne 0 si l'exécution s'est bien passée, -1 sinon.

Indication: plutôt que de recopier le tableau pour faire la récursion, on peut juste utiliser l'indice de la première et celui de la dernière case à trier.

Les deux tiers

Exercice 102. Écrire un algorithme récursif triant un tableau de longueur N de la manière suivante :

1. on note n le plus petit nombre entier supérieur ou égal à $N/3$ (donc, si $N \equiv 12$, $n \equiv 4$ et si $N \equiv 11$, $n \equiv 4$).
2. on trie d'abord les n premières cases ;
3. puis les n dernières cases ;
4. puis à nouveau les n premières cases.

Exercice 103. L'exécuter sur le tableau

30	45	29	30	6	12	8	4
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]	TAB[6]	TAB[7]

Exercice 104. Justifier la correction de cet algorithme (même informellement).

Exercice 105. Que pouvez-vous dire de la complexité de cet algorithme? Le raisonnement que nous avons fait pour le tri fusion est-il toujours valable?

Correction: Le raisonnement est plus proche : cette fois, on doit trier trois tableaux de tailles $2N/3$. On peut prouver que la complexité est approximativement en $O(N^{2.7})$, donc pire qu'un tri par insertion... ☺

Exercice 106. Programmez cet algorithme en C.

On attend une fonction de prototype `int tri(size_t longueur, int tab[longueur]);`.

Indication: plutôt que de recopier le tableau pour faire la récursion, on peut juste utiliser l'indice de la première et celui de la dernière case à trier.

Correction:

```

1 int deux_tiers(int n){
2     if (n%3 == 0) {
3         return 2*n/3-1;
4     } else {
5         return 2*n/3;
6     }
7 }
8
9 int un_tiers(int n){
10    return n/3;
11 }
12
13
14 int tri_long(size_t longueur, int tab[longueur], size_t debut, size_t fin){
15     if (fin - debut == 1) {
16         if (tab[debut] > tab[fin]) {
17             int swap = tab[debut];
18             tab[debut] = tab[fin];
19             tab[fin] = swap;
20         }
21     }
22 }
```

```
19     tab[fin] = swap;
20     print_tableau(longueur,tab);
21 }
22 } else {
23     tri_long(longueur,tab,debut,debut+deux_tiers(fin-debut+1));
24     tri_long(longueur,tab,debut+(fin-debut+1)/3,fin);
25     tri_long(longueur,tab,debut,debut+deux_tiers(fin-debut+1));
26 }
27
28     return EXIT_SUCCESS;
29 }
30
31 int tri(size_t longueur, int tab[longueur]){
32     if (longueur < 2) {
33         return EXIT_FAILURE;
34     } else {
35         return tri_long(longueur,tab,0,longueur-1);
36     }
37 }
```

☺

Exercice 107. Que concluez-vous de tout ce DM?

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.

4.1	Gloutonnerie	116
	Sac à dos fractionné et sac à dos	116
	Coloriage de graphes	118
	Rendu de monnaie	124
4.2	Backtracking	127
	Problème des dames	127
	Satisfiabilité	134
	NP -complétude	139
	Satisfiabilité des clauses de Horn	141
4.3	Le voyageur de commerce	142
	Programmation dynamique	142
	Approximations	143
4.4	Problème des couplages	143
	Couplages stables	143
	L'algorithme de Gale et Shapley	145

POUR l'instant, on a vu des problèmes qui avaient une solution à peu près unique¹. 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-réfrigérateurs), 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.

4.1 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.

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 monétaire². 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 acquise³. Par exemple, supposons que l'on a les objets suivants, avec leur volume, leur prix et

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).

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.

leur prix volumique :

	volume	prix	prix volumique
lingot d'or	6 L	5,4 M€	895 k€ L ⁻¹
lingot d'argent	4 L	28 k€	7 k€ L ⁻¹
lingot de palladium	8 L	6,0 M€	751 k€ L ⁻¹
lingot de platine	5 L	3,5 M€	692 k€ L ⁻¹
lingot de platine	5 L	3,5 M€	692 k€ L ⁻¹

En testant les différentes possibilités, 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 dos⁴. 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 $O(n \log(n))$), 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.

On peut écrire cet algorithme, en supposant l'inventaire trié par prix décroissant :

Entrées :

- un entier C ;
- un inventaire \mathcal{I} , trié par prix décroissant.

```

1 SacÀDos(C, I)
2   L ← Λ
3   tant que I ≠ Λ faire
4     e :: I) ← I
5     si volume(e) ≤ C alors
6       L ← e :: L
7       C ← C - volume(e)
8     fin
9   fin
10  retourner L

```

Sorties : un inventaire

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**.

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.

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



FIGURE 4.1 – Les pays membres de l’Union Européenne

Exercice 108. 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 4.1, où certains pays ne partageant pas de frontières ont des teintes quasi-identiques⁵ (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.

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

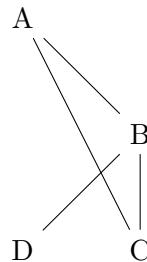
5. Ici, le choix a été fait que chaque pays soit colorié différemment; cela ne change rien au problème, comme on verra.

maritimes⁶, les zones disputées, l'outre-mer⁷... 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 simple⁸, 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 4.1 par le graphe de la Figure 4.2.

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/Grèce 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;
- 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 $0, \dots, n$, on peut représenter un graphe par un tableau à deux dimensions $(\text{TAB}[i][j])_{0 \leq i < n, 0 \leq j < n}$ tel que

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

s'il y a une arête entre le sommet i et le sommet j , et 0 sinon.

Ainsi, le graphe G de la page 119 serait représenté par un tableau de dimension 4 associant à chaque numéro le sommet de ce numéro : $[[A; B; C; D]]$ ainsi qu'un second tableau, de dimension 4×4 , que l'on va noter $(G[i][j])_{0 \leq i < 4, 0 \leq j < 4}$ et que l'on peut représenter par :

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*.

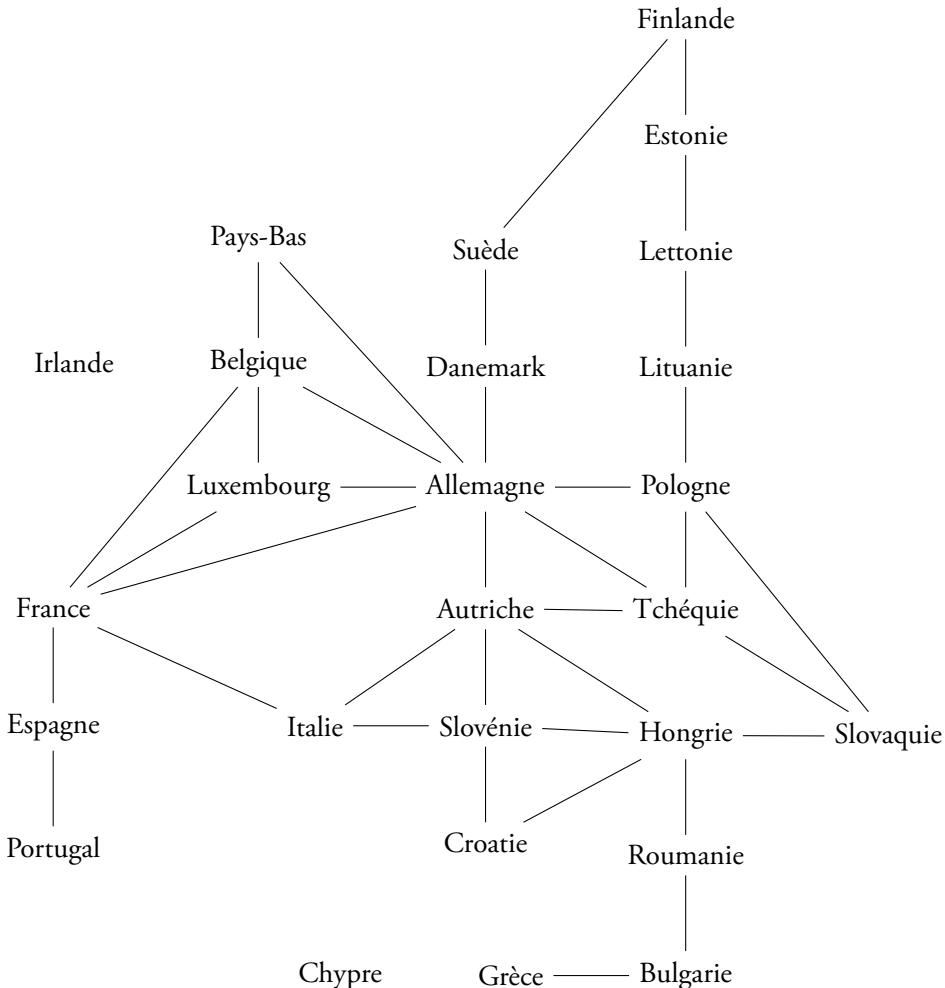


FIGURE 4.2 – Le graphe des frontières terrestres en Europe des pays membres de l'Union Européenne

	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 119 serait représenté par la liste

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

Exercice 109. Pour ces deux représentations, représentez le graphe des voisins de l'Union Européenne.

Exercice 110. 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 111. 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 droit⁹, c'est-à-dire que pour un tel tableau $(G[i][j])_{0 \leq i < N, 0 \leq j < N}$, pour chaque entiers $0 \leq i < N, 0 \leq j < N$,

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

Remarque 4. En C, le graphe G de la page 119 peut être défini avec un tableau à deux dimensions :

```

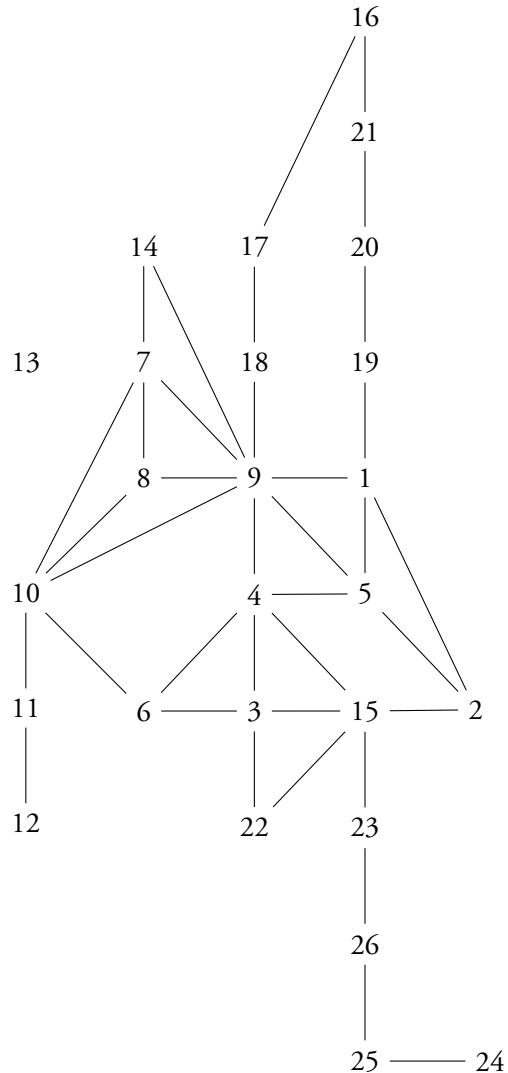
1 int g[4][4] = {
2     [0][0] = 0,
3     [0][1] = 1,
4     [0][2] = 1,
5     [0][3] = 0,
6     [0][0] = 0,
7     [1][1] = 1,
8     [1][2] = 1,
9     [1][3] = 0,
10    [1][0] = 0,
11    [2][1] = 1,
12    [2][2] = 1,
13    [2][3] = 0,
14    [3][0] = 0,
15    [3][1] = 1,
16    [3][2] = 1,
17    [3][3] = 0,
18 };
```

Nous pouvons revenir à notre problème : pour faire mieux que la solution triviale¹⁰, 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.

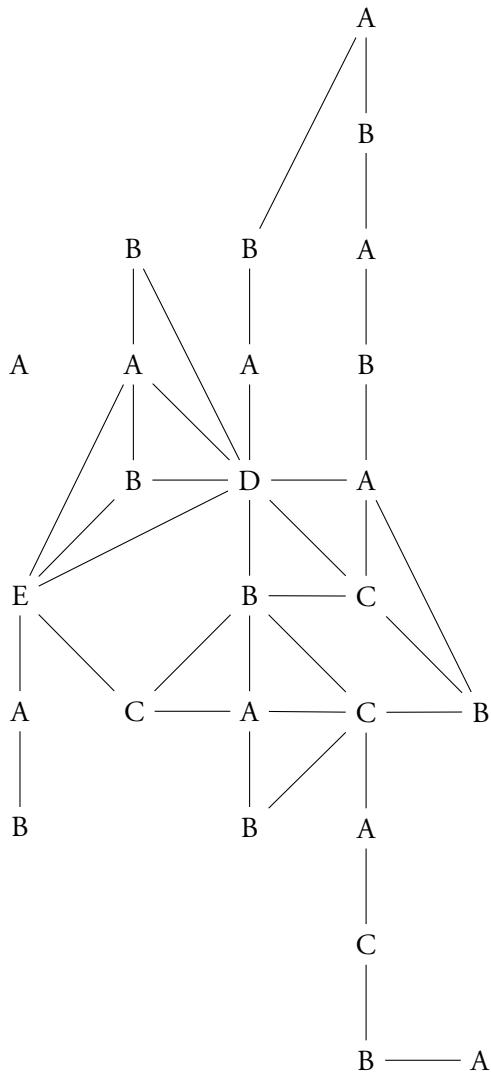
Supposons que les pays et les couleurs sont numérotées (pour distinguer, on numérotera les pays avec des nombres et les couleurs avec des lettres) : ainsi, plutôt que de dire « la Slovaquie est coloriée en rouge », on dira (par exemple) « 8 est D ». Supposons qu'on a ordonné les pays dans l'ordre Pologne,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

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 noeud. On n'a rien fait, la solution ne dépend pas du 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:

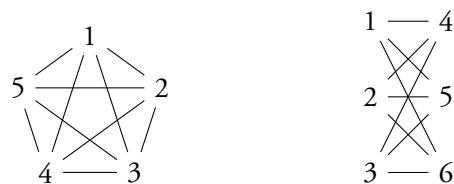


On a utilisé cinq couleurs dans ce coloriage. Ce n'est pas optimal (on peut en trouver un avec quatre couleurs), et en fait, le coloriage dépend de l'ordre que l'on a choisi pour les sommets.

Exercice 112. Trouver un ordre pour les sommets coloriant ce graphe avec seulement quatre couleurs.

Exercice 113. Écrire l'algorithme pour chacune des deux représentations des graphes.

Remarque 5 (théorème des quatre couleurs). On dit qu'un graphe est *planaire* s'il peut être dessiné sur un plan sans qu'aucune arête 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 $K_{3,3}$ 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-planaire (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 colorées 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 existe¹¹.

Exercice 114. Représenter $K_{3,3}$ et K_5 comme des tableaux et comme des listes.

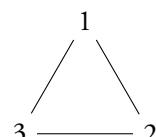
Exercice 115. On cherche à construire un emploi du temps pour une formation. Certains cours sont obligatoires, d'autres non. On attend la fin des inscriptions pédagogiques pour procéder à l'établissement de l'emploi du temps, ainsi, on connaît toutes les incompatibilités entre cours.

- 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 possibilité 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 116. Écrire comme une formule en logique propositionnelle que le graphe



n'est pas coloriable avec deux couleurs.

Rendu de monnaie

Exercice 117. 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.

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

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

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

rendre, on passe à celle d'après,... On suppose donc, dans l'Algorithm 28 que la liste des pièces possibles est triée par ordre décroissant. \odot

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

Algorithm 28: Rendu de monnaie — Algorithme glouton

Exercice 118. 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 :

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

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 119. 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 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. ⊕

4.2 BACKTRACKING

Pour l'instant, on a étudié des algorithmes gloutons, qui font le meilleur choix possible *localement* et où l'on espère que ce choix soit le meilleur *globalement*. Cependant, on a bien vu que ce n'était pas toujours le cas. Parfois, même, faire un choix nous amène dans une situation problématique, dans laquelle il n'est plus possible de continuer.

Prenons, pour fixer les idées, le cas du jeu d'échec (mais on pourrait prendre n'importe quel jeu à informations parfaite). Quand on réfléchit à quel coup jouer, on simule mentalement un coup, puis comment l'adversaire y répond, et ainsi de suite. Parfois, on se convainc que jouer un certain coup nous fait perdre sûrement : à ce moment-là, on veut revenir en arrière dans la simulation, pour jouer un autre coup que celui déjà considéré.

Ce mécanisme (faire un choix, en étudier les conséquences et, s'il était mauvais, revenir en arrière) est assez courant en algorithmique, et se nomme *backtracking*. On va l'étudier et l'appliquer.

Problème des dames

Illustrons ce mécanisme sur le *problème des dames*. On veut placer sur un damier de taille $n \times n$, n dames, et on veut qu'elles le fasse sans conflit (au sens du jeu d'échecs), c'est-à-dire qu'il n'y ait jamais deux dames sur la même ligne, la même colonne, ou la même diagonale¹². Peut-être a-t-on une solution pour réfléchir globalement au problème et trouver une solution : par exemple, on peut facilement montrer qu'il y a une unique solution pour $n = 1$; et qu'il n'y en a pas pour $n = 2$ (en effet, deux cases sont soit sur la même ligne, soit sur la même colonne, soit sur la même diagonale). Si on n'en trouve pas, on peut utiliser un algorithme de type backtracking : pour chaque dame, on va faire un choix et la positionner dans une position ne créant pas de conflit. Si on trouve une telle position, on continue, sinon, on retourne en arrière, au dernier choix qu'on a fait et on en fait un autre.

Par exemple, prenons le cas de du problème des quatre dames : on veut placer 4 dames sans qu'elles se menacent sur un damier 4×4 . Comme il y aura une dame par ligne, on peut supposer que faire un choix consiste à choisir la colonne où placer la dame de chaque ligne.

On peut placer la dame de la première ligne n'importe où.

- On commence par faire le choix de placer dans la première colonne. On ne peut donc placer la dame de la deuxième ligne, ni sur la première colonne (car il y a déjà une dame sur cette colonne) ni sur la deuxième (car il y a déjà une dame sur cette diagonale). On peut donc la placer sur la troisième ou sur la quatrième colonne.

- On fait le choix de la placer sur la troisième colonne. La dame de la troisième ligne ne peut pas être sur la première colonne (car en conflit avec celle de la ligne 1), ni sur la deuxième (car en conflit diagonal avec celle de la ligne 2), ni sur la troisième (ligne 2), ni sur la quatrième (ligne 2).

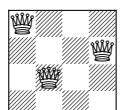
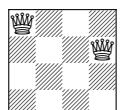
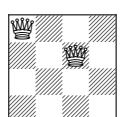
Ainsi, ce choix était impossible, et on revient en arrière.

- On fait le choix de placer la deuxième dame sur la quatrième colonne. La dame de la troisième ligne ne peut être ni sur la première, ni sur la quatrième colonne, car il y a des dames dans ces colonnes-là; et ne peut pas non plus être sur la troisième (par les deux diagonales).

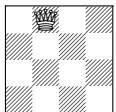
- On fait donc le choix de placer la troisième dame sur la deuxième colonne. La seule colonne sans dame est la troisième, mais elle est menacée diagonalement par celle de la troisième ligne.

Ainsi, ce choix était impossible, et on revient en arrière.

Ainsi, ce choix était impossible, et on revient en arrière.



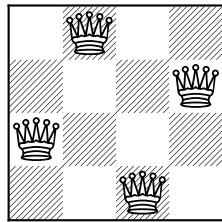
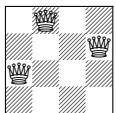
12. Et donc, selon les règles de mouvement du jeu d'échec, aucune ne pourrait se prendre mutuellement (en ignorant les couleurs...).



Ainsi, ce choix était impossible, et on revient en arrière.

- On place la première dame dans la deuxième colonne. La seconde dame est menacée sur les premières et troisièmes colonnes diagonalement. De plus, il y a déjà une dame sur la deuxième colonne.
- On fait le choix de placer la dame de la deuxième ligne sur la quatrième colonne. La troisième dame est menacée sur les deuxièmes, troisièmes, quatrièmes colonnes.
- On fait le choix de placer la troisième dame sur la première colonne.
La dernière dame ne peut être que sur la troisième colonne.
- On fait le choix de placer la quatrième dame sur la troisième colonne.

On a donc trouvé la solution :



Exercice 120. Continuer l'algorithme pour trouver une deuxième solution.

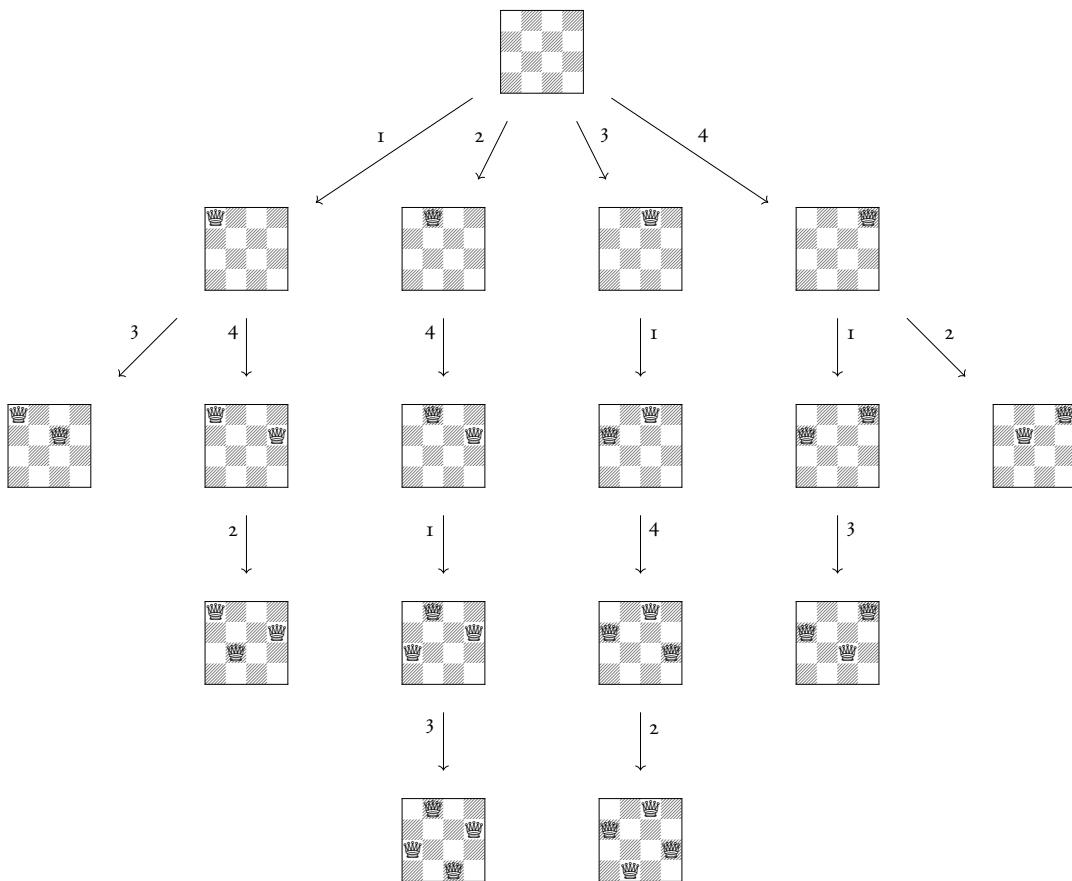
Sont-ce les seules?

