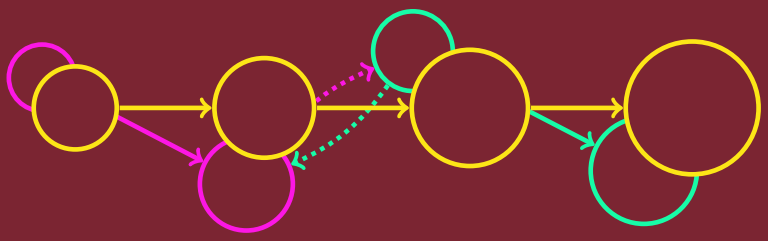


Algorithmes et structures de données

Notes de cours (IFT436)

Michael Blondin

28 novembre 2023

Avant-propos

La rédaction de ce document a été entamée à la session d'automne 2019 comme notes personnelles du cours « IFT436 – Algorithmes et structures de données » de l'Université de Sherbrooke, dans le but d'offrir des notes gratuites, libres et francophones. En particulier, la structure et le contenu de ces notes se basent sur le plan cadre établi par le Département d'informatique. Ainsi, ce document pourrait évoluer au gré des sessions.

Ces notes ne sont pas un livre: bien que je fasse un effort pour que ces notes soient lisibles sans assister au cours, et bien que l'entièreté du contenu du cours se trouve dans ce document, certains passages peuvent parfois être « expéditifs » par rapport aux explications, exemples et discussions qui surgissent en classe. Ainsi, ces notes devraient d'abord être considérées comme un complément aux séances de cours, par ex. pour réduire, voire éliminer, la prise de notes; comme références pour réaliser les devoirs; comme matériel de révision, etc.

Si vous trouvez des coquilles ou des erreurs dans le document, ou si vous avez des suggestions, n'hésitez pas à me les indiquer sur GitHub (en ajoutant un « *issue* ») ou par courriel à `michael.blondin@usherbrooke.ca`. Je remercie notamment Andréanne Allaire (A21), François Ladouceur (correcteur A21) et Luc Rainville (A21) pour l'identification de coquilles.

Légende

Observation.

Les passages compris dans une région comme celle-ci correspondent à des observations jugées intéressantes mais qui dérogent légèrement du contenu principal.

Remarque.

Les passages compris dans une région comme celle-ci correspondent à des remarques jugées intéressantes mais qui dérogent légèrement du contenu principal.

Les passages compris dans une région colorée sans bordure comme celle-ci correspondent à du contenu qui ne sera pas nécessairement présenté en classe, mais qui peut aider à une compréhension plus approfondie. Plusieurs de ces passages contiennent des preuves ou des propositions techniques.

Les exercices marqués par « ★ » sont considérés plus avancés que les autres.
Les exercices marqués par « ★★ » sont difficiles ou dépassent le cadre du cours.

L'icône «  » dans la marge fournit un lien vers du code associé au passage.

Les icônes «  » dans la marge permettent de naviguer vers les démonstrations des propositions et les solutions des exercices qui se trouvent en annexe.

Table des matières

Fondements

0

Rappel de notions mathématiques

Dans ce chapitre, nous rappelons certaines notions élémentaires de mathématiques discrètes et de logique utiles à la conception et l'analyse d'algorithmes.

0.1 Notation et objets élémentaires

0.1.1 Ensembles

Rappelons qu'un *ensemble* est une collection (finie ou infinie) d'éléments non ordonnés et sans répétitions. Par exemple, $\{2, 3, 7, 11, 13\}$ est l'ensemble des cinq premiers nombres premiers, $\emptyset$ est l'*ensemble vide*, $\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$ est l'ensemble des mots de taille deux formés des lettres `a` et `b`, et $\{0, 2, 4, 6, 8, \ldots\}$ est l'ensemble des nombres pairs non négatifs.

Nous utiliserons souvent des définitions en compréhension. Par exemple, l'ensemble des nombres pairs non négatifs peut s'écrire ainsi:

$$\{2n : n \in \mathbb{N}\}\,.$$

La taille d'un ensemble fini X est dénoté par $|X|$, par ex. $|\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}| = 3$ et $|\emptyset| = 0$. Nous écrivons $x \in X$ et $x \notin X$ afin de dénoter, respectivement, que x appartient et n'appartient pas à X. Nous écrivons $X \subseteq Y$ afin de dénoter que tous les éléments de X appartiennent à Y (*inclusion*), et nous écrivons $X \subset Y$ lorsque $X \subseteq Y$ et $X \neq Y$ (*inclusion stricte*). Lorsque $X \subseteq E$, où E est considéré comme un univers, nous écrivons $\overline{X}$ afin de dénoter le *complément*

$$\overline{X} \coloneqq \{e \in E : e \notin X\}\,.$$

Rappelons que $\cup$, $\cap$, $\setminus$ et $\times$ dénotent respectivement l'*union*, *intersection*, la *différence* et le *produit cartésien*, définis comme suit:

$$X \cup Y \coloneqq \{x : x \in X \vee x \in Y\}, \qquad X \setminus Y \coloneqq \{x : x \in X \wedge x \notin Y\},$$
$$X \cap Y \coloneqq \{x : x \in X \wedge x \in Y\}, \qquad X \times Y \coloneqq \{(x, y) : x \in X \wedge y \in Y\}.$$

L'*ensemble des sous-ensembles* de X est l'ensemble:

$$\mathcal{P}(X) \coloneqq \{Y : Y \subseteq X\}.$$

Par exemple, $\mathcal{P}(\{\mathsf{a},\mathsf{b}\}) = \{\emptyset, \{\mathsf{a}\}, \{\mathsf{b}\}, \{\mathsf{a},\mathsf{b}\}\}$ et $\mathcal{P}(\emptyset) = \{\emptyset\}$.

