Algorithmique et programmation

L2 MIASHS-Math

1 Correction d'un algorithme **2**

2 Complexité d'un algorithme **6**

3 Tris naïfs **10**

4 La récursivité **16**

5 Tri fusion : diviser pour régner **24**

6 Programmation dynamique **32**

7 Algorithmes simples sur les graphes **41**

8 Algorithmes de plus court chemin **48**

— 1 —

Correction d'un algorithme

Dans ce cours, on va s'intéresser à un certain nombre d'algorithmes. Un exemple classique que vous avez déjà vu est celui de la recherche du minimum d'une liste, que l'on écrit ici en utilisant un pseudo langage (cf. remarque suivante).

```
1 fonction minimum(L)
2 |  n ← longueur(L)
3 |  m ← L[0]
4 |  pour i ← 1 à n − 1 faire
5 |  |  si L[i] < m alors
6 |  |  |  m ← L[i]
7 |  retourner m
```

FIGURE 1.1 – Recherche du minimum dans une liste non vide L

Remarques sur le langage utilisé. Dans ce chapitre comme dans les suivants, nous allons utiliser un pseudo-langage algorithmique proche de Python. Ceci afin de s'abstraire le plus possible d'une syntaxe particulière. Toutefois, nous conserverons la plupart des structures de contrôle et quelques "habitudes" de ce langage comme pour l'échange de deux variables x et y que l'on représentera par $x, y \leftarrow y, x$. On utilisera aussi souvent quelques primitives qui correspondent à des fonctions usuelles dans n'importe quel langage (comme la longueur d'une liste, par exemple). Vous êtes fortement encouragé à implémenter en Python chaque exemple du cours afin de vous assurer que vous en avez bien compris le fonctionnement.

Étant donné un algorithme comme le précédent, on va se poser deux questions fondamentales :

- Celle de la *correction* : il s'agit de démontrer que l'algorithme fait bien ce qu'on veut (ici, renvoyer le minimum d'une liste non vide)
- Celle de la *complexité* : il s'agit d'avoir une borne sur le temps d'exécution de l'algorithme en fonction de la taille des objets en entrée (ici, la longueur de la liste).

Dans ce premier chapitre, on va se concentrer sur le premier point, à savoir la correction : comment démontrer qu'un algorithme fait bien ce qu'on veut ?

1.1 Définition des invariants de boucle

Lorsque l'on veut justifier qu'un programme fonctionne comme voulu, on doit en particulier décrire précisément ce que font les boucles du programme afin de justifier qu'à la sortie

de chacune d'entre elles, on a bien obtenu le résultat désiré. Par exemple, dans le programme ci-dessus, il faut justifier qu'à la sortie de la boucle sur i, la variable m est bien égale au minimum de la liste L. C'est à cela que servent les invariants de boucles.

Définition 1.1. *Un* ***invariant de boucle*** *est une propriété qui décrit ce que fait cette boucle à chacune de ses itérations.*

Pour démontrer qu'un invariant de boucle est vrai, on fait une preuve pas récurrence sur le nombre d'itérations de la boucle. Une fois l'invariant de boucle démontré, on peut décrire ce qui va se passer à la sortie de la boucle, ce qui est la raison pour laquelle on a introduit un invariant de boucle. On va maintenant donner deux exemples afin de préciser tout ça.

1.2 Exemple avec une boucle for

Reprenons l'exemple de la recherche du minimum (figure 1.1). On voit que notre fonction renvoie à la fin la variable m. Il s'agit donc de montrer qu'à la fin de l'exécution, la variable m est bien égale au minimum de la liste. Pour cela, on voit qu'il y a deux lignes où la valeur de m est modifiée : ligne 3, où elle prend la valeur $L[0]$, et ligne 6, où elle peut prendre la valeur $L[i]$.

Cette ligne 6 se trouve au sein d'une boucle, on va devoir faire une preuve par récurrence pour montrer qu'à chaque étape, les choses se passent correctement. Pour cela, on introduit donc un *invariant de boucle*, qui permet de préciser comment les choses se passent à chaque étape. On va donner deux manières de rédiger ; en général dans ce cours on préférera la deuxième, mais la première a sans doute le mérite d'être plus proche des preuves par récurrence auxquelles vous êtes habitués.

Première version de l'invariant de boucle

Voici un premier invariant de boucle pour notre fonction, où $k \in \{0, ..., n-1\}$:

Après la k-ième itération[1] *de la boucle* **pour** *de la ligne 4, la variable m est égale au minimum de $\{L[0], \ldots, L[k]\}$, et si $k \geq 1$ alors $i = k$.*

Preuve de l'invariant de boucle. Montrons que cet invariant de boucle est vrai par récurrence sur $k \in \{0, ..., n-1\}$:

- *Initialisation* : Si $k = 0$, la boucle n'a pas commencé et on est arrivé à la ligne 3, où l'on a $m = L[0]$, donc m est bien le minimum de $\{L[0], \ldots, L[0]\} = \{L[0]\}$.
- *Hérédité* : Soit $k \in \{0, \ldots, n-2\}$, supposons l'invariant de boucle vrai au rang k et montrons qu'il l'est également au rang $k+1$. Tout d'abord, il est clair qu'à la $k+1$-ème itération, on a $i = k+1$. Par hypothèse de récurrence, au début de cette itération $m = \min\{L[0], \ldots, L[k]\}$. Il y a alors deux possibilités :
  - $L[k+1] < m$, alors comme $m = \min\{L[0], \ldots, L[k]\}$, on déduit que $L[k+1] = \min\{L[0], \ldots, L[k+1]\}$, et comme m devient $L[k+1]$ à la ligne 6, on a bien que à la fin de la $k+1$-ème itération, $m = \min\{L[0], \ldots, L[k+1]\}$
  - $L[k+1] \geq m = \min\{L[0], \ldots, L[k]\}$, alors on ne change pas m qui est également le minimum de $\{L[0], \ldots, L[k+1]\}$.

Dans tous les cas, notre invariant de boucle est vrai à l'étape $k+1$, donc par récurrence il est vrai pour tout $k \in \{0, ..., n-1\}$. □

1. Une *itération* d'une boucle correspond au fait d'exécuter une fois son contenu. Le cas $k = 0$ correspond à ce qui se passe juste avant d'arriver à la boucle.

Maintenant que notre invariant de boucle est établi, le point clé est que l'on sait ce qui se passe à la sortie de la boucle, c'est à dire quand $k = n-1$: on a que $m = \min\{L[0], \dots, L[n-1]\}$, autrement dit m est bien le minimum de la liste L entière, ainsi notre fonction renvoie bien le minimum de L.

Deuxième version de l'invariant de boucle

Voici une autre manière de présenter les choses, où l'invariant de boucle ne parle que des variables i et m. On prétend qu'on a l'invariant de boucle suivant :

Après chaque itération de la boucle **pour** *de la ligne* 4*, la variable m est égale au minimum de $\{L[0], \dots, L[i]\}$.*

Pour montrer que cet invariant de boucle est vrai, on doit encore faire une récurrence sur le nombre d'itérations [2]

— Initialisation : Avant que la boucle commence, on avait $m = L[0]$ puis on a testé si $L[1] > m$, et on change m en $L[1]$ si c'est le cas : on a donc bien $m = \min\{L[0], L[1]\}$ après 1 itération.

— Récurrence : Supposons qu'après k itérations, m est égale au minimum de $\{L[0], \dots, L[i]\}$. Alors à l'itération suivante, i devient $i+1$, donc après la ligne 4 on a que m est égale au minimum de $\{L[0], \dots, L[i-1]\}$. Il y a alors deux possibilités à la ligne 5 :

  — $L[i] < m$, alors comme $m = \min\{L[0], \dots, L[i-1]\}$, on déduit que $L[i] = \min\{L[0], \dots, L[i]\}$, et comme m devient $L[i]$ à la ligne 6, on a bien que à la fin de la $k+1$-ème itération, $m = \min\{L[0], \dots, L[i]\}$

  — $L[i] \geq m = \min\{L[0], \dots, L[i-1]\}$, alors on ne change pas m qui est également le minimum de $\{L[0], \dots, L[i]\}$.

  Dans tous les cas, notre invariant de boucle est préservé à l'étape $k+1$, donc par récurrence il est vrai pour tout $k \in \geq 1$.

Maintenant que notre invariant de boucle est établi, on sait qu'on sortira de la boucle lorsque $i = n-1$, et à ce moment là on aura bien $m = \min\{L[0], \dots, L[n-1]\}$, ce qui permet de conclure comme précédemment.

1.3 Exemple avec une boucle *while*

Voici maintenant un exemple simple d'invariant de boucle dans le cas d'une boucle *while*. On considère le problème suivant : soit $(u_n)_{n\in\mathbb{N}}$ la suite définie par récurrence par $u_0 = 0$ et pour tout $n \in \mathbb{N}$, $u_{n+1} = 1 + u_n^2$. Étant donnée une entrée $A \in \mathbb{N}$, on veut un programme qui renvoie le premier entier $n \in \mathbb{N}$ tel que $u_n \geq A$.

Essayons maintenant de démontrer que le programme ci-dessus fonctionne comme voulu. On considère tout d'abord l'invariant de boucle suivant, où $k \in \mathbb{N}$:

Après la k-ième itération de la boucle **tant que** *de la ligne* 4*, la variable u est égale à u_k et la variable n est égale à k.*

On fait une nouvelle fois une preuve détaillée que l'invariant de boucle est vrai. Ce genre de preuve sera généralement omis dans la pratique.

2. Autrement dit, on montre par récurrence sur $k \geq 1$ que la propriété suivante $\mathcal{P}(k)$ est vraie : après k itérations, on a $m = \min\{L[0], \dots, L[i]\}$.

```
1 fonction premiern(A)
2 |  n ← 0
3 |  u ← 0
4 |  tant que u < A faire
5 |  |  n ← n + 1
6 |  |  u ← 1 + u * u
7 |  retourner n
```

FIGURE 1.2 – Recherche du premier entier n tel que $u_n \geq A$

Preuve de l'invariant de boucle. Montrons que cet invariant de boucle est vrai par récurrence sur $k \in \mathbb{N}$:

— *Initialisation* : Si $k = 0$, la boucle n'a pas commencé et on est arrivé à la ligne 3, on a donc $u = 0 = u_0$ et $n = 0$ comme voulu.

— *Hérédité* : Soit $k \in \mathbb{N}$, supposons l'invariant de boucle vrai au rang k et montrons qu'il l'est également au rang $k+1$. On a par hypothèse de récurrence qu'avant la $k+1$-ème itération, $u = u_k$ et $n = k$. Et pendant la $k+1$-ème itération, on a que n est augmenté de 1 (donc devient bien égal à $k+1$) et que u devient $1 + u^2 = 1 + u_k^2 = u_{k+1}$, donc l'invariant de boucle est vrai au rang $k+1$.

Par récurrence notre invariant de boucle vrai pour tout $k \in \mathbb{N}$. □

On peut maintenant terminer la preuve que le programme renvoie ce qu'on veut : la boucle tant que n'est pas réitérée dès lors que $u \geq A$, ce qui intervient tout de suite après la première itération où on a obtenu $u \geq A$. D'après notre invariant de boucle ceci arrive après la k-ème itération, où k est le premier entier tel que $u_k \geq A$, et on a alors $n = k$. On renvoie alors n comme voulu : notre programme fonctionne donc comme souhaité.

Exercice 1.2. En s'inspirant de la deuxième version de l'invariant de boucle dans la section précédente, écrire la preuve par récurrence que l'algorithme fonctionne en utilisant l'invariant de boucle suivant :

Après chaque itération de la boucle **tant que** *de la ligne* 4*, la variable u est égale à u_n.*

Remarque 1.3. *Dans les cas simples, on se contentera souvent de décrire l'invariant de boucle sans faire la preuve par récurrence.*

— 2 —

Complexité d'un algorithme

Dans le cours, nous souhaiterons souvent connaître le nombre d'opérations, ou pas de calculs, effectués par des algorithmes. C'est ce que l'on appellera la *complexité* de l'algorithme. Cela servira de mesure de leur efficacité : moins le nombre d'opérations est important, plus l'algorithme est considéré comme efficace. Notons que la taille de la mémoire utilisée peut-être aussi prise en compte concernant l'efficacité d'un algorithme.

Les algorithmes sont conçus pour toute entrée d'un certain "type" et non uniquement pour fonctionner sur une entrée en particulier : on ne fait pas, par exemple, un algorithme de tri pour une liste d'entiers spécifique mais, à priori, pour toute liste potentielle d'entiers. Il est donc raisonnable d'évaluer la complexité d'un algorithme de façon asymptotique, en soulignant la croissance du nombre de pas de calcul requis en fonction de la longueur de l'entrée.

Une façon habituelle de se repérer est de comparer cette complexité à quelques ordres de grandeurs bien connus.

2.1 Ordres de grandeur, comparaisons de fonctions

La notation $O(\cdot)$ est utilisée couramment en mathématiques pour comparer deux fonctions définies sur $\mathbb{R}$ (au voisinage d'un point, de $+\infty$ ou de $-\infty$). On en rappelle ci-dessous la définition dans le cadre spécifique qui nous intéresse pour la complexité des algorithmes, à savoir la comparaison au voisinage de $+\infty$ de deux fonctions définies sur $\mathbb{N}$ et à valeurs positives.

Définition 2.1. (NOTATION $O(\cdot)$) *Soit f, g deux fonctions de $\mathbb{N} \to \mathbb{R}^+$. On dit que $f = O(g)$ s'il existe $C > 0$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0$:*

$$f(n) \leqslant C \cdot g(n)\,.$$

On dit que f est dominée par g (au voisinage de $+\infty$).

Par exemple, $f = O(n)$ signifie qu'il existe $C > 0$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0$:

$$f(n) \leqslant C \cdot n\,.$$

Si $f = O(n^2)$, cela signifie qu'il existe $C > 0$, $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0$:

$$f(n) \leqslant C \cdot n^2\,.$$

La définition de $O()$ a des propriétés intéressantes qui permettront de déduire des résultats plus généraux sur les temps d'exécutions d'algorithmes par compositions des temps de calcul de leurs sous-algorithmes. On en liste quelques-unes ci-dessous.

Proposition 2.2. *Soit f, g, h des fonctions de $\mathbb{N} \to \mathbb{R}^+$. Si $f(n) = O(g(n))$ et $g(n) = O(h(n))$, alors $f(n) = O(h(n))$.*

Démonstration. Soient $C_1 > 0$ et $n_1 \in \mathbb{N}$ tel que $f(n) \leqslant C_1 \cdot g(n)$ pour tout $n \geqslant n_1$. De même, soit $C_2 > 0$ et $n_2 \in \mathbb{N}$ tel que $g(n) \leqslant C_2 \cdot h(n)$ pour tout $n \geqslant n_2$. Pour tout $n \geqslant \max\{n_1, n_2\}$, on a $f(n) \leqslant C_1 C_2 \cdot h(n)$. Cela montre f est dominée par h au voisinage de $+\infty$. □

Proposition 2.3. *Soit f, g, h, ℓ des fonctions de $\mathbb{N} \to \mathbb{R}^+$.*

- *Si $f(n) = O(h(n))$ et $g(n) = O(h(n))$ alors $f(n) + g(n) = O(h(n))$.*
- *Si $f(n) = O(h(n))$ et $g(n) = O(\ell(n))$ alors $f(n)g(n) = O(h(n)\ell(n))$.*

Démonstration. Soient $C_1 > 0$ et $n_1 \in \mathbb{N}$ tels que pour tout $n \geq n_1$, $f(n) \leq C_1 h(n)$, soient $C_2 > 0$ et $n_2 \in \mathbb{N}$ tels que pour tout $n \geq n_2$, $g(n) \leq C_2 h(n)$. Alors pour tout $n \geq \max(n_1, n_2)$, on a $f(n) + g(n) \leq (C_1 + C_2)h(n)$ et $f(n)g(n) \leq (C_1 C_2)h(n)$, ce qui montre bien les les deux points de la proposition. □

On peut alors généraliser facilement l'exercice précédent de la manière suivante.

Exemple 2.4. *Soit P un polynome de degré k, alors $P(n) = O(n^k)$. En effet, pour tout $l \leq k$, on a $n^l = O(n^k)$, donc $\sum_{l=0}^{k} a_l n^l = \sum_{l=0}^{k} O(n^k) = O(n^k)$.*

La proposition précédente, bien que simple, est très utile pour déterminer la complexité d'un algorithme lorsqu'on connaît la complexité de ses "parties" : par exemple pour montrer qu'un algorithme est en $O(n^k)$, il suffira de montrer que chacune de ses parties a une complexité en $O(n^k)$.

L'exercice qui suit permet de comparer entre elles les complexités qui peuvent apparaitre au sein de telles parties.

Exercice 2.5. Vérifier que $f(n) = O(g(n))$ dans chacun des cas suivants :

1. $f(n) = n^c$ et $g(n) = n^d$ avec $0 \leqslant c \leqslant d$;
2. $f(n) = \log(n)$ et $g(n) = n^\varepsilon$ pour $\varepsilon > 0$;
3. $f(n) = n \log n$ et $g(n) = n^{1+\varepsilon}$ pour $\varepsilon > 0$;
4. $f(n) = n^\alpha$ et $g(n) = c^n$ pour $\alpha \geqslant 0$ et $c > 1$.

2.2 Complexité des algorithmes : définition et exemples

Dans cette partie, on analyse la complexité de certains algorithmes simples. On parlera de *temps de calcul* même si dans les faits ce que l'on mesure est une borne sur le nombre d'opérations (comparaisons, affectation, addition, multiplication, etc) réalisées. Pour obtenir une analyse plus fine, il faudrait évaluer le coût de chacune de ces opérations par exemple en termes de nombres d'opérations élémentaires (sur les bits) et par rapport à la longueur des arguments impliqués.

L'analyse de complexité est une analyse *dans le pire cas*. La borne donnée est celle du nombre maximal d'opérations pour toutes les entrées de *même taille*.