On peut représenter les états successifs de l'algorithme comme un arbre :

- à la racine, on place l'état avant avoir fait le moindre choix;
- puis, pour chaque noeud, ses enfants sont les différents choix possibles.

On voit que les enfants immédiats de la racine placent la première dame, les petits-enfants la deuxième,...

Une solution est donc un descendant de niveau 4 de la racine.



Ainsi, on trouve les deux solutions en parcourant cet arbre en commençant par explorer jusqu'en bas la branche la plus à gauche, puis une à une chaque voisine. On appelle ce type de parcours un *parcours en profondeur*. On voit qu'on a pas du tout eu besoin d'énumérer toutes les manières possibles de placer quatre dames (il y en a 4^4 , c'est-à-dire 256) car on coupe toutes les branches impossibles aussi tôt qu'on peut. On voit aussi qu'on n'a pas procédé à une réflexion globale sur le problème (par exemple, on aurait pu se dire qu'une dame dans les grandes diagonales bloque trop de cases et donc rend impossible leur utilisation, et en déduire les deux solutions: j'insiste sur ce fait, quand on backtrack — et c'était aussi le cas pour des algorithmes gloutons —, on raisonne localement).

On voit aussi qu'on peut s'abstraire du problème et, de la même manière que tous les algorithmes gloutons sont à un certain sens les mêmes, on peut donner une forme générique de l'algorithme par *backtrack*. Pour cela, on doit supposer qu'on a:

- un ensemble fini de *niveaux* (ici, les lignes dans le damier);
- pour chaque niveau, un ensemble fini de choix (ici, les colonnes).

Ces deux éléments peuvent se dire autrement: on demande que les solutions potentielles s'écrivent comme des feuilles d'un arbre où les branchements sont des choix. Les feuilles et les nœuds internes de l'arbre nous donnent des configurations du problème.

- On a une propriété sur toutes les configurations, qui a trois caractéristiques:
 - parmi les feuilles de niveau maximal, elle est vraie exactement des solutions;
 - si la propriété est vraie d'une configuration, elle est vraie de tous les nœuds entre la racine et cette configuration;
 - si la propriété est vraie d'une configuration, il est facile de vérifier qu'elle est vraie de toutes celles que l'on peut atteindre en faisant un choix.

Dans l'exemple des dames, la propriété est « aucune dame ne menace d'autre dame ». On vérifie qu'elle a les trois caractéristiques citées :

- une solution est exactement une configuration maximale (toutes les dames sont placées) où aucune dame ne menace d'autre dame ;
- si dans une configuration, aucune dame n'en menace une autre, c'est aussi le cas si on retire une dame ;
- si dans une configuration, aucune dame ne menace d'autres dames, quand on fait le choix de placer une nouvelle dame, il suffit de vérifier que la nouvelle n'est pas menacée : c'est linéaire en le nombre de dames déjà placées.

Si ces éléments sont réunis, alors on peut utiliser du *backtrack*. On peut alors écrire l'Algorithme 29 qui s'utilise en appelant `SolutionBacktrack(Λ , 0)`

Entrées:

- la liste des choix déjà faits L ;
- le niveau courant n .

```

1 SolutionBacktrack(L, n)
2   si  $n$  est un niveau valide alors
3     pour chaque choix  $c$  de niveau  $n$  faire
4       si  $(c, L)$  satisfait la propriété alors
5         si SolutionBacktrack( $(c, L), n + 1 \neq \Lambda$  alors
6           retourner SolutionBacktrack( $(c, L), n + 1$ )
7         fin
8       fin
9     fin
10    retourner  $\Lambda$ 
11  sinon
12    retourner  $L$ 
13  fin
```

Sorties: une liste de choix

Algorithme 29: Algorithme générique de recherche d'une solution par *backtrack*

Exercice 121. En remarquant que deux dames, une en position (i, j) , l'autre en position (k, ℓ) , se menacent si et seulement si: $i == k$, $j == \ell$ ou $|i - k| == |j - \ell|$, adapter l'Algorithme 29 au problème des n dames.

Notation: $|\cdot|$ désigne la valeur absolue, c'est-à-dire la valeur d'un nombre en ignorant son signe. Par exemple, $|-3| == |3| == 3$.

Correction: Commençons d'abord par justifier la remarque : deux dames se menacent si :

- elles sont sur la même ligne, c'est-à-dire $i == k$;
- elles sont sur la même colonne, c'est-à-dire $j == \ell$;
- elles sont sur la même diagonale. Dans ce cas, $k - i$ désigne le nombre de case de décalage horizontal entre la première et la deuxième dame; et $|k - i|$ désigne le nombre de cases (sans tenir compte du sens) entre les deux. Donc deux dames sont sur la même diagonale si et seulement elles ont le même nombre de cases de décalage horizontalement et verticalement, c'est-à-dire :

$$|k - i| == |j - \ell|.$$

Ici, les niveaux sont les lignes, et les choix les colonnes. La première question à se poser est de savoir comment sont représentés les choix : on veut une liste des choix passés, on peut donc placer dans une liste uniquement les numéros des colonnes. Ainsi, la liste des choix amenant à la solution qu'on a trouvé sera :

$$(2, (0, (3, (1, \Lambda))))$$

car on a d'abord placé la dame de la ligne 0 dans la colonne 1 ; puis celle de la ligne 1 dans la colonne 3 ; celle de la ligne 2 dans la colonne 0 ; celle de la ligne 3 dans la colonne 2.

On va commencer par écrire un algorithme auxiliaire vérifiant qu'il est possible de placer une dame à une certaine colonne : autrement dit, au niveau n **PositionValide**(n , L, c) vaudra **VRAI** si placer la dame dans la colonne c est compatible avec la liste de choix L, et **FAUX** sinon.

Entrées:

- un entier n — la ligne courante ;
- une liste d'entiers L — la liste des choix déjà faits ;
- un entier c — la colonne choisie.

```

1 PositionValide( $n$ , L, c)
2    $m \leftarrow n$  tant que L  $\neq 0$  faire
3      $(e, L) \leftarrow L$ 
4      $m \leftarrow m - 1$ 
5     si  $e == c$  ou  $|e - c| == |n - m|$  alors
6       retourner FAUX
7     fin
8   fin
9   retourner VRAI

```

Sorties: un booléen

Maintenant, il ne nous reste plus qu'à spécialiser l'Algorithme 29 au cas des dames.

Entrées:

- un entier k ;
- une liste d'entiers L ;
- un entier n .

```

1 ProblèmeDesDames( $k$ , L, n)
2   si  $0 \leq n < k$  alors
3     pour  $c$  allant de 0 à  $k$  faire
4       si PositionValide( $n$ , L, c) == VRAI alors
5         si ProblèmeDesDames(L, c, n)  $\neq \Lambda$  alors
6           retourner ProblèmeDesDames(L, c, n)
7         fin
8       fin
9     fin
10    retourner  $\Lambda$ 
11  sinon
12    retourner L
13  fin

```

Sorties: une liste d'entiers

©

Exercice 122. Adaptez l'Algorithme 29 au cas où on veut chercher toutes les solutions et pas juste une seule.

Exercice 123. On a donc décidé que les niveaux étaient les lignes, et les choix les colonnes. Cela nous a fait ordonner les différents niveaux : d'abord la ligne tout en haut, puis celle juste en dessous,...

C'était un ordre au fond assez arbitraire. Modifiez l'ordre des lignes et voyez si cela amène à visiter plus ou moins de configurations.

De même en ordonnant les choix (c'est-à-dire les colonnes) différemment.

Exercice 124. Trouver une structure de données pertinente pour représenter le damier de manière à :

- trouver facilement si une case est menacée ;
- revenir facilement en arrière si on retire une dame.

Correction : Dans la version qu'on a donné précédemment, on a représenté les choix qu'on avait déjà faits par une liste, ce qui nous forçait à la reparcourir intégralement, et recalculer les incompatibilités à chaque fois qu'on voulait rajouter une autre dame.

On peut se dire qu'on va représenter les choix déjà faits autrement, par exemple un tableau. On peut représenter le damier avec un tableau à deux dimensions, en mettant un 0 dans chaque case ne contenant pas de dame, en un 1 dans chaque case en contenant une (**exercice** : représenter sous cette forme le damier solution qu'on a trouvé en backtrackant à la main page 128). À chaque étape de remplissage, on va donc rajouter des 1.

Cette solution permet un peu plus simplement de calculer si une position est licite (il suffit de lancer une boucle dans sa ligne, colonne et ses deux diagonales), mais nécessite encore de faire les calculs (**exercice** : donner l'algorithme trouvant une solution pour cette représentation).

On peut aller un peu plus loin et calculer les positions rendues impossibles par chaque ajout de dame, en les notant différemment pour pouvoir les retirer facilement. Par exemple, on peut stocker dans chaque case un couple constitué d'un signe (+ ou -) et d'un entier, le signe indiquant si la case peut être occupée, et, si la case ne peut pas être occupée, l'entier indique quelle est la plus petite dame rendant cela impossible.

Récapitulons : on va stocker dans chaque case d'un tableau $N \times N$ deux informations :

- un signe + ou - indiquant si la case est licite ;
- un entier, dont le sens varie selon le signe : si la case est licite, l'entier vaut 0 si la case est vide, est $i + 1$ si la dame de la i -ème ligne est dans cette case ; et si la case est illicite, l'entier vaut 0 si la case n'est pas utilisable inconditionnellement, sans aucune contrainte, et vaut $i + 1$ si la dame de la ligne i est la plus petite dame qui bloque cette case.

Commençons par écrire des algorithmes ajoutant et supprimant une dame.

ajout quand on ajoute une dame, il faut non seulement changer la case de la dame, mais aussi mettre à jour toute sa ligne, sa colonne et ses deux diagonales, pour indiquer que ce ne sont plus des positions licites à cause de cette dame. On va écrire un premier algorithme `RendIllicite` qui rend illicite une case licite, et ne modifie pas une case déjà illicite.

Entrées :

- un couple d'un signe et un entier ;
- un entier.

```

1 RendIllicite((σ, i), j)
2   | si σ == - alors
3   |   | retourner (σ, i)
4   | sinon
5   |   | retourner (-, j)
6   fin

```

Sorties : un couple d'un signe et d'un entier

Ensuite, on va écrire un algorithme `AjoutDame` qui prend en entrée un tableau $N \times N$ et deux entiers i et j , et place la dame de la i -ème ligne sur la j -ème colonne. Pour cela, on va parcourir la ligne, colonne, et les deux diagonales de la dame en question.

Cet algorithme est en $O(n)$: on doit agir sur les deux colonnes et les deux diagonales.

suppression inversement, pour supprimer une dame, on veut rendre licite uniquement les positions que cette dame avait bloquée, et pas celles qui étaient bloquées pour d'autres raisons.

Pour supprimer une dame sur la ligne i , il suffit de tester chaque case du damier, et rendre licite toutes celles qui étaient illicites à cause de la dame i (c'est-à-dire qui étaient de la forme $(-, i + 1)$) ; enfin, on rend la case de la dame qu'on vient de supprimer illicite, causée par la dame d'avant.

Cet algorithme est en $O(n^2)$: on agit sur le plateau entier.

Avec ces deux algorithmes écrits, l'adaptation de l'Algorithme 29 est immédiate : pour chaque ligne, on parcourt chaque colonne, si la position est licite (ce qui est explicite !) on ajoute une dame, et on continue sur la ligne d'après. Si aucune colonne n'est licite pour une ligne donnée, on supprime la dernière dame, et on continue. ☺

Exercice 125. Résoudre le problème du sac-à-dos en backtrackant.

Entrées:

- un tableau $(\text{TAB}[i][j])_{0 \leq i, j < N}$;
- un entier $0 \leq i < N$;
- un entier $0 \leq j < N$.

```

1 AjoutDame((TAB[i][j])0 ≤ i, j < N, i, j)
2   | (σ, k) ← TAB[i][j]
3   | si (σ, k) == (+, 0) alors
4   |   pour ℓ allant de 0 à N faire
5   |     TAB[i][ℓ] ← RendIllicite(TAB[i][ℓ], i + 1)
6   |     TAB[ℓ][j] ← RendIllicite(TAB[ℓ][j], i + 1)
7   |     si 0 ≤ i + ℓ < N et 0 ≤ j + ℓ < N alors
8   |       TAB[i + ℓ][j + ℓ] ← RendIllicite(TAB[i + ℓ][j + ℓ], i + 1)
9   |     fin
10  |     si 0 ≤ i - ℓ < N et 0 ≤ j + ℓ < N alors
11  |       TAB[i - ℓ][j + ℓ] ← RendIllicite(TAB[i - ℓ][j + ℓ], i + 1)
12  |     fin
13  |     si 0 ≤ i + ℓ < N et 0 ≤ j - ℓ < N alors
14  |       TAB[i + ℓ][j - ℓ] ← RendIllicite(TAB[i + ℓ][j - ℓ], i + 1)
15  |     fin
16  |     si 0 ≤ i + ℓ < N et 0 ≤ j - ℓ < N alors
17  |       TAB[i - ℓ][j - ℓ] ← RendIllicite(TAB[i - ℓ][j - ℓ], i + 1)
18  |     fin
19  |   fin
20  |   TAB[i][j] ← (+, i)
21  | sinon
22  |   | retourner erreur
23  | fin

```

Sorties: un tableau $N \times N$ ou une erreur

Entrées:

- un couple d'un signe et un entier;
- un entier.

```

1 RendLicite((σ, i), j)
2   | si σ == - et i == j alors
3   |   | retourner (+, 0)
4   | sinon
5   |   | retourner (σ, i)
6   | fin

```

Sorties: un couple d'un signe et d'un entier

Entrées:

- un tableau $(\text{TAB}[i][j])_{0 \leq i, j < N}$;
- un entier $0 \leq i < N$;
- un entier $0 \leq j < N$.

```

1 SuppressionDame((TAB[i][j])0 \leq i, j < N, i, j)
2   | (σ, k) ← TAB[i][j]
3   | si (σ, k) == (+, i) alors
4   |   | pour ℓ allant de 0 à N faire
5   |   |   | pour m allant de 0 à N faire
6   |   |   |   | TAB[ℓ][m] ← RendLicite(TAB[ℓ][m], i + 1)
7   |   |   | fin
8   |   | fin
9   |   | TAB[i][j] ← (-, i)
10  | sinon
11  |   | retourner erreur
12  | fin

```

Sorties: un tableau $N \times N$ ou une erreur

Correction: On trouve une solution. On modifie le dernier choix, si la solution est pire, on arrête, sinon, on continue. ☺

Satisfiabilité

Considérons une formule de logique propositionnelle (on précisera plus tard les définitions), c'est-à-dire une formule construite à partir de variables (pouvant valoir v ou f), de la **négation**, la **conjonction** et la **disjonction**. Par exemple, si a, b, c sont des variables,

$$\varphi := (a \vee b) \wedge (\neg c)$$

est une telle formule. Selon les valeurs que prennent les variables a, b et c , cette formule prend différentes valeurs, ce qu'on peut résumer par une table de vérité:

a	b	c	φ
F	F	F	F
F	F	V	F
F	V	F	V
F	V	V	F
V	F	F	V
V	F	V	F
V	V	F	V
V	V	V	F

Étant donnée une formule propositionnelle, on peut se poser un certain nombre de questions :

- est-elle *tautologie*, c'est-à-dire que quelle que soit la valeur des variables, la formule vaut tout le temps v?

Par exemple, le principe du tiers-exclus $a \vee \neg a$ est une tautologie : elle est vraie pour des raisons formelles, dérivant uniquement du sens de la négation et de la disjonction, et pas de la valeur de a .

- est-elle *contradictoire*, c'est-à-dire que quelle que soit la valeur des variables, la formule vaut tout le temps f?

Par exemple, le principe $a \wedge \neg a$ est une contradiction : elle est vraie pour des raisons formelles, dérivant uniquement du sens de la négation et de la conjonction, et pas de la valeur de a .

- est-elle *satisfiable*, c'est-à-dire qu'il existe une valeur des variables telle que la formule soit vraie.
- La formule $a \wedge \neg b$ est satisfiable : si $a \equiv \text{v}$ et $b \equiv \text{f}$, la formule est satisfaite.

Ce troisième problème a une importance particulière en algorithmique, et rentre dans la catégorie des problèmes qui nous intéressent dans ce chapitre : en effet, déterminer qu'une formule est satisfiable revient à construire une solution à un certain problème. On veut trouver comment assigner les différentes variables de manière à ce que les contraintes données par la formule soient satisfaites.

Exemple 13. Il est possible de satisfaire les formules :

- x_1 ou non- x_2 ;
- x_2 ou x_3 ;
- non- x_1 ou non- x_3 ;
- non- x_1 ou non- x_2 ou x_3

par $x_1 \equiv \text{f}$, $x_2 \equiv \text{f}$, $x_3 \equiv \text{v}$. On peut d'ailleurs ré-écrire ces formules en une seule :

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

Exemple 14. Il n'est pas possible de satisfaire les formules :

- x_1 ;
- non- x_1 .

On peut aussi les ré-écrire en une seule formule :

$$x_1 \wedge \neg x_1$$

De plus, énormément de problèmes peuvent s'écrire comme un problème de satisfiabilité. On a déjà vu un exemple : le [coloriage de graphes](#). On peut exprimer le fait qu'un graphe soit colorable avec n couleurs par un problème de satisfiabilité. Prenons un graphe particulier, avec d sommets. Pour chaque sommet v , on introduit n variables

$$v_1, v_2, \dots, v_n$$

signifiant chacune que le sommet v a la couleur i . On veut donc satisfaire, pour chaque sommet v , la formule

$$v_1 \vee v_2 \vee \dots \vee v_n$$

signifiant que chaque sommet a une couleur, ainsi que, pour chaque arête entre v et u , et chaque couleur i la formule

$$\neg v_i \vee \neg u_i$$

signifiant que les deux côtés de l'arête n'ont pas en même temps la même couleur.

Le graphe est colorable si et seulement si l'ensemble de formules ainsi induit est satisfiable.

Exemple 15. Considérons le graphe G de la page 119. Intéressons-nous au nombre de couleur minimal pour le colorier.

pour une couleur comme il y a quatre sommets et une seule couleur, on introduit quatre variables :

$$A_1, B_1, C_1, D_1$$

La formule que l'on veut satisfaire est donc :

$$\begin{aligned} & A_1 \wedge B_1 \wedge C_1 \wedge D_1 \\ & \wedge (\neg A_1 \vee \neg B_1) \\ & \wedge (\neg A_1 \vee \neg C_1) \\ & \wedge (\neg B_1 \vee \neg C_1) \\ & \wedge (\neg C_1 \vee \neg D_1) \end{aligned}$$

Cette formule n'est pas satisfiable (**exercice**: le prouver).

pour deux couleurs comme il y a quatre sommets et deux couleurs, on introduit huit variables:

$$A_1, B_1, C_1, D_1, A_2, B_2, C_2, D_2$$

La formule que l'on veut satisfaire est donc:

$$\begin{aligned} & (A_1 \vee A_2) \wedge (B_1 \vee B_2) \wedge (C_1 \vee C_2) \wedge (D_1 \vee D_2) \\ & \wedge (\neg A_1 \vee \neg B_1) \wedge (\neg A_2 \vee \neg B_2) \\ & \wedge (\neg A_1 \vee \neg C_1) \wedge (\neg A_2 \vee \neg C_2) \\ & \wedge (\neg B_1 \vee \neg C_1) \wedge (\neg B_2 \vee \neg C_2) \\ & \wedge (\neg C_1 \vee \neg D_1) \wedge (\neg C_2 \vee \neg D_2) \end{aligned}$$

Cette formule n'est pas satisfiable (**exercice**: le prouver).

pour trois couleurs comme il y a quatre sommets et trois couleurs, on introduit douze variables:

$$A_1, B_1, C_1, D_1, A_2, B_2, C_2, D_2, A_3, B_3, C_3, D_3$$

La formule que l'on veut satisfaire est donc:

$$\begin{aligned} & (A_1 \vee A_2 \vee A_3) \wedge (B_1 \vee B_2 \vee B_3) \wedge (C_1 \vee C_2 \vee C_3) \wedge (D_1 \vee D_2 \vee D_3) \\ & \wedge (\neg A_1 \vee \neg B_1) \wedge (\neg A_2 \vee \neg B_2) \wedge (\neg A_3 \vee \neg B_3) \\ & \wedge (\neg A_1 \vee \neg C_1) \wedge (\neg A_2 \vee \neg C_2) \wedge (\neg A_3 \vee \neg C_3) \\ & \wedge (\neg B_1 \vee \neg C_1) \wedge (\neg B_2 \vee \neg C_2) \wedge (\neg B_3 \vee \neg C_3) \\ & \wedge (\neg C_1 \vee \neg D_1) \wedge (\neg C_2 \vee \neg D_2) \wedge (\neg C_3 \vee \neg D_3) \end{aligned}$$

Cette formule est satisfiable, par exemple avec

$$A_1 \equiv B_2 \equiv C_3 \equiv D_1 \equiv v$$

et les autres variables égales à f .

On colorie donc les sommets A et D dans la couleur 1, le sommet B dans la couleur 2, et le sommet C dans la couleur 3.

Définition 7. On appelle *littéral* une variable propositionnelle ou sa négation. Un littéral est dit *positif* si c'est une variable, *négatif* si c'est une variable niée.

On appelle *clause* une disjonction de littéraux.

On dit qu'une formule est sous *forme normale conjonctive* si elle est une conjonction de clauses. Alternativement, on peut la voir comme un ensemble de clauses.

Théorème 3 (forme normale conjonctive). *Toute formule propositionnelle F peut se ré-écrire en une formule équivalente sous forme normale conjonctive.*

Démonstration. Par récurrence sur la structure des formules. ⊕

Exercice 126. Mettre les formules suivantes sous forme normale conjonctive:

- $\neg(x_1 \vee x_2)$;
- $x_1 \vee \neg(x_2 \wedge x_3)$.