0.1.2 Séquences

Une *séquence* est une suite d'éléments. Contrairement aux ensembles, les éléments d'une séquence sont ordonnés (par un *indice*) et peuvent se répéter. Nous décrivons les séquences à l'aide de crochets (plutôt que d'accolades pour les ensembles). Par exemple, $s = [\mathsf{k},\mathsf{a},\mathsf{y},\mathsf{a},\mathsf{k}]$ représente la chaîne de caractères « kayak » et $f = [1,1,2,3,5,8,\ldots]$ est la séquence de Fibonacci. Nous dénotons le $i^{\text{ème}}$ élément d'une séquence t par $t[i]$ (en débutant par 1), par ex. $s[1] = \mathsf{k}$ et $f[3] = 2$. La *taille* d'une séquence finie t est dénotée par $|t|$, par ex. $|s| = 5$. Nous écrivons $s[i:j]$ afin de dénoter la sous-séquence:

$$s[i:j] \coloneqq [s[i], s[i+1], \ldots, s[j]].$$

Par convention, $s[i:j] \coloneqq [\,]$ lorsque $i > j$. Nous écrivons $s + t$ afin de dénoter la séquence obtenue en ajoutant t à la suite de s (*concaténation*).

0.1.3 Nombres

Ensembles de nombres. Nous utiliserons les ensembles standards de nombres dont les nombres *naturels*, *entiers*, *rationnels* et *réels*:

$$\begin{aligned}
\mathbb{N} &= \{0,1,2,\ldots\},\\
\mathbb{Z} &= \mathbb{N} \cup \{-n : n \in \mathbb{N}\},\\
\mathbb{Q} &= \{a/b : a,b \in \mathbb{Z}, b \neq 0\},\\
\mathbb{R} &= \mathbb{Q} \cup \left\{\pi, \sqrt{2}, \ldots\right\}.
\end{aligned}$$

Nous restreindrons parfois ces ensembles, par ex. $\mathbb{R}_{>0}$ dénote l'ensemble des nombres réels positifs. L'intervalle des entiers de a à b est dénoté $[a..b] \coloneqq \{a, a+1, \ldots, b\}$.

Logarithmes et exponentielles. Le *logarithme en base* $b \in \mathbb{N}_{\geq 2}$ est la fonction $\log_b : \mathbb{R}_{>0} \to \mathbb{R}$ telle que $b^{\log_b(x)} = x$; autrement dit, l'inverse de l'exponentielle. Lorsque $b = 2$, nous écrivons simplement log sans indice. Rappelons quelques propriétés des logarithmes et des exponentielles.

Pour tous $x, y \in \mathbb{R}_{>0}$ et $a, b \in \mathbb{N}_{\geq 2}$, nous avons:

$$\log_b(xy) = \log_b(x) + \log_b(y), \qquad \log_b(x/y) = \log_b(x) - \log_b(y),$$

$$\log_b(x^y) = y \cdot \log_b(x), \qquad \log_a(x) = \frac{\log_b(x)}{\log_b(a)},$$

$$\log_b(1) = 0, \qquad x < y \iff \log_b(x) < \log_b(y).$$

Pour tout $b \in \mathbb{N}$ et tous $x, y \in \mathbb{R}_{\geq 0}$, nous avons:

$$b^x \cdot b^y = b^{x+y}, \qquad \frac{b^x}{b^y} = b^{x-y}, \qquad (b^x)^y = b^{x \cdot y}.$$

Arithmétique modulaire. Pour tous $n \in \mathbb{N}$ et $d \in \mathbb{N}_{>0}$, nous écrivons $n \div d$ afin de dénoter la *division entière* de n par d. Le *reste* de cette division est défini par $n \bmod d \coloneqq n - (n \div d) \cdot d$. Par exemple, $39 \bmod 2 = 1$ et $39 \bmod 5 = 4$.

0.1.4 Combinatoire

La *factorielle* d'un entier $n \in \mathbb{N}$ est définie par $n! \coloneqq 1 \cdot 2 \cdots n$ avec $0! \coloneqq 1$ par convention. Par exemple, $5! = 120$ et $6! = 720$. Il y a $n!$ façons d'ordonner n objets distincts. Par exemple, il y a $3! = 6$ façons de permuter $[\mathsf{a}, \mathsf{b}, \mathsf{c}]$:

$$[\mathsf{a}, \mathsf{b}, \mathsf{c}],\ [\mathsf{a}, \mathsf{c}, \mathsf{b}],\ [\mathsf{b}, \mathsf{a}, \mathsf{c}],\ [\mathsf{b}, \mathsf{c}, \mathsf{a}],\ [\mathsf{c}, \mathsf{a}, \mathsf{b}],\ [\mathsf{c}, \mathsf{b}, \mathsf{a}].$$

Le *coefficient binomial*, lu « k parmi n », est défini par:

$$\binom{n}{k} \coloneqq \frac{n!}{k!(n-k)!} \qquad \text{pour tous } n, k \in \mathbb{N} \text{ tels que } n \geq k.$$

Par convention, $\binom{n}{k} \coloneqq 0$ lorsque $k > n$ ou $k < 0$.