Les entrées des algorithmes peuvent être très variées : entiers, nombres réels, listes, mots, structures de données plus élaborées (comme des graphes ou des arbres). Stricto sensu, la longueur d'une entrée est le nombre de bits nécessaires pour le représenter en machine. Dans ce qui nous intéresse, on considérera des mesures plus "grossières" mais qui sont proches en pratique de la réalité. En voici quelques exemples :

- Pour un nombre (notamment entier) : la longueur de sa décomposition en binaire ;

- Pour une liste : son nombre d'éléments ou, si on veut être plus fin, la somme des longueurs des éléments qui la compose ;
- Pour un graphe : son nombre de sommets et d'arêtes ;
- Pour une matrice : sa dimension $n \times m$ ou, plus finement encore, la somme des longueurs de ses coefficients.

À chaque fois que l'on parle de complexité, il faudra donc préciser quelle est la taille d'entrée qui sert de référence. Pour un objet x, on notera $|x|$, sa longueur. Pour un algorithme A prenant en entrée x, si le calcul de A sur x s'arrête, on note $T(x)$ le nombre d'opérations élémentaires effectuées par A. Usuellement, on appelle $T(x)$ le temps de calcul de A sur x. La complexité d'un algorithme A est donc une fonction $f : \mathbb{N} \to \mathbb{N}$ qui à tout $n \in \mathbb{N}$ associe :

$$f(n) = \max\{T(x) \ : \ |x| = n\}.$$

C'est ce qu'on appelle la complexité dans le pire cas : pour une longueur n donnée, ce sont les entrées x qui prennent le plus de temps qui déterminent la valeur de $f(n)$.

Une fois la complexité exprimée en fonction de la taille des entrées, on peut étudier le comportement asymptotique de celle-ci.[1] On illustre quelques complexités d'algorithmes dans la suite de ce chapitre.

Recherche du minimum dans une liste Considérons une nouvelle fois l'algorithme de recherche du minimum dans une liste L.

```
1 fonction minimum(L)
2     n ← longueur(L)
3     m ← L[0]
4     pour i ← 1 à n − 1 faire
5         si L[i] < m alors
6             m ← L[i]
7     retourner m
```

FIGURE 2.1 – Recherche du minimum dans une liste non vide L

Chaque instruction conditionnelle à l'intérieur de la boucle effectue la comparaison $L[i] < m$, et l'affectation $m \leftarrow L[i]$ si le test est vrai. Ces instructions sont itérées $n-1$ fois et il n'y a pas d'autres instructions dans la boucle. Noter que le nombre d'étapes pour un tableau de taille n est toujours le même : il faut parcourir tout le tableau. L'algorithme a donc pour complexité $O(n)$, et cette borne est optimale.

Produit de matrices Supposons que A et B sont deux matrices carrées de taille $n \times n$. Les coefficients de la matrice $C = AB$ se calculent de la façon suivante : pour tout $i, j \in \{0, \ldots, n-1\}$,

$$C[i,j] = \sum_{k=0}^{n-1} A[i,k]B[k,j].$$

On peut implémenter ce calcul par l'algorithme suivant :

1. Il arrive parfois qu'on exprime la complexité en fonction de paramètres liés à la taille de l'entrée. Par exemple, il est usuel d'exprimer la complexité du produit de deux matrices de tailles $n \times n$ en fonction de n, bien que l'entrée soit de taille $2n^2$.

```
2  pour i ← 0 à n − 1 faire
4  |  pour j ← 0 à n − 1 faire
6  |  |  C[i, j] ← 0
8  |  |  pour k ← 0 à n − 1 faire
10 |  |  |  C[i, j] ← C[i, j] + A[i, k] × B[k, j]
```

FIGURE 2.2 – Produit de matrices $C = AB$

Le temps de calcul est borné par $O(n^3)$. En effet, ce temps est majoré par $n \cdot T_i$ où T_i, pour i fixé, est le temps de calcul du sous-programme commençant à la ligne 2. La valeur de T_i est donc bornée par $n \cdot (1 + T_{i,j})$, où $T_{i,j}$ désigne le temps de calcul du sous-programme commençant à la ligne 4 (pour i et j fixés). Enfin, $T_{i,j}$ est borné par $n \cdot T$ où T est le nombre d'opérations nécessaires pour exécuter l'instruction de la ligne 5. On voit que l'exécution de la ligne 5 requiert une addition et une multiplication, c'est-à-dire un nombre constant d'opérations. La complexité de l'algorithme est donc bien $O(n^3)$.

Notons qu'il s'agit de la complexité de *cet algorithme*. Si on parle de la complexité de la multiplication matricielle en général, celle-ci est plus faible : il existe des algorithmes plus efficaces.

2.3 A propos de l'optimalité des bornes de complexité

Quand on montre que la complexité d'un algorithme est $O(f(n))$, on a obtenu une *borne supérieure* sur le nombre de pas de calcul de cet algorithme. Mais il se peut que l'algorithme soit en fait plus rapide. Cela résulte d'une majoration trop large du nombre d'étapes utilisé par l'algorithme lors de son analyse.

Pour montrer que l'algorithme prend effectivement de l'ordre de $f(n)$ étapes sur une entrée de taille n dans le pire des cas, il faut montrer qu'il existe n_1 et $\beta > 0$ tel que pour tout $n \geqslant n_1$, il existe une entrée de taille n sur laquelle l'algorithme prend au moins $\beta f(n)$ étapes de calcul. Ceci montre que la borne $O(f(n))$ ne peut pas être améliorée. Un exemple simple est donné par l'exercice 3.1 du prochain chapitre.

Une mise en garde : il ne faut pas confondre la question de l'optimalité d'une borne de complexité d'un algorithme donné avec l'optimalité de l'algorithme pour un problème donné, qui est un problème beaucoup plus délicat. Afin d'illustrer ce point, revenons sur la multiplication de matrices.

Pour l'algorithme de multiplication de matrices décrit ci-dessus, on peut montrer que la borne de complexité en $O(n^3)$ est optimale, c'est-à-dire que c'est la meilleure borne de complexité *pour cet algorithme*. Par contre, cet algorithme n'est pas le meilleur permettant la multiplication de matrices : par exemple l'algorithme de Strassen est plus efficace (sa complexité est en $O(n^{2,808})$)). Il existe d'autres algorithmes eux-mêmes plus efficaces que l'algorithme de Strassen. En fait, on ne sait pas s'il existe un algorithme plus optimal que tous les autres parmi les algorithmes de multiplication de matrices [2]...

2. Actuellement, le meilleur algorithme connu est en $O(n^{2,373})$.

— 3 —

Tris naïfs

Dans ce chapitre, nous étudions un problème simple et important : le tri (*sorting* en anglais). Nous travaillerons sur des tableaux d'entiers pour fixer un cadre abstrait simple [1]. Nous n'utiliserons que peu les propriétés des entiers, en tous cas pas le fait qu'on puisse les additionner ou les multiplier.

Nous utiliserons le fait que les entiers sont *totalement ordonnés* : la relation $<$ vérifie

1. $\forall x \neq y \in \mathbb{N}$, $x = y$ ou bien $x < y$ ou bien $y < x$ (elle est anti-symétrique et il s'agit d'un ordre *total*) ;
2. $\forall x, y, z \in \mathbb{N},\ x < y$ et $y < z \Rightarrow x < z$ (transitivité).

Trier une liste L dans l'ordre croissant, c'est permuter les éléments de L pour les ranger dans l'ordre croissant. Par exemple, considérons la liste suivante :

position	1	2	3	4	5	6	7
valeur	5	2	-3	1	5	8	0

Après un tri par ordre croissant, on obtient la liste suivante :

position	1	2	3	4	5	6	7
valeur	-3	0	1	2	5	5	8

Dans toute la suite, nous trierons par ordre croissant, les algorithmes peuvent être adaptés très simplement pour trier par ordre décroissant.

Les algorithmes de tri que nous étudierons sont des tris *par comparaison* : leur exécution *ne dépend que de l'ordre* dans lequel sont initialement rangés les éléments de la liste à trier. De ce point de vue, trier $1, 5, 3$ n'est pas différent de trier $-1, 100, 20$.

Nous allons examiner d'abord des tris réputés naïfs. Ces algorithmes ont comme point commun de ne pas utiliser de structures de données compliquées.

Dans tout ce chapitre, on suppose qu'une liste L contenant n éléments est indexé de 0 à $n-1$, conformément à ce qui se fait dans la plupart des langages de programmation.

3.1 Recherche dans une liste

Pourquoi trier ? Parce qu'on a des algorithmes très efficaces de recherche dans une liste triée.

Définition d'un algorithmes de recherche. Un tel algorithme prend en entrée une liste L et un élément x et détermine si x apparaît dans L. Plus précisément, l'algorithme doit retourner un entier i tel que $L[i] = x$ si l'élément apparaît dans le tableau L et -1 sinon.

1. En Python, ces tableaux d'entiers seront représentés par des listes

Nous allons examiner successivement un algorithme de recherche séquentielle (qui fonctionne sur un tableau même s'il n'est pas trié) et un autre, plus rapide, de recherche dichotomique, qui s'applique uniquement au cas d'un tableau trié.

Recherche séquentielle Si nous devons chercher une valeur dans un tableau et que nous ne savons rien sur l'organisation du tableau, nous ne pouvons pas faire mieux que de passer en revue les valeurs représentées dans le tableau. Ceci se traduit par l'algorithme de la figure 3.1, où on renvoie -1 si x n'est pas dans le tableau.

```
1 fonction recherche(L, x)
2 |  pour i ← 0 à longueur(L) − 1 faire
3 |  |  si L[i] = x alors
4 |  |  |  retourner i
5 |  retourner −1
```

FIGURE 3.1 – Recherche séquentielle dans une liste

La correction découle du fait qu'au début de l'itération i (si l'algorithme n'est pas encore terminé), tous les éléments de la liste L d'indice $j < i$ vérifient $L[j] \neq x$. La complexité de cet algorithme est $O(\text{longueur}(L))$.

Exercice 3.1. Montrer que cette borne de complexité est optimale.

Recherche dichotomique dans un tableau trié en ordre croissant. Si le tableau est trié, on peut faire beaucoup mieux que dans le cas général : c'est la recherche dichotomique. Cette technique relève du principe appelé « diviser pour régner ».

```
1  fonction rechercheDichotomique(L, x)
2  |  d ← 0
3  |  f ← longueur(L) − 1
4  |  tant que d ⩽ f faire
5  |  |  m ← ⌊(d + f)/2⌋
6  |  |  si L[m] = x alors
7  |  |  |  retourner m
8  |  |  si L[m] < x alors
9  |  |  |  d ← m + 1
10 |  |  sinon
11 |  |  |  f ← m − 1
12 |  retourner −1
```

FIGURE 3.2 – Recherche dichotomique

Correction de la recherche dichotomique. Pour montrer la correction de l'algorithme, on montre l'invariant de boucle suivant :

Au début de chaque itération de la boucle **tant que** *de la ligne 3, si x est un élément de L alors la position de x est $\geq d$ et $\leq f$.*

On montre cette propriété par récurrence sur le nombre d'itérations : au début de la première itération, la propriété est clairement vérifiée puisque $d = 0$ et $f = \texttt{longueur}(L) - 1$.

Supposons la propriété vraie pour la k-ième itération, montrons qu'elle est vrai pour la $k+1$-ième itération.

Supposons donc que x est dans L et plaçons nous au début de la k-ième itération : la position de x est comprise entre d et f. Alors remarquons qu'à la ligne 5, m prend une valeur comprise entre d et f. Ces derniers sont modifiés dans deux cas :

— Si $L[m] < x$, comme x positionné entre d et f et la liste est triée, on déduit que x doit être entre $m+1$ et f, or $m+1$ est bien la nouvelle valeur prise par d (ligne 9) et f ne change pas : on a bien qu'à la fin de l'itération x est toujours entre d et f.

— Dans le dernier cas (ligne 10), on avait $L[m] > x$. On a que x est positionné entre d et f est la liste étant triée, on doit donc avoir que x est entre d et $m-1$. Or seul f change pour prendre la valeur $m-1$ (ligne 11), donc à la fin de l'itération, x est toujours entre d et f.

Dans les deux cas, on a bien que, sous l'hypothèse que x est dans L, x est positionné entre d et f au début de la $k+1$-ième itération.

Il s'agit maintenant de justifier que le programme termine et renvoie ce qu'on veut. Pour cela, on remarque qu'à chaque itération de la boucle while, la quantité $f-d$ diminue au moins de 1. En particulier, la boucle while ne peut pas boucler indéfiniment, car sinon on aurait $f < d$ après $n = \texttt{longueur}(L)$ itérations.

On peut sortir de la boucle while de deux manières :

— Soit parce que $L[m] = x$, auquel cas on renvoie m comme demandé.

— Soit parce qu'on finit par avoir $d > f$, mais alors x d'après l'invariant de boucle on ne pouvait avoir x dans L puisque sa position est toujours $\geq d$ et $\leq f$. C'est donc que x n'est pas dans f et alors on renvoie -1 (ligne 12), comme demandé.

Ceci conclut la preuve de la correction de notre algorithme. Remarquons que l'on utilise pas la formule précise qui donne m mais simplement le fait qu'il est toujours compris entre d et f au début de la boucle. Le calcul de la complexité va utiliser la formule précise.

Complexité de la recherche dichotomique. On va montrer que le nombre d'itérations de la boucle *tant que* (ligne 3) est au plus $1 + \lceil \log_2(n-1) \rceil$, où n est la longueur de la liste. Pour ceci, soit $k = \lceil \log_2(n-1) \rceil$, alors $\log_2(n-1) \leq k$ donc en passant à la puissance 2 on trouve $n - 1 \leq 2^k$ donc $n < 2^k$. Avant la première exécution de la boucle, $f - d = n - 1$. On remarque que à chaque étape de la boucle, $f - d$ devient $f' - d'$, avec $f' - d' \leq \frac{f-d}{2}$. Ainsi après k étapes, $f - d \leq \frac{n}{2^k} < 1$ donc $f - d = 0$ donc on a au plus une itération supplémentaire : la boucle termine en au plus $k + 1 = 1 + \lceil \log_2 n \rceil$ étapes. Or $f - d$ reste un entier, donc après au plus $k+1$ étapes on obtient $f = d$. Comme le coût d'une itération est $O(1)$, cela donne une complexité $O(1 + \lceil \log_2 n \rceil) = O(\log_2 n)$ pour l'algorithme de recherche dichotomique dans un tableau de longueur n.

Remarque 3.2. *La notation* $\log_2(x)$ *désigne le logarithme de* x *en base* 2*. Cette quantité est définie par* $\log_2(x) = \ln(x)/\ln(2)$ *où* $\ln(x) = \int_1^x u^{-1}\,du$ *est le logarithme népérien de* x *(ou logarithme naturel) (on peut aussi le définir comme la solution de l'équation en* y*,* $\exp(y) = \lim_{n\to\infty}(1 + y/n)^n = x$*). On a bien sûr* $\log_2(2) = 1$ *et d'une manière générale* $\log_2(2^k) = k$*. Si un entier* n *se décompose en* $n = \sum_{i=0}^d n_i 2^i$ *avec* $n_i \in \{0,1\}$ *et* $n_d = 1$*, on a*

$$2^d \leqslant n \leqslant \sum_{i=0}^{d} 2^i = 2^{d+1} - 1$$

donc

$$d \leqslant \log_2(n) < d + 1$$

soit $d = \lfloor \log_2(n) \rfloor$.

*Le nombre de chiffres binaires (*bits *pour* binary digits*) nécessaires pour représenter l'entier* n *est donc égal à* $\lfloor \log_2(n) \rfloor + 1$.

Exercice 3.3. Montrer que la borne de complexité en $O(\log_2 n)$ de l'algorithme de recherche dichotomique est optimale.

3.2 Tri à bulles

L'algorithme du tri à bulles pour une liste de longueur n consiste à balayer $n-1$ fois la liste de gauche à droite et effectuer l'opération suivante : si deux éléments consécutifs ne sont pas dans l'ordre, les permuter. L'algorithme est présenté en pseudo-code dans la figure 3.3.

```
1 fonction triBulles(L)
2 |  n ← longueur(L)
3 |  répéter n − 1 fois
4 |  |  pour j ← 0 à n − 2 faire
5 |  |  |  si L[j] > L[j + 1] alors  L[j], L[j + 1] ← L[j + 1], L[j]
6 |  retourner L
```

FIGURE 3.3 – Tri à bulles

Correction de l'algorithme. On montre la correction de l'algorithme en prouvant la propriété suivante : après k exécutions de la boucle interne, les k plus grands éléments sont triés dans les k dernières cases de la liste.

— Pour $k = 0$ c'est clair (la propriété est vide).

— Supposons l'avoir montré pour k et montrons le pour $k+1$. Notons L_k la liste obtenue après k itérations de la boucle. Il est immédiat par récurrence sur j que le plus grand élément de $L_k[1, \ldots, j]$ se trouve en position j au début de l'itération j (comparant les éléments j et $j+1$). Ainsi, à la fin de cette boucle interne, le plus grand élément de $L_k[1, \ldots, n-k]$ se trouve en position $n-k$. Dans la liste L_{k+1} ainsi obtenue, on a donc les $k+1$ plus grand éléments triés en dernières positions.

Après les $n-1$ itérations de la boucle interne, on a donc les $n-1$ plus grands éléments triés dans les positions de 2 à n. Le plus petit élément se trouve donc en position 1 et le tableau est trié. Ceci termine la preuve de correction de l'algorithme du tri à bulles.

Complexité. Sur une entrée de longueur n, le coût de l'algorithme est la somme de :

— coût de l'affectation de la variable n : $O(1)$ étapes de calcul ;

— coût des deux boucles imbriquées. Le corps de la boucle *pour* a un coût constant $O(1)$, et il est effectué $(n-1)$ fois. Le coût de la boucle *pour* est donc $O(n-1)$, autrement dit $O(n)$. Le coût total de la boucle *répéter* est donc $O(n^2)$.

En tout, la complexité est donc $O(1) + O(n^2)$, donc $O(n^2)$.

3.3 Tri par sélection (ou tri du maximum)

Le tri par sélection fonctionne de la façon suivante :

— À la première étape, on recherche la position p où se trouve le maximum de la liste. On échange cet élément avec l'élément en dernière position (position n).