Exercice 127. Traduire sous forme de formule en forme normale conjonctive l'article L262-4 du Code de l'action sociale et des familles¹³:

13. Ce qui suit n'est pas une citation littérale: je me suis permis d'harmoniser la typographie avec le reste du document.

Le bénéfice du revenu de solidarité active est subordonné au respect, par le bénéficiaire, des conditions suivantes :

1. être âgé de plus de vingt-cinq ans ou assumer la charge d'un ou plusieurs enfants nés ou à naître ;
2. être français ou titulaire, depuis au moins cinq ans, d'un titre de séjour autorisant à travailler. Cette condition n'est pas applicable :
 - aux réfugiés, aux bénéficiaires de la protection subsidiaire, aux apatrides et aux étrangers titulaires de la carte de résident ou d'un titre de séjour prévu par les traités et accords internationaux et conférant des droits équivalents ;
 - aux personnes ayant droit à la majoration prévue à l'article L. 262-9, qui doivent remplir les conditions de régularité du séjour mentionnées à l'article L. 512-2 du code de la sécurité sociale ;
3. ne pas être élève, étudiant ou stagiaire au sens de l'article 9 de la loi n° 2006-396 du 31 mars 2006 pour l'égalité des chances. Cette condition n'est pas applicable aux personnes ayant droit à la majoration mentionnée à l'article L. 262-9 du présent code ;
4. ne pas être en congé parental, sabbatique, sans solde ou en disponibilité. Cette condition n'est pas applicable aux personnes ayant droit à la majoration mentionnée à l'article L. 262-9.

On attend donc ces variables, pour chacune leur signification, et les formules les liant.

Exercice 128. La légende veut que cette énigme ait été inventée par Albert Einstein (1879–1955), publiée la première fois dans le magazine *Life* en 1962 : il y a cinq maisons dans une rue, de couleurs différentes. Dans chacune de ces maisons vit une personne de nationalité différente. Chacune de ces personnes boit une boisson différente, fume une marque de cigare différente et a un animal domestique différent. On sait de plus que :

- Le Britannique vit dans la maison rouge.
- Le Suédois a des chiens.
- Le Danois boit du thé.
- La maison verte est directement à gauche de la maison blanche.
- Le propriétaire de la maison verte boit du café.
- La personne qui fume des Pall Mall élève des oiseaux.
- Le propriétaire de la maison jaune fume des Dunhill.
- La personne qui vit dans la maison du centre boit du lait.
- Le Norvégien habite dans la première maison.
- L'homme qui fume des Blend vit à côté de celui qui a des chats.
- L'homme qui a un cheval est le voisin de celui qui fume des Dunhill.
- Celui qui fume des Bluemaster boit de la bière.
- L'Allemand fume des Prince.
- Le Norvégien vit juste à côté de la maison bleue.
- L'homme qui fume des Blend a un voisin qui boit de l'eau.

On se demande qui a le poisson.

Il y a bien des manières de résoudre l'énigme de Exercice 128. Une d'entre elles peut suivre la stratégie suivante :

- commencer par ré-écrire le problème en un problème de satisfiabilité ;
- résoudre ce problème de satisfiabilité par un algorithme.

Concentrons-nous sur la première partie. Introduisons des variables qui seront vraies si la caractéristique correspondante l'est. On va introduire une variable par maison et par caractéristique : On peut par

exemple introduire des variables c_r^1, c_b^1, \dots signifiant respectivement que la maison 1 est rouge, blanche, ... c_r^2, c_b^2, \dots signifiant la même chose pour la deuxième maison, ... On se rend compte que le nom des couleurs n'ont aucune importance. En fait, on peut plutôt numérotter les couleurs 1, 2, 3, 4, 5, et introduire directement :

- une famille de variables $(c_j^i)_{1 \leq i \leq 5, 1 \leq j \leq 5}$ pour les couleurs ;
- une famille de variables $(n_j^i)_{1 \leq i \leq 5, 1 \leq j \leq 5}$ pour les nationalités ;
- une famille de variables $(b_j^i)_{1 \leq i \leq 5, 1 \leq j \leq 5}$ pour les boissons ;
- une famille de variables $(f_j^i)_{1 \leq i \leq 5, 1 \leq j \leq 5}$ pour les cigares que l'on fume ;
- une famille de variables $(a_j^i)_{1 \leq i \leq 5, 1 \leq j \leq 5}$ pour les animaux ;

Puis l'on écrit :

- des formules signifiant que chaque maison a chaque caractéristique :

$$c_1^1 \vee c_2^1 \vee c_3^1 \vee c_4^1 \vee c_5^1$$

signifiant que la maison 1 a soit la couleur 1, soit la couleur 2, soit la couleur 3, soit la couleur 4, soit la couleur 5, et

$$a_1^3 \vee a_2^3 \vee a_3^3 \vee a_4^3 \vee a_5^3$$

signifiant de même que la maison 3 abrite soit l'animal 1, soit...

- des formules signifiant que chaque caractéristique est unique :

$$\neg c_1^1 \vee \neg c_1^2$$

signifiant par exemple que la maison 1 et la maison 2 n'ont pas toutes les deux la couleur 1.

- et enfin des formules codant les informations qu'on a en plus. Par exemple, la première information « Le Britannique vit dans la maison rouge. » peut, si Britannique est la troisième nationalité, et rouge la première couleur, être écrit par l'ensemble de formules :

$$\begin{aligned} &\neg n_3^1 \vee c_1^1 \\ &\neg n_3^2 \vee c_1^2 \\ &\neg n_3^3 \vee c_1^3 \\ &\neg n_3^4 \vee c_1^4 \\ &\neg n_3^5 \vee c_1^5 \end{aligned}$$

signifiant que, pour chaque maison, soit elle n'est pas habitée par un Britannique, soit elle est rouge.

Exercice 129. Donner le système complet de formules booléennes à satisfaire.

Exercice 130. Donner une manière de coder n'importe quelle grille de sudoku en une formule booléenne.

Ainsi, si on a un algorithme générique qui est capable de satisfaire des formules booléennes, on sait répondre à ce problème. En règle générale, on voudra un algorithme qui, étant donné un ensemble de variables et de formules, réponde soit que l'ensemble n'est pas satisfiable, soit donne des valeurs à chaque variables de manière à satisfaire toutes les formules.

Exercice 131. Donner une représentation des formules en forme normale conjonctive sur n variables et un algorithme, qui étant donnés un entier n , une formule F à n variables en forme normale conjonctive, renvoie une assignation des n variables à v ou f si la formule est satisfiable, et autre chose sinon.

Correction: On commence par la représentation des formules sous forme normale conjonctive, et pour être précis, on commence par se demander ce qu'on voudra faire avec de telles formules. On va vouloir observer chaque clause, l'évaluer, et prendre des décisions selon son évaluation; autrement dit, on va vouloir considérer chaque clause indépendamment et l'évaluer. On fait donc la décision de coder une formule comme une liste de clauses, et de commencer par une première question: on veut une représentation des clauses et des assignations partielles de variables et, étant donnée une clause et une assignation, être capable de déterminer si cette clause est insatisfiable (auquel cas l'assignation des variables doit être rejetée) ou non.

On veut pouvoir accéder rapidement à la valeur de chaque variable (donc une liste serait malpratique) et le nombre de variables est connu d'avance (donc un tableau n'a pas son principal défaut): on va donc représenter l'assignation partielle des variables comme un tableau, et la valeur de la i -ème variable est stockée dans la i -ème case. On va écrire 0 si la valeur de la variable n'a pas encore été fixée. Une clause, elle, est de longueur arbitraire. On va donc la coder par une liste contenant le numéro des variables — en positif si elles ne sont pas niées, en négatif sinon¹⁴. Ainsi, la clause

$$x_1 \vee \neg x_4 \vee x_8 \vee \neg x_1$$

va être représentée par la liste

$$[1; -4; 8; -1].$$

On peut donc écrire l'algorithme suivant, qui vérifie si une clause est insatisfiable, vue l'assignation partielle.

Entrées:

- une liste C de couples d'un signe et un entier positif;
- un tableau $(VAL[i])_{0 \leq i < N+1}$ contenant v, f ou 0.

```

1 ClauseInsatisfiable(C, (VAL[i])0 \leq i < N+1)
2   tant que C ≠ Λ faire
3     |   ( $\sigma, \ell$ ) :: C ← C
4     |   si ( $\sigma \equiv +$  et  $VAL[\ell] \equiv f$ ) ou ( $\sigma \equiv -$  et  $VAL[\ell] \equiv v$ ) alors
5       |       retourner v
6     |   fin
7   |   fin
8   retourner f

```

Sorties: un booléen

On va donc, pour chaque variable:

- faire un choix de sa valeur (vraie ou fausse);
- vérifier si le choix produit une clause insatisfiable;
- si c'est le cas, on change de choix
- sinon on continue
- si on n'a plus de choix disponible, on *backtrack*.

On va donc faire un algorithme récursif, qui retourne soit un tableau (l'assignation partielle des variables), soit un symbole spécifique, signifiant qu'on a échoué (on va choisir \bullet).



NP-complétude

Le problème de la satisfiabilité a une caractéristique assez étonnante:

- il est compliqué à résoudre: l'algorithme qu'on a donné, par *backtrack* peut, dans certains cas, se retrouver à énumérer toutes les configurations possibles. Comme il y a, pour chaque variable, deux manières de l'assigner (par v ou par f), il y a un nombre exponentiel de configurations possibles, ce qui fait un algorithme exponentiel en le nombre de variables;
- si on a une configuration (la valeur de chaque variable), vérifier si c'est ou non une solution est facile (**exercice**: en donner la complexité): il suffit d'évaluer chaque clause.

On peut donner une définition plus précise de ces deux caractéristiques. Les problèmes comme la satisfiabilité (faciles à vérifier mais *a priori* difficiles à résoudre) sont appelés des problèmes **NP-complets**. La question ouverte la plus célèbre de l'informatique théorique (et une des plus célèbres des

14. Cela veut dire qu'on doit numérotter les variables à partir de 1 et pas de 0!

Entrées:

- une liste F de listes de couples d'un signe et un entier positif;
- un tableau $(VAL[i])_{0 \leq i < N+1}$ contenant v , F ou \bullet ;
- un entier $i \leq N$.

```

1 Satisfiabilité( $F, (VAL[i])_{0 \leq i < N+1}, i$ )
2    $VAL[i] \leftarrow v$ 
3   insatisfiable  $\leftarrow_F$  tant que  $F \neq \Lambda$  faire
4     |    $C :: F \leftarrow F$ 
5     |   insatisfiable  $\leftarrow$  insatisfiable ou ClauseInsatisfiable( $C, (VAL[i])_{0 \leq i < N+1}$ )
6   fin
7   si insatisfiable  $\equiv_F$  alors
8     |   si  $i \equiv N$  alors
9       |     retourner  $(VAL[i])_{0 \leq i < N+1}$ 
10    sinon
11      |      $v \leftarrow$  Satisfiabilité( $F, (VAL[i])_{0 \leq i < N+1}, i$ )
12      |     si  $v \neq \bullet$  alors
13        |       retourner  $v$ 
14      |     fin
15    fin
16  sinon
17    |    $VAL[i] \leftarrow F$ 
18    |   insatisfiable  $\leftarrow_F$  tant que  $F \neq \Lambda$  faire
19      |     |    $C :: F \leftarrow F$ 
20      |     |   insatisfiable  $\leftarrow$  insatisfiable ou ClauseInsatisfiable( $C, (VAL[i])_{0 \leq i < N+1}$ )
21    fin
22  si insatisfiable  $\equiv_F$  alors
23    |   si  $i \equiv N$  alors
24      |     retourner  $(VAL[i])_{0 \leq i < N+1}$ 
25    sinon
26      |     retourner Satisfiabilité( $F, (VAL[i])_{0 \leq i < N+1}, i$ )
27    fin
28  sinon
29    |     retourner  $\bullet$ 
30  fin
31 fin
Sorties: un tableau ou  $\bullet$ 
```

mathématiques¹⁵⁾ est de prouver que ces problèmes sont vraiment plus difficiles à résoudre qu'ils ne le sont à vérifier (ou à l'inverse, qu'ils sont simples à résoudre). Cette question peut être lue comme une conjecture précise posant la question de la créativité: pour résoudre rapidement un problème comme la satisfiabilité, peut-on suivre une méthode mécanique, ou a-t-on besoin d'autre chose?

Exemple 16. Le problème du sac-à-dos est un problème **NP-complet**, ou pour être précis, le problème suivant est **NP-complet**: « étant donné une liste d'objets, leur prix et leur volume, une capacité C ainsi qu'un seuil S, est-il possible de remplir un sac de capacité C avec des objets de la liste — sans les fractionner — de manière à ce que leur prix dépasse le seuil? ».

Exemple 17. Un voyageur de commerce veut passer par un certain nombre de villes. Il n'a d'essence que pour faire un certain nombre de kilomètres: il doit donc choisir rigoureusement dans quel ordre visiter ces villes de manière à faire le trajet le plus court possible.

Le problème (dit du *voyageur de commerce*) « étant donné un tableau de dimension $N \times N$ représentant des distances entre des points, et un seuil S, y-a-t'il un trajet passant par tous les points de longueur inférieure ou égale à S? » est **NP-complet**.

On voit qu'en fait, tous les problèmes de logistique sont facilement **NP-complets**.

Satisfiabilité des clauses de Horn

Néanmoins, il existe des cas où les formules sont suffisamment simples, et on peut utiliser des algorithmes plus efficaces. C'est le cas par exemple des clauses de Horn.

Définition 8 (clauses de Horn). Une clause est dite *de Horn* si un de ses littéraux au plus est positif.

En général, on note les clauses de Horn avec l'*implication*: en effet, en logique classique, l'implication $a \Rightarrow b$ est équivalente à $\neg a \vee b$, donc une clause

$$\neg x_1 \vee \neg x_2 \vee \dots \vee \neg x_n \vee y$$

dont tous les littéraux sauf un sont négatifs peut être vue comme une implication

$$x_1, \dots, x_n \Rightarrow y.$$

On a donc à gauche de la flèche, les variables qui sont dans des littéraux négatifs, et à droite, celle positive.

On peut imaginer assez facilement un algorithme résolvant le problème de la satisfiabilité restreinte aux clauses de Horn: en effet, pour chaque clause, si tous les littéraux négatifs valent F (et donc, leurs variables sous-jacentes valent V), on doit affecter la variable du littéral positif à V . On va donc parcourir les clauses dont tous les littéraux négatifs valent F , assigner leur littéral négatif à V , et recommencer (vu que de nouveaux littéraux négatifs vont valoir F). Quand on a terminé, on peut affecter toutes les autres variables à F .

Exemple 18. Si on a quatre clauses de Horn:

$$\begin{aligned} x_1, x_2 &\Rightarrow x_3 \\ x_3 &\Rightarrow x_4 \\ x_4, x_1 &\Rightarrow \\ &\Rightarrow x_1 \\ x_1 &\Rightarrow x_5 \end{aligned}$$

15. C'est, par exemple, un des sept *Millennium Problems* du *Clay Institute* dont la résolution donne droit à un prix d'un million de dollars américains.

Une seule (l'avant-dernière) a toutes ses variables en position négative affectées à v . Donc on affecte x_1 à v .

La dernière clause se retrouve avec toutes ses variables en position négatives affectées à v , donc on affecte x_5 à v .

On peut affecter toutes les variables qui restent à F et vérifier qu'on satisfait bien toutes les clauses.

Toute la difficulté est de faire ça efficacement : c'est-à-dire trouver facilement quelles clauses considérer, et mettre à jour les autres. On veut donc :

- classer les clauses par leur nombres croissant de littéraux négatifs ne valant pas F
- savoir, pour chaque variable, dans quelle clause il apparaît.

Exercice 132. Programmer un algorithme résolvant le problème de la satisfiabilité des clauses de Horn en temps linéaire.

4.3 LE VOYAGEUR DE COMMERCE

Revenons au problème du voyageur de commerce. Il est possible de le résoudre en backtracking : on trouve une première solution, puis on essaye de l'améliorer en changeant l'ordre des villes. Le problème est que rien ne nous dit que si on empire la solution localement, ça ne nous permettra pas, ensuite, de l'améliorer. Autrement dit, on va devoir énumérer toutes les permutations possibles des villes. S'il y a n villes, il y a $n! \equiv n \times (n - 1) \times \dots \times 3 \times 2 \times 1$ permutations. On peut montrer que ce nombre est nettement pire qu'exponentiel donc même un algorithme exponentiel serait une amélioration.

Programmation dynamique

On peut maintenant introduire une dernière technique : la programmation dynamique. Regardons-la d'abord sur le cas du voyageur de commerce, avant d'apprendre sa forme générale. On voit d'abord que le problème est un cas particulier d'un problème plus général : trouver le plus court chemin entre deux points fixés devant passer par un certain nombre de points (c'est le cas où les deux bornes sont égales). On ne peut pas appliquer facilement la méthode diviser-pour-régner : en effet, il faudrait savoir, dans ce cas, où découper notre chemin en deux listes de villes, et choisir la ville où découper paraît aussi difficile que résoudre le problème général. On veut plutôt pouvoir diviser pour chaque ville, résoudre le problème pour chaque division, et choisir ensuite la division la plus pertinente. Une idée pour faire ça est, plutôt que de diviser au milieu, choisir la dernière ville. Pour cela introduisons les notations suivantes :

- si L est une liste de villes, et v une ville, on note $L \setminus \{v\}$ la liste contenant tous les éléments de L sauf v ;
- si L est une liste de villes, et v_1 et v_2 deux villes, on note $\text{ch}_L(v, v')$ le plus court chemin entre v et v' dont tous les nœuds intermédiaires sont exactement les éléments de L , dans n'importe quel ordre, exactement une fois;
- si c est un chemin, on va noter $|c|$ sa longueur;
- $d(v, v')$ la distance entre deux villes v et v' .

Le problème du voyageur de commerce revient donc à calculer la longueur de $\text{ch}_{L \setminus \{v\}}(v, v)$ pour une ville quelconque. Elle n'a pas d'importance, en effet, ce chemin bouclant, on pourrait partir de n'importe où. Fixons-la donc et appelons-la 0. On veut donc calculer

$$|\text{ch}_{L \setminus \{0\}}(0, 0)|.$$

Ce chemin a un dernier élément (0), et un avant dernier, v qui est dans la liste $L \setminus \{0\}$. On remarque la propriété suivante :

$$|\text{ch}_{L \setminus \{0\}}(0, 0)| \equiv |\text{ch}_{L \setminus \{0, v\}}(0, v)| + d(v, 0).$$

On peut même aller plus loin : v est intégralement caractérisé par cette équation. Autrement dit :

$$|\text{ch}_{L \setminus \{0\}}(0, 0)| \equiv \min_{v \in L \setminus \{0\}} (|\text{ch}_{L \setminus \{0, v\}}(0, v)| + d(v, 0)).$$

Exercice 133. En déduire un algorithme exponentiel qui décide le problème du voyageur de commerce.

Exercice 134. On voit que certains calculs sont refaits très souvent. Proposer une structure de données pour les stocker.

Approximations

Dans certains cas, si on ne sait pas résoudre le problème de départ rapidement (par exemple parce qu'il est **NP-complet**)

4.4 PROBLÈME DES COUPLAGES

Couplages stables

On va s'intéresser maintenant à un problème que l'on peut résoudre autrement que localement. Il est apparu suite à la massification de l'enseignement supérieur, au sortir de la seconde guerre mondiale : chaque étudiant·e candidate à un certain nombre d'universités, et sur des critères qui lui sont propres ; de l'autre côté, les universités classent les étudiants, là encore, sur des critères particuliers. On se demande donc comment assigner *optimallement* les étudiant·es à des formations.¹⁶

Avant même de pouvoir trouver un algorithme qui réponde à la question, il faut la préciser, autrement dit *spécifier* le problème. On doit se demander :

- est-ce que chaque étudiant·e doit classer toutes les formations ?
- est-ce que chaque formation doit classer toutes les étudiant·es ?
- est-ce que les classements doivent être des ordres totaux ou non ?
- et surtout, que veux dire optimal ?

Attardons-nous un peu sur la dernière question : on voudrait, dans l'idéal, un système qui assure à la fois de prendre en compte des goûts et des capacités de chacun, et permettant d'affecter des étudiant·es dans des formations ouvrant sur les besoins d'emploi du futur tout en permettant de s'émanciper... Mais soyons honnête, la détermination de ce qui est un besoin social, a fortiori dans le futur, et de comment les différents besoins s'articulent, de comment les financer, et ainsi de suite, ne sont pas des questions mathématiques. On va donc se donner un critère d'optimalité purement interne, rapporté uniquement aux listes de préférences exprimées.

On va donc supposer que chaque étudiant·e donne un classement des formations ; et que parallèlement, chaque formation donne un classement des étudiant·es. On ne s'intéressera pas une seule seconde à la manière dont ces classements sont obtenus¹⁷, on suppose qu'ils existent.

Exercice 135. Montrez que, sans perte de généralité, on peut supposer que :

- chaque étudiant·e classe toutes les formations ;
- chaque formation classe toutes les étudiant·es ;
- il y a autant de formations que d'étudiant·es.

16. Le sujet n'a pris une acuité particulière en France depuis la loi du 8 mars 2018 relative à l'orientation et à la réussite des étudiants (loi ORE), et on a pu entendre tout et n'importe quoi à ce sujet. Il convient néanmoins de distinguer la procédure d'affectation dans des formations (qui est un sujet assez technique) avec le problème politique qui est derrière : de 2014 à 2019, il y a 234 700 étudiant·es supplémentaires en France sans augmentation de l'offre de formation (à titre de comparaison, l'UPEC forme 38 000 étudiant·es : on aurait pu construire l'équivalent de plus d'une UPEC par an pour absorber ce flux).

17. On peut imaginer une étudiante tirant au sort parmi toutes les formations, ou une autre choisissant celle offrant le meilleur salaire en sortie (ce qui sont deux algorithmes...), ou n'importe quoi d'autre, moins algorithmique.

C'est du fait de ces simplifications que le problème est souvent nommé « problème des mariages » : dans les États-Unis des années 1960, on pouvait faire l'hypothèse que la population était divisée en deux groupes, et qu'un membre d'un groupe ne pouvait se marier qu'avec un membre de l'autre¹⁸. On va se tenir, dans le reste du texte à la formulation plus neutre de *problème des couplages*.

On va donc supposer dorénavant qu'on a deux populations : les A (que l'on notera par des voyelles minuscules) et les B (que l'on notera par des consonnes minuscules) que l'on supposera avoir le même nombre d'éléments. Chaque a classe tous les éléments de B, autrement dit, on se donne pour chaque $a \in A$ un ordre total \leq_a sur les éléments de B. De même, on se donne un ordre total \leq_b sur A pour chaque élément de B.