Le coefficient binomial donne le nombre de façons de choisir k éléments parmi n éléments, sans tenir compte de leur ordre. Par exemple, il y a $\binom{4}{2} = 6$ façons de choisir deux éléments parmi l'ensemble $\{$●, ■, ▲, ◆$\}$:

$$\{●, ■\},\ \{●, ▲\},\ \{●, ◆\},\ \{■, ▲\},\ \{■, ◆\},\ \{▲, ◆\}.$$

0.2 Techniques de preuve

0.2.1 Preuves directes

La technique de preuve probablement la plus simple consiste à prouver un énoncé directement à partir de définitions et de propositions déjà connues. Par exemple, démontrons la proposition suivante à l'aide d'une preuve directe:

Proposition 1. *Pour tous $m, n \in \mathbb{N}$, si mn est impair, alors m et n sont impairs.*

Démonstration. Soient $m, n \in \mathbb{N}$ tels que mn est impair. Il existe $a, b \in \mathbb{N}$ et $r, s \in \{0, 1\}$ tels que $m = 2a + r$ et $n = 2b + s$. Nous avons:

$$\begin{aligned} mn &= (2a + r)(2b + s) \\ &= 4ab + 2as + 2br + rs \\ &= 2(2ab + as + br) + rs \end{aligned}$$

Puisque mn est impair, nous avons $rs = 1$, et donc $r = s = 1$ car $r, s \in \{0, 1\}$. Par conséquent, $m = 2a + 1$ et $n = 2b + 1$, et ainsi m et n sont impairs. □

0.2.2 Infirmations par contre-exemple

Il s'avère parfois possible d'infirmer un résultat à l'aide d'un contre-exemple. On peut se convaincre qu'une affirmation de la forme « $\forall x \in X : \varphi(x)$ » est fausse en exhibant un élément $x \in X$ tel que $\varphi(x)$ est faux.

Exemple.

Considérons la fonction $f(n) = 1+2+\ldots+n+\lceil (n+1)^2/9 \rceil$. En évaluant $f(0) = 1$, $f(1) = 2$, $f(2) = 4$ et $f(3) = 8$, il est tentant de conjecturer que la description de f est une façon tordue de spécifier la fonction 2^n. Ce n'est pas le cas, puisque $f(4) = 13$.

Nous avons donc infirmé notre conjecture « $\forall n \in \mathbb{N} : f(n) = 2^n$ » en identifiant le contre-exemple $n = 4$ et en argumentant qu'il enfreint la propriété: $f(4) = 13 \neq 16 = 2^4$.

0.2.3 Preuves par contradiction

L'une des techniques de preuve les plus utilisées consiste à supposer que l'énoncé dont nous cherchons à prouver la véracité est faux, puis en dériver une contradiction afin de conclure que l'énoncé était finalement vrai. Voyons un exemple d'une telle preuve:

Proposition 2. *Soit s une séquence de n éléments provenant de $\mathbb{R}$. Il existe un élément inférieur ou égal à la moyenne de s; autrement dit, il existe $i \in [1..n]$ tel que $s[i] \leq \frac{1}{n}\sum_{j=1}^{n} s[j]$.*

Démonstration. Dans le but d'obtenir une contradiction, supposons qu'il n'existe aucun tel i. Nous avons donc $s[i] > \frac{1}{n}\sum_{j=1}^{n} s[j]$ pour tout $i \in [1..n]$. Ainsi:

$$\begin{aligned}
\sum_{i=1}^{n} s[i] &> \sum_{i=1}^{n} \left(\frac{1}{n} \sum_{j=1}^{n} s[j] \right) && \text{(par hypothèse)} \\
&= \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} s[j] \\
&= \frac{1}{n} \cdot n \cdot \sum_{j=1}^{n} s[j] && \text{(car aucun terme ne dépend de } i\text{)} \\
&= \sum_{j=1}^{n} s[j].
\end{aligned}$$

Nous en concluons que la somme des éléments de s est strictement supérieure à elle-même, ce qui est une contradiction. Cela conclut la preuve. □

0.2.4 Preuves par induction

Rappelons la technique de *preuve par induction*. Soit $\varphi\colon \mathbb{N} \to \{\texttt{faux}, \texttt{vrai}\}$ un prédicat. Afin de démontrer que $\varphi(n)$ est vrai pour tout $n \geq b$, il suffit de démontrer que:

— $\varphi(b)$ est vrai;

— $\varphi(n) \implies \varphi(n+1)$ est vrai pour tout $n \geq b$.

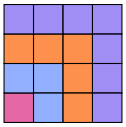
Ces deux étapes se nomment respectivement *cas de base* et *étape d'induction*.

Voyons quelques exemples de preuves par induction.