— A la deuxième étape, on recherche la position p où se trouve le maximum de la sous-liste $L[1, \ldots, n-1]$. On échange cet élément avec l'élément en position $n-1$.
— ...
— À la dernière étape on échange les éléments de la sous-liste $L[1, 2]$ s'ils ne sont pas dans l'ordre.

On commence par écrire l'algorithme permettant de rechercher la position du maximum dans une partie du tableau.

Recherche du maximum. La recherche de la position du maximum se fait en parcourant la liste.

```
1 fonction posMax(L, d, f)
2 |  p ← d
3 |  pour i ← d + 1 à f faire
4 |  |  si L[i] > L[p] alors
5 |  |  |  p ← i
6 |  retourner p
```

FIGURE 3.4 – Recherche de la position du maximum dans le sous-tableau allant des indices d à f (inclus).

La correction de l'algorithme `posMax` se démontre de manière similaire à celle de la recherche du minium (cf. Figure 1.1). Elle repose sur l'invariant de boucle suivant : ≪ Lorsqu'on termine la i-ème itération de la boucle *pour* , la valeur courante de $L[p]$ contient le maximum des valeurs contenues dans le tableau L entre les positions d et $d + i$ ≫. La complexité de l'algorithme est $O(f - d + 1)$.

Algorithme du tri par sélection. Le pseudo-code de l'algorithme est donné dans la Figure 3.5.

```
1  fonction triSelection(L)
3  |  n ← longueur(L)
5  |  pour i ← n − 1 à 1 par pas de −1 faire
7  |  |  p ← posMax(L, 1, i)
9  |  |  L[p], L[i] ← L[i], L[p]
11 |  retourner T
```

FIGURE 3.5 – Tri par sélection

Correction de l'algorithme. La correction de l'algorithme repose sur la propriété suivante (invariant de boucle) :≪ Lorsqu'on commence une itération de la boucle *pour*, les positions i à $n - 1$ contiennent les $n - i$ plus grandes valeurs du le tableau T. Ces valeurs sont rangées en ordre croissant ≫.

Cette proriété se démontre par récurrence en utilisant la correction de l'algorithme `posMax`.

Complexité. La ligne 3 a un coût $O(i)$ et la ligne 4 un coût $O(1)$. Le coût total de cet algorithme est donc $O(1) + O(\sum_{i=2}^{n}(i + 1)) = O(n^2)$.

3.4 Conclusion

Les algorithmes de tri à bulles et de tri par sélection ont une complexité $O(n^2)$, tout comme le tri par insertion étudié en TD. De plus, nous avons vu que cette borne sur la complexité est optimale : ces algorithmes et les différentes variantes étudiées prennent effectivement de l'ordre de n^2 pas de calcul dans le pire cas sur des listes de taille n. Nous allons voir dans les chapitres suivants des méthodes permettant de trier plus rapidement.

— 4 —
La récursivité

4.1 Principe et exemples

On a l'habitude en mathématiques de définir certains objets par récurrence. C'est une approche naturelle pour les suites et certaines fonctions combinatoires. L'idée de la programmation récursive est de tirer profit de ces approches et de proposer un style de programmation où on décrit comment calculer une fonction pour une certaine valeur de paramètre en fonction de la valeur de cette fonction pour des paramètres plus petits.

Un premier exemple : la factorielle Considérons la fonction bien connue, factorielle, définie pour tout entier strictement positif n, par

$$n! = \prod_{i=1}^{n} i = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1.$$

En $n = 0$, on a $0! = 1$. La définition alternative suivante rend plus immédiat l'aspect récurrent :

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \text{ pour } n > 0 \end{cases}$$

Cette approche se traduit naturellement en un algorithme, que l'on dira récursif, pour calculer la fonction factorielle (cf. figure 4.1).

```
1 fonction fact(n)
2 |  si n = 0 alors
3 |  |  retourner 1
4 |  sinon
5 |  |  retourner n × fact(n − 1)
```

FIGURE 4.1 – Factorielle : programme récursif

On voit que dans l'algorithme de la figure 4.1, fact(n) fait appel à fact($n-1$) qui fait à son tour appel à fact($n-2$) etc. Le processus pourrait continuer indéfiniment mais l'algorithme est pourvu d'un test, basé sur les valeurs initiales de la fonction, qui va permettre d'arrêter l'enchaînement d'appels : si $n = 0$ alors on renvoie un résultat connu à l'avance (en l'occurrence 1).

L'algorithme ci-dessus est intuitivement clair et très simple. Il est en quelque sorte *déclaratif* : il se fonde sur la définition de la fonction et n'explicite pas vraiment comment calculer celle-ci. Ce qui se passe en coulisse, c'est à dire comment ces programmes sont évalués par la machine, mérite d'être expliqué au moins rapidement.

L'enchaînement des appels récursifs peut se visualiser à travers ce que l'on nomme *l'arbre des appels récursifs* suivant (arbre qui dans le cas de la fonction factorielle est très simple).

```
1 fonction fact(3)
2   si 3 = 0 alors
3     retourner 1
4   sinon
5     retourner
        3 × fact(2)
```

```
1 fonction fact(2)
2   si 2 = 0 alors
3     retourner 1
4   sinon
5     retourner
        2 × fact(1)
```

```
1 fonction fact(1)
2   si 1 = 0 alors
3     retourner 1
4   sinon
5     retourner
        1 × fact(0)
```

```
1 fonction fact(0)
2   si 0 = 0 alors
3     retourner 1
4   sinon
5     retourner
        0 × fact(−1)
```

6

2

1

1

Évaluation d'un programme récursif Au cours de l'évaluation d'une fonction récursive, le système doit gérer de multiples appels à la fonction avec des valeurs de paramètres différents et se souvenir, lorsqu'un appel est terminé, de l'appel "parent" à qui renvoyer les résultats. Pour cela, chaque appel récursif dispose en quelque sorte d'une copie locale des variables de la fonction. Ainsi, cela permet aux calculs réalisés pendant un appel avec une certaine valeur de paramètre de ne pas interférer avec le calcul de la fonction pour d'autres valeurs de paramètres. Pour gérer les appels, le système utilise une pile d'exécution (ou pile d'environnement) qui fonctionne pour schématiser de la façon suivante :

— A chaque appel de fonction, le contexte (valeur des variables, adresse de la fonction appelante) est sauvegardé. On dit qu'il est empilé.

— A chaque retour, on dépile le contexte pour retrouver l'ensemble des informations utiles pour continuer la calcul au niveau supérieur.

Un second exemple : la suite de Fibonacci Cette suite bien connue se définit par la schéma de récurrence suivant :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \text{ pour } n \geqslant 0 \end{cases}$$

Cette définition peut être exploitée facilement pour concevoir un algorithme récursif de calcul du terme de rang n de la suite (cf figure 4.2). Ce terme dépend cette fois de deux des termes précédents (et nécessite donc de connaître au moins deux valeurs initiales F_0 et F_1 pour énoncer des conditions d'arrêts).

```
1 fonction fibo(n)
2   si n = 0 alors
3     retourner 0
4   sinon si n = 1 alors
5     retourner 1
6   sinon
7     retourner fibo(n − 1) + fibo(n − 2)
```

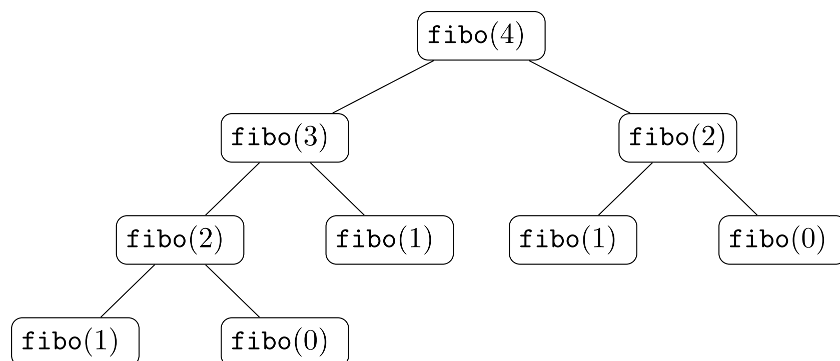
FIGURE 4.2 – Fibonacci : programme récursif

Notez que l'imbrication des conditions ne sert à rien, on aurait pu donner la forme "à plat" présentée dans la figure 4.3.

```
1 fonction fibo(n)
2   si n = 0 alors  retourner 0
3   si n = 1 alors  retourner 1
4   retourner fibo(n − 1) + fibo(n − 2)
```

FIGURE 4.3 – Fibonacci : version alternative du programme récursif

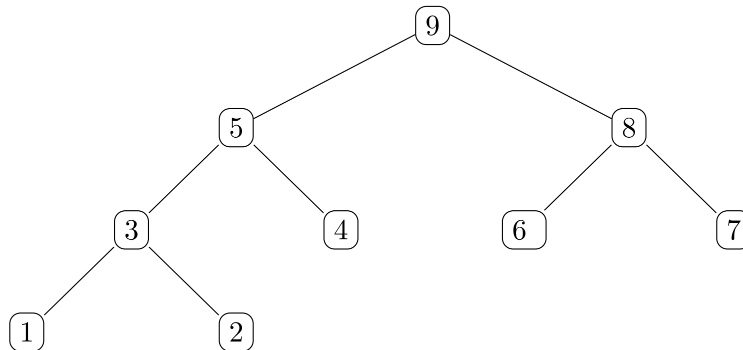
L'évaluation d'un programme avec plusieurs appels récursifs est plus compliquée à comprendre. L'enchaînement des appels récursifs est donné par *l'arbre des appels* ci-dessous :



La valeur d'un noeud n'est connue que lorsque les valeurs de tous les noeuds plus petits sont connus. L'ordre d'évaluation des fils dans l'arbre est donné par l'ordre des appels

récursifs. Par exemple, pour l'évaluation de `fibo`(4) on calcule, dans l'algorithme tel que nous l'avons donné, d'abord `fibo`(3) puis `fibo`(2)). Cet ordre d'évaluation correspond à un parcours de l'arbre d'appel dit "parcours en profondeur postfixé" : on visite les noeuds à partir de sa racine en choisissant toujours le fils le plus à gauche non visité et en remontant dès qu'un sous-arbre a été visité complètement. Lors du dernier passage en un noeud, la valeur de celui-ci devient disponible : ceci est normal, la valeur d'un noeud ne peut être connue que lorsque la valeur de tous ses fils est connue !

On reprend notre arbre d'appel (de l'exemple), en numérotant chacun des noeuds pour indiquer l'ordre dans lequel la valeur de ceux-ci est calculée et donc disponible pour le noeud parent (qui appelle ces valeurs) :



Implémenté tel quel, cet algorithme est particulièrement inefficace : les valeurs de la fonction sont systématiquement recalculées. Sur l'exemple, `fibo`(2) est appelé 2 fois pour connaître `fibo`(4).

En fait, si on note a_n le nombre d'additions faites par cet algorithme sur l'entrée n, la suite (a_n) est définie par $a_0 = a_1 = 0$ et pour $n \geqslant 2$, $a_n = a_{n-1} + a_{n-2} + 1$. On en déduit que

$$a_n \sim \alpha \left(\frac{1+\sqrt{5}}{2} \right)^n$$

pour un certain réel $\alpha > 0$. Ainsi, cet algorithme fait un nombre exponentiel d'additions.

Il est bien sûr facile de calculer F_n en calculant successivement les termes $F_0, F_1, F_2, \ldots$ par une boucle, ce qui ne fait que $n - 1$ additions (pour $n \geqslant 2$). L'algorithme récursif vu ci-dessus est donc particulièrement inefficace.

Remarque sur la structure d'une fonction récursive On peut remarquer qu'un programme récursif doit présenter un ou plusieurs cas de base, appelés aussi conditions d'arrêt. Le ou les appels à lui-même doivent permettrent de se ramener au cas de base (si ce n'est pas le cas, le programme récursif et mal conçu et peut alors entrer dans une boucle infinie).

4.2 Méthode de mémoïsation

En adaptant directement la définition de la suite de Fibonacci pour obtenir une fonction calculant le n-ième terme de cette suite, nous avons vu que la fonction obtenue est exponentielle car elle recalcule de nombreuses fois les mêmes termes de la suite. Pour remédier en partie à cette inefficacité, on peut employer une technique assez générale de *mémorisation* (le terme consacré est *mémoïsation*) : la première fois qu'une valeur d'une fonction f est calculée, on stocke celle-ci dans un tableau (ou une structure de donnée adéquate). A chaque

appel potentiel pour un paramètre k, on teste alors si la valeur $f(k)$ a déjà été calculée en inspectant le tableau. Si c'est le cas, on récupère celle-ci et on ne la recalcule pas.

Pour l'exemple du calcul de la suite de Fibonacci, cela pourrait s'illustrer abstraitement par la fonction présenté dans la figure 4.4.

```
1  fonction fiboM(n, T, Vu)
2      si n = 0 alors retourner 0
3      si n = 1 alors retourner 1
4      si Vu(n − 2) = Faux alors
5          T(n − 2) ← fiboM(n − 2, T, Vu)
6          Vu(n − 2) = Vrai
7      si Vu(n − 1) = Faux alors
8          T(n − 1) ← fiboM(n − 1, T, Vu)
9          Vu(n − 1) = Vrai
10     retourner T(n − 1) + T(n − 2)
```

FIGURE 4.4 – Fibonacci : programme récursif avec mémoïsation. La liste Vu indique si on a déjà calculé la valeur (elle devrait être initialisée à $Faux$) et T collecte les valeurs calculées. Notez qu'on a besoin d'une structure de donnée pour laquelle les modifications de T et Vu sont visibles en sortie des appels de fonctions (être mutable suffit).

4.3 Du récursif à l'itératif : cas de la récursion terminale

Tout programme récursif peut être réécrit en un programme équivalent qui n'utilise pas d'appel récursif (on parlera de programme *itératif*) : une méthode générale consiste à gérer "à la main" la pile des appels du programme récursif correspondant. Mais ceci est très technique.

Il existe d'autres cas où le passage d'un programme récursif à un programme itératif est plus aisé. C'est le cas pour algorithme de type récursif terminal.

Un programme est récursif terminal s'il contient un seul appel récursif et ne contient aucune instruction à exécuter après (ou plutôt, au retour de celui-ci). En d'autres termes, la solution est fournie par le résultat de l'appel récursif. Un programme récursif terminal a donc globalement la forme suivante où g, s sont des fonctions et $c(x)$ une expression[1].

```
1  fonction f(x)
2      si c(x) alors retourner g(x)
3      retourner f(s(x))
```

FIGURE 4.5 – Programme récursif terminal

Un algorithme itératif équivalent est donné par la fonction de la figure 4.6

Algorithme d'Euclide L'algorithme d'Euclide permet de calculer le pgcd de deux nombres. Il est présenté ci-dessous dans sa version récursive.

1. Noter qu'on ne sait pas, sans information supplémentaire sur g et s si le programme va s'arrêter. Pour que cela soit le cas, il faut que $s(x)$ soit en quelque sorte plus petit ou plus "simple" que x : si, par exemple, x est un entier, une condition suffisante est $s(x) < x$

```
1 fonction f(x)
2 |  tant que c(x) = Faux faire
3 |  |  x ← s(x)
4 |  retourner g(x)
```

FIGURE 4.6 – Programme itératif correspondant au programme récursif terminal

```
1 fonction euclidRec(a, b)
2 |  si b = 0 alors retourner a
3 |  retourner euclidRec(b, a mod b)
```

La méthode de transformation d'un algorithme récursif en itératif expliquée ci-dessus s'applique directement, on est bien dans le cas d'un récursivité terminale.

Exercice 4.1. *(Avant de lire la suite)* Déterminer quelles sont les fonctions c, g et s dans le cas de cette fonction, puis effectuer la transformation expliquée ci-dessus pour transformer un algorithme récursif terminal en un algorithme itératif.

On obtient l'algorithme itératif suivant.

```
1 fonction euclid(a, b)
2 |  r ← a mod b
3 |  tant que r ≠ 0 faire
4 |  |  a ← b
5 |  |  b ← r
6 |  |  r ← a mod b
7 |  retourner b
```

Complexité de l'algorithme d'Euclide Pour évaluer le nombre de divisions réalisé par l'algorithme d'Euclide, on adopte une démarche différente de ce que l'on a fait précédemment en examinant à quelle vitesse les valeurs de reste et de quotient (valeurs successives de a et b dans la boucle) décroissent d'un passage à l'autre. Soit n, le nombre d'étapes de l'algorithme. On pose $a = r_0$ et $b = r_1$. On a :

$$\begin{cases} r_0 & = r_1 \cdot q_1 + r_2 \\ r_1 & = r_2 \cdot q_2 + r_3 \\ r_2 & = r_3 \cdot q_3 + r_4 \\ \vdots & \vdots \vdots \\ r_{n-2} & = r_{n-1} \cdot q_n + r_n \\ r_{n-1} & = r_n \cdot q_{n+1} + r_{n+1} \end{cases}$$

où $r_{n+1} = 0, r_n = (a, b)$ (on note (a, b) le pgcd de a et b). Comme r_2 est le reste de la division de r_0 par r_1, on a : $r_2 < r_1$ et donc $r_2 < \frac{1}{2}r_0 = \frac{1}{2}b$. Pour la même raison, quelque soit $i \leqslant n - 1$, $r_i < \frac{1}{2}r_{i-2}$. On montre facilement par induction que quelque soit i tel que $2i \leqslant n - 1$:

$$r_{2i} < \left(\frac{1}{2}\right)^i b.$$

Si m l'entier maximal tel que $n \geqslant 2m$ (i.e. $n = 2m$ ou $n = 2m + 1$). Alors $1 \leqslant (a, b) = r_n < (\frac{1}{2})^m b$ et donc $b > 2^m$. Comme $2m + 1 \geqslant n$, on a : $m \geqslant \frac{n-1}{2}$ et donc $b > 2^{(n-1)/2}$. Dans tous les cas, $\log_2(b) > \frac{n-1}{2}$.

Que nous dit cette dernière équation ? Que le nombre d'étapes, n, est borné par un $O(\log_2(b))$.