Exemple 19. Soit $A := \{a, e, i, o\}$ et $B := \{b, c, d, f\}$.

On se donne les quatre ordres sur B :

$$\begin{aligned} d &>_a c >_a f >_a b \\ c &>_e b >_e d >_e f \\ c &>_i f >_i b >_i d \\ d &>_o b >_o f >_o c \end{aligned}$$

et les quatre ordres sur A :

$$\begin{aligned} a &>_b e >_b o >_b i \\ i &>_c a >_c o >_c e \\ i &>_d e >_d o >_d a \\ e &>_f a >_f i >_f o \end{aligned}$$

On veut obtenir un *couplage* entre les deux ensembles, c'est-à-dire une fonction

$$\mathfrak{C} : A \rightarrow B$$

qui associe de manière univoque un élément de B à un élément de A et vice versa¹⁹. On veut aussi que ce couplage soit optimal, ce qui peut vouloir dire plusieurs choses :

- on peut considérer que parmi les A, il y a un élément dictatorial, et que ce qui compte est que sa préférence soit réalisée. Dans le cas de l'exemple ci-dessus, on dirait que n'importe quelle fonction est optimale tant que

$$\mathfrak{C}(a) \equiv d;$$

- plus généralement, on peut considérer que satisfaire le plus de A possible en leur donnant leur premier choix est la seule chose qui compte. Dans le cas de l'exemple, on imposerait donc que

$$\mathfrak{C}(a) \equiv d \text{ ou } \mathfrak{C}(o) \equiv d$$

et

$$\mathfrak{C}(e) \equiv c \text{ ou } \mathfrak{C}(i) \equiv c.$$

Mais ce critère ne nous permettrait pas de choisir entre les autres solutions ;

- on peut imaginer le complexifier, prendre en compte les seconds choix... voire même les choix des B, mais il devient assez difficile de trouver un critère objectif ;
- de même, on peut imaginer privilégier les B, ou encore minimiser les éléments obtenant leur dernier choix...

18. D'autres temps ou d'autres lieux auraient donné d'autres conclusions (HÉRITIER-AUGÉ et COPET-ROUGIER 1990).

19. C'est-à-dire qu'à chaque a correspond exactement un b et vice-versa.

On va plutôt prendre une autre définition, en commençant par l'autre sens, et définir comment on peut améliorer un couplage. Soit a un élément de A , et $\mathfrak{C} : A \rightarrow B$ un couplage. Si le parèdre²⁰ $\mathfrak{C}(a)$ de a est le premier choix de a , du point de vue de a , ce couplage ne peut pas être amélioré, même si pour $\mathfrak{C}(a)$, a était tout en bas de la liste : il ne sera pas évident qu'un couplage améliorant la situation pour $\mathfrak{C}(a)$ soit globalement meilleur. Si, par contre, a préfère un autre élément, c'est-à-dire qu'il existe e dans A tel que :

$$\mathfrak{C}(e) >_a \mathfrak{C}(a),$$

(c'est-à-dire que a préfère le parèdre de e au sien) alors que

$$a >_{\mathfrak{C}(e)} e$$

(c'est-à-dire que le parèdre de e aurait préféré a), alors on obtient un couplage plus satisfaisant en remplaçant les parèdres de a et de e .

Définition 9 (GALE et SHAPLEY 1962). Soit $\mathfrak{C} : A \rightarrow B$ un couplage. Une *instabilité* est une paire (a, e) d'éléments de A où a préfère le parèdre de e au sien et le parèdre de e préfère a à e telle que

$$\begin{aligned}\mathfrak{C}(e) &>_a \mathfrak{C}(a) \\ a &>_{\mathfrak{C}(e)} e\end{aligned}$$

Un couplage est dit *stable* s'il n'y a pas d'instabilités.

Exemple 20. Supposons qu'on ait deux étudiant·es, Camille et Sam, et deux formations, Créteil et Sceaux. Camille préférerait aller à Créteil, et Créteil a classé Camille en première position.

Le couplage qui affecte Camille à Sceaux et Sam à Créteil est instable : si Camille et l'administration de Créteil s'en rendent compte, on demandera à changer.

On va donc se limiter à chercher des couplages stables. Il n'est pas évident qu'il en existe, ni même qu'il soit facile d'en calculer.

Exercice 136. Donner un couplage stable pour les préférences de l'Exemple 19.

Encore une fois, rien ne dit qu'un couplage stable soit particulièrement pertinent du point de vue de la société.

L'algorithme de Gale et Shapley

Pour prouver qu'il existe toujours une solution, on va en construire une : autrement dit, on va donner un algorithme qui trouve systématiquement un couplage stable, étant donné les listes de préférences. Pour une analyse beaucoup plus poussée, on pourra se rapporter à (KNUTH 1976b).

La première question à se poser est de savoir comment représenter les données : on doit manipuler deux ensembles contenant le même nombre d'éléments, et les préférences de chacun des éléments dans l'autre ensemble. On peut donc, quitte à supposer qu'il y a un ordre sur les deux ensembles, représenter chaque élément par un entier, plus petit que le nombre d'éléments. Chaque préférence est donc une permutation de ces entiers, qu'on peut représenter par un tableau. On a deux possibilités pour représenter une telle préférence : soit on met les éléments dans leur ordre de préférences, et par exemple, la préférence

$$d > c > f > b$$

si on suppose les éléments classés par ordre alphabétique, pourra être représenté par le tableau :

20. Terme, généralement utilisé pour les dieux, ayant l'avantage d'être épicène.

2	1	3	0
0	1	2	3

L'autre possibilité est, pour chaque élément, de placer son niveau, et donc, de représenter la même préférence par :

3	1	0	2
0	1	2	3

On va choisir la deuxième représentation, car, si elle ne nous permet pas facilement de savoir quel est le premier choix, elle nous permet de savoir qu'elle était le rang d'un élément (et donc d'évaluer le couplage).

Comme on veut représenter une préférence (sur B) pour chaque élément de A et vice versa, on va utiliser deux tableaux à deux dimensions, l'un donnant les préférences de A sur B et inversement. Pour des raisons techniques, on va donc prendre des tableaux de dimensions $N \times N + 1$, en se laissant de la place pour un élément en plus.

Exercice 137. Donner les deux tableaux représentant les préférences de l'Exemple 19.

Avant de donner l'Algorithme 30, donnons-en une description à très haut niveau : on commence par coupler tous les éléments de B avec un élément fictif \mathbb{F} (à qui on assigne le numéro N), que l'on considère comme le dernier choix de tous les éléments de B. Pour chaque élément a de A, on prend son premier choix b , si b préfère a à son parèdre actuel a' , on accouple b avec a et on assigne a' à a . Tant que a n'est pas \mathbb{F} , on supprime a des préférences de b et on recommence²¹.

Exercice 138. Exécutez l'algorithme sur les préférences de l'Exemple 19. On attend pas une trace, seulement la suite des parèdres.

Exercice 139. Montrer que si on retire un b à la liste des préférences d'un a , alors aucun couplage stable n'associe b et a .

Exercice 140. Appelons \mathcal{C} le résultat de l'exécution.

Montrer que si un a préfère b à $\mathcal{C}(a)$, c'est que pendant l'exécution, b s'en est séparé.

Exercice 141. Montrer que deux b, b' ne peuvent être couplés avec le même a .

Exercice 142. Montrer que la situation d'un b n'empire pas au cours de l'exécution.

Exercice 143. Montrer que la liste de préférence d'un a ne devient jamais vide au cours de l'exécution.

Exercice 144. En déduire que le couplage obtenu par l'exécution est stable.

Exercice 145. La situation était parfaitement symétrique entre les deux populations, néanmoins, l'algorithme en choisit une, qui fait des propositions à l'autre. Qu'en pensez-vous ?

Cet algorithme est plus ou moins le cœur de ParcourSup (et Admission PostBac avant lui, et sans doute 36 15 RAVEL, et...). Néanmoins, l'algorithme réellement effectué est plus complexe (et n'a sûrement aucune propriété mathématique raisonnable) du fait que certaines filières ont des quotas (de boursier·es ou de lauréat·es de bacs techniques), et que d'autres dispositifs forcent à réservé des places

21. On voit ici la difficulté d'écrire un algorithme en français...

Entrées:

- un entier N;
- deux tableaux d'entiers $(\text{CHOIX_A}[i][j])_{\substack{0 \leq i < N \\ 0 \leq j < N+1}}$ et $(\text{CHOIX_B}[i][j])_{\substack{0 \leq i < N \\ 0 \leq j < N+1}}$.

1 CouplageStable(N, $(\text{CHOIX_A}[i][j])_{\substack{0 \leq i < N \\ 0 \leq j < N+1}}$, $(\text{CHOIX_B}[i][j])_{\substack{0 \leq i < N \\ 0 \leq j < N+1}}$)

```

2   | nouveau (COUPLAGE[i])0 \leq i < N
3   | pour i allant de 0 à N faire
4   |   | COUPLAGE[i] ← N
5   | fin
6   | pour k allant de 0 à N faire
7   |   | a ← k + 1
8   |   | tant que a ≠ N faire
9   |   |   | pour i allant de 0 à N faire
10  |   |   |   | si CHOIX_A[a][i] == 0 alors
11  |   |   |   |   | b ← i
12  |   |   |   | fin
13  |   | fin
14  |   | si CHOIX_B[b][a] < CHOIX_B[b][COUPLAGE[b]] alors
15  |   |   | a ↔ COUPLAGE[b]
16  |   | fin
17  |   | si a ≠ N alors
18  |   |   | c ← CHOIX_A[a][b]
19  |   |   | pour i allant de 0 à N faire
20  |   |   |   | si CHOIX_A[a][i] > c alors
21  |   |   |   |   | CHOIX_A[a][i] ← CHOIX_A[a][i] - 1
22  |   |   |   | fin
23  |   |   | fin
24  |   |   | CHOIX_A[a][b] ← N
25  |   | fin
26  | fin
27  | fin

```

Sorties: un tableau $(\text{COUPLAGE}[i])_{0 \leq i < N}$

Algorithme 30: Algorithme de Gale et Shapley

(par exemple, les 10% de bachelier·es ayant eu les meilleurs résultats au bac de leur lycée ont droit à intégrer une filière sélective — vu que les résultats du bac arrivent bien plus tard que la période des admissions, il faut réserver des places à l'aveugle).

En résumé, l'algorithme de ParcourSup est une adaptation brisant toutes les propriétés mathématiques connues d'un algorithme n'ayant a priori pas de propriétés sociales souhaitables.

La pensée algorithmique

En guise de conclusion, on étudie certaines institutions comme l’État ou le marché, d’un point de vue algorithmique, et on voit que ce point de vue plaide contre le solutionnisme naïf.

5.1	Complexité	150
5.2	Conséquences	150
5.3	Monnaie	151
	Le problème du consensus byzantin	152
	Rémunération de la vérification	154
	Jetons non-fongibles	154
	Contrats intelligents	155
	Petite tératologie	156
5.4	Optimisation combinatoire	157

5.1 COMPLEXITÉ

ON A VU que tous les problèmes n'ont pas la même difficulté intrinsèque: par exemple, on a énoncé le théorème que le problème du tri ne peut pas être résolu en moins de $O(n \log n)$ comparaisons dans le pire des cas. On a, par exemple, aussi donné un algorithme quadratique (c'est-à-dire en $O(n^2)$), où n est le nombre de sommets) pour le problème du coloriage de graphe¹. On peut donc classer les problèmes par leurs complexités.

On appelle *classe de complexité* un ensemble de problèmes qui peuvent être résolus, dans le pire des cas, par un algorithme d'une complexité donnée. On peut par exemple considérer la classe « linéaire » des problèmes pouvant se résoudre par un algorithme linéaire.

Il est généralement considéré qu'un problème est faisable s'il existe un algorithme polynomial le résolvant, c'est à dire un algorithme dont la complexité est linéaire, quadratique, cubique,... On appelle cette classe **P**. Il y a de nombreux problèmes dans cette classe, par exemple:

- le tri ;
- le coloriage de graphe ;
- la connexité d'un graphe — c'est-à-dire l'existence d'un chemin entre deux sommets ;
- la satisfiabilité des clauses de Horn ;

et d'autres problèmes que nous verrons.

Inversement, on a aussi vu une autre classe de complexité, la classe **NP** des problèmes pour lesquels si on a une solution potentielle, il est facile (c'est-à-dire il est polynomial) de vérifier que cette solution l'est vraiment. Dedans, il y a :

- tous les problèmes dans **P** (en effet, s'il est facile de trouver une solution, il est aussi facile de vérifier) ;
- la factorisation de nombres entiers en facteurs premiers: étant donné un nombre, on veut le représenter comme un produit de nombre les plus petits possibles. Multiplier des nombres pour vérifier que la factorisation est correcte est facile (polynomial) mais trouver une telle factorisation est plus dure ;
- le problème SAT de la satisfiabilité des formules booléennes ;
- le problème du voyageur de commerce.

Pour les deux derniers problèmes, on sait même que ces problèmes sont, à un sens précis, les plus durs de la classe **NP**: on dit qu'ils sont **NP-complets**.

On peut donc résumer la situation par le schéma suivant :

On conjecture en général que $\mathbf{P} \neq \mathbf{NP}$. Cette question, posée depuis ... n'a pas vraiment connu de progrès depuis. On n'a aucune idée de comment attaquer le problème.

5.2 CONSÉQUENCES

Ça semble un aveu d'échec terrible. Ça a néanmoins un certain nombre de conséquences intéressantes. On appelle une *fonction-trappe* une fonction qui est facile à calculer, et dont l'inverse est bien définie mais difficile à calculer. Si $\mathbf{P} \equiv \mathbf{NP}$, il n'existe pas de telle fonction. Sinon, on conjecture qu'il en existe et que la multiplication de nombres premiers en est une.

Il est très intéressant d'avoir une telle fonction. En effet, elle permet de *chiffrer*: le concept même du chiffrement est qu'il soit facile de chiffrer un message, facile de le déchiffrer (avec la clef), difficile de le décrypter (sans la clef). Si $\mathbf{P} \equiv \mathbf{NP}$, il n'existe pas de fonctions-trappes, et il n'est donc pas réaliste de faire du chiffrement.

Cela a encore d'autres conséquences: le chiffrement sert en effet à beaucoup de choses — par exemple aux signatures électroniques.

1. C'est l'algorithme glouton, qui trouve une solution, mais pas forcément la solution optimale.

5.3 MONNAIE

Observons plus en détail un nouveau problème, celui de la monnaie. Nous utilisons quotidiennement au moins deux systèmes monétaires avec des principes assez différentes, mais néanmoins intégrés : la monnaie physique, et la monnaie électronique. Avant de les décrire, voyons quels sont les problèmes qu'ils cherchent à résoudre² :

1. il faut tout d'abord être capable de savoir la quantité d'argent possédée par chaque agent, et que cette information ne soit pas sujette à controverse. Autrement dit, tout le monde doit être capable de reconnaître ce qui est une unité monétaire de ce qui ne l'est pas ;
2. il faut que ces unités aient un·e propriétaire non-équivoque ;
3. il faut ensuite pouvoir transférer cet argent d'un agent à un autre, et que seul le possesseur soit capable de le faire ;
4. enfin, il faut pouvoir s'assurer qu'une unité dépensée ne peut pas être re-dépensée.

On peut tout de suite se demander comment ces quatre problèmes sont résolus par les systèmes monétaires habituels :

pour la monnaie physique les unités monétaires sont représentées par des objets particuliers (pièces et billets), difficiles à produire mais dont il est relativement facile de vérifier qu'ils sont authentiques. La possession est la possession physique, et l'échange en suit les mêmes règles : un objet physique n'est pas reproductible sans ressources.

On voit que le fonctionnement nécessite crucialement la pouvoir d'un État qui s'arroge le monopole de battre monnaie et des poursuites contre les faussaires.

pour la monnaie fiduciaire les unités monétaires sont représentées comme des lignes dans les livres de compte d'une banque. C'est la banque qui s'assure que chaque unité monétaire reste à son propriétaire, n'est échangée que sous son contrôle et n'est jamais dupliquée. Les banques ensuite, ont des comptes soient les unes chez les autres, soit dans une banque centrale pour effectuer des échanges entre leurs clientèles respectives.

On voit, là aussi, que le fonctionnement nécessite un système très régulé pour contrôler la tenue de tout ces comptes.

In fine, ces deux systèmes reposent sur la puissance d'un État, et nécessite donc de lui faire confiance : un État désargenté pourra battre mauvaise monnaie, finançant ses dépenses par l'inflation, ou même plus simplement diminuer les dépôts bancaires (par exemple, en 2013 à Chypre, tous les dépôts bancaires de plus de 100 000 € ont été diminués de 47,5%). Si une solution tout à fait raisonnable pour éviter d'avoir à placer une confiance déraisonnable dans l'État peut consister à le démocratiser, certains courants politiques veulent l'abolir, tout en gardant tel quel le système monétaire³.

Pour ne plus avoir à faire confiance à qui que ce soit, l'idée est très simple : tout le monde va garder le registre de toutes les transactions. Ainsi, un État maléfique ne pourra pas réduire la valeur de nos comptes en banques : nous, bons citoyens amoureux de la liberté, pourrons toujours constater que dans nos comptes, nous avons l'argent et donc le dépenser. Donc, le premier principe de la chaîne de blocs est d'être décentralisée : tout le monde garde un registre de toutes les transactions :

<1905-12-26> Swann verse 200 à Odette
 <1905-12-27> Odette verse 30 au fleuriste

2. Rappel : avant même de décrire un algorithme particulier, on doit toujours chercher à savoir quel problème il résout, afin qu'ensuite on puisse évaluer s'il le résout bien.

3. Donc, quelque part, pour comprendre tout ce qui va suivre, il faut s'imaginer dans un ranch, avec des *pick-ups* et des AR-15, en train d'écouter Mini Thin.

et on ne valide une transaction que si elle est cohérente avec tout le passé, sinon, on la rejette. Nos quatre problèmes ne sont pas encore réglés : en effet, on ne peut pas calculer combien chacun a : il faut aussi des éléments spéciaux du registre permettant de le faire commencer en donnant, pour chaque acteur, une somme initiale. Par ailleurs, rien ne garantit que la somme ne soit débloquée que par l'agent la possédant. Pour régler ce problème, on exige que chaque transaction soit signée par le propriétaire du compte versant la somme. Ainsi, un registre va ressembler à quelque chose comme :

Initialisation	Swann possède 1000			
Initialisation	Le fleuriste possède 500			
<1905-12-26>	Verser 200 à Odette	<i>Elle ressemble à la fille de Jéthro !</i>	Swann	
<1905-12-27>	Verser 30 au fleuriste		Odette	

À chaque nouvelle transaction, la personne dépensant envoie à tout le monde la nouvelle ligne (avec donc, une date, l'adresse du compte à qui verser, la somme, un éventuel commentaire et signe numériquement la transaction). Tout le monde peut donc vérifier que Swann avait les 200 nécessaires pour payer, et accepter la transaction.

On a donc, juste avec la signature électronique, réglé tous nos problèmes... en en créant un nouveau : en effet, que se passe-t-il si un acteur mal intentionné dépense deux fois la même somme au même moment, en faisant deux transactions qui seraient chacune valide séparément, mais pas prises ensemble ? Il faut se donner un mécanisme permettant de valider seulement une de ces transactions⁴, et de propager ce choix dans tous les registres.

Le problème du consensus byzantin

Ce problème se décompose en plusieurs cas :

- soit on a reçu les deux transactions incompatibles, et il suffit de les rejeter ;
- soit on a reçu une seule des deux, et il faut pouvoir faire accepter à tous les autres registres du réseau qu'elle est la bonne.

Une manière pour faire ceci est d'avoir un registre ayant un statut particulier, et seules les transactions que ce registre-là accepte sont considérées acceptées. On pourrait l'appeler « Banque », voire « État », et on se retrouverait dans une situation dure à défendre idéologiquement.

On cherche donc à résoudre le problème suivant : mettre d'accord un certain nombre d'agents indépendants de manière arbitraire (souvent, on n'a pas de raison de choisir une transaction plutôt qu'une autre...) sans qu'il y ait une autorité pour choisir. On appelle ce problème le *problème du consensus byzantin*⁵, et c'est un problème assez classique de l'informatique distribuée.

La solution de la chaîne de blocs est d'utiliser un processus de validation suffisamment long mais avec une durée aléatoire pour que deux validations n'aient jamais lieu au même moment. Plus précisément, tout nœud (qui tient un registre) passe son temps à recevoir des nouvelles transactions. Dès qu'il en a reçu assez (pour fixer les idées, disons 10), il peut chercher à les valider. Considérons le bloc de transactions suivant :

4. On considérera qu'une transaction n'est validée qu'une fois qu'elle est acceptée par tout le réseau, ce qui n'a aucune raison d'être instantané : un peu comme un chèque, qui peut être refusé.

5. Sans doute car l'armée byzantine est réputée avoir tellement de généraux (enfin, plutôt de stratégies) qu'elle n'a pas vraiment de hiérarchie.

!	"	#	\$	%	&	,	()	*	+	,	-	.	/	
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	~	¤
1100000	1100001	1100010	1100011	1100100	1100101	1100110	1100111	1101000	1101001	1101010	1101011	1101100	1101101	1101110	1101111

FIGURE 5.1 – Les caractères dans la norme ASCII

<1905-12-26>	Veser 200 à Odette	<i>Elle ressemble à la fille de Jéthro!</i>	Swann
<1905-12-27>	Veser 30 au fleuriste		Odette
<1905-12-27>	Veser 40 à Elstir		Odette
<1905-12-27>	Veser 40 à Gilberte		Odette
<1905-12-28>	Veser 30 à Morel		Charlus
<1905-12-28>	Veser 30 au fleuriste		le narrateur
<1905-12-28>	Veser 30 à Albertine	<i>Pour un peignoir</i>	le narrateur
<1905-12-28>	Veser 30 à Andrée		le narrateur
<1915-01-13>	Veser 40 à Saint-Loup		Gilberte
<1915-08-06>	Veser 2000 à Jupien		Charlus

Pour le valider, on va commencer par trouver un problème difficile à résoudre mais dont la solution est facile à vérifier qui corresponde à ce bloc. Pour cela, on va se servir de la manière dont les données sont codées pour un ordinateur: tout est, *in fine*, représenté sous forme de suites de 0 et de 1, selon des normes. Par exemple, la norme ASCII, présentée Figure 5.1, code les caractères sur 7 bits. On peut donc lire un « A » comme la suite de bits 1000001. Cette suite de bits peut aussi bien être lue comme un nombre, en l'occurrence 65. En poussant un peu, on peut voir chaque bloc comme un ou plusieurs nombres; et on peut calculer sur ses nombres. Par exemple, si on voit le bloc comme deux nombres (par exemple, en coupant le bloc par son milieu), on peut vouloir calculer leur somme.