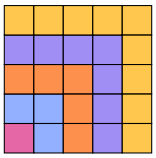
Carré d'un nombre. En inspectant quelques valeurs, la somme des n premiers nombres impairs semble égaler n^2:

$$\begin{aligned} 1 &= 1 = 1^2, \\ 1+3 &= 4 = 2^2, \\ 1+3+5 &= 9 = 3^2, \\ 1+3+5+7 &= 16 = 4^2. \end{aligned}$$

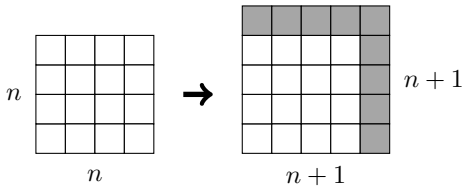
Cherchons à démontrer que cela est vrai *pour tout* $n \in \mathbb{N}_{\geq 1}$. Considérons le cas où $n = 4$. Nous pouvons visualiser graphiquement que $1+3+5+7 = 4^2$:



En ajoutant 9 cases aux carré précédent, nous obtenons un carré de 5^2 cases:



Cela nous apprend que $1+3+5+7+9 = 5^2$ et ainsi que la conjecture est vraie pour $n = 5$. Cependant, nous n'apprenons rien sur $n = 6$. En ajoutant cette fois 11 cases, nous obtiendrions un carré de 6^2 cases. En continuant de procéder de cette manière, nous pourrions couvrir toutes les valeurs de n. Malheureusement, il y en a une infinité et nous ne terminerions jamais. Cependant, ce processus se généralise de façon plus abstraite: en ajoutant $2n+1$ cases à un carré de n^2 cases, nous obtenons un carré de $(n+1)^2$ cases:



Ainsi, il nous suffit de démontrer que:

— Notre conjecture est vraie pour le premier cas, c-à-d. $n = 1$ (*cas de base*);

— Si notre conjecture est vraie pour les n premiers nombres impairs, alors elle est vraie pour les $n+1$ premiers nombres impairs (*étape d'induction*).

Autrement dit, nous pouvons démontrer notre conjecture par induction:

Proposition 3. *La somme des n premiers nombres impairs est égale à n^2. Autrement dit, $\sum_{i=1}^{n}(2i-1) = n^2$ pour tout $n \in \mathbb{N}_{\geq 1}$.*

Démonstration. Posons $s_n \coloneqq \sum_{i=1}^{n}(2i-1)$ pour tout $n \in \mathbb{N}_{\geq 1}$. Démontrons que $s_n = n^2$ par induction sur n.

Case de base ($n = 1$). Nous avons $s_n = s_1 = 2 \cdot 1 - 1 = 1 = 1^2 = n^2$.

Étape d'induction. Soit $n \geq 1$. Supposons que $s_n = n^2$. Nous avons:

$$\begin{aligned}
s_{n+1} &= \sum_{i=1}^{n+1}(2i-1) && \text{(par définition de } s_{n+1}\text{)} \\
&= \sum_{i=1}^{n}(2i-1) + (2(n+1)-1) \\
&= \sum_{i=1}^{n}(2i-1) + (2n+1) \\
&= s_n + (2n+1) && \text{(par définition de } s_n\text{)} \\
&= n^2 + (2n+1) && \text{(par hypothèse d'induction)} \\
&= (n+1)(n+1) \\
&= (n+1)^2.
\end{aligned}$$

□

Observation.

La proposition précédente donne lieu à un algorithme qui calcule le carré d'un entier naturel.

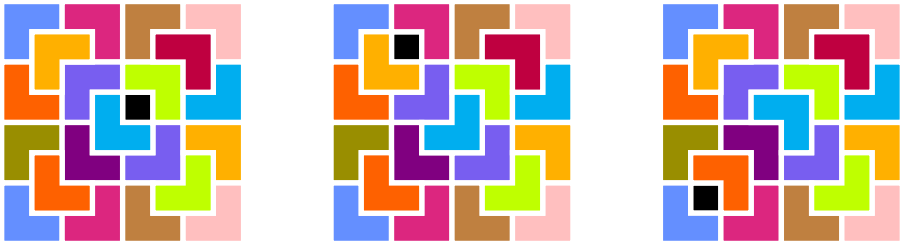
Algorithme 1 : Calcul d'un carré à partir de nombres impairs.

Entrées : $n \in \mathbb{N}_{\geq 1}$
Résultat : n^2

```
1 c ← 0; m ← 1
2 faire n fois
3 |  c ← c + m; m ← m + 2
4 retourner c
```

Pavage. Voyons un autre exemple de preuve par induction, appliqué cette fois à une autre structure discrète. Considérons le scénario suivant. Nous avons une *grille* de $2^n \times 2^n$ cases que nous désirons paver à l'aide de *tuiles* de trois cases en « forme de L »:

Il est *impossible* d'accomplir cette tâche puisqu'un pavage contient forcément un nombre de cases divisible par 3, alors que la grille contient 2^{2n} cases, donc un nombre qui n'est pas un multiple de 3. Par exemple, voici quelques tentatives de pavage d'une grille de taille 8×8:



Dans chaque cas, toutes les cases sont pavées à l'exception d'*une seule* case (colorée en noire). Cela porte à croire qu'il est toujours possible de paver la totalité d'une grille $2^n \times 2^n$ à l'exception d'une seule case *de notre choix*. Démontrons cette conjecture:

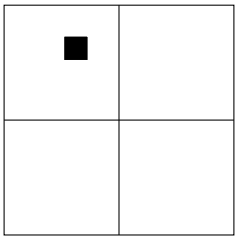
Proposition 4. *Pour tout $n \in \mathbb{N}$ et pour tous $i, j \in [1..2^n]$, il est possible de paver la totalité d'une grille $2^n \times 2^n$ à l'exception de la case (i, j).*

Démonstration. Nous prouvons la proposition par induction sur n.