On peut montrer que cette borne est atteinte lorsque $a = F_{n+1}$ et $b = F_n$ où F_n et F_{n+1} sont respectivement les n-ième et $(n+1)$-ième nombres de la suite de Fibonacci. Dans ce cas, il est facile de voir que le premier reste obtenu est F_{n-2}, le deuxième F_{n-3}, etc. Ainsi, l'algorithme prend de l'ordre de n étapes sur cette entrée, c'est-à-dire de l'ordre de $C \cdot \log_2 b$ étapes. Ainsi, la borne supérieure $O(\log_2 b)$ obtenue sur la complexité de cet algorithme ne peut être améliorée : il n'est pas possible de faire une analyse plus fine de la complexité de cet algorithme pour améliorer cette borne.

Factorielle Noter que le calcul récursif de la factorielle vu précédemment n'est pas une récusivité terminale : le calcule de $n!$ fait un appel récursif pour calculer $(n-1)!$ mais fait encore une opération (la multiplication par n) pour retourner le résultat. On peut cependant transformer l'algorithme récursif de la factorielle en ajoutant un accumulateur pour le transformer en algorithme récursif terminal, puis appliquer la méthode ci-dessus. On obtient l'algorithme itératif suivant pour calculer la factorielle :

```
1 fonction fact(n)
2 |  r ← 1
3 |  tant que n ≠ 0 faire
4 |  |  r ← n × r
5 |  |  n ← n − 1
6 |  retourner r
```

FIGURE 4.7 – Programme itératif pour la factorielle

4.4 Récursivité mutuelle ou croisée

Deux programmes récursifs peuvent avoir des définitions interdépendantes : chacun des deux peut faire appel à l'autre dans sa définition. Considérons l'exemple classique de la figure 4.8 de test de la parité d'un entier (sans opération arithmétique autre que le prédécesseur) :

```
1 fonction pair(n)
2 |  si n = 0 alors retourner Vrai
3 |  retourner impair(n − 1)

4 fonction impair(n)
5 |  si n = 0 alors retourner Faux
6 |  retourner pair(n − 1)
```

FIGURE 4.8 – Récursivité mutuelle : exemple

A l'exécution, on constate que `pair`(n) renvoie *Vrai* is n est pair et *Faux* si n est impair (`impair`(n) a un comportement opposé). Comme annoncé, la fonction `pair` fait appel à la fonction `impair` et réciproquement. Chaque appel à l'autre fonction est réalisé avec un paramètre *plus petit*. Cela permet de montrer facilement que les deux fonctions s'arrêtent sur toute entrée (entière - nous n'avons pas géré les erreurs de paramètres d'entrée) et renvoient un résultat.

Bien que les fonctions `pair` et `impair` ne fassent pas appel directement à elles-mêmes, il s'agit bien d'un comportement récursif.

4.5 Objets définis récursivement

Un cas où il est très naturel (et beaucoup plus simple) d'utiliser des algorithmes récursifs est quand manipule des objets définis récursivement, tel que des arbres. Vous étudierez des exemples de ce type en TD.

— 5 —

Tri fusion : diviser pour régner

Le principe du « diviser pour régner » consiste à résoudre un problème pour une entrée de taille n en le divisant en sous-problèmes et en combinant les réponses pour ces sous-problèmes. On en a vu un exemple simple avec la méthode de recherche dichotomique d'un élément dans une liste triée.

Le tri fusion représente une autre illustration convaincante de ce principe du « diviser pour régner » dans le domaine du tri. Il va permettre de trier une liste de taille n avec une complexité $O(n \log n)$.

5.1 Fusion de deux listes triées

L'opération fondamentale pour ce tri est la fusion de deux listes déjà triées.

Principe et pseudo-code

On dispose de deux listes d'entiers A et B indexées de 0 à $n-1$ et 0 à $m-1$ respectivement et on souhaite construire une liste C triée de taille $n+m$ constituée des éléments de A et de B.

Une première approche pourrait être de former une liste à $n+m$ éléments en recopiant les éléments de A puis ceux de B et en appliquant à cette liste une des méthodes de tri vues auparavant. Ce qui serait un gâchis ne tenant pas compte du fait que A et B sont triés.

On adopte plutôt le principe plus économique suivant :

— On maintient à chaque instant des indices i, j, k : $0 \leqslant i < n$, $0 \leqslant j < m$ et $0 \leqslant k < n+m-1$

— Pour toute valeur de k, on compare les éléments dits "courants" $A[i]$ et $B[j]$. Si $A[i] < B[j]$, on met $A[i]$ en position k de la liste C et on incrémente i et k, sinon on met $B[j]$ et on incrémente j et k.

— Lorsqu'une des listes est complètement parcourue, on insère les éléments de celui qui reste dans C.

L'idée est que si A et B sont triés, on va placer dans C les éléments de A et de B dans l'ordre dans lequel ils apparaissent déjà dans ces listes, ce qui permet de considérer à chaque fois un seul élément de A et un seul élément de B : les premiers à ne pas avoir encore trouvé leur place dans C.

Exemple 5.1. *Ici $n = 5$ et $m = 5$. On considère les listes A et B suivantes :*

A	*1*	*4*	*7*	*8*	*10*
B	*2*	*3*	*5*	*6*	*9*

On veut les fusionner, c'est à dire former une liste C qui contient à la fois les éléments de A et de B triés en ordre croissant. Le tableau suivant montre l'entrelacement des deux listes A et B fournissant C.

C	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
A	*1*			*4*			*7*	*8*		*10*
B		*2*	*3*		*5*	*6*			*9*	

L'algorithme de la figure 5.1 met en oeuvre le principe décrit ci-dessus. Il est écrit de manière à rendre plus simple la preuve de la correction puisqu'on a un seul invariant de boucle [1]

```
1  fonction fusion(A, B)
2     n ← longueur(A)
3     m ← longueur(B)
4     Creer liste C[0 ... n + m − 1]
5     i ← 0
6     j ← 0
7     pour k ← 0 à n + m − 1 faire
8        si i > n alors
9           C[k] ← B[j]
10          j ← j + 1
11       sinon si j > m alors
12          C[k] ← A[i]
13          i ← i + 1
14       sinon si A[i] < B[j] alors
15          C[k] ← A[i]
16          i ← i + 1
17       sinon
18          C[k] ← B[j]
19          j ← j + 1
20    retourner C
```

FIGURE 5.1 – Fusion de deux listes.

Correction

Pour montrer la correction de l'algorithme, on va établir qu'une propriété reste vraie à l'issue de chaque passage dans la boucle principale (i.e. un≪ invariant de boucle ≫).

On montre l'invariant suivant :

Invariant 5.2. *À la fin d'une itération de la boucle* pour *:*

(a) On a $k = i + j - 1$ et les indices de 0 à k de C contiennent les éléments
   - *de A entre 0 et $i - 1$ et*
   - *de B entre 0 et $j - 1$ et*

1. Une manière plus naturelle de l'écrire serait de faire d'abord une boucle "tant que $i < n$ et $j < m$" où k est incrémenté à chaque étape, puis d'insérer ensuite les éléments restants.

triés en ordre croissant.

(b) Les éléments de A d'indice $\geqslant i$ et les éléments de B d'indice $\geqslant j$ sont tous supérieurs ou égaux à $C[k]$.

DÉMONSTRATION. (invariant). À la première itération on a deux cas, selon que $A[0] < B[0]$ ou $A[0] \geqslant B[0]$. Les deux cas sont symétriques, supposons donc $A[0] < B[0]$. Dans ce cas, à la fin de la première itération, on a $i = 1$, $j = 0$, $k = 0$ et $C[0] = A[0]$. On a donc bien la propriété (a). Comme A et B sont triés, $A[i'] \geqslant A[0] = C[0]$ pour tout $i' \geqslant i = 1$ et $B[j'] \geqslant B[0] \geqslant A[0] = C[0]$ pour tout $j' \geqslant j = 0$. La propriété (b) est également vérifiée.

Supposons avoir montré la propriété pour $k = \ell$ (à l'issue de l'itération ℓ) et montrons la propriété pour $k = \ell + 1$.

Premier cas : une des listes est déjà épuisé. Le corps de la boucle *pour* ajoute en position $\ell + 1$ la plus petite valeur contenue dans la liste qui n'est pas épuisé. D'après l'hypothèse de récurrence (b), cette valeur est supérieure ou égale à $C[\ell]$ donc à la fin de l'itération, la sous-liste $C[0, \ldots, \ell + 1]$ est trié : on a montré (a). De plus, comme la liste qui n'était pas épuisée est trié, les valeurs restant dans cette liste sont toutes au moins égales à $C[\ell + 1]$: ceci montre (b).

Second cas : aucune des deux listes n'est épuisée. L'alternative détermine le plus petit élément des deux sous-listes qui n'ont pas encore été rangés dans C, cet élément est inséré en position $\ell + 1$. Par l'hypothèse de récurrence (a), la liste $C[0, \ldots, \ell]$ est triée et d'après l'hypothèse de récurrence (a), $C[\ell + 1] \geqslant C[\ell]$. La liste $C[1, \ldots, \ell + 1]$ est donc triée, ce qui montre (a). De plus cet élément $C[\ell + 1]$ est plus petit que tous les éléments restant dans A et B car les listes A et B sont triées, ce qui prouve (b). □

Lorsque on sort de la boucle *pour*, on a effectué $|A| + |B|$ itérations, C contient tous les éléments de A et B triés en ordre croissant d'après la propriété (a) de l'invariant de boucle. Ainsi, la procédure de fusion est correcte.

Complexité

La fusion de deux listes triées de longueur n et m requiert $O(n + m)$ étapes, car chaque itération de la boucle fait un nombre constant d'étapes.

5.2 Tri fusion

L'algorithme de fusion vu ci-dessus, qui tire parti du fait que les deux listes sont triées, représente un gros progrès par rapport à l'idée naïve, concaténer puis trier. Ce progrès est assez impressionnant pour nous conduire à élaborer une méthode de tri plus économe que les tris naïfs.

Dans sa version récursive, l'algorithme de tri fusion est donné dans la figure 5.2. Il consiste pour trier A à diviser A en deux listes de taille sensiblement égales, à les trier puis à fusionner les listes obtenues.

Regardons d'un peu plus près ce qu'il en est des tailles des sous-listes L_1 et L_2 dans l'algorithme en fonction de la parité de n.

Si n est pair, il s'écrit $n = 2k$ pour un certain $k > 0$. Dans ce cas $m = \lfloor n/2 \rfloor = k$ et $L_1 = L[0 : k - 1]$ et $A_2 = A[k : 2k - 1]$ sont deux listes de taille $\lfloor n/2 \rfloor$ (remarquer que dans ce cas : $\lfloor n/2 \rfloor = \lceil n/2 \rceil = k$). Si $n = 2k + 1$ est impair alors $m = \lfloor n/2 \rfloor = \lfloor (2k+1)/2 \rfloor = k$ et $A_1 = A[0 : k-1]$ est une liste de taille $k = \lfloor n/2 \rfloor$ et $A_2 = A[k : 2k]$ est de taille $k+1 = \lceil n/2 \rceil$. Cette remarque sera utile pour l'évaluation de complexité.

```
1  fonction triFusion(L)
2  |  n ← longueur(L)
3  |  si n ⩽ 1 alors retourner L
4  |  m ← ⌊n/2⌋
5  |  L1 ← L[0: m − 1]
6  |  L2 ← L[m: n − 1]
7  |  L1 ← triFusion(L1)
8  |  L2 ← triFusion(L2)
9  |  L ← fusion(L1, L2)
10 |  retourner T
```

FIGURE 5.2 – Algorithme pour le tri fusion. $L[i : j]$ fait référence à une liste comportant les éléments de L des indices i à j *inclus*.

Correction

On vérifie la terminaison et la correction de l'algorithme de tri fusion par récurrence sur la taille de la liste à trier.

Si la liste est de taille au plus 1, l'algorithme termine et trie correctement.

Soit $n > 0$. Supposons avoir montré que l'algorithme termine et trie correctement les listes de taille au plus n. Si on traite une liste de taille $n + 1$, les deux appels récursifs traitent des sous-listes de taille au plus $\lceil (n + 1)/2 \rceil \leqslant n$. Par hypothèse de récurrence, ces deux listes sont triées correctement. Comme la fusion traite correctement les paires de listes qui lui sont soumises, le résultat final est correct.

Complexité

Théorème 5.3. *Pour trier une liste de longueur n, l'algorithme de tri par fusion effectue $O(n \log n)$ étapes dans le pire des cas.*

Démonstration. Notons $t(n)$ le nombre d'étapes de calcul que prend l'algorithme de tri fusion (dans le pire cas) sur une liste de taille n. On va montrer qu'on peut choisir $C > 0$ une constante assez grande telle que pour tout $n \geqslant 2$, on ait

$$t(n) \leqslant Cn \log_2 n.$$

Ceci montrera que $t(n) = O(n \log_2 n)$.

Tout d'abord, on choisit C assez grand pour que l'inégalité soit vraie pour $n \in \{2, 3\}$. On va voir quelle condition est nécessaire sur C pour que la récurrence passe.

Soit $n \geqslant 4$ et supposons avoir montré que $t(p) \leqslant Cp \log_2 p$ pour tout $p \in \{2, 3, \ldots, n-1\}$. Le nombre d'étapes du tri fusion sur une entrée donnée (de taille au moins 4) est la somme de trois contributions :

— temps de couper la liste en deux parties ;

— temps de trier récursivement chaque partie ;

— temps de fusionner les deux parties triées.

Couper une liste de longueur n en deux listes de taille moitié environ prend un temps linéaire, donc majoré par αn pour une certaine constante α. L'algorithme de fusion a une complexité linéaire aussi : il existe donc β tel que la fusion de A et B prend un temps au plus βn si la somme des longueurs de A et B est égale à n.

Ainsi, on obtient

$$t(n) \leqslant \alpha n + t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \beta n.$$

Comme $2 \leqslant \lfloor n/2 \rfloor \leqslant \lceil n/2 \rceil \leqslant n-1$, on a par récurrence

$$t(\lfloor n/2 \rfloor) \leqslant C \lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor)$$

et

$$t(\lceil n/2 \rceil) \leqslant C \lceil n/2 \rceil \log_2(\lceil n/2 \rceil)$$

On a donc

$$t(n) \leqslant (\alpha+\beta)n + Cn \log_2(\lceil n/2 \rceil) \leqslant (\alpha+\beta)n + Cn \log_2 \frac{n+1}{2}.$$

Or

$$(\alpha+\beta)n + Cn \log_2 \frac{n+1}{2} \leqslant Cn \log_2 n$$

si et seulement si $(\alpha+\beta) \leqslant C \log_2 \frac{2n}{n+1}$. La fonction $x \mapsto \log_2 \frac{2x}{x+1}$ est croissante sur $\mathbb{R}^+$ donc on a $\log_2 \frac{2n}{n+1} \geqslant \log_2 \frac{8}{5}$ pour $n \geqslant 4$. Ainsi, il suffit de prendre $C \geqslant (\alpha+\beta)/\log_2(8/5)$. □

5.3 Optimalité du tri fusion

On rappelle qu'un tri par comparaisons est un tri où tous les tests sur L portent sur l'ordre relatif de deux éléments de L, c'est-à-dire des tests du type "$L[i] < L[j]$".

Les algorithmes de tri à bulles, tri par sélection, tri par insertion et tri fusion appartiennent tous à la classe des tris par comparaison. Nous allons montrer que le tri fusion est optimal pour les tris de cette classe.

Théorème 5.4. *Tout tri par comparaison a une complexité supérieure ou égale à $C \cdot n \log_2 n$ pour une constante $C > 0$.*

Démonstration. Le déroulement d'un algorithme de tri par comparaison sur une liste de longueur n peut être développé en un arbre binaire A_n dont les noeuds sont étiquetés par des tests de la forme "$L[i] < L[j]$?" avec $i, j \in \{1, \ldots, n\}$ et $i \neq j$.

La racine de A_n est étiquetée par le premier test qui est effectué par l'algorithme sur une liste de taille n. Noter que ce test est bien indépendant du contenu de la liste L. Par exemple, l'algorithme de tri à bulles décrit au chapitre 2 commence par effectuer le test "$L[1] < L[0]$?".

Le sous-arbre gauche de A_n correspond au déroulement de l'algorithme dans le cas où L satisfait le premier test, le sous-arbre droit au cas où L ne satisfait pas ce test. Pour reprendre l'exemple du tri à bulles, le second test consiste à comparer les éléments en position 1 et 2.

— Si $L[1] < L[0]$, les éléments en position 1 et 0 sont échangés. La nouvelle liste L' est de la forme

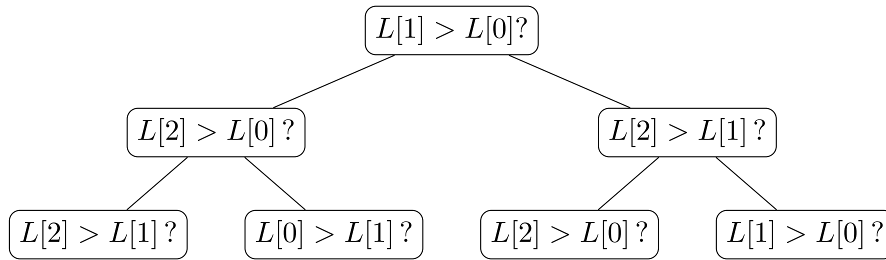
$$L' = [L[1], L[0], L[2]].$$

Le second test correspond donc à "$L[2] < L[0]$?", où $L[0]$ et $L[2]$ font référence aux valeurs de la liste initiale (comme c'est le cas dans tout l'arbre A_n).

— Si $L[1] \geqslant L[0]$, aucun échange n'est réalisé dans la liste et le second test est "$L[2] > L[1]$?".

À titre d'exemple, la figure 5.3 représente l'arbre A_3 correspondant au déroulement du tri à bulles sur une liste de longueur 3.