Plutôt que de voir le bloc comme une instance du problème de l'addition, on va plutôt le voir comme une instance d'un problème **NP**-complet, c'est-à-dire pour lequel il est difficile à trouver une solution, mais une fois qu'on a une solution, loisible de la vérifier. Ainsi, le protocole prévoit une manière de lire le bloc comme une instance d'un problème, et chaque vérificateur, après avoir vérifié que les transactions sont cohérentes, se met à chercher une solution. Dès qu'un vérificateur l'a trouvée, il place sa solution à la fin du bloc, rajoute des informations comme la date, et signe le tout. Ainsi, le bloc une fois vérifié ressemble à quelque chose comme :

<1915-09-12>

<1905-12-26>	Verser 200 à Odette	<i>Elle ressemble à la fille de Jéthro!</i>	Swann
<1905-12-27>	Verser 30 au fleuriste		Odette
<1905-12-27>	Verser 40 à Elstir		Odette
<1905-12-27>	Verser 40 à Gilberte		Odette
<1905-12-28>	Verser 30 à Morel		Charlus
<1905-12-28>	Verser 30 au fleuriste		le narrateur
<1905-12-28>	Verser 30 à Albertine	<i>Pour un peignoir</i>	le narrateur
<1905-12-28>	Verser 30 à Andrée		le narrateur
<1915-01-13>	Verser 40 à Saint-Loup		Gilberte
<1915-08-06>	Verser 2000 à Jupien		Charlus

SOLUTION DE L'INSTANCE DU PROBLÈME CI-DESSUS

Céleste Albaret

si la vérificatrice s'appelle Céleste Albaret.

La vérificatrice envoie ce bloc à tout le reste du réseau. Les autres nœuds vérifient que ce bloc est cohérent avec le reste de la chaîne (et donc, qu'aucune transaction n'a déjà été validée, ou que les transactions ne sont pas incohérentes). Si jamais deux blocs contenant les mêmes transactions sont validés par deux vérificatrices différentes, on prend le plus vieux des blocs (ce qui rend l'horodatage indispensable) : le problème calculatoire difficile sert à rendre improbable que deux vérificatrices arrivent exactement en même temps.

Ainsi, le registre des transactions va prendre la forme d'une chaîne de bloc, chacun ayant été vérifié une première fois par quelqu'un, mais ensuite par tout le monde. Cela résout (de manière assez baroque) le problème du consensus byzantin.

Rémunération de la vérification

La vérification d'un nouveau bloc est une activité très lourde calculatoirement, qui coûte cher aussi bien en terme d'équipement que d'électricité. On peut donc rémunérer les vérificatrices : signer un nouveau bloc rapporte un peu d'unités monétaires au compte de la signature⁶. Comme les auteures de ces choses ont un fétichisme du métal précieux, on appelle cette activité de vérification *miner*.

Jetons non-fongibles

On se dit que, tant qu'à faire, on peut se servir de ce registre pour noter non seulement des transactions monétaires, mais aussi un peu ce qu'on veut : si on rajoute une entrée qui ne modifie les comptes de personne, ça ne met pas en danger le système monétaire, mais ça nous permet de rendre public cette entrée, et la faire horodater par des vérificateurs qui sont, *a priori*, difficiles à coordonner. On va donc faire précédé de  chaque transaction monétaire, et autoriser des transactions qui ne commence pas par ce symbole, mais par exemple par  des évènements impressionnantes. De la même manière que le reste, ces évènements sont signés par quelqu'un, puis le tout est vérifié ; ce qui signifie simplement qu'on vérifie que la signature est correcte, mais absolument pas que l'évènement a eu lieu.

6. Ces unités peuvent soit être prélevées des transactions (ce sont donc des frais de transaction) soit être créées *ex nihilo* (auquel cas c'est un mécanisme de création monétaire). Dans tous les cas, ça permet de transformer de l'électricité (et donc de l'argent du vrai monde) en une unité monétaire.

<1915-09-12>

<1850-11-12>		#897347	Sonate pour piano et violon	Vinteuil
<1905-12-26>		200 à Odette	Elle ressemble à la fille de Jéthro!	Swann
<1905-12-27>		30 au fleuriste		Odette
<1905-12-27>		40 à Elstir		Odette
<1905-12-27>		40 à Gilberte		Odette
<1905-12-28>		30 à Morel		Charlus
<1905-12-28>		30 au fleuriste		le narrateur
<1905-12-28>		30 à Albertine	Pour un peignoir	le narrateur
<1905-12-28>		30 à Andrée		le narrateur
<1906-07-12>		Dreyfus est réhabilité		Le Petit Parisien
<1908-01-01>		#897347 à Verdurin		Vinteuil
<1908-01-01>		2000 à Vinteuil	Pour #897347	Verdurin
<1915-01-13>		40 à Saint-Loup		Gilberte
<1915-08-06>		2000 à Jupien		Charlus

SOLUTION DE L'INSTANCE DU PROBLÈME CI-DESSUS

Céleste Albaret
 génère 10

On voit qu'ici, Céleste Albaret n'a fait que vérifier que le Petit Parisien prétendait que Dreyfus est réhabilité, sans l'avoir vérifié elle-même. On imagine qu'après tout un jeu de réputation assure que certains signataires ont plus de poids que d'autres quand ils annoncent des événements.

On peut continuer encore plus loin, et se dire que l'on peut se servir de transactions pour créer des unités qui pourront ensuite être vendues. Vu qu'encore une fois, le vocabulaire est tinté d'idées étranges, on reprend la métaphore du jeton de casino. Tout le monde peut donc créer un jeton (qui ne sera qu'une transaction spéciale) qui a un identifiant unique et un commentaire le décrivant, et il appartient à la personne qui l'a signée. Cette personne pourra ensuite l'échanger, comme n'importe quelle unité monétaire, mais lors de l'échange, l'identité du jeton sera préservée (ce qui n'est pas le cas pour les unités monétaires, qui sont toutes interchangeables). Dans le bloc ci-dessus, on a noté les transactions de création de tels jetons avec le signe .

Contrats intelligents

Pour l'instant, on a donc trois types de transactions, qui ont toutes un effet instantané (si elles ont un effet). On peut imaginer faire des transactions plus compliquées :

des transactions différences on peut programmer qu'une transaction n'ait lieu que si un évènement a lieu, ou à une certaine date. Une application possible peut-être pour des paris sportifs : on programme une transaction qui n'a lieu que si un bloc apparaît contenant l'observation d'une victoire, et elle est annulée sinon ;

des transaction bloquées on peut imaginer qu'une transaction doive être débloquée par une autre signature avant d'être déclenchée. On peut imaginer un notaire, ce qui remplacerait un dépôt.

Dans tous les cas, on voit qu'on fait jouer directement par le système monétaire des activités qui avaient un intermédiaire — mais dans tous les cas, si on a supprimé l'infrastructure qui garde en dépôt le montant du pari ou la somme pour un achat immobilier, on garde l'arbitre des courses ou le

notaire... Donc une suppression des intermédiations toute relative, mais qui permettrait, par exemple, de supprimer la direction générale des finances publiques, en mettant le code des impôts dans chaque transaction générant des revenus.

Se pose une question informatique intéressante si on veut faire cela: une fois qu'on veut remplacer tout le système monétaire, il existe des transactions arbitrairement compliquées; on peut donc imaginer écrire des transactions dans un langage de programmation générique, permettant de varier les conditions pour lesquelles les transactions seront validées, voire leur montant. On a tout de suite un problème purement informatique: en effet, on sait⁷ qu'un langage de programmation peut soit:

- permettre d'écrire tous les programmes qui terminent, mais aussi des programmes qui ne terminent pas;
- ne pas permettre d'écrire tous les programmes qui terminent.

Autrement dit, soit on limite les transactions possibles, soit il est possible d'avoir des transactions dont on ne sait pas quelles valeurs elles versent et à qui, ce qui met en danger le système monétaire dans son ensemble. Tout aussi grave, on peut avoir des transactions dont le calcul est très long, bloquant tous les ordinateurs en train de vérifier et les empêchant de calculer.

On a donc deux solutions, soit on bride l'expressivité du langage de programmation (et donc, on doit conserver la DGFIP), soit on se donne un moyen de forcer le calcul à être fini. Pour cela, on décide que chaque étape de calcul (dans la trace d'exécution) coûte une certaine somme et chaque personne désirant faire une transaction doit allouer une somme qui sera payée aux vérificateurs faisant vraiment le calcul. Si cette somme arrive à bout, la transaction est annulée (mais le prix de transaction bien dépensé). Ainsi, on arrive à des programmes exécutés par les vérificateurs, pouvant déclencher des transactions arbitrairement compliquées (tant qu'il reste des frais à déboursier).

Que se passe-t-il si le programme était mal écrit, par exemple avec un bug, ou une arnaque? Rien. Il n'y a aucune manière (interne) de rattraper ça. Soit on doit accepter une transaction douteuse, soit on intervient (socialement...) pour rétablir ce que la communauté veut. Autrement dit, soit on considère que tous les contrats sont parfaits et non-ambigus et couvrent toutes les situations possibles, soit on doit re-donner une place à des institutions ayant une place privilégiée dans l'administration de la chaîne de blocs. De la même manière, si on veut que les contrats conclus dans la chaîne aient un impact sur le reste du monde... il faut s'en donner les moyens et sûrement s'appuyer sur des institutions.

On n'abolit pas l'État si facilement.

Petite tématologie

Ce problème a attiré et n'est pas particulièrement récent. Les premières chaînes de blocs permettant de servir de bases de données décentralisées de transactions datent de 1998 (*Bitgold* et *B-money*); *Bitcoin*, l'application utilisant la chaîne de bloc la plus connue date de 2009. Aujourd'hui, des centaines de protocoles utilisent des chaînes de blocs. Citons-en quelques un :

- *Bitcoin*, créée par un anonyme utilisant le pseudonyme de Satoshi Nakamoto. Le langage de transaction y est très simple (n'est pas Turing-complet). La plupart des noeuds du réseau font tourner la même implémentation, *Bitcoin Core*. Un nouveau bloc est créé en moyenne toutes les 10 minutes. Il y a au maximum 21 millions de bitcoins, ce qui veut dire qu'à chaque création de bloc, la récompense pour avoir validé devient plus faible (aujourd'hui, c'est 6,25 bitcoins pour un bloc); et aussi que, en tant que monnaie, le *bitcoin* est intrinsèquement déflationiste⁸.

7. Cf le cours de fondements de l'informatique.

8. Il y a de moins en moins de monnaie produite, donc posséder de la monnaie est de plus en plus intéressant (plutôt que posséder n'importe quoi d'autre): les prix vont avoir tendance à baisser. En conséquence, il n'est jamais intéressant de dépenser si on peut décaler: il suffit d'attendre le dernier moment, le prix aura baissé d'ici-là. Plus dramatique, il n'est pertinent d'investir seulement dans des investissements ayant des rendements énormes: en effet, s'il suffit d'attendre, sans rien faire, pour que la valeur de son argent ait augmenté, à quoi bon prendre des risques...

Le fondateur ayant disparu sans laisser de trace (mais en possédant toujours une belle somme), Bitcoin est plus ou moins gelé.

- *Ethereum*, la deuxième plus grosse monnaie (en capitalisation) basée sur une chaîne de blocs. C'est le lieu des expérimentations : jetons non-fongibles, contrats intelligents, etc. Le langage des contrats est Turing-complet, et on doit payer pour exécuter chaque étape (comptées dans des traces d'exécution : de fait, il y a deux langages de programmation pour écrire des contrats — un de bas niveau, *Ethereum Virtual Machine code*, dans lequel on peut compter facilement les différentes étapes et un de plus haut niveau, *Solidity*, dont les instructions sont traduites dans le premier).

En 2016, un énorme contrat intelligent a eu lieu sur Ethereum (en réalité, le premier, et plus ou moins la vitrine commerciale d'Ethereum) : plus de 150 M\$ y ont été investis. Une fois validé, il n'y a aucune manière de modifier un contrat, ce qui veut dire qu'il doit être parfait. Évidemment, ça n'a pas été le cas, et quelqu'un a pu se servir d'un bug pour voler 50 M\$. La communauté autour d'Ethereum a alors réagi (en utilisant des institutions sociales hors de la chaîne de blocs) et a proposé de ré-écrire le passé : inspecter une à une toutes les transactions et modifier celles qui posent problème, en mettant des faux horodatages, et en resignant tout à chaque étape. De fait, la communauté s'est retrouvé à devoir choisir entre deux registres, l'un correspondant à l'histoire, l'autre falsifié mais correspondant aux intentions des personnes socialement bien considérées, et individuellement, chaque noeud du réseau a du choisir quelle histoire il préférait. Il existe encore aujourd'hui deux versions d'Ethereum, une dans laquelle les voleurs ont volé, une dans laquelle le contrat a été bien écrit⁹.

On voit donc qu'à la froide perfection des contrats intelligents, on a substitué autre chose.

- je suis obligé de mentionner *Tezos*, qui résout ce problème en intégrant des mécanismes d'auto-modification (en gros, les noeuds peuvent voter pour modifier le protocole), intégrant donc des institutions sociales dans le fonctionnement même de la chaîne de bloc (il faut imaginer de temps en temps, un bloc qui change les règles du jeu — c'est un classique des jeux à boire). De plus, Tezos a un intérêt fort pour la vérification des contrats intelligents, ce qui explique qu'ils financent indirectement beaucoup de l'écosystème *Ocaml* et *Coq*, ou qu'ils embauchent des personnes de ma communauté scientifique.

5.4 OPTIMISATION COMBINATOIRE

9. Et donc, pour tout ce qui n'a pas divergé, les mêmes transactions et donc les mêmes comptes des deux côtés. Combien valent ces unités monétaires qui se sont fait dédoubler du jour au lendemain ?

Deuxième partie

Annexes

Compléments de logique

Formules, prédicat, relation, quantificateurs, connecteurs

Définition 10 (cardinal). Le *cardinal* d'un ensemble fini est le nombre d'éléments de cet ensemble. Si E est un ensemble, on le note

$$\text{Card}(E).$$

On ne cherchera pas à définir le cardinal d'un ensemble infini.

Proposition 12. Soient E et F deux ensembles finis. On a :

$$\text{Card}(E) + \text{Card}(F) == \text{Card}(E \cup F) + \text{Card}(E \cap F).$$

Compléments sur le langage C

On a besoin de manipuler des tableaux — pour le Chapter 2 et le Chapter 3. On introduit donc les constructions syntaxiques nécessaires en C et donne leur sémantique. On introduit aussi des fonctions permettant de mesurer le temps d'exécution.

B.1	Tableaux : la pratique	164
	Syntaxe	164
	Appels fonctionnels	165
B.2	Remplir un tableau aléatoirement	166
B.3	Mesurer le temps d'exécution d'une fonction	166
B.4	Traduire un algorithme en C	166

Le langage C est *standardisé*, c'est-à-dire qu'il est défini par des standards, édités en l'occurrence par l'ISO, l'Organisation Internationale de Normalisation. Le plus récent s'appelle « ISO/IEC 9899:2018 Information technology — Programming languages — C », publié en juin 2018, et plus connu sous le nom de C17. Il décrit ce que sont des programmes valides, et comment ces programmes doivent être exécutés.¹ Ce standard est disponible, mais n'est pas gratuit (et cher: 198 francs suisses...); néanmoins, on peut trouver facilement sur le web le dernier brouillon tel qu'il a été voté par le comité de standardisation. C'est un document de 515 pages qui nécessite une bonne connaissance du C et de la compilation pour être compris.

Il est assez rare de voir du C correct, en particulier suivant les dernières normes : on voit souvent des mélanges venant du C++ (un langage de programmation différent) ou d'anciennes versions (en particulier sur [stackoverflow](#)). Un bon livre sur le C moderne est (GUSTEDT 2020).

B.1 TABLEAUX : LA PRATIQUE

Syntaxe

En C, un tableau a exactement la même définition que celle que nous avons adopté dans tout ce cours : c'est une série de cases contiguës de la mémoire, chacune pouvant prendre une valeur différente. Néanmoins, une contrainte existe, plus forte que ce que nous avons explicitement dit jusqu'à présent : chaque case doit avoir la même taille, autrement dit, le même type.

Aussi, une déclaration de tableau en C doit contenir non seulement la longueur du tableau, mais aussi le type de ces cases : on aura donc des tableaux de type `int` et de longueur 4, c'est-à-dire 4 cases consécutives de la mémoire de type `int`, par exemple le tableau

6	-8	35	42
tab[0]	tab[1]	tab[2]	tab[3]

On le déclare en C par :

```
1 int tab[4] = {
2     [0] = 6 ,
3     [1] = -8,
4     [2] = 35,
5     [3] = 42,
6 };
```

ou encore, de manière plus condensée,

```
1 int tab[4] = {6, -8, 35, 42};
```

On préférera si possible la première écriture, plus explicite. En C, comme dans ce cours, les tableaux commencent à 0² : ainsi, les cases d'un tableau `tab[4]` de quatre éléments s'appellent

`tab[0], tab[1], tab[2], tab[3]`

ce qui, là encore, correspond aux conventions adoptées dans le cours. On manipule ensuite chaque case comme n'importe quelle variable. Ainsi, toujours pour ce même tableau, on peut écrire :

1. Néanmoins des programmes invalides peuvent être exécutés par certaines plateformes, et des programmes dont l'exécution est laissée indéfinie peuvent avoir des plateformes définissant leur exécution. Le standard n'est qu'un *minimum* pour mériter de s'appeler C.

2. Au-delà de mes opinions grandiloquentes sur la question, c'est la raison principale pour laquelle j'ai adopté cette convention dans ce cours.

```

1 tab[2] = 3;
2 if (tab[2] == 1) {
3     tab[3] = 3;
4 } else {
5     tab[0] = -1;
6 }
```

Mais surtout, on peut se servir d'une variable de boucle pour parcourir tous les éléments d'un tableau. Les indices possibles d'un tableau peuvent être contenus dans une variable de type `size_t`:

```

1 for(size_t i = 0; i < 4; i++) {
2     tab[i] = tab[i] + 1;
3 }
```

Le type `size_t` peut contenir un entier positif compris entre 0 et `SIZE_MAX`. Sur ma machine, `SIZE_MAX` vaut:

```
1 printf("%lu", SIZE_MAX);
```

```
18446744073709551615
```

ce qui signifie qu'un tableau peut avoir au plus ce nombre de cases. Le standard C ne spécifie pas cette longueur.

Appels fonctionnels

Si on veut passer un tableau comme argument à une fonction, il suffit de lui donner son nom: `f(tab)`; applique bien la fonction `f` au tableau `tab`. Néanmoins, on fait rarement ainsi: la fonction ainsi appelée ne peut pas connaître la longueur du tableau. En général, donc, les fonctions demandent, en plus du tableau, qu'on leur passe aussi la longueur en argument. Donc, pour déclarer que la fonction `f` prend comme argument un tableau d'`int` d'une longueur quelconque et renvoie un `int`, on écrira:

```
1 int f(size_t longueur, int tab[longueur]);
```

c'est-à-dire qu'en plus du tableau `tab`, on se donne une longueur. Ensuite, si `tab` est de longueur 4, on appellera la fonction par l'instruction `f(4,tab)`.

Une difficulté est présente dans cet usage: en effet, les tableaux sont toujours passés en [appel par référence](#), ce qui veut dire qu'une fonction qu'on appelle sur un tableau peut modifier ledit tableau; ainsi:

```

1 int changement(size_t longueur, int tab[longueur]){
2     tab[0] = 1;
3     return 0;
4 }
5
6 int main() {
7     int tab[3] = {
8         [0] = 0,
9         [1] = 1,
10        [2] = 2,
11    };
12    printf("%d\n",tab[0]);
13    changement(3,tab);
14    printf("%d\n",tab[0]);
15    return 0;
16 }
```

renvoie

```
0
1
```

Exercice 146. Écrire une fonction qui affiche le contenu d'un tableau d'**int**.

Correction:

```
1 int print_tableau(size_t longueur, int tab[longueur]){
2     for(size_t i = 0; i < longueur; i++){
3         printf("%d ",tab[i]);
4     }
5     printf("\n");
6     return 0;
7 }
```

☺

B.2 REMPLIR UN TABLEAU ALÉATOIUREMENT

Pour faire des tests, on veut remplir aléatoirement des tableaux, pour pouvoir les trier et mesurer le temps de tri.

Pour faire cela, on va créer une fonction **tableau_aleatoire** prenant en entrée un tableau et sa taille et renvoyant 0, si elle arrive à le remplir avec des valeurs aléatoires, 1 sinon.

```
1 int tableau_aleatoire(size_t longueur, int tab[longueur]){
2     srand(time(NULL));
3     for (size_t i = 0; i < longueur; i++) {
4         tab[i] = rand();
5     }
6     return 0;
7 }
```

B.3 MESURER LE TEMPS D'EXÉCUTION D'UNE FONCTION

La fonction **clock_t clock(void)** de la librairie **time.h** mesure le nombre de tics d'horloge depuis le lancement du programme. Il suffit donc de récupérer deux fois ce nombre, avant et après l'exécution d'une certaine fonction **void f(void)** pour pouvoir récupérer le nombre de tics d'horloge utilisés par la fonction. On le divise ensuite par le nombre de tics d'horloge par seconde (qui est la constante **CLOCKS_PER_SEC**) pour obtenir le nombre de secondes pris par ce calcul.

```
1 clock_t t;
2 t = clock();
3 f();
4 t = clock() - t;
5 double temps = ((double)t)/CLOCKS_PER_SEC;
6
7 printf("%f secondes\n", temps);
```

B.4 TRADUIRE UN ALGORITHME EN C

Tous les algorithmes que l'on a écrit en pseudo-code insistent sur les entrées et les sorties. C'est aussi le cas d'un programme écrit en C: chaque fonction spécifie dans son type le type et des noms pour ses entrées ainsi que sa sortie.

Graphes de données expérimentales

On s'intéresse à comment présenter des données expérimentales. C'est une compétence essentielle.

Présenter des données expérimentales est un art compliqué.

Examens

Plein d'examens!

D.1	2020→2021 Semestre 1	170
	Sommes	170
	Maximum	172
	Emacs	174
D.2	2020→2021 Semestre 2	178
	Sommes partielles	178
	Tri rapide de Hoare	181
D.3	2021→2022 Semestre 1 Partiel	184
D.4	2021→2022 Semestre 1 Examen	186
	Invariant de boucle	186
	Implémenter des listes chaînées	187
D.5	2021→2022 Semestre 2 Partiel	191
	Bibliothèque	191
	Typographier un paragraphe	193
D.6	2021→2022 Semestre 2 Examen	193
	Sudoku	193
	Bibliothèque	194

D.1 2020→2021 SEMESTRE I

Sommes

Considérons l'algorithme suivant :

Entrées: un entier N

```

1 Inversion(N)
2   nouveau a
3   a ← 0
4   pour i allant de 0 à N faire
5     a ← a + i
6   fin
7   retourner a
Sorties: un entier

```

Exercice 147. Donner un invariant de boucle pour la boucle allant de la ligne 4 à la ligne 6.