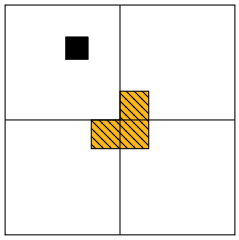
Cas de base ($n = 0$). La grille possède une seule case qui est forcément la case (i, j). Le pavage ne nécessite donc aucune tuile.

Étape d'induction. Soit $n \geq 0$. Considérons une grille de taille $2^{n+1} \times 2^{n+1}$. Supposons que toute grille de taille $2^n \times 2^n$ soit pavable à l'exception d'une case arbitraire.

Découpons notre grille en quatre sous-grilles de même taille. Remarquons que chacune d'elles est de taille $2^n \times 2^n$. La case (i, j) se trouve dans l'une des quatre sous-grilles. Par exemple, la case (i, j) se trouve ici (colorée en noire) dans la sous-grille supérieure gauche:



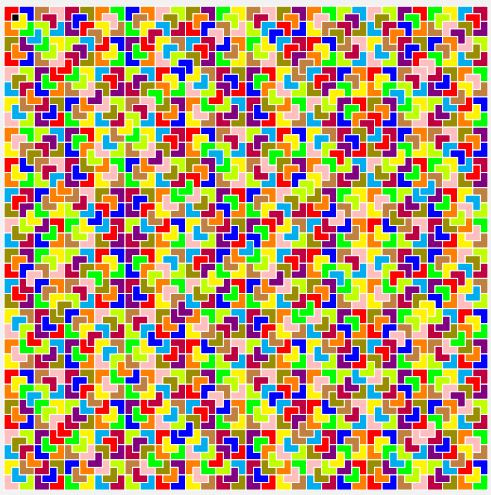
Retranchons une case à chacune des trois autres sous-grilles de façon à former un « L ». Par exemple, dans le cas précédent, nous retranchons ces trois cases hachurées en couleur:



Nous pavons ces trois cases retranchées par une tuile. Remarquons qu'exactement une case a été retranchée à chaque sous-grille. De plus, chaque sous-grille est de taille $2^n \times 2^n$. Ainsi, par hypothèse d'induction, chaque sous-grille peut être pavée, ce qui donne un pavage de la grille entière. □

Observation.

La preuve précédente est *constructive*: elle ne montre pas seulement l'existence d'un pavage, elle explique également comment en obtenir un. En effet, l'étape d'induction correspond essentiellement à quatre appels récursifs d'un algorithme, dont les paramètres sont (n, i, j), jusqu'à son cas de base qui est $n = 0$. Par exemple, une implémentation simple en PYTHON pave la grille suivante de taille 64×64, avec $(i, j) = (2, 2)$, en quelques millisecondes:



Observation.

La preuve démontre également indirectement que $2^{2n} \bmod 3 = 1$ pour tout $n \in \mathbb{N}$. Comme exercice supplémentaire, tentez de le démontrer par induction sur n (sans considérer le concept de pavage).

Jeu de Nim. Considérons un *jeu de Nim* à deux participant·e·s où:

- il y a deux piles contenant chacune $n \in \mathbb{N}_{\geq 1}$ allumettes;
- les participant·e·s jouent en alternance;
- à chaque tour, un·e participant·e choisit une pile et retire autant d'allumettes que désiré de cette pile (mais au moins une);
- la personne qui vide la dernière pile gagne.

Nous appelons la première personne *Alice* et la deuxième personne *Bob*. Montrons que Bob possède toujours une stratégie gagnante. Pour ce faire, nous utiliserons l'*induction généralisée* qui se prête mieux à ce type de problème. Afin de démontrer qu'un prédicat φ est vrai pour tout $n \geq b$, on démontre que:

- $\varphi(b)$ est vrai;
- $(\forall m \in [b..n] \;\; \varphi(m)) \implies \varphi(n+1)$ est vrai pour tout $n \geq b$.

Autrement dit, dans l'étape d'induction, on suppose que $\varphi(b), \varphi(b+1), \ldots, \varphi(n)$ sont tous vrais et on montre que $\varphi(n+1)$ est vrai.

Proposition 5. *Bob possède une stratégie gagnante au jeu de Nim pour tout $n \in \mathbb{N}_{\geq 1}$, où n désigne le nombre d'allumettes initialement dans chaque pile.*

Démonstration. Nous montrons que Bob gagne s'il retire toujours la même quantité d'allumettes qu'Alice.

Cas de base ($n = 1$). Le seul coup possible pour Alice consiste à retirer une allumette d'une pile. Bob peut donc retirer la dernière allumette et gagner.