Un algorithme de tri par comparaison fonctionnant en temps $t(n)$, lorsqu'on regarde son fonctionnement sur les listes de longueur n, peut-être représenté par un arbre binaire de profondeur $t(n)$. Dans chaque feuille de l'arbre, puisque l'algorithme de tri est terminé, on connaît la position relative de tous les éléments de T. Plus précisément, on connaît une




```
1 fonction triBulles(L)
2   n ← longueur(L)
3   pour i ← 0 à n − 2 faire
4     pour j ← 0 à n − i − 1 faire
5       si L[j] > L[j + 1] alors  L[j], L[j + 1] ← L[j + 1], L[j]
6   retourner T
```

FIGURE 5.3 – Arbre des tests de l'algorithme du tri à bulles rappelé ci-dessus sur une liste de longueur 3.

permutation à appliquer à T permettant de ranger les éléments dans l'ordre croissant (dans le cas où les éléments de T sont tous distincts, cette permutation est unique).

Chaque permutation de n éléments doit apparaître dans au moins une feuille de l'arbre A_n. Ainsi, l'arbre A_n a au moins $n!$ feuilles. Comme un arbre binaire de profondeur h a au plus 2^h feuilles et que l'arbre A_n a une hauteur majorée par $t(n)$, on en déduit

$$t(n) \geqslant \text{hauteur}(A_n) \geqslant \log_2 (\text{nombre de feuilles de } A_n) \geqslant \log_2(n!).$$

Autrement dit, un algorithme de tri par comparaison doit, au moins sur certaines entrées de longueur n, faire au moins $\log_2(n!)$ tests.

Par la technique classique de comparaison entre séries et intégrales, il existe $C > 0$ tel que $\log_2(n!) \geqslant C \cdot nlog_2n$ ce qui démontre le théorème. □

5.4 Produit rapide de polynômes

La méthode "diviser pour régner" est une technique générale consistant à résoudre une problème de la manière suivante :

— Découper le problème initial en sous-problèmes plus petits ;

— Appliquer la méthode récursivement pour résoudre chacun des sous-problèmes (lorsque le problème est suffisamment petit, il est résolu directement sans appel récursif) ;

— Recombiner les solutions obtenues pour les sous-problèmes afin d'en déduire la solution du problème initial.

Le tri fusion est un excellent exemple d'application de ce paradigme : le point important dans cet algorithme est la recombinaison, qui correspond à l'étape de fusion de deux listes triées.

Nous allons présenter une seconde mise en oeuvre de ce principe, le produit rapide de polynômes.

Problème du produit de polynômes. Étant donné deux polynômes

$$A = \sum_{i=0}^{n} a_i X^i \quad \text{et} \quad B = \sum_{j=0}^{m} b_j X^j$$

données par la liste de leurs coefficients, le problème du produit de polynômes consiste à retourner le polynôme produit

$$AB = \sum_{k=0}^{n+m} \left(\sum_{i+j=k} a_i b_j \right) X^k$$

donné lui aussi par la liste de ses coefficients.

Il est laissé en exercice d'écrire l'algorithme naïf et de vérifier que sa complexité est $O(nm)$. Le but de la suite est de construire un algorithme plus efficace pour ce problème.

Mise en oeuvre du principe diviser pour régner. Pour simplifier la présentation, nous allons supposer que les polynômes A et B sont de même degré, de la forme $n = 2^k - 1$. Ainsi A et B ont chacun 2^k coefficients.

On peut écrire de manière unique $A = A_0 + X^{2^{k-1}} A_1$ et $B = B_0 + X^{2^{k-1}} B_1$ avec A_0 et B_0 de degré strictement inférieur à 2^{k-1}. Leur produit peut alors s'écrire

$$\begin{aligned} AB &= (A_0 + X^{2^{k-1}} A_1)(B_0 + X^{2^{k-1}} B_1) \\ &= A_0 B_0 + X^{2^{k-1}} (A_0 B_1 + A_1 B_0) + X^{2^k} A_1 B_1. \end{aligned}$$

L'idée du produit rapide de polynômes repose sur le principe suivant. Pour calculer le produit AB, il est suffisant de calculer les trois polynômes

$$A_0 B_0, \quad A_0 B_1 + A_1 B_0, \quad A_1 B_1.$$

Ceci nécessite à priori 4 produits de deux polynômes de degré 2^{k-1}, mais on peut le faire avec les seuls trois produits

$$A_0 B_0, \quad (A_0 + A_1)(B_0 + B_1), \quad A_1 B_1,$$

au prix de quelques additions de polynômes supplémentaires (mais les additions ne sont pas très coûteuses comparées aux produits). En effet,

$$A_0 B_1 + A_1 B_0 = (A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1.$$

Bien sûr, pour obtenir un produit de polynômes plus rapide que par la méthode naïve, ces produits doivent être calculés récursivement (en utilisant la même méthode consistant à couper en deux chaque polynôme).

Complexité du produit rapide de polynômes. Notons $T(n)$ le nombre d'étapes de calcul de l'algorithme de produit rapide de polynômes défini ci-dessus, pour le produit de deux polynômes ayant chacun $n = 2^k$ coefficients.

Le coût de la procédure se décompose en deux parties :

— Coût des 3 appels récursifs pour multiplier deux polynômes de tailles $n/2 = 2^{k-1}$;

— Coût de tous les autres opérations : construction des polynômes A_0, A_1, B_0, B_1 mais aussi additions de polynômes (pour construire $A_0 + A_1$ par exemple, ou encore pour construire AB à partir des 3 polynômes $A_0B_0, A_0B_1 + A_1B_0, A_1B_1$).

Le premier point a un coût $3T(n/2)$ et le second $O(n)$. Ainsi, on obtient la récurrence

$$T(n) \leqslant 3 \cdot T(n/2) + a \cdot n$$

pour une constante $a > 0$. Pour $n = 2^k$, on obtient

$$\begin{aligned} T(2^k) &\leqslant 3T(2^{k-1}) + a \cdot 2^k \\ &\leqslant 3(3T(2^{k-2}) + a \cdot 2^{k-1}) + a \cdot 2^k \\ &\leqslant 3^2 T(2^{k-2}) + a \cdot (3 \cdot 2^{k-1} + 2^k) \end{aligned}$$

Par récurrence on obtient

$$\begin{aligned} T(2^k) &\leqslant 3^k \cdot T(1) + a \cdot \left(\sum_{i=0}^{k-1} 3^i 2^{k-i} \right) \\ &\leqslant 3^k \left(T(1) + a \sum_{i=0}^{k-1} (2/3)^{k-i} \right). \end{aligned}$$

Le terme de droite étant majoré par une constante, on obtient $T(2^k) = O(3^k)$, autrement dit

$$T(n) = O\left(n^{\log_2 3}\right).$$

La complexité obtenue est donc $O(n^{1.6})$, à comparer à la complexité de la méthode naïve qui est en $O(n^2)$.

— 6 —

Programmation dynamique

Dans ce chapitre nous étudions les principes de la programmation dynamique. Il s'agit d'une méthode assez générale pour résoudre des problèmes en combinant les solutions de divers sous-problèmes. À la différence du paradigme "diviser pour régner", cette méthode s'applique dans le cas où les sous-problèmes se chevauchent.

On commence par le problème introductif du calcul d'une suite double définie par récurrence.

6.1 Calculer une suite définie par une récurrence sur deux indices

Dans cette section, f est une fonction : $\mathbb{N}^3 \to \mathbb{N}$.

On suppose que l'on dispose d'un algorithme F qui prend en arguments les représentations de trois entiers x, y, z et renvoie (la représentation de) la valeur de $f(x, y, z)$. Cette fonction est considérée comme une boîte noire. Nous pourrons utiliser la procédure F (l'appeler) sans connaître la manière dont elle est réalisée. Dans la suite f sera souvent une fonction assez élémentaire.

A partir de la fonction f, on définit des suites à deux indices de la façon suivante : on suppose données $(a_{0,j})_{j\in\mathbb{N}}$ et $(a_{i,0})_{i\in\mathbb{N}}$, et on pose :

$$a_{i+1,j+1} = f(a_{i,j+1}, a_{i+1,j}, a_{i,j}). \tag{6.1}$$

On commence par vérifier que ce procédé définit bien une suite et une seule.

Lemme 6.1. *Si deux suites $(a_{i,j})_{i,j\in\mathbb{N}}$ et $(b_{i,j})_{i,j\in\mathbb{N}}$ satisfont les relations de récurrence :*

$$a_{i+1,j+1} = f(a_{i,j+1}, a_{i+1,j}, a_{i,j}) \quad \text{et} \quad b_{i+1,j+1} = f(b_{i,j+1}, b_{i+1,j}, b_{i,j}) \tag{6.2}$$

et $a_{0,i} = b_{0,i}$ et $a_{i,0} = b_{i,0}$ pour tout $i \in \mathbb{N}$ alors $a_{i,j} = b_{i,j}$ pour tous $i, j \in \mathbb{N}$.

Démonstration. L'argument naturel procède par *récurrence*. La difficulté est de décider *sur quoi porte cette récurrence*. Les suites sont à deux indices. Il faudra développer une récurrence sur ces deux indices. On peut l'organiser de plusieurs façons. Une manière standard de procéder consiste à imbriquer deux récurrences, une récurrence dite *externe* sur i, une récurrence dite *interne* sur j.

Démonstration directe par récurrence.

Par récurrence (externe) sur i, on cherche à établir pour tout $i \in \mathbb{N}$, la propriété P_i définie par $\forall j \in \mathbb{N}, a_{i,j} = b_{i,j}$.

Par définition, P_0 est vérifiée. Si P_i est vérifiée pour $i \leqslant i_0$, alors on établit P_{i_0+1} par récurrence (interne) sur $j \in \mathbb{N}$.

On définit la propriété $Q_j^{i_0+1}$: $a_{i_0+1,j} = b_{i_0+1,j}$. On a clairement par définition que P_{i_0+1} est vérifiée si $Q_0^{i_0+1}, Q_1^{i_0+1}, \ldots, Q_j^{i_0+1}, \ldots$, sont vérifiés i.e

$$P_{i_0+1} = \bigwedge_{j\in\mathbb{N}} Q_j^{i_0+1}$$

où $\bigwedge$ est le symbole logique "ET". On note que $Q_0^{i_0+1}$ est vérifiée d'après l'énoncé car $a_{i_0+1,0} = b_{i_0+1,0}$. Pour montrer que si $Q_j^{i_0+1}$ est vérifiée pour $j \leqslant j_0$ alors $Q_{j_0+1}^{i_0+1}$ est vérifiée, on observe que

$$\begin{aligned} a_{i_0+1,j_0+1} &= f(a_{i_0,j_0+1}, a_{i_0+1,j_0}, a_{i_0,j_0}) \\ &= f(b_{i_0,j_0+1}, a_{i_0+1,j_0}, b_{i_0,j_0}) \quad \text{car } P_{i_0} \text{ est vérifée} \\ &= f(b_{i_0,j_0+1}, b_{i_0+1,j_0}, b_{i_0,j_0}) \quad \text{car } Q_{j_0}^{i_0+1} \text{ est vérifée} \\ &= b_{i_0+1,j_0+1} \,. \end{aligned}$$

On a donc la propriété P_{i_0+1} d'établie. Ce qui achève la preuve par récurrence.

Argument indirect (démonstration par l'absurde)
Supposons qu'il existe au moins un couple $i, j \in \mathbb{N}$ avec $a_{i,j} \neq b_{i,j}$. On choisit parmi ces couples (i, j), un couple minimal (i_0, j_0) (pour tout autre couple (i, j) tel que $a_{i,j} \neq b_{i,j}$, on a $i > i_0$ ou $j > j_0$). On a $i_0 > 0$ et $j_0 > 0$. Mais alors $a_{i_0-1,j_0} = b_{i_0-1,j_0}, a_{i_0,j_0-1} = b_{i_0,j_0-1}, a_{i_0-1,j_0-1} = b_{i_0-1,j_0-1}$. Et on aboutit à la contradiction :

$$a_{i_0,j_0} = f(a_{i_0-1,,j_0}, a_{i_0,,j_0-1}, a_{i_0-1,,j_0-1}) = f(b_{i_0-1,,j_0}, b_{i_0,,j_0-1}, b_{i_0-1,,j_0-1}) = b_{i_0,,j_0} \,.$$

□

On suppose que l'on dispose d'un algorithme ALGOA qui prend en argument un entier i et retourne $a_{i,0}$ et d'un algorithme ALGOB qui prend en argument un entier j et retourne $a_{0,j}$.

On peut alors proposer un algorithme récursif ALGOREC qui prend en arguments $n, m \in \mathbb{N}$ et calcule $a_{n,m}$ où la suite $(a_{i,j})_{i,j\in\mathbb{N}}$ est définie par la récurrence (6.1) et des conditions initiales $(a_{i,0})_i$ et $(a_{0,i})_i$ pour $i \in \mathbb{N}$ (disponibles grâce aux procédures à un argument ALGOA et ALGOB). Cependant, cet algorithme n'est pas efficace car on tombe dans le même piège que celui vu avec la suite de Fibonacci : les même valeurs de la suite sont calculées un nombre exponentiel de fois.

```
1 fonction algoRec(i, j)
2 |  si i = 0 alors retourner algoA(i)
3 |  si j = 0 alors retourner algoB(j)
4 |  retourner F(algoRec(i − 1, j), algoRec(i, j − 1), algoRec(i − 1, j − 1))
```

FIGURE 6.1 – Version récursive du calcul de la suite

Il existe de façons d'obtenir un algorithme efficace : appliquer le principe de mémoïsation à l'algorithme récursif ci-dessus, ou calculer la table de toutes les valeurs de la suite double dont on a besoin "dans le bon ordre". C'est ce qui est fait dans l'algorithme TDITER (TD signifie ici *tableau dynamique*). Cet algorithme itératif construit une *table bi-dimensionnelle* T indexée par $\{0, \ldots, n\} \times \{0, \ldots, m\}$. Lorsque l'algorithme termine $T[i, j] = a_{i,j}$ pour $i \leqslant n$ et $j \leqslant m$.

```
1 fonction algoIter(n, m)
2   Creer tableau T[0...n, 0...m]
3   pour i ← 0 à n faire  T[i, 0] ← algoA(i)
4   pour j ← 0 à m faire  T[0, j] ← algoB(j)
5   pour i ← 1 à n faire
6     pour i ← j à m faire
7       T[i, j] ← F(T[i − 1, j], T[i, j − 1], T[i − 1, j − 1])
8   retourner T
```

FIGURE 6.2 – Version itérative : elle illustre le principe de la programmation dynamique. Pour résoudre le problème « $a_{n,m}$ », on est amené à résoudre une collection de sous-problèmes, le calcul des $a_{i,j}$ pour $i \leqslant n$ et $j \leqslant m$ dans un certain ordre. On mémorise dans une table les solutions de ces sous-problèmes. On organise la résolution de ces sous-problèmes de façon à ce que les informations nécessaires soient disponibles lorsqu'on veut calculer $a_{i,j}$.

6.2 Plus longue sous-suite commune

Nous allons maintenant étudier un problème algorithmique sur les mots.

Formulation

Soit Σ un alphabet fini. On note Σ^* l'ensemble des mots sur l'alphabet Σ.

Dans la suite, contrairement à Python, on indexera un mot w de 1 à $|w| = \texttt{longueur}(w)$, la longueur du mot w. Pour éviter toute confusion, étant donné $i \leqslant \texttt{longueur}(u)$, on notera $u[1, ..., i]$ le sous-mot de u entre les indices 1 et i inclus i.e. $u[1, ..., i] = u[1] \cdots u[i]$. Par définition, $u[1, \ldots, 0]$ désignera le mot vide.

Une *sous-suite commune* à deux mots $u, v \in \Sigma^*$ est une suite $w \in \Sigma^*$, telle qu'il existe deux suites strictement croissantes d'indices $1 \leqslant i_1 < i_2 < \ldots < i_{|w|}$ et $1 \leqslant j_1 < j_2 < \ldots < j_{|w|}$ vérifiant

$$w_k = u_{i_k} = v_{j_k} \text{ pour } 1 \leqslant k \leqslant |w| \,.$$

Le problème de la *plus longue sous-suite commune* (LCS en anglais pour Longest Common Subsequence) consiste à déterminer une sous-suite de longueur maximale commune à u et v.

Observons qu'il peut exister plusieurs sous-suites communes de longueur maximale.

Exemple 6.2. *Considérons deux mots*

$$u = ttatatgcgt$$

et

$$v = tatcccctta,$$

une plus longue sous-suite commune à ces deux mots est tattt (mais aussi tatct).

	1	2	3	4	5	6	7	8	9	10
$u =$	**t**	*t*	**a**	**t**	*a*	**t**	*g*	*c*	*g*	**t**
	t		*a*	*t*		*t*				*t*
$v =$	**t**	**a**	**t**	*c*	*c*	*c*	*c*	**t**	**t**	*a*

Elle correspond aux indices $i_1 = 1, i_2 = 3, i_3 = 4, i_4 = 6, i_5 = 10$ *de* u *et aux indices* $j_1 = 1, j_2 = 2, j_3 = 3, j_4 = 8, j_5 = 9$ *de* v.

On peut envisager l'approche suivante : pour k allant de 1 à $\min\{|u|, |v|\}$ énumérer les sous-suites de longueur k de u et v et tester si elles sont égales (on se ramène à un problème d'égalité de mots). Remarquons qu'il y a $2^{|u|}$ sous-suites de u. Cette approche donnera donc des algorithmes de complexité exponentielle.

Remarque 6.3. *Le problème de la plus longue sous-suite commune est un problème qui a de nombreuses applications. C'est par exemple un problème que doit résoudre tout ≪ système de gestion de versions ≫. Un système de gestion de versions doit être capable d'extraire une sous-suite maximale de lignes communes à deux fichiers. C'est ce que fait la commande* diff *d'*Unix. *On retrouve ce besoin aussi dans certains algorithmes de compression de données ou dans des problèmes d'alignement de séquences ADN.*

Propriété fondamentale

Le lemme suivant est crucial pour la construction d'un bon algorithme de calcul de la plus longue sous-suite commune.