Correction: Soit i un entier, $0 \leq i < N$. Au début de la i -ème boucle, la variable a contient la somme des entiers entre 0 et i (exclu), soit :

$$a == \sum_{j=0}^{i-1} j.$$

☺

Exercice 148. Donner un algorithme prenant en entrée un tableau d'entiers et calculant la somme des éléments du tableau.

Correction:

☺

Entrées: un tableau d'entiers $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N

```

1 Somme((\text{TAB}[i])_{0 \leq i < N})
2   nouveau a
3   a ← 0
4   pour i allant de 0 à N faire
5     a ← a + TAB[i]
6   fin
7   retourner a
Sorties: un entier

```

Exercice 149. L'exécuter sur

4	5	30	18
TAB[0]	TAB[1]	TAB[2]	TAB[3]

Correction: Les deux variables sont a et i , la variable de la boucle. On va représenter en une seule ligne la fin d'un tour de boucle et le début du suivant.

LIGNE	SUIVANTE	<i>a</i>	<i>i</i>
1	2		
2	3		
3	4	0	
4	5	0	0
5	6	4	0
6-4	5	4	1
5	6	9	1
6-4	5	9	2
5	6	39	2
6-4	5	39	3
5	6	57	3
6	7	57	
7	Retourner 57		

⊕

Maximum

Exercice 150. Écrivez un algorithme prenant un tableau d'entiers en entrée et renvoyant le plus grand entier du tableau.

Correction: On parcourt le tableau et on stocke le maximum des éléments qu'on a déjà rencontré. On compare cet élément à celui courant.

On renvoie une erreur si le tableau est vide. On aurait aussi pu affecter une valeur particulière (en général, on considère que la borne supérieure de l'ensemble vide est plus petite que tous les autres éléments possibles).

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N	
1	MaxTableau($(\text{TAB}[i])_{0 \leq i < N}$)
2	si $N == 0$ alors
3	retourner une erreur
4	sinon
5	nouveau a
6	$a \leftarrow \text{TAB}[0]$
7	pour i <i>allant de</i> 1 <i>à</i> N faire
8	si $\text{TAB}[i] > a$ alors
9	$a \leftarrow \text{TAB}[i]$
10	fin
11	fin
12	retourner a
13	fin
Sorties: un entier ou une erreur	



Exercice 151. Exécutez cet algorithme sur le tableau

4	5	30	18
TAB[0]	TAB[1]	TAB[2]	TAB[3]

Correction: Les variables sont a et i .

LIGNE	SUIVANTE	a	i
1	2		
2	5		
5	6		
6	7	4	
7	8	4	1
8	9	4	1
9	10	5	1
10	11	5	1
11-7	8	5	2
8	9	5	2
9	10	30	2
10	11	30	2
11-7	8	30	3
8	11	30	3
11	12	30	
12	Retourner 30		



Exercice 152. Écrivez un algorithme prenant une liste d'entiers en entrée et renvoyant le plus grand entier de la liste.

Correction: Ici aussi, on va considérer que calculer le maximum de la liste vide n'a pas de sens et provoque une erreur. On va donc procéder en deux temps et faire un algorithme récursif qui renvoie le maximum entre un entier fixé et le maximum de la liste, et encapsuler cet algorithme récursif.

Entrées:

- une liste d'entiers L;
- un entier a.

```

1 MaxListeRec(L, a)
2   si L == Λ alors
3     retourner a
4   sinon
5     (e, Q) ← L
6     si e > a alors
7       retourner MaxListeRec(Q, e)
8     sinon
9       retourner MaxListeRec(Q, a)
10    fin
11 fin

```

Sorties: un entier

Entrées: une liste d'entiers L.

```

1 MaxListe(L)
2   si L == Λ alors
3     retourner une erreur
4   sinon
5     (e, Q) ← L
6     retourner MaxListeRec(Q, e)
7 fin

```

Sorties: un entier ou une erreur



Exercice 153. Exécutez cet algorithme sur la liste



Correction: Tous les appels à d'autres procédures sont terminaux. **MaxListe** comme **MaxListeRec** ont deux variables, e et Q.

	LIGNE	SUIVANTE	e	Q
<code>MaxListe((4, (5, (30, (18, Λ)))))</code>	1	2		
	2	5		
	5	6	4	(5, (30, (18, Λ)))
	6	*	4	(5, (30, (18, Λ)))
<code>MaxListeRec((5, (30, (18, Λ))), 4)</code>	1	2		
	2	5		
	5	6	5	(30, (18, Λ))
	6	7	5	(30, (18, Λ))
	6	*	5	(30, (18, Λ))
<code>MaxListeRec((30, (18, Λ)), 5)</code>	1	2		
	2	5		
	5	6	30	(18, Λ)
	6	7	30	(18, Λ)
	7	*	30	(18, Λ)
<code>MaxListeRec((18, Λ), 30)</code>	1	2		
	2	5		
	5	6	18	Λ
	6	7	18	Λ
	7	*	18	Λ
<code>MaxListeRec(Λ, 30)</code>	1	2		
	2	3		
	3		Retourner 30	

⊕

Emacs

Dans l'éditeur de texte `emacs` (mais sans doute d'autres), le texte est représenté comme une liste. Ainsi, la phrase

C'est toi ?

pourra être représentée par la liste

`(C, (' , (e, (s, (t, (, (t, (o, (i, (, (? , Λ))))))))))`

Ainsi, une fonction d'affichage du texte prend un à un les caractères et les affiche individuellement les uns à la suite des autres.

Exercice 154. Justifiez le choix d'une structure de liste chaînée par opposition à un tableau.

Correction: Un tableau est de longueur fixe : on devrait soit changer de tableau à chaque fois qu'on va ajouter ou supprimer une lettre, soit fixer une longueur maximale pour une ligne. ⊕

Un éditeur de texte ne fait pas qu'afficher du texte : on édite aussi ! En particulier, une action d'édition courante consiste à supprimer des caractères ; une autre à en rajouter. On sait que supprimer et ajouter des éléments dans une liste, ailleurs qu'à sa tête, est coûteux.

Aussi, la représentation du texte est plus subtile : non seulement on représente le texte, mais aussi la position du curseur (l'endroit où l'utilisateur va supprimer, ajouter ou modifier des caractères) : on ne veut pas représenter la chaîne de caractères ci-dessus, mais plutôt

C'est t•oi ?

si le pointeur (qu'on note par un •) est situé entre le deuxième t et le o.

La solution a été prise de représenter chaque ligne de texte par deux listes: une représentant les lettres avant le curseur de droite à gauche, une autre représentant les lettres après le curseur, de gauche à droite. On représente donc la ligne ci-dessus par:

$$((t, (, (t, (s, (e, (' , (c, \Lambda))))))), \\ (o, (i, (, (? , \Lambda))))))$$

Une ligne est donc représentée par une paire de listes. On appelle cette structure un *zipper*.

On rappelle la notation $[a_1, \dots, a_n] := (a_1, (\dots, (a_n, \Lambda)))$, à n'utiliser que si vous êtes sûr·es de vous.

Exercice 155. Donner les zippers représentant les trois lignes suivantes:

Sal•ut
S•alut
•Salut

Correction: Sal•ut est représenté par

$$((l, (a, (s, \Lambda))), \\ (u, (t, \Lambda)))$$

S•alut par

$$((s, \Lambda), \\ (a, (l, (u, (t, \Lambda)))))$$

•Salut par

$$(\Lambda, \\ (s, (a, (l, (u, (t, \Lambda))))))$$

⊕

Exercice 156. Écrire un algorithme DécaleDroite prenant en entrée un zipper (représentant une ligne) et renvoyant, en sortie, un zipper représentant la même ligne, mais où le curseur a été déplacé d'un caractère vers la droite. Si le curseur est déjà à la fin de la ligne, il reste à sa place.

Correction:

Un zipper est, par définition, une paire de deux listes: on n'a donc pas à vérifier s'il est vide (contrairement à une liste) avant de le décomposer.

Entrées: un zipper Z. 1 DécaleDroite(Z) 2 (L, R) ← Z 3 si R == \Lambda alors 4 retourner (L, R) 5 sinon 6 (e, Q) ← R 7 retourner ((e, L), Q) 8 fin Sorties: un zipper
--

⊕

En général, une version de cet algorithme est appelée quand on appuie sur la touche → d'un clavier.

Exercice 157. Exécuter cet algorithme sur le zipper représentant Sal•ut.

Correction: Les variables sont L, R, e et Q.

LIGNE	SUIVANTE	L	R	e	Q
1	2				
2	3	(1, (a, (S, Λ)))	(u, (t, Λ))		
3	6	(1, (a, (S, Λ)))	(u, (t, Λ))		
6	7	(1, (a, (S, Λ)))	(u, (t, Λ))	u	(t, Λ)

On retourne (u, (1, (a, (S, Λ)))), (t, Λ))

⊕

Exercice 158. Écrire un algorithme DécaleGauche prenant en entrée un zipper (représentant une ligne) et retournant, en sortie, un zipper représentant la même ligne, mais où le curseur a été déplacé d'un caractère vers la gauche. Si le curseur est déjà au début de la ligne, il reste à sa place.

Correction:

Entrées: un zipper Z.

```

1 DécaleGauche(Z)
2   | (L, R) ← Z
3   | si L ==  $\Lambda$  alors
4   |   | retourner (L, R)
5   | sinon
6   |   | (e, Q) ← L
7   |   | retourner (Q, (e, R))
8   | fin
  
```

Sorties: un zipper

⊕

En général, une version de cet algorithme est appelée quand on appuie sur la touche ← d'un clavier.

Exercice 159. Exécuter cet algorithme sur le zipper représentant •Salut.

Correction: Les variables sont L, R, e et Q.

LIGNE	SUIVANTE	L	R	e	Q
1	2				
2	3	Λ	(S, (a, (1, (u, (t, Λ))))))		
3	4	Λ	(S, (a, (1, (u, (t, Λ))))))		

On retourne (Λ , (S, (a, (1, (u, (t, Λ)))))).

⊕

Exercice 160. Donner un algorithme Suppression supprimant le caractère à gauche du curseur. Si le curseur est déjà au début de la ligne, on ne fait rien.

Correction:

Entrées: un zipper Z.
1 Suppression (Z)
2 (L, R) \leftarrow Z
3 si L == Λ alors
4 retourner (L, R)
5 sinon
6 (e, Q) \leftarrow L
7 retourner (Q, R)
8 fin
Sorties: un zipper

⊕

En général, une version de cet algorithme est appelée quand on appuie sur la touche de suppression.

Exercice 161. Écrire un algorithme Transpose qui échange la lettre située à gauche du curseur avec la lettre située à droite du curseur.

Correction:

Entrées: un zipper Z.
1 Transpose (Z)
2 (L, R) \leftarrow Z
3 si L == Λ <i>ou</i> R == Λ alors
4 retourner (L, R)
5 sinon
6 (e, Q) \leftarrow L
7 (f, S) \leftarrow R
8 retourner ((f, Q), (e, S))
9 fin
Sorties: un zipper

⊕

Dans `emacs`, cette fonction est appelée par l'accord C-t (c'est-à-dire, appuyer en même temps sur la touche contrôle et la touche t). C'est une fonction très partique quan dn fait beaucoup de fautes de frappe.

Exercice 162. Quelle est la complexité de ces quatre opérations ?

Correction: Toutes ces opérations se font en temps constant, c'est-à-dire en complexité O(1). En effet, aucun des algorithmes que nous n'avons écrit n'utilise de boucle ou de récursion.

⊕

Exercice 163. Que pensez-vous de cette représentation des lignes de texte ?

Correction: Cette représentation est efficace du point de vue de la complexité: on peut faire beaucoup d'opérations courantes en temps constant.

⊕

D.2 2020→2021 SEMESTRE 2

Sommes partielles

On considère le problème suivant: on a des entiers et un entier K, et on veut en sélectionner un sous-ensemble égal à K (on peut voir ça comme une variante du problème du sac à dos: on a des objets, leur masse, et on veut sélectionner un sous-ensemble d'objets dont la masse totale vaut K).

Par exemple, si on veut sélectionner un sous-ensemble de total 8 de 1, 3, 5, 8, une solution peut être de sélectionner 3 et 5; une autre de sélectionner 8.

Exercice 164. Une première solution (peu efficace) consiste à énumérer tous les sous-ensembles possibles de l'ensemble d'entiers, calculer leur somme et comparer.

Écrire un algorithme appliquant cette stratégie.

Correction: Avant d'écrire l'algorithme, on doit commencer par se demander comment on va représenter les données, en l'occurrence, comment on va représenter l'ensemble d'entiers. Comme cette stratégie est peu efficace, on se doute que des listes comme des tableaux seront mauvais, on se laisse donc guider uniquement par la considération suivante: on va vouloir considérer des sous-ensembles de longueur arbitraire. On va donc utiliser des listes. Décomposons le problème. On va écrire:

1. un algorithme prenant en entrée une liste L et renvoyant toutes les sous-listes de L sous la forme d'une liste;
2. un algorithme calculant la somme d'une liste;
3. un algorithme appliquant une certaine procédure à tous les éléments d'une liste;
4. enfin, l'algorithme demandé.

L'algorithme **SousListes** prend une liste en entrée, calcule toutes les sous-listes de la queue, puis leur rajoute ou non la tête. Pour faciliter la lecture, on a distingué typographiquement les listes (en capitales, comme M) des listes de listes (en capitales en *gras de tableau noir*, comme M).

Entrées: une liste L

```

1 SousListes(L)
2   | si L == Λ alors
3   |   | retourner Λ
4   | sinon
5   |   | M ← Λ
6   |   | (a, L) ← L
7   |   | si L == Λ alors
8   |   |   | retourner ([a], ([], Λ))
9   |   | sinon
10  |   |   | N ← SousListes(L)
11  |   |   | tant que N ≠ Λ faire
12  |   |   |   | (N, N) ← N
13  |   |   |   | M ← ((a, N), (N, M))
14  |   |   | fin
15  |   | fin
16  |   | retourner M
17 | fin
```

Sorties: une liste de listes

Il ne nous reste qu'à calculer la somme de chacune de ces listes. Pour cela, on commence par calculer la somme d'une liste.

Et on utilise une procédure d'ordre supérieur, prenant en entrée une liste et une procédure à appliquer à chaque élément de la liste. Elle est connue sous le nom de **Map**.

Ainsi, **Map(L, Somme)** calcule une liste dont chaque élément est la somme de chaque liste de L. On pouvait programmer ça sans découper.



Entrées: une liste d'entiers L

```

1 Somme(L)
2   | si L == Λ alors
3   |   | retourner 0
4   | sinon
5   |   | (a, L) ← L
6   |   | retourner a + Somme(L)
7   | fin
```

Sorties: un entier

Entrées:

- une liste L d'éléments de type A ;
- une procédure f prenant des A en entrée, et ayant des B en sortie.

```

1 Map(L, f)
2   | si L == Λ alors
3   |   | retourner Λ
4   | sinon
5   |   | (a, L) ← L
6   |   | retourner ((a, f(a)), Map(L, f))
7   | fin
```

Sorties: une liste de paires d'éléments de type A et de type B

Entrées:

- une liste L d'entiers ;
- un entier K.

```

1 SousEnsemble(L, K)
2   | M ← Map(SousListes(L), Somme)
3   | tant que M == Λ faire
4   |   | ((a, b), M) ← M
5   |   | si b == K alors
6   |   |   | retourner a
7   |   | fin
8   | fin
9   | retourner une erreur
```

Sorties: une liste d'entiers ou une erreur

Exercice 165. L'exécuter pour trouver une somme de 8 parmi les sous-ensembles de 1, 3, 5, 8.

Correction: Si on cherche à exécuter `SousEnsemble(8, [1, 3, 5, 8])`, on va d'abord calculer toutes les sous-listes de $[1, 3, 5, 8]$, ce que l'on va faire en calculant récursivement les sous-listes de $[3, 5, 8]$, puis de $[5, 8]$, puis de $[8]$. En suivant les appels récursifs, on va donc avoir :

sous-listes de $[8]$ $[8], []$

sous-listes de $[5, 8]$ $[5, 8], [8], [5], []$

sous-listes de $[3, 5, 8]$ $[3, 5, 8], [5, 8], [3, 8], [8], [3, 5], [5], [3], []$

sous-listes de $[1, 3, 5, 8]$ $[1, 3, 5, 8], [3, 5, 8], [1, 5, 8], [5, 8], [1, 3, 8], [3, 8], [1, 8], [8], [1, 3, 5], [3, 5], [1, 5], [5], [1, 3], [3], [1], []$

Une fois qu'on applique la procédure calculant les sommes à cette longue liste, on obtient :

$([1, 3, 5, 8], 17), ([3, 5, 8], 16), ([1, 5, 8], 14), ([5, 8], 13), ([1, 3, 8], 12), ([3, 8], 11), ([1, 8], 9), ([8], 8), ([1, 3, 5], 9), ([3, 5], 8), ([1, 5], 6), ([5], 5), ([1, 3], 4), ([3], 3), ([1], 1), ([], 0)$

On supprime ensuite les différentes têtes de cette liste jusqu'à en trouver une de somme 8, ce qui va être la liste $[8]$. ⊕

Exercice 166. Une meilleure stratégie peut être de *backtracker*.

Écrire un algorithme appliquant cette stratégie.

Correction: On va faire ce qu'on fait à chaque fois qu'on backtracks : on considère chaque élément de la liste, si c'est possible de l'ajouter, on l'ajoute et on continue, sinon, on ne l'ajoute pas et on continue. Si aussi bien l'ajouter que ne pas l'ajouter était impossible, on revient en arrière. Ainsi, pour reprendre notre vocabulaire, les niveaux sont les éléments et il y a deux choix, prendre l'élément ou ne pas la prendre.

Entrées:

- une liste L d'entiers ;
- une liste L' d'entiers ;
- un entier K .

```

1 SousEnsembleRecursif(L, L', K)
2   | si  $K == 0$  alors
3   |   | retourner []
4   | sinon
5   |   | si  $K < 0$  ou  $L == \Lambda$  alors
6   |   |   | retourner une erreur
7   |   | sinon
8   |   |   |  $(a, L) \leftarrow L$ 
9   |   |   | si SousEnsembleRecursif( $L, (a, L')$ ,  $K - a$ ) est une erreur alors
10  |   |   |   | retourner SousEnsembleRecursif( $L, L'$ ,  $K$ )
11  |   |   | sinon
12  |   |   |   | SousEnsembleRecursif( $L, (a, L')$ ,  $K - a$ )
13  |   |   | fin
14  |   | fin
15  | fin

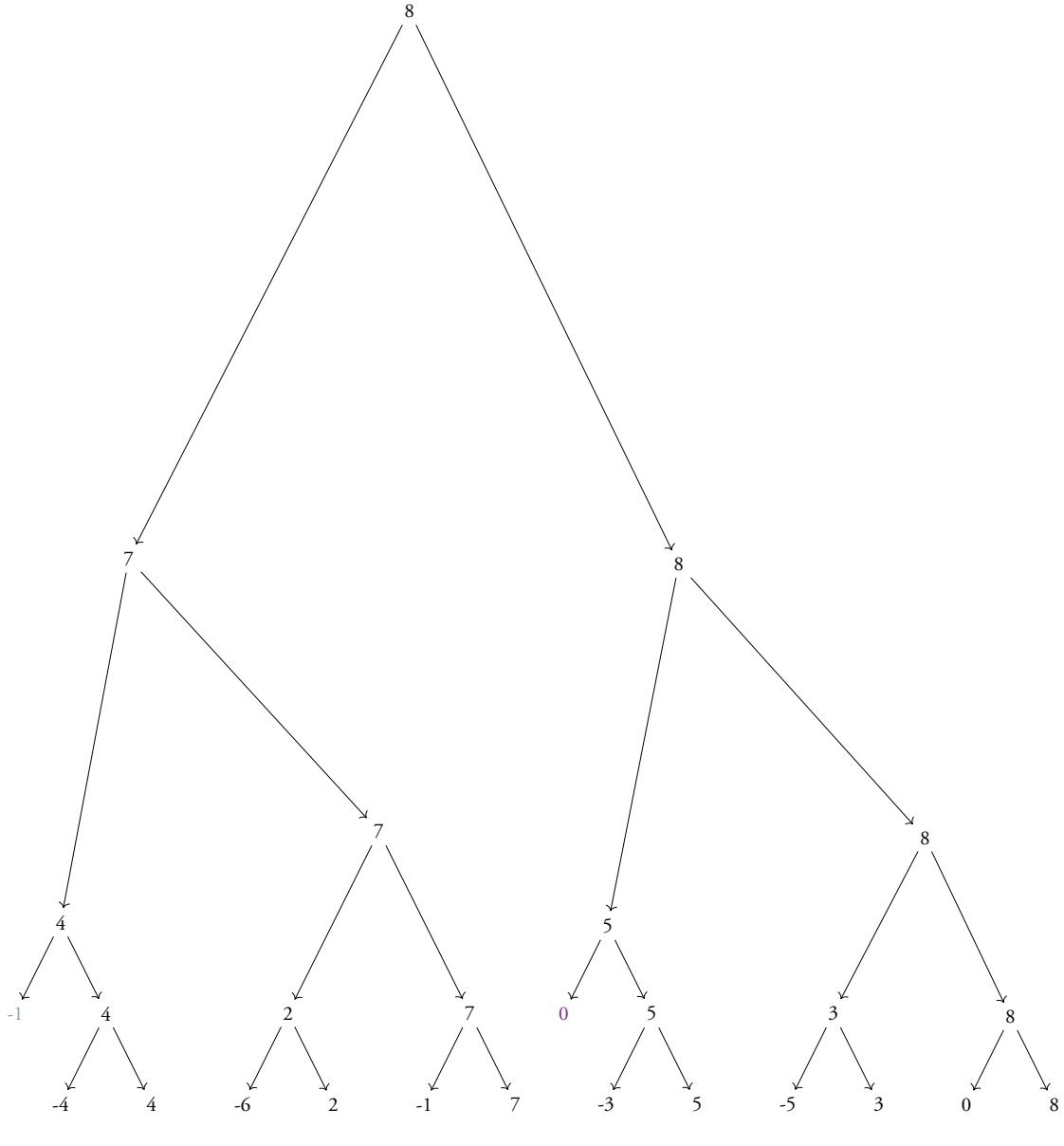
```

Sorties: une liste d'entiers ou une erreur

On exécute `SousEnsembleRecursif(L, Λ , K)`. ⊕

Exercice 167. Représentez l'arbre des sous-ensembles considérés, si on veut une somme de 8 parmi les sous-ensembles de 1, 3, 5, 8.