Étape d'induction. Soit $n \geq 1$. Supposons que la stratégie fonctionne pour toute valeur initiale $m \in [1..n]$. Considérons l'instance du jeu où il y a initialement $n+1$ allumettes dans chaque pile. Si Alice retire $n+1$ allumettes d'une pile, alors Bob peut retirer les $n+1$ allumettes de l'autre pile et gagner. Sinon, si elle retire $k \in [1..n]$ allumettes d'une pile, alors Bob peut retirer k allumettes de l'autre pile. Il reste donc $m := n + 1 - k$ allumettes dans *chaque* pile. Puisque $m \in [1..n]$, par hypothèse d'induction, Bob peut gagner le jeu. □

0.3 Exercices

0.1) Quelle est la taille de l'ensemble $\{\{\mathsf{a},\mathsf{b}\},\{\emptyset,\{1,2\}\},\{\emptyset,\{\emptyset,\emptyset\}\}\}$?

0.2) Soient les ensembles $X \coloneqq \{1,5,6,\mathsf{a},9,23,\mathsf{c}\}$ et $Y \coloneqq \{-23,3,5,9,\mathsf{a}\}$. Donnez le contenu des ensembles $X \cup Y$, $X \cap Y$, $X \setminus Y$ et $Y \setminus X$.

0.3) Donnez tous les éléments de $\mathcal{P}(\{\mathsf{a},\mathsf{b},\mathsf{c}\})$.

0.4) Montrez que la somme d'un nombre pair et d'un nombre impair est forcément impaire.

0.5) Montrez que si une séquence de nombre réels contient au moins deux éléments distincts, alors elle contient au moins un élément *strictement inférieur* à sa moyenne.

0.6) ★ Montrez que $\sqrt{2} \notin \mathbb{Q}$ par contradiction.

0.7) Montrez que $|\mathcal{P}(X)| = 2^{|X|}$ pour tout ensemble fini X.

0.8) Considérons une interface graphique qui contient n cases à cocher. Vous considérez les remplacer par une seule liste déroulante afin de gagner de l'espace à l'écran. Combien d'éléments doit contenir cette liste?

0.9) Montrez que $n! > 2^n$ pour tout $n \in \mathbb{N}_{\geq 4}$.

0.10) Démontrez la formule de Pascal: $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$ pour tous $k, n \in \mathbb{N}$ tels que $n \geq k$.

0.11) La formule de Pascal permet de conclure que $\binom{n}{k} \in \mathbb{N}$ pour tous $k, n \in \mathbb{N}$. Pourquoi?

0.12) Montrez que $\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{n} = 2^n$ pour tout $n \in \mathbb{N}$.

0.13) Montrez que la somme des n premiers nombres pairs non négatifs donne $n^2 - n$. Tentez de le prouver d'au moins deux manières différentes: (1) par induction sur n; (2) directement en utilisant la proposition 3.

0.14) Montrez que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ pour tout $n \in \mathbb{N}_{>0}$.

0.15) Montrez que $2^{2n} \bmod 3 = 1$ pour tout $n \in \mathbb{N}$.

0.16) Montrez qu'il y a $|X|^n$ séquences de taille n composées d'éléments d'un ensemble fini X.

0.17) Montrez que 1$ et 3$ sont les seuls montants entiers qui ne peuvent pas être payés à l'aide de pièces de 2$ et de billets de 5$.

0.18) Identifiez l'erreur dans la « preuve » classique suivante qui « démontre » que tous les chevaux d'un groupe sont forcément de la même couleur:

Démontrons l'énoncé par induction sur le nombre n de chevaux. Si $n = 1$, alors il n'y a qu'une seule couleur. Supposons donc que l'énoncé soit vrai pour $n \geq 1$ chevaux et montrons que c'est aussi le cas pour $n+1$ chevaux. Considérons deux chevaux du groupe: *Alice* et *Bob*. Retirons Alice du groupe. Il reste n chevaux. Par hypothèse d'induction, tous ces chevaux sont de la même couleur que Bob. Faisons maintenant revenir Alice et retirons Bob du groupe. Il reste n chevaux. Par hypothèse d'induction, tous ces chevaux sont de la même couleur qu'Alice. Ainsi, Alice, Bob et tous les autres chevaux sont de la même couleur. □

0.19) L'escalade a fait son entrée aux Jeux olympiques d'été de 2020. La compétition était constituée de trois épreuves: vitesse, bloc et voie. Dans une telle compétition, le score d'un·e athlète correspond au produit de ses positions, par ex. une personne qui termine deuxième, quatrième et troisième, obtient un pointage de $2 \cdot 4 \cdot 3 = 24$. Le plus petit score l'emporte.

(a) Ce système de pointage multiplicatif est-il équivalent à un système additif?

(b) Le système de pointage dans lequel on somme le logarithme de chaque position est-il équivalent à un système additif?

(c) Les plus petit et plus grand pointages du système multiplicatif sont 1 et n^3 lorsqu'il y a n participant·e·s. Est-ce que tous les pointages de $[1..n^3]$ sont possibles?

0.20) Montrez qu'un nombre non négatif est un multiple de 3 ssi la somme de ses chiffres (en base 10) est un multiple de 3.

0.21) Démontrez, par induction sur $n \geq 1$, que tout arbre binaire de n sommets a une hauteur d'au moins $\log n$.

0.22) Démontrez cette identité bien connue des séries géométriques: $\sum_{i=0}^{n} r^i = (1 - r^{n+1})/(1 - r)$ pour tout $r \neq 1$ et $n \in \mathbb{N}$.