Lemme 6.4. (OPTIMALITÉ DES SOUS-STRUCTURES) *Soit $w = w_1 \dots w_k$ une plus longue sous-suite commune à $u = u_1 \dots u_n$ et $v = v_1 \dots v_m$.*

1. *Si $u_n = v_m$, alors $w_k = u_n = v_n$ et $w[1, \dots, k-1]$ est une plus longue sous-suite commune à $u[1, \dots, n-1]$ et $v[1, \dots, m-1]$;*
2. *Si $u_n \neq v_m$, alors $w_k \neq u_n$ ou $w_k \neq v_m$ et on a :*
   — *Si $w_k \neq u_n$, w est une plus longue sous-suite commune à $u[1, \dots, n-1]$ et v ;*
   — *Si $w_k \neq v_m$, w est une plus longue sous-suite commune à u et $v[1, \dots, m-1]$.*

Démonstration. On va montrer successivement les deux points.

1. Si $u_n = v_m$, supposons par l'absurde que $w_k \neq u_n$. Alors $w_k \neq v_m$ aussi, et on pourrait alors ajouter $u_n = v_m$ à w pour obtenir une sous-suite commune à u et v strictement plus longue que w : ceci est absurde. On a donc $w_k = u_n = v_n$.
   Remarquons maintenant que $w[1, \dots, k-1]$ est une sous-suite commune à $u[1, \dots, n-1]$ et $v[1, \dots, m-1]$. Si elle n'était pas de longueur maximale, on pourrait ajouter $u_n = v_m$ à cette suite et on obtiendrait une sous-suite commune à u et v plus longue que w, absurde.
2. Si $u_n \neq v_m$, on ne peut pas avoir $w_k = u_n$ et $w_k = v_m$.
   Supposons $w_k \neq u_n$. Alors w est une sous-suite commune à $u[1, \dots, n-1]$ et v. Elle est nécessairement de longueur maximale car une sous-suite commune à $u[1, \dots, n-1]$ et v est aussi une sous-suite commune à u et v, et w est une sous-suite commune à u et v de longueur maximale.
   Le cas $w_k \neq v_m$ se démontre de manière symétrique.

□

On définit A un tableau à deux dimensions indexé par

$$\{0, \dots, \texttt{longueur}(u)\} \times \{0, \dots, \texttt{longueur}(v)\},$$

où $A[i, j]$ est la longueur de la plus longue sous-suite commune à $u[1, \dots, i]$ et $v[1, \dots, j]$.

On remarque que les valeurs du tableau sont croissantes (au sens large) lorsque les valeurs d'indices augmentent :

$$A[i, j] \leqslant A[i', j'] \text{ si } i \leqslant i' \text{ et } j \leqslant j' \ .$$

Le lemme précédent permet d'obtenir une relation de récurrence sur les valeurs du tableau A.

Proposition 6.5. *On a* $A[0,j] = A[i,0] = 0$ *pour tout* $0 \leqslant i \leqslant n$ *et* $0 \leqslant j \leqslant m$*. De plus, pour tout* $0 \leqslant i < n$ *et* $0 \leqslant j < m$*, on a :*

$$A[i+1,j+1] = \begin{cases} A[i,j]+1 \text{ si } u[i+1] = v[j+1] \\ \max\{A[i+1,j], A[i,j+1]\} \text{ sinon} \end{cases}$$

Démonstration. Pour $i = 0$, le mot $u[1,\dots,0]$ est le mot vide et donc $A[0,j] = 0$ pour tout j. De même, $A[i,0] = 0$ pour tout i car $v[1,\dots,0]$ est le mot vide. Montrons maintenant la relation de récurrence.

Si $u_{i+1} = v_{j+1}$, on a $A[i+1,j+1] = A[i,j]+1$ d'après le point 1 du Lemme 6.4. Dans le cas $u_{i+1} \neq v_{j+1}$, alors $A[i+1,j+1] = \max\{A[i+1,j], A[i,j+1]\}$ d'après le point 2 du Lemme 6.4. □

La dernière proposition suggère une méthode de construction du tableau A : la ligne $A[0,j]$, pour tout $j \leqslant m$ et la colonne $A[i,0]$ pour tout $i \leqslant n$ ne demandent aucun calcul. Il suffit ensuite de progresser ligne à ligne, par indice de colonne croissant et d'utiliser la proposition.

Algorithme, programmation dynamique

La fonction `dynPLSC` (cf. figure 6.3) prend en entrée deux mots u et v et renvoie à la fois le tableau A tel que $A[i,j]$ contient la longueur de la plus longue sous-suite commune à $u[1,\dots,i]$ et $v[1,\dots,j]$. L'algorithme renvoie aussi un tableau L, de même dimension que A traçant pour le calcul de $A[i+1,j+1]$ lequel parmi $A[i,j] + \mathbf{1}_{u[i+1]=v[j+1]}$, $A[i+1,j]$ ou $A[i,j+1]$ contient le maximum. On le notera en disant que $L[i+1,j+1]$ est influencé par la case positionnée à l'ouest, le nord ou le nord-ouest. Cette information permettra d'extraire la valeur d'une plus longue sous-suite commune à u et v : ce calcul est effectué par la fonction `PLSC` (cf. figure 6.6).

```
1  fonction dynPLSC(u, v)
2  | n, m ← longueur(u), longueur(v)
3  | Créer tableaux A[0...n, 0...m] et L[1...n, 1...m]
4  | pour i ← 0 à n faire A[i,0] ← 0
5  | pour j ← 0 à m faire A[0,j] ← 0
6  | pour i ← 1 à n faire
7  | | pour i ← j à m faire
8  | | | si u[i] = v[j] alors
9  | | | | A[i,j] ← A[i−1, j−1] + 1
10 | | | | L[i,j] ← nord-ouest
11 | | | sinon si A[i−1, j] > A[i, j−1] alors
12 | | | | A[i,j] ← A[i−1, j]
13 | | | | L[i,j] ← nord
14 | | | sinon
15 | | | | A[i,j] ← A[i, j−1]
16 | | | | L[i,j] ← ouest
17 | retourner A, L
```

FIGURE 6.3 – Calcul de la plus longue sous-suite commune

$u\backslash v$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1
2	0	1	1	1	1	1	1
3	0	1	2	2	2	2	2
4	0	1	2	3	3	3	3
5	0	1	2	3	3	3	3

FIGURE 6.4 – Tableau A après exécution du programme DYNPLSC sur les entrées $u = atgcg$ et $v = agcccc$.

$u\backslash v$	1	2	3	4	5	6
1	no	o	o	o	o	o
2	n	o	o	o	o	o
3	n	no	o	o	o	o
4	n	n	no	o	o	o
5	n	n	n	o	o	o

FIGURE 6.5 – Tableau L après exécution du programme `dynPLSC` sur les mêmes données $u = atgcg$ et $v = agcccc$. Les cases étiquetées `no` donnent les indices d'une plus longue sous suite commune : $u[1], u[3], u[4]$ soit agc. Le programme `PLSC` renverra cette sous-suite.

```
1  fonction PLSC(L, u, v)
2    z, i, j ← '', longueur(u), longueur(v)        // z initialisé au mot vide
3    tant que min(i, j) > 0 faire
4      si L[i, j] = nord-ouest alors
5        z ← u[i] + z                              // caractère u[i] ajouté en tête de z
6        i ← i − 1
7        j ← j − 1
8      sinon si L[i, j] = ouest alors
9        j ← j − 1
10     sinon
11       i ← i − 1
12   retourner z
```

FIGURE 6.6 – Cette fonction renvoie une plus longue sous-séquence commune.

Coût de l'algorithme

Proposition 6.6. *Le calcul de la plus longue sous-suite commune à deux mots de longueurs n et m requiert $O(n \cdot m)$ opérations.*

Démonstration. Le calcul est effectué par les fonctions `dynPLSC` et `PLSC`. Les deux ensembles d'opération les plus coûteuses sont générés par les deux boucles imbriquées de `dynPLSC` et par la boucle `while` de `PLSC`. Dans le premier cas, il y a $n \cdot m$ passages dans le corps des boucles (chacun de ses passage requière un nombre constant d'opérations), dans le second cas, il y a au plus $n + m$ itérations. □

6.3 Le problème du sac-à-dos

Le problème du *sac-à-dos* est un problème d'optimisation algorithmique classique repré– sentant un grand nombre de problèmes se présentant dans la vie courante. Il se présente informellement de la façon suivante. Des objets sont numérotés de 1 à n. Les entiers positifs p_i et v_i représentent respectivement le poids et la valeur de l'objet numéro $i \in \{1, \ldots, n\}$. La capacité du sac (qu'on peut voir comme le poids maximum qui peut être porté) est notée C. On suppose cette capacité entière.

Le problème est de former une collection d'objets $L \subseteq \{1, \ldots, n\}$ de valeur maximale mais de poids total inférieur à la capacité du sac C. C'est-à-dire que L doit satisfaire simultanément deux contraintes :

— La somme des poids des objets de L doit être inférieure ou égale à C ;

— Parmi tous les ensembles L' satisfaisant la première condition, L est l'un de ceux dont la somme des valeurs de ces objets est maximale.

Pour décrire formellement une façon de remplir le sac-à-dos, c'est-à-dire un sous-ensemble $L \subseteq \{1, ..., n\}$, on utilise un codage binaire $x = (x_1, x_2, ..., x_n) \in \{0,1\}^n$. On a $x_i = 1$ si l'élément i est mis dans le sac, c'est-à-dire si $i \in L$, et $x_i = 0$ sinon.

Le poids et la valeur associés au remplissage peuvent être calculés à partir de x : la valeur totale contenue dans le sac est

$$\sum_{i \in L} v_i = \sum_{i=1}^{n} x_i v_i,$$

la somme des poids des objets choisis est :

$$\sum_{i \in L} p_i = \sum_{i=1}^{n} x_i p_i.$$

Le problème du « Sac-à-Dos » consiste à chercher un vecteur $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, réalisant le maximum de la valeur :

$$\sum_{i=1}^{n} x_i v_i$$

sous la contrainte de poids :

$$\sum_{i=1}^{n} x_i p_i \leqslant C.$$

On définit un tableau T de dimensions $(n+1) \times (C+1)$ où $T[i,k]$ est la valeur maximale d'un sac-à-dos qui ne contiendrait que des objets d'indices $\leqslant i$ et de poids total $\leqslant k$ lorsque $i \in \{0, \ldots, n\}$ et $k \in \{0, \ldots, C\}$:

$$T[i,k] = \max \left\{ \sum_{j \in L} v_j \ : \ L \subseteq \{1, ..., i\} \text{ et } \sum_{j \in L} p_j \leqslant k \right\}.$$

Bien sûr $T[0,k] = 0$ pour tout k tel que $0 \leqslant k \leqslant C$ et $T[i,0] = 0$ pour tout i tel que $0 \leqslant i \leqslant n$.

Chaque paire $(i,k) \in \{0, \ldots, n\} \times \{0, \ldots, C\}$ définit un sous-problème du problème de départ. Et $T[i,k]$ définit la valeur optimale d'une solution de ce sous-problème.

On veut d'abord comparer la valeur du remplissage optimal à partir des objets d'indices $\leqslant i$ dans un sac de capacité k et la valeur du remplissage optimal à partir des objets d'indices $\leqslant i+1$ dans un sac-à-dos de capacité k ou $k - p_{i+1}$.

Proposition 6.7. *On a :*

$$T[i+1,k] = \max \begin{cases} \max\{T[i,k], v_{i+1} + T[i,k-p_{i+1}]\} & si\ k \geqslant p_{i+1} \\ T[i,k] & sinon \end{cases}$$

Démonstration. Si $k < p_{i+1}$ aucun sous-ensemble $L \subseteq \{1,\ldots,i+1\}$ contenant l'élément $i+1$ ne satisfait la contrainte $\sum_{j \in L} p_j \leqslant k$: on a donc $T[i+1,k] = T[i,k]$.

On suppose maintenant que $k \geqslant p_{i+1}$. On va montrer que les sous-ensembles de $\{1,\ldots,i+1\}$ contenant $i+1$, de poids au plus k, et réalisant la valeur maximale possible sous cette contrainte ont pour valeur $v_{i+1}+T[i,k-p_{i+1}]$. On va ensuite montrer que les sous-ensembles de $\{1,\ldots,i+1\}$ ne contenant pas $i+1$, de poids au plus k, et réalisant la valeur maximale possible sous cette contrainte ont pour valeur $T[i,k]$. Ensemble, ces deux propriétés montrent que $T[i+1,k]$ est le maximum de ces deux valeurs.

Point 1. Considérons un sous-ensemble L tel que $i+1 \in L$, de poids au plus k, et maximisant la valeur dans le sac-à-dos. Sa valeur v' vérifie

$$v' = \sum_{j \in L} v_j = \sum_{j \in L \setminus \{i+1\}} v_j + v_{i+1} \,.$$

On a $\sum_{j \in L \setminus \{i+1\}} v_j \leqslant T[i,k-p_{i+1}]$. Si on avait par ailleurs $\sum_{j \in L \setminus \{i+1\}} v_j < T[i,k-p_{i+1}]$ il existerait un sous-ensemble L' de $\{1,\ldots,i\}$ de poids au plus $k-p_{i+1}$ et de valeur strictement supérieure à $v'-v_{i+1}$. Mais alors $L' \cup \{i+1\}$ aurait une valeur $> v'$, absurde. On a montré que $v' = v_{i+1} + T[i,k-p_{i+1}]$.

Point 2. Considérons un sous-ensemble L tel que $i+1 \notin L$, de poids au plus k, et maximisant la valeur dans le sac-à-dos. Sa valeur v' vérifie $v' \leqslant T[i,k]$ car le poids total des éléments de L est au plus k, et $L \subset \{1,\ldots,i\}$. Si on avait $v' < T[i,k]$, ceci contredirait la maximalité de v', donc $v' = T[i,k]$. □

L'algorithme de la figure 6.7 effectue $O(n \cdot C)$ opérations et détermine la valeur optimale d'un remplissage vérifiant la contrainte de capacité, soit $T[n,C]$.

```
1  fonction sac-a-dos(p, v, C)
2  |  n ← longueur(p)
3  |  Créer tableau T[0...n, 0...C]
4  |  pour i ← 0 à n faire  T[i,0] ← 0
5  |  pour j ← 0 à C faire  T[0,j] ← 0
6  |  pour i ← 1 à n faire
7  |  |  pour k ← 1 à C faire
8  |  |  |  si k ⩾ p[i] alors
9  |  |  |  |  T[i,k] ← max{T[i−1,k], v[i] + T[i−1,k−p[i]]}
10 |  |  |  sinon
11 |  |  |  |  T[i,k] ← T[i−1,k]
12 |  retourner T
```

FIGURE 6.7 – Sac-à-dos : calcul de la table.

On peut modifier l'algorithme pour qu'il détermine une collection d'objets qui réalise la valeur optimale d'un remplissage vérifiant la contrainte de capacité. Il suffit pour cela d'utiliser la même technique que celle qui nous a permis de reconstruire une solution optimale

pour les sous-séquence communes : calculer un tableau annexe U en même temps que le tableau T pour ce souvenir d'où vient le maximum.

Une case du tableau $U[i,k]$ pourrait avoir dans ce cas une valeur booléenne : `Vrai` si une solution optimale pour le problème de paramètres (i,k) peut être obtenue en prenant l'objet i, c'est-à-dire si $T[i,k] \leqslant v_{i+1} + T[i, k - v_{i+1}]$, et `Faux` sinon.

Il est laissé en exercice de modifier l'algorithme pour calculer le tableau U et d'écrire une fonction qui, à partir de U, reconstruit une solution optimale.

Remarques sur le coût de l'algorithme L'algorithme `sac-a-dos` peut sembler à la fois simple et efficace. Il faut noter qu'il ne constitue pas une réponse complètement satisfaisante (du point de vue de la complexité) au problème posé.

Si nous devions représenter les données $C, p_1, \ldots, p_n, v_1, \ldots, v_n$ du problème sur une machine binaire, nous utiliserions :

— $\log_2 C$ bits pour représenter C ;

— $\sum_{i=1}^{n} \log_2 p_i$ bits pour représenter les p_i ;

— $\sum_{i=1}^{n} \log_2 v_i$ bits pour représenter les v_i.

En fait il faudrait utiliser un peu plus de symboles binaires pour indiquer la ponctuation. On peut donc majorer le nombre de bits nécessaires à la représentation des données par

$$2n \left(1 + \log_2 \max\{\max_{i \leqslant n} p_i, \max_{i \leqslant n} v_i\} \right) + \log_2 C.$$

Le nombre d'opérations effectuées par notre algorithme n'est pas majoré par une puissance de la taille des données. On peut en effet se contenter d'envisager les données qui vérifient $\log_2 \max(\max_{i \leqslant n} p_i, \max_{i \leqslant n} v_i) \leqslant \log_2 W$ et $\frac{1}{2} \log_2 C \leqslant n \leqslant \log_2 C$. Dans ce cas $n \log_2 C \leqslant (\log_2 C)^2$ majore la taille de la représentation binaire des données. Mais alors pour tout $r \in \mathbb{N}$,

$$\frac{n \times C}{(n \log_2 C)^r} \geqslant \frac{C}{2(\log_2 C)^{2r-1}}.$$

Le membre droit tend vers $+\infty$ lorsque C tend vers l'infini. Au delà de cela, on peut voir que le nombre d'opérations, de l'ordre de $C \log_2 C$, est exponentiellement plus grand que l'entrée du problème, qui est de l'ordre de $(\log_2 C)^2$.