Correction: Chaque choix va être binaire : considérer ou non le prochain élément. On va représenter dans l'arbre l'objectif qu'il nous reste à atteindre.



⊕

Tri rapide de Hoare

Le *tri rapide* a été publié par Tony Hoare en 1962. C'est un tri récursif. Pour trier un tableau, il commence par choisir un élément (le *pivot*), puis séparer le tableau entre les éléments plus grands et les éléments plus petits que le pivot.

Ainsi, si 9 est le pivot, le tableau

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

sera découpé en deux tableau, l'un contenant 5, l'autre 30; 30. On va ensuite trier séparément les deux côtés.

Pour éviter de recopier les données du tableau, on va trier en place.

Exercice 168. Écrire un algorithme, qui, étant donné un tableau d'entiers $(\text{TAB}[i])_{0 \leq i < N}$, renvoie un tableau $(\text{TAB}'[i])_{0 \leq i < N}$ tel que :

- les éléments des deux tableaux soient identiques, dans un ordre différent ;
- si on appelle ℓ la position de $\text{TAB}[0]$ dans $(\text{TAB}'[i])_{0 \leq i < N}$, tous les éléments de $(\text{TAB}'[i])_{0 \leq i < N}$ d'indice inférieurs à ℓ sont inférieurs ou égaux à $\text{TAB}[0]$ et ceux d'indice supérieur lui sont supérieurs ou égaux.

Autrement dit, on a placé le pivot $\text{TAB}[0]$ en ℓ , et le tableau est partitionné selon le pivot.

Cet algorithme peut être écrit tel qu'il soit en temps linéaire.

Indication : on peut partir des deux côtés du tableau et stocker deux positions i et j . On incrémente i tant qu'on n'a pas trouvé un élément plus grand que le pivot : dès qu'on en a trouvé un, on décrémente j en cherchant un élément plus petit que le pivot. On échange les éléments dès qu'on a trouvé les deux.

Correction:

Entrées: un tableau d'entiers $(\text{TAB}[i])_{0 \leq i < N}$.

1 PartitionnePivot $((\text{TAB}[i])_{0 \leq i < N})$

2 $p \leftarrow \text{TAB}[0]$

3 $j \leftarrow N - 1$

4 **pour** i allant de 1 à j **faire**

5 **si** $\text{TAB}[i] > p$ **alors**

6 **tant que** $\text{TAB}[j] > p$ et $j > i$ **faire**

7 $j \leftarrow j - 1$

8 **fin**

9 $\text{TAB}[i] \leftrightarrow \text{TAB}[j]$

10 **fin**

11 **fin**

12 $\text{TAB}[0] \leftrightarrow \text{TAB}[j]$ **retourner** $(\text{TAB}[i])_{0 \leq i < N}$

Sorties: un tableau d'entiers



Exercice 169. Exécutez cet algorithme sur le tableau représenté ci-dessus, avec 9 comme pivot.

Exercice 170. Adaptez cet algorithme pour qu'il ne fasse le partitionnement qu'entre deux cases données du tableau, (le nouvel algorithme doit donc avoir comme entrée non seulement le tableau mais aussi une case de début — qui sera son pivot — et une case de fin).

Correction:



Exercice 171. En déduire un algorithme de tri.

Correction: Il suffit d'appeler de partitionner selon le pivot et d'appeler récursivement de chaque côté.



Exercice 172. Sur le tableau

9	30	5	30	18	2	90	32
TAB[0]	TAB[1]	TAB[2]	TAB[3]	TAB[4]	TAB[5]	TAB[6]	TAB[7]

représentez l'arbre des appels récursifs causés par cet algorithme (si, pour trier ce tableau, vous devez d'abord appeler la procédure avec d'autres paramètres en entrée, qui elle-même devra faire appel,... cela constitue un arbre).

Entrées:

- un tableau d'entiers $(\text{TAB}[i])_{0 \leq i < N}$;
- deux indices i et j

```

1 PartitionnePivot((TAB[i])0 ≤ i < N, i, j)
2   p ← TAB[i]
3   pour ℓ allant de  $i$  à  $j$  faire
4     si TAB[ℓ] > p alors
5       tant que TAB[j] > p et  $j > \ell$  faire
6         |   j ← j - 1
7       fin
8       TAB[ℓ] ↔ TAB[j]
9     fin
10   fin
11   TAB[i] ↔ TAB[j] retourner (TAB[i])0 ≤ i < N

```

Sorties: un tableau d'entiers**Entrées:**

- un tableau d'entiers $(\text{TAB}[i])_{0 \leq i < N}$;
- deux indices i et j

```

1 TriRapide((TAB[i])0 ≤ i < N, i, j)
2   p ← TAB[i]
3   k ← j pour ℓ allant de  $i$  à  $k$  faire
4     si TAB[ℓ] > p alors
5       tant que TAB[j] > p et  $k > \ell$  faire
6         |   k ← k - 1
7       fin
8       TAB[ℓ] ↔ TAB[k]
9     fin
10   fin
11   TAB[i] ↔ TAB[k]
12   ( $\text{TAB}[i]$ )0 ≤ i < N ← TriRapide((TAB[i])0 ≤ i < N, i, k)
13   ( $\text{TAB}[i]$ )0 ≤ i < N ← TriRapide((TAB[i])0 ≤ i < N, k, j)
14   retourner ( $\text{TAB}[i]$ )0 ≤ i < N

```

Sorties: un tableau d'entiers

D.3 2021→2022 SEMESTRE I PARTIEL

On notera les chaînes de caractères entre guillemets droits simples ''. La chaîne vide sera désignée par ''.

On suppose qu'on a une procédure Longueur qui prend en entrée une chaîne ce caractères et renvoie sa longueur (en tenant compte de tous les signes : lettres, ponctuation,...).

Ainsi, l'exécution de Longueur('Bonjour') vaudra 7.

Considérons l'algorithme suivant :

Entrées: un tableau de chaînes de caractères $(\text{TAB}[i])_{0 \leq i < N}$

- 1 **Algorithme** $((\text{TAB}[i])_{0 \leq i < N})$
- 2 **pour** i allant de 0 à N **faire**
- 3 **si** $\text{Longueur}(\text{TAB}[i]) > 10$ **alors**
- 4 **retourner** $\text{TAB}[i]$
- 5 **fin**
- 6 **fin**
- 7 **retourner** ''

Sorties: ???

Exercice 173. Quel est le type de la sortie?

Correction: La sortie est soit un élément du tableau (donc une chaîne de caractères), soit '' (qui est aussi une chaîne de caractères). C'est donc, dans tous les cas, une chaîne de caractères. ☺

Exercice 174. L'exécuter sur le tableau à trois cases $(\text{TAB}[i])_{0 \leq i < 3}$ dont les trois cases valent, respectivement

$$\begin{aligned}\text{TAB}[0] &:= \text{'Bonjour'}, \\ \text{TAB}[1] &:= \text{'les'}, \\ \text{TAB}[2] &:= \text{'masterant · es'}.\end{aligned}$$

Correction:

LIGNE	i
1	
2	0
3	0
6	0
2	1
3	1
6	1
2	2
3	2
4	On retourne 'masterant · es'

☺

Barème:

— $\frac{1}{3}$ pour la bonne sortie

Exercice 175. Que réalise cet algorithme?

Correction: Cet algorithme renvoie la première chaîne de caractères du tableau dont la longueur est supérieure à 10, et la chaîne vide si aucune case du tableau ne fait plus de dix caractères. ☺

Exercice 176. Adapter cet algorithme pour qu'il renvoie la plus longue chaîne de caractère du tableau.

Correction:



Entrées: un tableau de chaînes de caractères $(\text{TAB}[i])_{0 \leq i < N}$

```

1 PlusLongueChaîne((TAB[i])0 ≤ i < N)
2   ℓ ← 0
3   pour i allant de 0 à N faire
4     |   si Longueur(TAB[i]) > ℓ alors
5       |     |   u ← i
6       |     |   ℓ ← Longueur(TAB[i])
7     |   fin
8   fin
9   retourner TAB[u]
```

Sorties: une chaîne de caractère

Exercice 177. Adapter cet algorithme pour qu'il renvoie la liste des plus longues chaînes de caractère du tableau (c'est-à-dire que si deux chaînes sont aussi longues, on veut renvoyer les deux).

Correction:



Entrées: un tableau de chaînes de caractères $(\text{TAB}[i])_{0 \leq i < N}$

```

1 PlusLonguesChaînes((TAB[i])0 ≤ i < N)
2   ℓ ← 0
3   L ← Λ
4   pour i allant de 0 à N faire
5     |   si Longueur(TAB[i]) > ℓ alors
6       |     |   L ← [TAB[i]]
7       |     |   ℓ ← Longueur(TAB[i])
8     |   sinon
9       |     |   si Longueur(TAB[i]) ≡ ℓ alors
10      |       |     |   L ← TAB[i] :: L
11      |     |   fin
12    |   fin
13  fin
14 retourner L
```

Sorties: une liste de chaîne de caractère

Exercice 178. L'exécuter sur le tableau à trois cases $(\text{TAB}[i])_{0 \leq i < 3}$ dont les trois cases valent, respectivement

```

TAB[0] := 'Bonjour',
TAB[1] := 'les',
TAB[2] := 'masters'.
```

Correction:

LIGNE	<i>i</i>	ℓ	L
1			
2		0	
3	0		Λ
4	0	0	Λ
5	0	0	Λ
6	0	0	[‘Bonjour’]
7	0	6	[‘Bonjour’]
12	0	6	[‘Bonjour’]
13	0	6	[‘Bonjour’]
4	1	6	[‘Bonjour’]
5	1	6	[‘Bonjour’]
9	1	6	[‘Bonjour’]
11	1	6	[‘Bonjour’]
12	1	6	[‘Bonjour’]
13	1	6	[‘Bonjour’]
4	2	6	[‘Bonjour’]
5	2	6	[‘Bonjour’]
9	2	6	[‘Bonjour’]
10	2	6	[‘masters’, ‘Bonjour’]
11	2	6	[‘masters’, ‘Bonjour’]
12	2	6	[‘masters’, ‘Bonjour’]
13	2	6	[‘masters’, ‘Bonjour’]
4	3	6	[‘masters’, ‘Bonjour’]
14	On retourne [‘masters’, ‘Bonjour’]		

⊕

Exercice 179. Une chaîne de caractère est composée de caractères. Comment pouvez-vous représenter une chaîne de caractère par des structures de données que vous connaissez. Réfléchissez aux opérations naturelles pour de telles chaînes, et leur complexité.

D.4 2021→2022 SEMESTRE I EXAMEN

Invariant de boucle

Considérons l'algorithme suivant :

Entrées: un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de nombres

```

1 Mystère((TAB[i])0 ≤ i < N)
2   pour i allant de 0 à N faire
3     si TAB[N - (i + 1)] > 10 alors
4       TAB[N - (i + 1)] ← 10
5     fin
6   fin
7   retourner (TAB[i])0 ≤ i < N
```

Sorties: un tableau de nombres

Exercice 180. Que fait cet algorithme ?

Donnez un invariant de boucle pour le prouver.

Correction: Cet algorithme tronque à 10 les éléments du tableau d'entrée, c'est-à-dire que tous les éléments supérieurs à 10 sont remplacés par 10.

Pour trouver un invariant de boucle, on constate tout d'abord que la boucle parcourt le tableau en commençant par les cases de grand indice, donc l'invariant sera exprimé sur les cases plus grandes que $N - i$. De plus, on veut exprimer deux choses sur ces cases :

- si, dans le tableau d'entrée, elles sont inférieures à 10, alors elles sont identiques dans le tableau modifié;
- si, dans le tableau d'entrée, elles sont inférieures à 10, elles sont remplacées par 10 dans le tableau modifié.

Il nous faut donc donner un nom au tableau initial, avant toutes modifications. Disons $(\text{INITIAL}[i])_{0 \leq i < N}$ (il est de même taille que $(\text{TAB}[i])_{0 \leq i < N}$).

On a donc, au i -ème tour de boucle, que, pour toute case de $(\text{TAB}[i])_{0 \leq i < N}$ d'indice strictement plus grand que $N - (i + 1)$, la valeur de cette case est égale au minimum entre 10 et la valeur de la case du tableau initial de même indice. On peut noter cette propriété

$$(P_i) : \forall j, N - (i + 1) < j < N \implies \text{TAB}[j] \equiv \min(10, \text{INITIAL}[j])$$

⊕

Barème:

- l'algorithme est correctement décrit

Implémenter des listes chaînées

On suppose que la mémoire d'un ordinateur contient des cases qui peuvent chacune contenir une valeur, et qu'on sait distinguer les cases par des *adresses*: des numéros distincts pour chacune des cases. On voit bien dans ce cadre comment programmer des tableaux: pour un tableau $(\text{TAB}[i])_{0 \leq i < N}$ de longueur N , on choisit N cases non-utilisées contiguës, et il suffit de connaître la position de la première case pour accéder à toutes les autres. Ainsi, si on a accès à un langage de bas niveau ne possédant qu'une notion d'adresse, on peut construire un langage de plus haut niveau, plus abstrait, ayant une notion de tableau.

Pour programmer des listes chaînées, c'est un peu plus compliqué. En effet, tout le principe des listes chaînées est que les cases ne sont pas placées contigument, mais au contraire, que chaque chaînon de la liste contient une valeur et la position du prochain chaînon. On se demande comment construire des listes dans un langage ne possédant qu'une notion de tableaux (comme, par exemple, le C).

On va commencer par se donner un tableau dans lequel nous travaillerons: ainsi, tout ce que nous ferons se fera dans ce tableau, qui représentera, pour nous, l'intégralité de la mémoire (cela signifie que la longueur des listes chaînées que l'on voudra faire sera limitée par la taille de ce tableau). Appelons-le $(\text{MEM}[i])_{0 \leq i < N}$ et supposons que N est très grand. De plus, on va supposer qu'initialement, toutes les cases de $(\text{MEM}[i])_{0 \leq i < N}$ contiennent une valeur spéciale, •.

On va représenter chaque chaînon d'une liste de la manière suivante:

- on stocke la valeur de ce chaînon dans une case;
- on stocke l'adresse du prochain chaînon dans la case suivante.

De ce fait, on voit que stocker un chaînon va utiliser deux cases. On va systématiquement stocker les valeurs dans les cases paires, et l'adresse du prochain chaînon dans la case impaire suivante.

On va considérer qu'une liste est représentée par l'adresse de son premier chaînon.

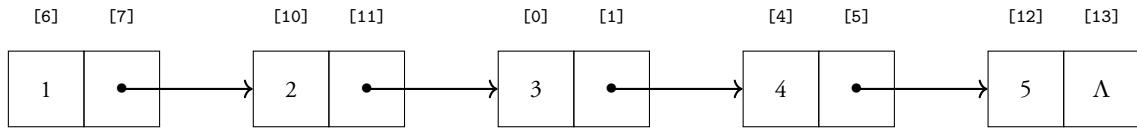
Exercice 181. Quelle liste est représentée par le tableau $(\text{MEM}[i])_{0 \leq i < N}$ dessiné ici et dont le premier chaînon est à l'adresse 6 :

3	4	•	•	4	12	1	10	•	•	2	0	5	•	•
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]

Écrire cette liste avec des cases et des flèches, avec des couples écrits :: et avec des [].

Correction: La liste commence en l'adresse 6, sa première valeur est donc 1. La case suivante, 7, contient donc l'adresse du prochain chaînon: 10. On peut continuer à suivre ainsi les chaînons, jusqu'à l'adresse 12, qui contient la valeur 5. La prochaine case contient •, on considère donc que c'est la fin de la liste.

On fait le choix ici de représenter au dessus des chaînons la position dans le tableau, mais c'est uniquement pour insister.



Représentée avec la syntaxe ::, cela donne la liste $1 :: 2 :: 3 :: 4 :: 5 :: []$, et avec celle en crochets $[1, 2, 3, 4, 5]$. Dans ce cas-là, ça n'a pas vraiment de sens de placer les adresses. D'une certaine manière, la représentation du tableau dans l'énoncé est la plus concrète, celle avec des chaînons et des flèches est un niveau d'abstraction au dessus, et celles finales sont les plus abstraites. ☺

Exercice 182. Écrire un algorithme d'accès qui prenne en entrée le tableau $(\text{MEM}[i])_{0 \leq i < N}$, l'adresse d'un premier chaînon et un entier k et qui retourne la $k^{\text{ème}}$ élément de la liste.

Correction: Comme on travaille sur une liste (structure récursive!), on peut partir d'une idée récursive: si on cherche le $0^{\text{ème}}$ élément de la liste commençant à l'adresse a , il nous suffit de renvoyer $\text{MEM}[a]$. Sinon, on cherche le $k - 1^{\text{ème}}$ élément de la liste commençant à l'adresse indiquée à la case d'adresse $a + 1$. Autrement dit:

Entrées:

- le tableau de la mémoire $(\text{MEM}[i])_{0 \leq i < N}$;
- une adresse a ;
- un entier k .

```

1 Accès(((MEM[i])0 ≤ i < N, a, k)
2   si k ≡ 0 alors
3     |   retourner MEM[a]
4   fin
5   si MEM[a + 1] ≡ • alors
6     |   retourner erreur
7   fin
8   retourner Accès((MEM[i])0 ≤ i < N, MEM[a + 1], k - 1)
```

Sorties: un élément ou une erreur

On pourrait tout-à-fait le faire itérativement. ☺

Exercice 183. L'exécuter sur le tableau $(\text{MEM}[i])_{0 \leq i < N}$ suivant

3	4	•	•	-3	12	1	10	12	0	2	0	5	•	•
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]

l'adresse 8 et l'entier 3.

Correction: Je me permets ici d'aller vite, et de représenter juste les appels successifs récursifs:

- Accès($(\text{MEM}[i])_{0 \leq i < N}$, 8, 3)
- Accès($(\text{MEM}[i])_{0 \leq i < N}$, 0, 2)
- Accès($(\text{MEM}[i])_{0 \leq i < N}$, 4, 1)
- Accès($(\text{MEM}[i])_{0 \leq i < N}$, 12, 0)

Et donc, on renvoie 5. ☺

On veut maintenant pouvoir créer un nouveau chaînon. Pour cela, on veut pouvoir prendre deux cases inutilisées de $(\text{MEM}[i])_{0 \leq i < N}$, ce qui nécessite de savoir où est la première case non-utilisée. On va donc stocker en mémoire l'adresse de la première case non-utilisée, et quand on voudra ajouter un chaînon, on écrira à cette adresse. De même, quand on détruit une case, on veut la rendre dans le pool

des cases non-utilisées. Cela veut dire qu'on doit, en rendant une nouvelle case libre, écrire dedans l'ancienne adresse de la première case inutilisée.

Ainsi, les cases libres auront la structure suivante :

- une plage continue de cases n'ayant jamais utilisées, à la fin du bloc de mémoire ;
- une liste chaînée de cases vides, dont la dernière case contient l'adresse de la première case de la plage ;
- une variable contenant l'adresse de la première case de la liste chaînée (si elle est non-vide), ou de la première case de la plage.

Exercice 184. Écrire un algorithme d'ajout d'un élément qui prenne en entrée le tableau $(\text{MEM}[i])_{0 \leq i < N}$, l'adresse d'un premier chaînon, l'adresse de la première case vide, un entier k , un élément a et qui ajoute a en $k^{\text{ème}}$ position dans la liste.

On veut que cet algorithme renvoie l'adresse du premier chaînon (de la liste) et l'adresse de la première case vide.

Correction : On a deux problèmes à régler : premièrement, on cherche à prendre la première case vide et trouver l'adresse de la deuxième case vide. Il y a deux possibilités : soit la case immédiatement après la première case vide contient un \bullet , auquel cas, cette case fait partie d'une plage vide, et la prochaine case vide est la suivante ; soit cette case contient une adresse et la prochaine case vide est la case pointée par cette adresse. Deuxièmement, on veut rajouter un élément à une liste. Encore une fois, on peut penser récursivement : ajouter un élément en tête de liste est facile.

Entrées :

- le tableau de la mémoire $(\text{MEM}[i])_{0 \leq i < N}$;
- une adresse ℓ ;
- une adresse v ;
- un entier k ;
- un élément a

```

1 Ajout(((MEM[i])0 ≤ i < N, ℓ, v, k, a)
2   si k ≡ 0 alors
3     | MEM[v] ← a
4     | ℓ' ← v
5     | si MEM[v + 1] ≡ • alors
6     |   | v ← v + 2
7     | sinon
8     |   | v ← MEM[v + 1]
9     | fin
10    | MEM[ℓ' + 1] ← ℓ
11    | retourner (ℓ', v)
12  fin
13  si MEM[ℓ + 1] ≡ • alors
14    | retourner erreur
15  fin
16  (ℓ', v') ← Ajout((MEM[i])0 ≤ i < N, MEM[ℓ + 1], v, k - 1, a)
17  retourner (ℓ, v')
```

Sorties : un couple de deux adresses ou une erreur

⊕

Exercice 185. Écrire un algorithme de suppression d'un élément qui prenne en entrée le tableau $(\text{MEM}[i])_{0 \leq i < N}$, l'adresse d'un premier chaînon, l'adresse de la première case vide, un entier k et qui supprime l'élément en $k^{\text{ème}}$ position dans la liste.

On veut que cet algorithme renvoie l'adresse du premier chaînon (de la liste) et l'adresse de la première case vide.

Correction:



Entrées:

- le tableau de la mémoire $(\text{MEM}[i])_{0 \leq i < N}$;
- une adresse ℓ ;
- une adresse v ;
- un entier k ;
- un élément a

```

1 Suppression(((\text{MEM}[i])_{0 \leq i < N}, \ell, v, k, a)
2   | si  $k \equiv 0$  alors
3   |   |  $\ell' \leftarrow \text{MEM}[\ell + 1]$ 
4   |   | si  $\ell' \equiv \bullet$  alors
5   |   |   | retourner erreur
6   |   | sinon
7   |   |   |  $\text{MEM}[\ell + 1] \leftarrow v$ 
8   |   |   | retourner  $(\ell', \ell)$ 
9   |   | fin
10  |   | fin
11  |   | si  $\text{MEM}[\ell + 1] \equiv \bullet$  alors
12  |   |   | retourner erreur
13  |   | fin
14  |   |  $(\ell', v') \leftarrow \text{Suppression}((\text{MEM}[i])_{0 \leq i < N}, \text{MEM}[\ell + 1], v, k - 1, a)$ 
15  |   |  $\text{MEM}[\ell + 1] \leftarrow \ell'$ 
16  |   | retourner  $(\ell, v')$ 
```

Sorties: un couple de deux adresses ou une erreur

Exercice 186. Écrire un algorithme prenant en entrée l'adresse de la première case vide, un élément a et renvoyant l'adresse d'une liste ne contenant que a , et l'adresse de la première case vide.