0.23) Le numéro d'assurance maladie d'une personne au Québec est constitué (selon Wikipédia) de ces quatre lettres et huit chiffres:

— les trois premières lettres du nom de famille à la naissance,

— la première lettre du prénom,

— les deux derniers chiffres de l'année de naissance,

— le mois de naissance (+50 pour les femmes),

— le jour de naissance,

— un numéro permettant de distinguer les personnes pour lesquelles les dix premiers caractères sont identiques,

— un chiffre-preuve.

Supposons que vous tentez d'obtenir le code QR de vaccination COVID-19 associé au numéro d'assurance maladie d'une personne. Si vous connaissez son identité, combien y a-t-il de combinaisons à essayer? Et si vous ne connaissez que son nom complet et savez qu'elle est née en 2000 ou 2001? Si tester une combinaison prend une milliseconde, combien de temps suffit afin d'obtenir le code QR?

0.24) Considérons un jeu de Nim à deux participant·e·s où:

- il y a une pile de $n \in \mathbb{N}_{\geq 1}$ allumettes;
- les participant·e·s jouent en alternance;
- à chaque tour, un·e participant·e retire une, deux ou trois allumettes de la pile;
- la personne qui vide la pile gagne.

Montrez que:

- la première personne possède une stratégie gagnante lorsque n n'est pas un multiple de 4; et
- la deuxième personne possède une stratégie gagnante lorsque n est un multiple de 4.

0.25) Montrez que $n^2 \leq 2^n$ pour tout $n \geq 4$.

0.26) Soit $i(n)$ la somme des n premiers nombres impairs. Soit $p(n)$ la somme des n premiers nombres pairs. Par exemple, $i(3) = 1 + 3 + 5 = 9$ et $p(3) = 0 + 2 + 4 = 6$. Nous avons déjà démontré que $i(n) = n^2$ (voir section 0.2.4). Que vaut $p(n)$? Démontrez votre conjecture.

0.27) ★ Nous disons qu'une grille de $m \times n$ cases est *pavable* s'il est possible de la paver entièrement à l'aide de tels *triominos*:

Une grille de $m \times n$ cases est dite *quasi-pavable* s'il est possible de la paver entièrement peu importe la case qu'on lui retranche. Par exemple, à la section 0.2.4, nous avons vu que toute grille $2^k \times 2^k$ est quasi-pavable. Démontrez les affirmations suivantes.

(a) Toute grille $3 \times 2k$, où $k \geq 1$, est pavable.

(b) Toute grille $6 \times k$, où $k \geq 2$, est pavable.

(c) La grille 14×14 est quasi-pavable.

(d) La grille 7×7 est quasi-pavable.

(e) La grille 5×5 n'est pas quasi-pavable.

(f) Toute grille $m \times n$, où $m, n \in \{4, 7, 10\}$, est quasi-pavable.

(g) Toute grille $(3a + 1) \times (3b + 1)$, où $a, b \geq 1$ est quasi-pavable.

(h) Toute grille $m \times n$, où $m, n \in \{8, 11\}$, est quasi-pavable.

(i) Toute grille $(3a + 2) \times (3b + 2)$, où $a, b \geq 2$ est quasi-pavable.

(j) Une grille $m \times n$ est quasi-pavable ssi

$$(m = n = 1) \vee (m = n = 2) \vee \\ (m, n \notin \{1, 2, 5\} \wedge m \bmod 3 = n \bmod 3 \in \{1, 2\}).$$

Cet exercice est inspiré d'une démonstration des affirmations (a), (b) et (c) proposée par Louis Desruisseaux (A23).

Remarque.

La preuve du quasi-pavage des grilles $2^k \times 2^k$ est attribuée à Golomb [Gol54]. Le cas des grilles $m \times m$ a été résolu dans [CJ86]. L'exercice ci-dessus gère plus généralement les grilles $m \times n$ et répond ainsi à la question ouverte 2 de [CJ86] (qui semble déjà avoir été résolue par J. Marshall Ash et Solomon W. Golomb).

1

Analyse des algorithmes

Ce chapitre porte sur l'analyse du *temps d'exécution* ainsi que du *bon fonctionnement* des algorithmes. Nous introduirons plusieurs outils permettant d'accomplir une telle analyse: la notation asymptotique ($\mathcal{O}$, Ω, Θ); la règle du maximum; la règle des polynômes; la règle de la limite; et les invariants. Ces notions seront accompagnées d'exemples. Nous mettons l'emphase sur l'analyse d'algorithmes *itératifs*; nous couvrirons les algorithmes récursifs au chapitre 5.

1.1 Temps d'exécution

1.1.1 Temps en fonction d'un paramètre

Soient $\mathcal{A}$ un algorithme et f la fonction telle que $f(x)$ dénote le nombre d'opérations élémentaires exécutées par $\mathcal{A}$ sur entrée x. Le *temps d'exécution de $\mathcal{A}$ dans le pire cas* est la fonction $t_{\max} : \mathbb{N} \to \mathbb{N}$ telle que:

$$t_{\max}(n) \coloneqq \max \{ f(x) : \text{entrée } x \text{ de taille } n \} .$$

Autrement dit, $t(n)$ indique le *plus grand* nombre d'opérations exécutées parmi *toutes* les entrées de taille n. Nous considérerons parfois aussi le *temps d'exécution dans le meilleur cas*, défini similairement par:

$$t_{\min}(n) \coloneqq \min \{ f(x) : \text{entrée } x \text{ de taille } n \} .$$

Par défaut, le terme « temps d'exécution » fera référence au pire cas lorsque nous ne spécifierons pas le cas dont il s'agit.