Remarque 6.8. *La performance de la programmation dynamique sur le problème du sac-à-dos est une bonne illustration des performances de la programmation dynamique. On ne connaît toujours pas de méthode capable de résoudre efficacement le problème du sac-à-dos c'est-à-dire fonctionnant en temps polynomial en la représentation en binaire de l'entrée. Pour plus de détails consulter les chapitres autour de la question $P = NP$ des ouvrages de complexité.*

— 7 —
Algorithmes simples sur les graphes

7.1 Graphes non orientés

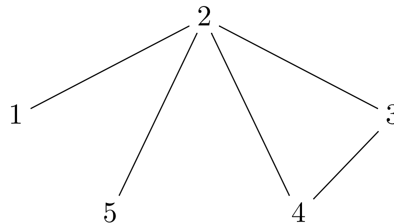
Définition 7.1. *Un* graphe simple non orienté *est un couple* (V, E) *avec* V *un ensemble et* E *un ensemble de paires à deux éléments de* V*. L'ensemble* V *est l'ensemble des sommets et* E *l'ensemble des arêtes ;* $\{x, y\} \in E$ *est appelé l'arête reliant* x *à* y*. Si* $e = \{u, v\}$ *on dit que* u *et* e *sont adjacents et que* u *est voisin de* v*. On note* $\Gamma_G(v) = \{u \in V \ : \ \{u, v\} \in E\}$ *: c'est le voisinage de* v *dans* G*. Le* degré *d'un sommet* $v \in V$ *est*

$$d_G(v) = |\{u \in V \ : \ \{u, v\} \in E\}| = |\Gamma_G(v)| \, .$$

Représentation graphique d'un graphe Le graphe $G = (V, E)$ avec

$$V = \{1, 2, \ldots, 5\}, \ E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{2, 4\}, \{2, 5\}\}$$

est représenté comme ceci :



Quelques graphes usuels

— Pour $n \geqslant 1$, le *graphe complet* à n sommets K_n. Il est défini par $K_n = (V, E)$ avec $V = [n]$ et $E = \{\{i, j\} \,|\, 1 \leqslant i < j \leqslant n\}$. Par exemple :

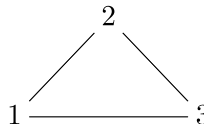
⋆ K_1 :

1

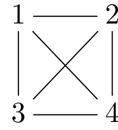
⋆ K_2 :

1 —— 2

⋆ K_3 :



⋆ K_4 :

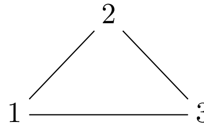


— Pour $n \geqslant 3$, le *cycle* à n sommets C_n défini par $C_n = (V, E)$ avec $V = [n]$ et

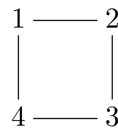
$$E = \{\{1,2\}, \{2,3\}, \dots, \{n-1,n\}, \{n,1\}\}.$$

Par exemple :

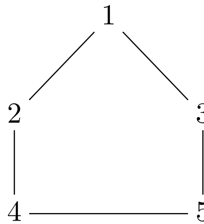
⋆ C_3 :



⋆ C_4 :



⋆ C_5 :



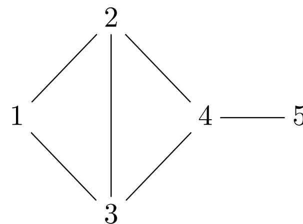
— Pour $n \geqslant 0$, le chemin de longueur n P_n défini par $P_n = (V, E)$ avec $V = \{0, 1, \dots, n\}$ et $E = \{\{i, i+1\} \ : \ 0 \leqslant i < n\}$.

Matrice d'adjacence d'un graphe Soit $G = (V, E)$ un graphe avec $V = \{v_1, \dots, v_n\}$.

Définition 7.2. *La matrice d'adjacence de G est la matrice M de taille $n \times n$ définie par :*

$$M_{i,j} = \begin{cases} 1 \ si \ \{v_i, v_j\} \in E \\ 0 \ sinon. \end{cases}$$

Exemple Considérons le graphe suivant G :



Sa matrice d'adjacence est égale à :

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Remarque 7.3. *La matrice d'adjacence d'un graphe non orienté est une matrice symétrique et avec des* 0 *sur la diagonale.*

Représentation d'un graphe en algorithmique Pour manipuler un graphe en algorithmique, il faut se donner la liste des sommets et des arêtes. On peut supposer que l'ensemble des sommets est $\{1, \ldots, n\}$ quitte à les renommer. Il faut donc se donner le nombre de sommets n, et l'ensemble des arêtes.

La première façon de représenter le graphe est de donner la matrice d'adjacence du graphe comme décrit ci-dessus (en Python, on peut représenter une matrice par une liste de listes). Si le graphe a n sommets et m arêtes, notons que cette représentation a toujours une taille $\Theta(n^2)$.

La seconde façon de se donner un graphe est par liste d'adjacence. Pour chaque sommet v, on donne la liste des voisins de v. Ainsi, pour le graphe précédent à 5 sommet, on se donnerait la liste

$$[[2, 3], [1, 3, 4], [1, 2, 4], [2, 3, 5], [4]].$$

La première liste correspond à la liste de tous les voisins du sommet 1, la deuxième liste à la liste des voisins du sommets 2, etc.

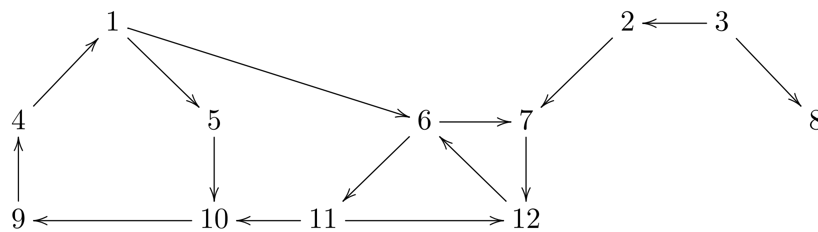
Remarquons que cette seconde représentation est plus économique. La taille de cette représentation est $\Theta(n + m)$: si le graphe a peu d'arêtes, la taille de cette représentation est significativement plus petite. (En revanche, dans le cas où le graphe aurait toutes les arêtes, il aurait de l'ordre de n^2 arêtes et les deux représentations ont une taille qui est du même ordre de grandeur $\Theta(n^2)$.)

Graphe orienté

Définition 7.4. *Un graphe orienté est un couple $G = (V, A)$ avec $A \subset V \times V$. Les éléments de A sont appelés arcs.*

On les représente graphiquement comme les graphes non orientés mais en mettant une flèche sur les arcs : un arc (u, v) est représenté par une flèche allant de u vers v.

Exemple



On note $\Gamma_G^+(v) = \{w \in V \ : \ (v, w) \in A\}$. Le *degré sortant* d'un sommet $v \in V$ est

$$\mathrm{d}_G^+(v) = \left|\Gamma_G^+(v)\right|.$$

Si $(u, v) \in A$, c'est-à-dire s'il existe un arc de u vers v, nous diront que v est un successeur de u. Ainsi, $\Gamma^+(u)$ est l'ensemble des successeurs de u.

Chemins orientés et circuits Un chemin dans un graphe orienté est une suite de sommets $(u_0, u_2, \ldots, u_p)$ tel que $(u_i, u_{i+1}) \in A$ pour tout $i \in \{0, \ldots, p-1\}$. Ce chemin est dit fermé si $p > 0$ et $u_0 = u_p$. Un chemin fermé est appelé un circuit.

Un chemin $(u_0, \ldots, u_p)$ est appelé chemin élémentaire si tous les u_i sont distincts. Un circuit élémentaire est un chemin fermé $(u_0, u_2, \ldots, u_p)$ avec $\{u_0, \ldots, u_{p-1}\}$ distincts.

Exercice Donner la liste de tous les circuits élémentaires du graphe orienté précédent. Donner un exemple de circuit qui n'est pas élémentaire.

Représentation d'un graphe orienté en algorithmique On représente un graphe orienté informatiquement de la même manière qu'un graphe non orienté. Par la matrice d'adjacence ou par la liste d'adjacence. Dans le cas de la matrice d'adjacence, on aura $M_{i,j} = 1$ si (i,j) est un arc et $M_{i,j} = 0$ sinon. Cette matrice n'est plus nécessairement symétrique. Dans le cas d'une représentation par liste d'adjacence, un graphe $G = (V, A)$ avec $V = \{1, \ldots, n\}$ sera représenté par $[L_1, \ldots, L_n]$, une liste de n listes telle que L_i contient la liste des sommets de l'ensemble $\Gamma^+(i) = \{j \in V \ : \ (i,j) \in A\}$.

7.2 Parcours en profondeur

Dans cette section, on va s'intéresser à la question suivante : étant donné un graphe orienté et un sommet v, quels sont les sommets accessibles depuis v via un chemin ?

L'idée est la suivante, pour un graphe dont l'ensemble des sommets est $\{1, ..., n\}$: on garde en mémoire une liste `Visité` des sommets qu'on a déjà rencontrés

pour éviter de boucler (un tableau indicé de 1 à n qui contient Vrai si le sommet a été visité, Faux sinon).

On se donne aussi une liste `Àexplorer` qui contient les sommets dont on va explorer les voisins plus tard. Au début `Àexplorer=[v]`, et `Visité[i]=Faux` pour tout $i \neq v$, et `Visité[v]=Vrai`. Tant que la liste `Àexplorer` n'est pas vide, on prend le dernier élément u de `Àexplorer`, et pour tous les voisins w de u qui n'ont pas encore été visités, on les ajoute à la fin de `Àexplorer` et on les marque comme visités.

Dans le pseudo code qui suit, on suppose comme en Python, on a des fonctions

- `L.pop()` qui renvoie le dernier élément de la liste et l'enlève de la liste
- `L.append(x)` qui ajoute un élément `x` en fin de liste,

Le programme renvoie, étant donné un graphe G et un sommet v, une liste `Visité` de booléens telle que `Visité`$[w]$ est vrai ssi il existe un chemin de v à w.

```
1  fonction parcoursProfondeur(G, v)
2  |  Àexplorer ← [v]
3  |  pour tous u sommet de G faire
4  |  |  Visité[u] ← Faux
5  |  Visité[v] ← Vrai
6  |  tant que Àexplorer non vide faire
7  |  |  u ← Àexplorer.pop()
8  |  |  pour tous w ∈ Γ+(u) faire
9  |  |  |  si non Visité[w] alors
10 |  |  |  |  Àexplorer.append(w)
11 |  |  |  |  Visité[w] ← Vrai
12 |  retourner Visité
```

Correction de l'algorithme. Convenons qu'un sommet u est *visité* si `Visité[u]` est égal à `Vrai`. La vérification de la correction de l'algorithme se montre via l'invariant de boucle suivant : pour tout i, à la fin de la i-ième itération de la boucle tant que, tous les éléments visités sont reliés à v, et la liste `Àexplorer` contient uniquement des sommets visités. La démonstration de cet invariant de boucle par récurrence est laissée en exercice.

On voit aussi que pour chaque sommet, au moment ou il est marqué comme visité, il est aussi ajouté dans la liste `Àexplorer`, et ne pourra ensuite plus y retourner. Comme à chaque étape de la boucle, on enlève un élément de `Àexplorer`, on voit qu'en au plus n itérations (où n est le nombre de sommets), on sort de la boucle en tant que. En particulier, notre algorithme termine bien.

Il s'agit maintenant de montre que pour tout sommet s, on a que s est visité ssi il existe un chemin de v vers s.

D'après notre invariant de boucle, il est clair que si s est visité, il existe un chemin de v vers s. La réciproque demande un peu plus de travail et se prouve par l'absurde : supposons avoir un sommet s non visité, mais tel qu'il existe un chemin de v vers s. Notons c un tel chemin. Soit alors w le premier élément de c non visité, comme v est visité on a nécessairement $w \neq v$, et on peut donc considérer le prédécesseur t de w dans c. Comme w est le premier élément de c non visité, t est visité. Mais alors, t a du être ajouté à la liste `Àexplorer` à un moment, et donc comme `Àexplorer` finit par être vide, à un moment t était en fin de liste et donc on a eu l'affectation $u \leftarrow t$ dans le programme. Ainsi, on aurait du marquer w comme visité, ce qui est contradictoire.

On conclut que les sommets visités sont exactement les sommets s pour lequels il existe un chemin de v à s, ce qu'il fallait démontrer.

Complexité de l'algorithme. Si le graphe a n sommets, la première boucle pour tous u sommet de G a une complexité en $O(n)$, et la boucle tant que est itérée au plus n fois comme expliqué précédemment. Elle contient une exploration des voisins de u, qui se fait en $O(n)$, et on conclut donc que la complexité du parcours en profondeur est en $O(n^2)$.

Remarquons que si le graphe est donné par liste adjacence, l'exploration des voisins de chaque u se fait une seule fois, et nécessite autant d'étape de calculs que u a d'arêtes sortantes. Ainsi la complexité est en $O(n+m)$, où m désigne le nombre d'arêtes.

7.3 Détection d'un circuit dans un graphe orienté

Considérons un graphe orienté $G = (V, A)$. Notre but est de déterminer s'il contient un circuit. On peut tout d'abord remarquer la propriété suivante.

Proposition 7.5. *Un graphe orienté possède un circuit si et seulement s'il possède un circuit élémentaire.*

Démonstration. Un circuit élémentaire est un cas particulier de circuit, ce qui démontre le sens de droite à gauche. Réciproquement, supposons que le graphe orienté G possède un circuit. Considérons $(u_0, u_1, \ldots, u_p = u_0)$ un circuit de longueur minimale (ie avec p minimal). Si on avait deux sommets identiques $u_i = u_j$ avec $i < j$, alors $(u_i, u_{i+1}, \ldots, u_j = u_i)$ formerait un circuit de longueur strictement plus courte : c'est absurde. Les sommets sont donc tous distincts et le circuit $(u_0, u_1, \ldots, u_p)$ est donc élémentaire. □

Algorithme de détection de circuit On décrit maintenant un algorithme qui prend en entrée un graphe orienté $G = (V, A)$ et retourne *vrai* s'il possède un circuit et *faux* sinon. L'algorithme se décrit ainsi :

- Tant qu'il existe un sommet de degré sortant nul :
  - Choisir un tel sommet v et le supprimer ;
- Si le graphe obtenu est vide retourner *faux* sinon retourner *vrai*

Notons que supprimer un sommet u d'un graphe (V, A) signifie supprimer ce sommet de la liste des sommets V et supprimer bien sûr tous les arcs de la forme (v, u) et tous les arcs de la forme (u, v). Ainsi, si $G = (V, A)$, le graphe $G \setminus \{u\}$ est le graphe $(V \setminus \{u\}, A \cap (V \setminus \{u\}) \times (V \setminus \{u\}))$.

Correction de l'algorithme. On montre d'abord les deux lemmes suivants.

Lemme 7.6. *Soit $G = (V, A)$ un graphe orienté tel que $d^+(u) > 0$ pour tout $u \in V$. Alors le graphe G possède un circuit.*

Démonstration. On part d'un sommet quelconque $v_0 \in V$. Comme son degré sortant est non nul, on peut considérer v_1 tel que $(v_0, v_1) \in A$. Par récurrence, on construit donc un chemin $(v_0, v_1, \ldots, v_i)$ dans le graphe. Comme le graphe a un nombre fini de sommets, il existe une première étape q tel que v_q a déjà été vu, c'est-à-dire tel que $v_q = v_p$ avec $p < q$. Ainsi G possède le circuit $(v_p, v_{v+1}, \ldots, v_q)$. □

Lemme 7.7. *Soit $G = (V, A)$ un graphe orienté et $v \in V$ tel que $d^+(v) = 0$. Le graphe G est sans circuit si et seulement si le graphe $G \setminus \{v\}$ est sans circuit.*

Démonstration. Si $G \setminus \{v\}$ possède un circuit alors G aussi (le même circuit).

Réciproquement supposons que G possède un circuit C. Si le circuit C passait par v, il serait de la forme $(\ldots, v, v', \ldots)$, on aurait donc $(v, v') \in A$, ce qui impliquerait $d^+(v) > 0$. C'est absurde. Ainsi le circuit C ne passe pas par v et c'est aussi un circuit de $G \setminus \{v\}$. □

On va maintenant montrer la correction de l'algorithme par récurrence sur n, le le nombre de sommets de $G = (V, A)$.

— Si $n = 0$ est sans circuit et l'algorithme est correct.

— Supposons maintenant que G possède $n > 0$ sommets. Si $d^+(u) > 0$ pour tout $u \in V$, l'algorithme s'arrête immédiatement et retourne vrai ce qui est la réponse correcte d'après le Lemme 7.6. Sinon, l'algorithme enlève un sommet v du graphe vérifiant $d^+(v) = 0$ et continue l'algorithme sur $G \setminus \{v\}$. Comme le nombre de sommets de $G \setminus \{v\}$ est $n-1 < n$, l'algorithme est correct pour $G \setminus \{v\}$ par hypothèse de récurrence. Il retourne *faux* si $G \setminus \{v\}$ sans circuit et *vrai* sinon. Mais d'après le Lemme 7.7, G est sans circuit si et seulement si $G \setminus \{v\}$ est sans circuit. L'algorithme est donc correct pour G.

Implémentation de l'algorithme. Plutôt que de retirer effectivement le sommet du graphe, il est plus simple de marquer les sommets que l'on va retirer. On va donc prendre un tableau de booléens (indexé par les sommets) initialement à *faux* pour tous les sommets et "marquer" les sommets que l'on retire (mettre à *vrai* le tableau pour ce sommet). L'algorithme peut alors se décrire de la façon suivant.

— Initialisation : aucun sommet n'est marqué

— Tant qu'il existe un sommet dont tous les successeurs sont marqués :

  — Choisir un tel sommet v et le marquer

— Si tous les sommets v sont marqués, retourner *faux* sinon retourner *vrai*

Complexité de l'algorithme. Notons n le nombre de sommets du graphes et m son nombre d'arcs. On marque les sommets en gardant un tableau de booléens $M[1, \ldots, n]$: $M[i]$ est à *vrai* si le sommet i est marqué est *faux* sinon. Initialiser le tableau M prend un temps $O(n)$.

Prenons le cas où le graphe est représenté par une matrice d'adjacence. Pour déterminer s'il existe un sommet dont tous les successeurs sont marqués, il faut inspecter chaque ligne

de la matrice une par une. Cela prend un temps $O(n^2)$. Marquer un sommet prend un temps $O(1)$. Ainsi, une itération de la boucle *tant que* prend a pour complexité $O(n^2)$. Comme on fait au plus n itérations de cette boucle (car il y a au plus n sommets à marquer), la complexité totale est $O(n^3)$.

Si le graphe est donné par une liste d'adjacence, la complexité est $O(n^2 + nm)$ car une itération de la boucle *tant que* coûte $O(n + m)$.

7.4 Tri topologique

Considérons un graphe orienté $G = (V, A)$ à n sommets sans circuit. On voudrait ordonner les sommets de sorte que si on les dispose dans cet ordre sur une droite, tous les arcs vont de la gauche vers la droite. C'est ce qu'on appelle un tri topologique.