Au bout d'un moment, à force de faire des ajouts et des suppressions, la mémoire aura une structure particulièrement compliquée.

Supposons par exemple que la liste représente une file d'attente, et qu'elle est remplie et vidée selon les règles suivantes :

- la liste est remplie à un rythme régulier, avec des numéros consécutifs. Ainsi, le premier élément à arriver sera le 0, puis le 1, ... ;
- il y a deux processus qui vident la liste :
 - le premier, le processus A traite et efface le premier élément de la liste. Traiter un élément prend autant de temps que prennent quatre éléments pour arriver.
 - le second, le processus B, traite et efface le premier élément de la liste à contenir un numéro pair. Traiter un élément prend autant de temps que prennent deux éléments pour arriver.

Si les deux processus sont libres en même temps, c'est le processus A qui traite en premier.

Ainsi, la liste va prendre les valeurs suivantes :

1. []
2. [0] (un premier élément arrive)
3. [0, 1] (un second arrive, A commence à traiter 0)

- | | |
|------------------------|---|
| 4. [0, 1, 2] | (un troisième arrive) |
| 5. [0, 1, 2, 3] | (un quatrième arrive, B commence à traiter 2) |
| 6. [1, 2, 3, 4] | (un cinquième arrive, A commence à traiter 1) |
| 7. [1, 3, 4, 5] | (un sixième arrive, B commence à traiter 4) |
| 8. [1, 3, 4, 5, 6] | (un septième arrive) |
| 9. [3, 4, 5, 6, 7] | (un huitième arrive, A commence à traiter 3, B à traiter 6) |
| 10. [3, 4, 5, 6, 7, 8] | (un neuvième arrive) |
| 11. [3, 4, 5, 7, 8, 9] | (un dixième arrive, B commence à traiter 8) |

Représenter l'état de la mémoire (donc l'état du tableau $(\text{MEM}[i])_{0 \leq i < N}$ et l'adresse du premier chaînon) à l'étape 11, en supposant qu'à l'étape 1, le tableau $(\text{MEM}[i])_{0 \leq i < N}$ était intégralement vide.

On veut simplifier la mémoire de temps en temps, en ré-écrivant chaque liste de manière à ce que ses chaînons soient adjacents et dans l'ordre.

Exercice 187. Écrire un algorithme de suppression d'un élément qui prenne en entrée le tableau $(\text{MEM}[i])_{0 \leq i < N}$, l'adresse d'un premier chaînon, l'adresse de la première case vide, et qui déplace la liste de manière à ce que tous ces chaînons soient adjacents et dans l'ordre.

On veut que cet algorithme renvoie l'adresse du premier chaînon (de la liste) et l'adresse de la première case vide.

D.5 2021→2022 SEMESTRE 2 PARTIEL

Bibliothèque

La bibliothèque planifie son déménagement : elle veut placer ses livres en le moins de places possible. Plus précisément, on suppose que la bibliothèque possède n livres, que l'on nomme b_0, b_1, \dots, b_{n-1} . On note ℓ_i la largeur du livre b_i et h_i sa hauteur. Les livres doivent être rangés dans l'ordre donné sur des étagères qui sont toutes de largeur L .

Exercice 188. On suppose que tous les livres ont la même hauteur h .

Donner un algorithme glouton donnant une répartition des livres par étagères.

Correction : Un algorithme glouton fait le meilleur choix possible sur le coup, puis passe au choix d'après. Dans le cas du rangement de la bibliothèque, cela veut dire prendre chaque livre dans l'ordre, et pour chaque livre faire le choix suivant :

- soit le placer sur l'étagère courante ;
- soit ouvrir une nouvelle étagère.

On fait le choix optimal localement, c'est-à-dire qu'on ouvre une nouvelle étagère uniquement si le livre ne rentre pas sur l'étagère courante.

Si jamais un livre est plus large qu'une étagère (et donc ne rentrera nulle part), on renvoie une erreur. On commence par écrire un premier algorithme qui remplit une étagère.

⊕

Exercice 189. Justifier les structures de données utilisées pour représenter les entrées et les sorties.

Correction : On a choisi de représenter l'entrée comme une liste de livres, chaque livre étant représenté par une paire composée de sa hauteur et de sa largeur : on aurait pu utiliser un tableau, en effet, on a jamais besoin de considérer les livres autrement que séquentiellement.

On a choisi de représenter chaque étagère par la liste des livres qu'on a placé dessus (vu qu'on ne sait pas a priori combien de livres on placera sur chaque étagère), et donc l'ensemble des étagères par une liste de listes de livres.

⊕

Exercice 190. L'exécuter sur des livres de hauteur 1, et de largeur (dans l'ordre) 15, 18, 6, 20, 3, 5, 8, 30, 12 ; pour des étagères de largeur 30.

Entrées:

- une liste Livres de couples d'entiers;
- un entier L;
- un entier dispo.

```

1 RemplirÉtagère(Livres, L, dispo)
2   | si Livres ≡ Λ alors
3   |   | retourner []
4   | fin
5   | (h, l) :: Livres ← Livres
6   | si l > L alors
7   |   | retourner une erreur
8   | fin
9   | si l ≤ dispo alors
10  |   | (E, R) ← RemplirÉtagère(Livres, L, dispo - L)
11  |   | retourner ((h, l) :: E, R)
12  | fin
13 retourner [], Livres

```

Sorties: un couple de listes de couples d'entiers ou une erreur

Entrées:

- une liste Livres de couples d'entiers;
- un entier L.

```

1 Étagères(Livres, L)
2   | si Livres ≡ Λ alors
3   |   | retourner []
4   | fin
5   | (E, Livres) ← RemplirÉtagère(Livres, L, L)
6   | retourner E :: Étagères(Livres, L)

```

Sorties: une liste de listes de couples d'entiers ou une erreur

Correction: Cela va produire la liste suivante :

$$[[[(1, 15)]; [(1, 18; (1, 6))]; [(1, 20; (1, 3); (1, 5))]; [(1, 8)]; [(1, 30)]; [(1, 12)]]]$$

⊕

Exercice 191. Montrer que cet algorithme est optimal au sens suivant: il utilise le moins possible d'étagères.

Correction: Cet algorithme est optimal pour un seul livre: en effet, il le place seul sur une étagère. Supposons qu'il soit optimal pour un nombre fixé de livres. Si on en a un de plus, alors il n'utilise une autre étagère que s'il ne peut pas faire autrement.

⊕

Exercice 192. Supposons maintenant que les livres ont des hauteurs différentes. Notre but n'est plus d'optimiser le nombre d'étagères, mais de réussir à mettre le plus d'étagères possibles sur chaque armoire (et donc le plus de livres possibles!).

On définit donc l'*encombrement* d'une étagère comme la hauteur de son plus grand livre (qui va donc déterminer la place verticale que va prendre l'étagère), et l'encombrement total comme la somme des encombrements des étagères (qui dont va être la hauteur totale utilisée par les livres).

Montrer que l'algorithme glouton précédent n'est pas optimal pour cette mesure : donner un exemple où il amène à encombrer plus que nécessaire.

Exercice 193. Proposer un algorithme qui soit optimal pour cette mesure. Donner sa complexité.

Exercice 194. Changeons encore de point de vue : supposons maintenant que tous les livres ont la même hauteur, et que la hauteur totale disponible est fixée. Autrement dit, c'est la largeur des étagères qu'on cherche à minimiser.

Reprenons, on veut ranger les livres en k étagères (et k est fixé) de largeur L , et on veut minimiser L . Donner un algorithme récursif qui résolve ce problème.

Tout ça pour dire que ces différentes notions d'optimalité sont tout aussi légitimes.

Typographier un paragraphe

En typographie, on doit mettre en forme des paragraphes : c'est-à-dire qu'on a du texte, et qu'on doit décider où couper le paragraphe en plusieurs lignes. On va, pour simplifier, supposer qu'on n'a pas de césures, c'est-à-dire qu'on ne coupe jamais une ligne au milieu d'un mot. On suppose qu'on a donc des mots, chacun d'une certaine longueur (on mesure la longueur en caractères, et on ne préoccupe pas de la chasse) ; et que de même, les lignes ont une longueur M fixée, et qu'entre chaque mot et à la fin de la ligne (autant qu'il en faut pour remplir), il y a des espaces.

On veut évidemment mettre en forme les paragraphes de manière optimale. On doit donc trouver une mesure pour cette optimalité.

Exercice 195. On commence par se dire qu'on veut minimiser le nombre total d'espaces à la fin des lignes (donc les caractères de perdus), sauf sur la dernière. Donner un algorithme glouton optimal pour ce critère.

Exercice 196. Justifier les structures de données pour les entrées et les sorties.

Exercice 197. L'expérience de quelques siècles de typographie a montré que, globalement, c'est assez moche : l'algorithme glouton a tendance à produire des lignes avec trop d'espaces. L'expérience a montré que le cube du nombre d'espaces à la fin d'une ligne était une bonne mesure de sa laideur.

On veut donc minimiser la somme des cubes des nombres d'espaces à la fin des lignes (sauf la dernière).

Est-ce que l'algorithme glouton précédent est optimal pour cette mesure ?

Exercice 198. Proposer un algorithme pour résoudre ce problème, avec cette mesure.

D.6 2021→2022 SEMESTRE 2 EXAMEN

Sudoku

Une *grille de sudoku pleine* est un carré 9×9 , subdivisé en neuf carrés 3×3 remplis de nombres entre 1 et 9 et telle que :

- aucune ligne
- aucune colonne
- aucun des petits carrés

ne contiennent deux fois le même chiffre.

Une *grille de sudoku valide* est un carré 9×9 , partiellement rempli de nombres entre 1 et 9, pouvant être complété en une grille de sudoku pleine.

Une *grille de sudoku gauche* est un carré 9×9 , partiellement rempli de nombres entre 1 et 9, sans qu'on sache si elle respecte les règles de constructions.

Exercice 199. Quelles structures de données peut-on utiliser et pourquoi ? Donner des exemples.

Exercice 200. Écrire un algorithme prenant en entrée une grille de sudoku valide, les coordonnées d'une de ses cases, et qui renvoie la ou les valeurs pouvant la remplir si on complète la grille.

Exercice 201. Décrire une stratégie permettant de remplir une grille de sudoku valide.

Bibliothèque

La bibliothèque planifie son déménagement : elle veut placer ses livres en le moins de places possible. Plus précisément, on suppose que la bibliothèque possède n livres, que l'on nomme b_0, b_1, \dots, b_{n-1} . On note ℓ_i la largeur du livre b_i et h_i sa hauteur. Les livres sont donnés dans un ordre et doivent être rangés dans l'ordre donné sur des étagères qui sont toutes de largeur L .

Dans la suite, on appellera étagère une planche où on pose les livres.

Exercice 202. On suppose que tous les livres ont la même hauteur h .

Donner un algorithme glouton donnant les livres, dans l'ordre, que l'on doit poser sur la première étagère. Prenez en compte l'exercice 3 pour choisir les entrées et les sorties.

Exercice 203. L'exécuter sur des livres de hauteur 1, et de largeur (dans l'ordre) 15, 6, 18, 20, 3, 5, 8, 30, 12 ; pour des étagères de largeur 30.

Exercice 204. En déduire un algorithme glouton donnant comment les livres doivent être placés sur toutes les étagères.

Exercice 205. Montrer que, si on a placé les livres en utilisant le moins possible d'étagères, et qu'on rajoute un livre de manière gloutonne, alors le nouveau placement des livres utilise encore le moins possible d'étagères.

En déduire que l'algorithme glouton est optimal au sens suivant : il utilise le moins possible d'étagères.

Exercice 206. Supposons maintenant que les livres ont des hauteurs différentes. Notre but n'est plus d'optimiser le nombre d'étagères, mais de réussir à mettre le plus d'étagères possibles sur chaque armoire (et donc le plus de livres possibles!).

On définit donc l'*encombrement* d'une étagère comme la hauteur de son plus grand livre (qui va donc déterminer la place verticale que va prendre l'étagère), et l'encombrement total d'un rayonnage (le meuble contenant plusieurs étagères) comme la somme des encombremens des étagères (qui dont va être la hauteur totale utilisée par les livres).

Montrer que l'algorithme glouton précédent n'est pas optimal pour cette mesure : donner un exemple où il amène à encombrer un rayonnage plus que nécessaire.

Exercice 207. Pour un livre donné $0 \leq i < n - 2$, donner une relation entre l'encombrement minimal entre le livre numéro i et le dernier livre, et l'encombrement minimal entre le livre numéro $i + 1$ et le dernier livre.

En déduire un algorithme qui soit optimal pour cette mesure. Donner sa complexité.

Exercice 208. Changeons encore de point de vue : supposons maintenant que tous les livres ont la même hauteur, et que la hauteur totale disponible est fixée. Toutes les étagères ont même largeur, mais inconnue : c'est la largeur des étagères qu'on cherche à minimiser.

Reprenons, on veut ranger les livres en k étagères (et k est fixé) de largeur L , et on veut minimiser L . Donner un algorithme récursif qui résolve ce problème.

Tout ça pour dire que ces différentes notions d'optimalité sont tout aussi légitimes.

Bibliographie

- Fabio ACERBI, éd. (2007). *Euclide. Tutte le opere*. Bompiani.
- Nicolas AUGER, Vincent JUGÉ, Cyril NICAUD et Carine PIVOTEAU (2018). « On the Worst-Case Complexity of TimSort ». 26th Annual European Symposium on Algorithms. In : *ESA 2018*. Sous la dir. d'Yossi AZAR, Hannah BAST et Grzegorz HERMAN. T. 112. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 4:1-4:13.
- David GALE et Lloyd SHAPLEY (1962). « College Admissions and the Stability of Marriage ». In : *The American Mathematical Monthly* **69**.1, p. 9-15.
- Juan Luis GASTALDI (2018). « Boole's untruth tables: The formal conditions of meaning before the emergence of propositional logic ». In : *Logic in Question*.
- Georges GONTHIER (2008). « Formal Proof—The Four-Color Theorem ». In : *Notices of the AMS* **55** (11), p. 1382-1393.
- Jens GUSTEDT (2020). *Modern C*. 2^e éd. Manning, p. 324.
- Françoise HÉRITIER-AUGÉ et Élisabeth COPET-ROUGIER (1990). *Les Complexités de l'alliance*.
- Georges IFRAH (1981). *Histoire Universelle des Chiffres*. Robert Laffont.
- Roman JAKOBSON et Claude LÉVI-STRAUSS (1962). « « Les Chats » de Charles Baudelaire ». In : *L'Homme* **2** (1), p. 5-21.
- Donald E. KNUTH (avr. 1976a). « Big Omicron and Big Omega and Big Theta ». In : *SIGACT News* **8**.2, p. 18-24.
- (1976b). *Mariages stables et leurs relations avec d'autres problèmes combinatoires. Introduction à l'analyse mathématique des algorithmes*. Les Presses de l'Université de Montréal.
- Kazimierz KURATOWSKI (1930). « Sur le problème des courbes gauches en Topologie ». In : *Fundamenta Mathematicae* **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).
- Jonathan SWIFT (1726). *Travels into Several Remote Nations of the World. In Four Parts. By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships*.
- Klaus WAGNER (1937). « Über eine Eigenschaft der ebenen Komplexe ». In : *Mathematische Annalen* **114** (1), p. 570-590.

Index

- algorithme, 13
 - glouton, 116
 - d'Euclide, 19, 20
 - incrémentiel, 90
- algorithmique, 6
- appel par référence, 165
- assignation, 8
- backtracking*, 127
- boucle, 9
 - corps de —, 9
- cardinal, 81, 82, 86–88, 161
- clause, 136
 - de Horn, 141
- clef, 68, 74
- complexité, 10, 28, 69
 - classe de —, 150
- conjonction, 134
- constante, 28
- couplage, 144
- disjonction, 134
- diviser pour régner, 72, 73, 93
- enregistrement, 68, 74
- exponentielle, 29
- fonction, 6, 7, 66
 - calculable, 7
- forme normale conjonctive, 136
- graphe, 119
 - arête, 119
 - planaire, 123
 - sommet, 119
- implication, 141
- implémentation, 13
- invariant de boucle, 54, 80
- linéaire, 28, 69
- liste
 - queue de —, 35, 45
 - tête de —, 35, 46
 - chaînée, 35, 45, 46, 74
- littéral, 136
- logarithme, 28
- matrice, 121
- modèle de calcul, 7
- multiplication, 8, 10, 12
- NP-complet**, 117, 139
- négation, 134
- optimal, 116
- ordre, 67
- parcours
 - en profondeur, 129
- pile, 46, 74
- problème
 - des dames, 127
 - du coloriage de graphes, 119, 135
 - du consensus byzantin, 152
 - du sac à dos, 116
 - du sac à dos fractionné, 117
 - du voyageur de commerce, 141
- procédure, 10
- programme, 7
- pseudo-code, 15, 21
- quadratique, 29, 93
- recherche dichotomique, 71

réalisabilité, 13
récursion, 35, 45

- algorithme récursif, 45
- structure récursive, 45
- terminale, 39

tableau, 29, 34, 74
test, 8
trace

- d'exécution, 8, 17

tri, 66

- fonction de —, 68, 79
- en place, 90
- fusion, 93
- par insertion, 89
- par énumération, 74, 90
- stable, 89, 90

variable, 8, 15

Liste des Algorithmes

1	Multiplication par addition itérée	10
2	Algorithme d'Euclide	21
3	Tableau — accès	31
4	Tableau — modification	31
5	Tableau — ajout d'un élément	32
6	Tableau — suppression d'un élément	32
7	Tableau — fusion de deux listes	33
8	Liste — accès itératif	37
9	Liste — accès récursif	38
10	Liste — modification	39
11	Liste — ajout d'un élément	43
12	Liste — suppression d'un élément	43
13	Liste — fusion de deux listes	44
14	Liste — longueur	44
15	Pile — accès à la tête	46
16	Pile — modification de la tête	46
17	Pile — ajout d'une tête	46
18	Pile — suppression d'une tête	47
19	Inversion d'un tableau	49
20	Inversion d'un tableau	49
21	Recherche dichotomique	72
22	Tri par énumération	76
23	Tableau — remplissage avec des zéros	80
24	Tri par insertion	90
25	Fusion de tableaux triés	94
26	Tri fusion — la procédure récursive naïve	95
27	Fusion de listes triés	103
28	Rendu de monnaie — Algorithme glouton	125
29	Algorithme générique de recherche d'une solution par <i>backtrack</i>	130
30	Algorithme de Gale et Shapley	147

Table des matières

I Fondamentaux	3
1 Algorithmes, programmes, fonctions, données	5
1.1 Fonctions	6
1.2 Programmes	7
1.3 Multiplication	8
Multiplication par addition itérée	8
Multiplication posée	10
1.4 Algorithmes	12
1.5 Notations	15
1.6 Exécution	17
1.7 Résumé	17
1.A L'algorithme d'Euclide	19
1.B PGCD	21
La division euclidienne	22
Exécution	23
Terminaison	23
Correction	23
Programmation	23
2 Structures de données 1	25
2.1 Éléments de complexité algorithmique	26
2.2 Tableaux	29
Récupération & modification	31
Adjonction & suppression	31
Fusion	33
Nombre d'éléments	34
Résumé	34
2.3 Listes chaînées	34
Récupération & modification	36
Adjonction & suppression	41
Fusion	43
Longueur	44
Résumé	44
2.4 La récursion	45
2.5 Piles	45

2.6	Résumé	46
2.A	Inversion Correction	48
2.B	D'un tableau à une liste et vice-versa	52
2.C	La multiplication en base 10	55
	Représentation des nombres	55
	Petite multiplication	62
	Grande multiplication	63
3	Tris	65
3.1	Le problème du tri	66
	Relation d'ordre	66
	Enregistrements, clefs et tris	68
3.2	Algorithmes de tri: comparaisons et implémentations	68
3.3	Une application du tri: la recherche dichotomique	71
3.4	Énumération	74
	L'algorithme	74
	Complexité	79
	Les invariants de boucle	79
	Correction	81
	Résumé	89
3.5	Insertion	89
	Correction	91
	Complexité	92
	Résumé	93
3.6	Fusion	93
	Correction	99
	Complexité	99
	Résumé	100
3.7	Borne inférieure de complexité des tris par comparaison	100
3.8	Résumé	102
3.A	Améliorer le tri fusion	103
	Fusion de listes	103
	Rajouter le prochain élément	104
	Fusion équilibrée	106
	Partir de briques plus longues	108
3.B	Chanson sur mon drôle de tri	109
	La moitié plus un	111
	Les deux tiers	112
4	Faire des choix	115
4.1	Gloutonnerie	116
	Sac à dos fractionné et sac à dos	116
	Coloriage de graphes	118
	Rendu de monnaie	124
4.2	Backtracking	127
	Problème des dames	127
	Satisfiabilité	134
	NP-complétude	139
	Satisfiabilité des clauses de Horn	141
4.3	Le voyageur de commerce	142

Programmation dynamique	142
Approximations	143
4.4 Problème des couplages	143
Coupages stables	143
L'algorithme de Gale et Shapley	145
5 La pensée algorithmique	149
5.1 Complexité	150
5.2 Conséquences	150
5.3 Monnaie	151
Le problème du consensus byzantin	152
Rémunération de la vérification	154
Jetons non-fongibles	154
Contrats intelligents	155
Petite tératologie	156
5.4 Optimisation combinatoire	157
II Annexes	159
A Compléments de logique	161
B Compléments sur le langage C	163
B.1 Tableaux : la pratique	164
Syntaxe	164
Appels fonctionnels	165
B.2 Remplir un tableau aléatoirement	166
B.3 Mesurer le temps d'exécution d'une fonction	166
B.4 Traduire un algorithme en C	166
C Graphes de données expérimentales	167
D Examens	169
D.1 2020→2021 Semestre 1	170
Sommes	170
Maximum	172
Emacs	174
D.2 2020→2021 Semestre 2	178
Sommes partielles	178
Tri rapide de Hoare	181
D.3 2021→2022 Semestre 1 Partiel	184
D.4 2021→2022 Semestre 1 Examen	186
Invariant de boucle	186
Implémenter des listes chaînées	187
D.5 2021→2022 Semestre 2 Partiel	191
Bibliothèque	191
Typographier un paragraphe	193
D.6 2021→2022 Semestre 2 Examen	193
Sudoku	193
Bibliothèque	194

Bibliographie	195
Index	197
Liste des algorithmes	199
Table des matières	200