Autrement dit, on cherche à déterminer une bijection π de V sur $\{1, \ldots, n\}$ telle que pour tout $i, j \in V$,

$$(i, j) \in A \Rightarrow \pi(i) < \pi(j).$$

Exercice : montrer que ce n'est pas possible si G contient un circuit.

On suppose maintenant que G est sans circuit. On va voir qu'on peut adapter l'algorithme décrit dans la section précédente pour effectuer un tri topologique de G, c'est-à-dire calculer une bijection π ayant la propriété décrite ci-dessus.

L'algorithme se décrit ainsi. Sur un graphe G à n sommets :

- Pour i de n à 1 par pas de -1 :
  - Choisir un sommet v de degré sortant nul
  - $\pi(v) \leftarrow i$
  - $G \leftarrow G \setminus \{v\}$
- Retourner π

La correction de l'algorithme se fait par récurrence sur le nombre de sommets. Pour un graphe à 1 sommet c'est clair. Supposons avoir montré la propriété pour un graphe à n sommets et montrons le pour un graphe $G = (V, A)$ à $n + 1$ sommets.

Comme le graphe G est sans circuit, un sommet de degré sortant nul existe dans G à la première étape existe (d'après le Lemme 7.6). Soit v le sommet choisi par l'algorithme et posons $G' = G \setminus \{v\}$. Remarquons que G' est aussi sans circuit (d'après le Lemme 7.7).

L'algorithme pose $\pi(v) = n + 1$ puis continue avec le graphe G' (ce qui revient à appliquer l'algorithme en entier à G'). L'algorithme étant correct pour un graphe à n sommets, l'algorithme ci-dessus appliqué à G' permet d'obtenir un tri topologique de G', c'est-à-dire une bijection π' de $V \setminus \{v\}$ sur $\{1, \ldots, n\}$ telle que pour tout $i \neq j$ avec $i, j \in V \setminus \{v\}$,

$$(i, j) \in A \Rightarrow \pi'(i) < \pi'(j).$$

Le bijection π retournée par l'algorithme sur G est définie par $\pi(v) = n+1$ et $\pi(u) = \pi'(u)$ pour tout $u \in V \setminus \{v\}$.

Vérifions que π réalise bien un tri topologique.

- Pour $(u, u') \in A$ avec $u, u' \in V \setminus \{v\}$, on a $\pi(u) < \pi(u')$;
- Pour tout arc $(u, v) \in A$, on a $\pi(u) < \pi(v)$ car $\pi(u) \leqslant n$ et $\pi(v) = n + 1$;
- Il n'existe par d'arc de la forme (v, u) dans A car $d^+(v) = 0$.

La complexité du tri topologique est la même que la complexité de détection de circuit vue précédemment.

— 8 —

Algorithmes de plus court chemin

On considère un graphe orienté $G = (V, A)$. Ce graphe est en plus pondéré, ce qui signifie qu'à chaque arc $(u, v) \in A$ est associé un poids $w(u, v)$.

On appelle longueur d'un chemin $(u_0, u_1, \ldots, u_p)$ la somme des poids des arcs le long du chemin :

$$\sum_{i=0}^{p-1} w(u_i, u_{i+1}).$$

La distance d'un sommet u à un sommet v, notée $\delta(u, v)$, est la longueur minimale d'un chemin de u à v.

8.1 Plus courts chemins d'une source unique à tous les autres sommets

Étant donné $s \in V$, appelé sommet source, on veut calculer la distance la plus courte de ce sommet à tous les autres sommets du graphe : c'est-à-dire que pour tout $v \in V$, on veut calculer $\delta(s, v)$. On voudrait aussi, étant donné un sommet v, pouvoir construire facilement un chemin de longueur minimale de s à v.

On va utiliser deux tableaux indexés par l'ensemble des sommets V : un tableau d qui à la fin de l'algorithme contiendra les distances des différents sommets à s, et un tableau p. À la fin de l'algorithme $p[v]$ contiendra un prédécesseur de v le long d'un chemin de longueur minimale de s à v. Ainsi, si on pose $u_0 = v$ et $u_{i+1} = p[u_i]$ et si m est le plus petit entier tel que $u_m = s$, alors $u_m, u_{m-1}, \ldots, u_1, u_0$ sera un chemin de longueur minimale de s à v.

Initialisation Au début, on pose $d[s] = 0$ et $d[v] = +\infty$ pour tout $v \neq s$. On pose aussi $p[v] = \text{null}$ pour tout v. Cela signifie que $p[v]$ n'est pas encore défini.

Traitement d'un arc La traitement de l'arc (u, v) consiste à tester si, pour aller de s à v, on ne peut pas obtenir un chemin plus court que ce qui a été trouvé jusque là en allant de s à u, puis en prenant l'arc (u, v) : si c'est la cas, les tableaux d et p sont mis à jour en conséquence. L'algorithme est décrit Figure 8.1.

```
1 fonction traitementArc(G, u, v)
2 |  si d[v] > d[u] + w(u, v) alors
3 |  |  d[v] ← d[u] + w(u, v)
4 |  |  p[v] ← u
```

Figure 8.1 – Traitement d'un arc. On suppose que la fonction a accès aux poids du graphe w ainsi qu'aux tableaux d et p qui peuvent être modifiés par cette fonction.

La propriété fondamentale du traitement d'un arc est résumée dans le lemme suivant.

Lemme 8.1. *Si* $(s = u_0, \ldots, u_p)$ *est un plus court chemin de* u_0 *à* u_p *et si on fait une séquence de traitements d'arc* $a_1, \ldots, a_q$ *qui contient comme sous-séquence les arcs*

$$(u_0, u_1), (u_1, u_2), \ldots, (u_{p-1}, u_p)$$

(après la phase d'initialisation décrite ci-dessus), alors $d[u_p] = \delta(s, u_p)$ *à la fin de l'algorithme.*

Démonstration. Notons tout d'abord que la traitement d'un arc ne peut que diminuer $d[v]$, et qu'on a toujours $\delta(s, v) \leqslant d[v]$ au cours de l'algorithme.

On montre la propriété par récurrence sur p. Pour $p = 0$ c'est clair car $d[s] = 0$ initialement et le traitement d'une arête ne peut que diminuer $d[s]$. A la fin de l'algorithme, on a donc $d[s] = 0 = \delta(s, s)$.

Supposons avoir montré la propriété pour un chemin de p arcs et montrons le pour un chemin de $p+1$ arcs : notons $(u_0, \ldots, u_p, u_{p+1})$ ce chemin. Une première partie de l'algorithme traite dans l'ordre les p premiers arcs de ce chemin. A ce stade, on a $d[u_p] = \delta(s, u_p)$ par hypothèse de récurrence.

La seconde partie de l'algorithme traite une suite d'arcs dans laquelle apparaît (au moins une fois) l'arc (u_p, u_{p+1}). Avant le traitement de cet arc, on a toujours $d[u_p] = \delta(s, u_p)$ car chaque traitement d'arc ne peut que diminuer $d[u_p]$, mais jamais en dessous de $\delta(s, u_p)$. Après le traitement de l'arc (u_p, u_{p+1}), on a

$$\delta(s, u_{p+1}) \leqslant d[u_{p+1}] \leqslant d[u_p] + w(u_p, u_{p+1}) = \delta(s, u_p) + w(u_p, u_{p+1}) = \delta(s, u_{p+1}).$$

L'égalité de droite vient du fait que $(u_0, \ldots, u_p, up + 1)$ est un plus court chemin de u_0 à u_{p+1}. Ainsi, $d[u_{p+1}] = \delta(s, u_{p+1})$ après le traitement de cet arc. Les traitements d'arcs suivant (s'il y en a) ne modifieront plus cette valeur, comme nous l'avons déjà vu. □

Algorithme de Bellman-Ford On présente une première version où tous les arcs ont un poids positif ou nul. Cet algorithme consiste à faire une séquence de traitement d'arcs universelle au sens où elle contiendra toute séquence d'arcs de longueur $n - 1$.

```
1 fonction bellmanFordPositif(G, s)
2 | n ← nombre de sommets de G
3 | initialiser d et p pour la souce s
4 | pour i ← 1 à n − 1 faire
5 | | pour tous (u, v) arc de G faire
6 | | | traitementArc(G, u, v)
```

FIGURE 8.2 – Algorithme de calcul des plus courts chemins d'une source s à tous les autres sommets (cas de poids positifs).

Correction de l'algorithme Soit $v \in V$. Il existe nécessairement un plus court chemin de s à v ne passant pas deux fois pas le même sommet (sinon, on pourrait le raccourcir en supprimant le cycle correspondant). Soit $(s = u_0, u_1, \ldots, u_p = v)$ un tel plus court chemin. La séquence de traitements d'arcs faite dans l'algorithme de Bellman-Ford contient la séquence d'arcs correspondante comme sous-séquence, puisqu'elle traite $n - 1$ fois de suite tous les arcs, et $p \leqslant n - 1$. D'après le Lemme 8.1, on a donc $d[v] = \delta(s, v)$ à la fin de l'algorithme.

Complexité de l'algorithme Notons n le nombre de sommets et m le nombre d'arcs du graphe G. L'initialisation prend un temps $O(n)$. Le traitement d'un arc prend un temps $O(1)$. Comme on fait la boucle principale $n-1$ fois, la complexité totale est $O(n+(n-1)t)$ où t est la complexité d'une itération de la boucle principale. Pour effectuer cette boucle principale, il faut parcourir tous les arcs. Si le graphe est donné par une matrice d'adjacence, cela prend un temps $t = O(n^2)$. Si le graphe est donné par liste d'adjacences, parcourir toutes les arêtes prend un temps $t = O(n + m)$.

Ainsi, l'algorithme de Bellman-Ford a une complexité $O(n^3)$ si le graphe est donné par sa matrice d'adjacence, et $O(nm + n^2)$ s'il est donné par listes d'adjacence.

Variante : plus courts chemins d'une source unique à tous les autres dans un DAG Un DAG (Directed Acyclic Graph) est un graphe orienté sans circuit. Dans ce cas spécial, on peut améliorer l'algorithme de Bellman-Ford en faisant moins de traitements d'arcs.

L'algorithme est présenté à la Figure 8.3. La complexité ainsi que la correction de l'algorithme sont laissés en exercice.

```
1 fonction plusCourtsCheminsDAG(G, s)
2 |  Trier les sommets de G dans l'ordre topologique
3 |  initialiser d et p pour la souce s
4 |  pour tous u sommet de G (dans l'ordre topologique) faire
5 |  |  pour tous v tel que (u, v) arc de G faire
6 |  |  |  traitementArc(G, u, v)
```

FIGURE 8.3 – Algorithme de calcul des plus courts chemins d'une source s à tous les autres sommets dans un DAG.

```
1 fonction bellmanFord(G, s)
2 |  n ← nombre de sommets de G
3 |  initialiser d et p pour la souce s
4 |  pour i ← 1 à n − 1 faire
5 |  |  pour tous (u, v) arc de G faire
6 |  |  |  traitementArc(G, u, v)
7 |  pour tous (u, v) arc de G faire
8 |  |  si d[v] > d[u] + w(u, v) alors retourner Faux
9 |  retourner Vrai
```

FIGURE 8.4 – Algorithme de calcul des plus courts chemins d'une source s à tous les autres sommets (cas de poids positifs ou négatifs), avec détection du cas où un circuit de poids négatif accessible depuis la source.

Cas des poids positifs et négatifs L'algorithme de Bellman-Ford peut être adapté au cas où les arcs ont des poids positifs et négatifs. Dans ce cas, la distance de la source s à tous les autres noeuds est bien définie uniquement si un circuit de poids strictement négatif n'est pas accessible depuis s.

L'algorithme est donné Figure 8.4. Cet algorithme renvoie *vrai* si aucun circuit de poids négatif n'est accessible depuis s et dans ce cas, d et p contiennent les même informations que dans l'algorithme précédent (distance à s d'un sommet et prédécesseur le long d'un plus

court chemin). Si un circuit de poids négatif est accessible depuis s, l'algorithme renvoie *faux*.

La première partie de l'algorithme est similaire à la première version de l'algorithme de Belmann-Ford. La seconde partie a pour but de détecter la présence d'un circuit de poids négatif.

Correction de l'algorithme. Un circuit de poids négatif est un circuit $(c_0, \ldots, c_p = c_0)$ tel que

$$\sum_{i=0}^{p-1} w(c_i, c_{i+1}) < 0.$$

Si aucun circuit de poids négatif n'est accessible depuis s, la première partie de l'algorithme calcule bien les distances (la démonstration faite dans le cas de poids strictement positifs est valide aussi). Il reste à montrer que dans ce cas, l'algorithme retourne *vrai*. Comme $d[u] = \delta(s, u)$ pour tout sommet u, l'inégalité $d[u] \leqslant d[v] + w(u, d)$ est équivalente à $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$. Or

$$\delta(s, v) \leqslant \delta(s, u) + \delta(u, v) \leqslant \delta(s, u) + w(u, v).$$

La première inégalité est l'inégalité triangulaire et la seconde et la seconde vient de la majoration $\delta(u, v) \leqslant w(u, v)$. L'algorithme retourne donc *vrai*.

Réciproquement, supposons qu'un circuit de poids négatif $(c_0, \ldots, c_p = c_0)$ est accessible depuis s. Comme ce circuit est accessible, on a $d[c_i] < +\infty$ pour tout i à la fin de l'algorithme. Supposons par l'absurde que l'algorithme retourne *vrai*. Cela signifie que

$$d[c_i] \leqslant d[c_{i+1}] + w(c_i, c_{i+1})$$

pour tout $i \in \{0, \ldots, p-1\}$, ce qui peut se réécrire

$$d[c_i] - d[c_{i+1}] \leqslant w(c_i, c_{i+1}).$$

En sommant ces p inégalités, on obtient

$$0 \leqslant \sum_{i=0}^{p-1} w(c_i, c_{i+1}) = w(c_0, \ldots, c_p)$$

ce qui est absurde car le circuit $(c_0, \ldots, c_p)$ est de poids strictement négatif. L'algorithme retourne donc *faux* dans ce cas, ce qui termine la preuve de correction de l'algorithme.

Complexité de l'algorithme. La complexité de l'algorithme est la même que celle de la première version de Bellman-Ford, la seconde partie de l'algorithme ayant une complexité négligeable par rapport à la première partie.

Application à la programmation linéaire La programmation linéaire consiste à chercher la valeur maximale d'une fonction linéaire $\sum_{i=1}^{n} c_i x_i$ à l'intérieur d'une région définie par des contraintes affines du type $\sum_{i=1}^{n} a_i^{(j)} x_i + a_0^{(j)} \leqslant 0$ (pour $j \in \{1, \ldots, p\}$).

Une question en particulier est de savoir si l'ensemble des poids $x \in \mathbb{R}^n$ vérifiant toutes les contraintes est vide ou non : c'est ce qu'on appelle le problème de faisabilité. Pour ce problème, la fonction $\sum_{i=1}^{n} c_i x_i$ ne joue aucun rôle.

L'algorithme de Bellman-Ford vu ci-dessus permet de résoudre le problème de faisabilité lorsque les contraintes ont une forme particulière. Plus précisément, quand toutes les contraintes sont des *contraintes de différence*, c'est-à-dire de la forme

$$x_j - x_i \leqslant \beta_{i,j}.$$

Considérons un tel système de contraintes que l'on note S. Pour un ensemble de contraintes de ce type sur n variables $x_1, \ldots, x_n$, on définit le graphe orienté pondéré suivant. Les sommets sont $v_0, v_1, \ldots, v_n$ (noter qu'il y a un sommet de plus que le nombre de variables), et les arcs sont les suivants : pour chaque contrainte $x_j - x_i \leqslant \beta_{i,j}$, on ajoute l'arc (v_i, v_j) de poids $\beta_{i,j}$. Enfin, on ajoute l'ensemble d'arcs $\{(v_0, v_i) \ : \ 1 \leqslant i \leqslant n\}$, tous de poids 0.

Proposition 8.2. *Soit un S un système de contraintes de différence et G le graphe associé. S'il n'existe pas de solution $(x_1, \ldots, x_n) \in \mathbb{R}^n$ satisfaisant S, l'algorithme de Bellman-Ford appliqué au graphe G avec pour source le sommet v_0 retourne faux. Sinon, il retourne vrai et le point $(d[v_1], \ldots, d[v_n])$ est une solution du système S.*

Démonstration. Supposons tout d'abord que l'algorithme retourne *vrai*. Dans ce cas, comme tous les sommets sont accessibles depuis v_0, nous avons $d[v_i] < +\infty$ pour tout $i \in \{1, \ldots, n\}$ à la fin de l'algorithme. Nous allons montrer que dans ce cas, le point $z = (z_1, \ldots, z_n)$ défini par $z_i = d[v_i]$ est une solution du système S. Pour cela, il faut que toutes les contraintes de différence soient satisfaites par z. La contrainte $x_j - x_i \leqslant \beta_{i,j}$ est satisfaite par z si et seulement si $d[v_j] - d[v_i] \leqslant w(i,j)$ (on rappelle que le poids de l'arc (i,j) est $\beta_{i,j}$ par définition de G). Or c'est bien le cas car on a supposé que l'algorithme de Bellman-Ford retourne *vrai*, cette contrainte est donc vérifiée sinon l'algorithme retournerait *faux* lors de la seconde phase.

Supposons maintenant que l'algorithme retourne *faux*. Cela signifie qu'il existe un cycle de poids strictement négatif dans G (accessible depuis v_0). Notons $(v_{i_0}, \ldots, v_{i_p} = v_{i_0})$ ce cycle de poids négatif. On a donc dans S les contraintes

$$x_{i_{j+1}} - x_{i_j} \leqslant \gamma_j$$

pour $0 \leqslant j \leqslant p-1$ avec $\sum_{j=0}^{p-1} \gamma_j < 0$. Une solution du système S devrait donc satisfaire l'inégalité obtenue en sommant toutes ces inégalités, soit

$$0 = \sum_{j=0}^{p-1} (x_{i_{j+1}} - x_{i_j}) \leqslant \sum_{i=0}^{p-1} \gamma_j < 0.$$

Cette inégalité ne peut pas être satisfaite, le système S n'a donc pas de solution. □