Par exemple, analysons le temps d'exécution de l'algorithme 2 qui retourne la valeur maximale d'une séquence non vide. Nous considérons les opérations suivantes comme élémentaires: l'affectation, la comparaison, l'addition et l'accès à un élément d'une séquence. La première ligne exécute toujours 3 opérations élémentaires (deux affectations et un accès). La boucle s'exécute toujours précisément $n-1$ fois, et en particulier exécute $n-1$ comparaisons. Si la condition du **si** n'est *jamais* satisfaite, alors le corps de la boucle exécute 4 opérations

Algorithme 2 : Algorithme de calcul du maximum d'une séquence.

Entrées : séquence s de $n \in \mathbb{N}_{>0}$ entiers
Sorties : valeur maximale apparaissant dans s

$i \leftarrow 2, max \leftarrow s[1]$
tant que $i \leq n$
  si $s[i] > max$ **alors** $max \leftarrow s[i]$
  $i \leftarrow i + 1$
retourner max

élémentaires (un accès, une comparaison, une addition et une affectation). Si la condition du **si** est *toujours* satisfaite, alors le corps de la boucle exécute 6 opérations élémentaires (deux accès, une comparaison, deux affectations et une addition). Ainsi, le temps d'exécution est compris entre $3+5(n-1)$ et $3+7(n-1)$. Autrement dit:

$$5n - 2 \leq t_{\min}(n) \leq t_{\max}(n) \leq 7n - 4 \qquad \text{pour tout } n \geq 1.$$

Intuitivement, l'algorithme 2 fonctionne donc en temps linéaire par rapport à n, c'est-à-dire que peu importe l'entrée, le temps d'exécution est d'environ n à certaines constantes près. À la section suivante, nous formaliserons cette notion et développerons des outils afin de faciliter l'analyse du temps d'exécution.

1.1.2 Temps en fonction de plusieurs paramètres

Pour certains algorithmes, la « taille » d'une entrée dépend de plusieurs paramètres, par ex. le nombre de lignes m et de colonnes n d'une matrice; le nombre d'éléments m d'une séquence et le nombre de bits n de ses éléments; le nombre de sommets m et d'arêtes n d'un graphe, etc. Pour ces algorithmes, la notion de temps est étendue naturellement:

$$t_{\max}(n_1, \ldots, n_k) \coloneqq \max\{f(x) : \text{entrée } x \text{ de taille } (n_1, \ldots, n_k)\},$$
$$t_{\min}(n_1, \ldots, n_k) \coloneqq \min\{f(x) : \text{entrée } x \text{ de taille } (n_1, \ldots, n_k)\}.$$

Par exemple, considérons l'algorithme 3 qui calcule la valeur maximale apparaissant dans une matrice de taille $m \times n$. Analysons le cas où la condition du **si** est *toujours* satisfaite. La première boucle de l'algorithme est toujours exécutée m fois. À sa première itération, la seconde boucle est exécutée $n-1$ fois, et pour ses itérations subséquentes, la seconde boucle est exécutée n fois. Le corps de la seconde boucle exécute 6 opérations élémentaires. La première boucle exécute ensuite 3 opérations supplémentaires. Observons finalement que chaque tour d'une boucle exécute une comparaison. Au total, nous obtenons:

$$\begin{aligned} t_{\max}(m, n) &\leq 4 + \overbrace{(1 + 7(n-1) + 3)}^{1^{\text{ère}} \text{ itér. boucle princip.}} + \overbrace{(m-1)(1 + 7n + 3)}^{\text{autres itér. boucle princip.}} \\ &= 7mn + 4m - 3. \end{aligned}$$

Algorithme 3 : Algorithme de calcul du maximum d'une matrice.

Entrées : matrice **A** d'entiers de taille $m \times n$, où $m, n \in \mathbb{N}_{>0}$
Sorties : valeur maximale parmi toutes les entrées de **A**

$i \leftarrow 1,\ j \leftarrow 2,\ max \leftarrow \mathbf{A}[1,1]$
tant que $i \leq m$
  tant que $j \leq n$
    si $\mathbf{A}[i,j] > max$ **alors** $max \leftarrow \mathbf{A}[i,j]$
    $j \leftarrow j + 1$
  $i \leftarrow i + 1$
  $j \leftarrow 1$
retourner max

Dans le cas où la condition du **si** n'est *jamais* satisfaite, l'analyse demeure la même à l'exception des 6 opérations du corps de la seconde boucle qui deviennent 4 opérations. Ainsi:

$$\begin{aligned} t_{\min}(m,n) &\geq 4 + \overbrace{(1 + 5(n-1) + 3)}^{1^{\text{ère}}\text{ itér. boucle princip.}} + \overbrace{(m-1)(1+5n+3)}^{\text{autres itér. boucle princip.}} \\ &= 5mn + 4m - 1. \end{aligned}$$

Nous en concluons donc que pour tous $m, n \geq 1$:

$$5mn + 4m - 1 \leq t_{\min}(m,n) \leq t_{\max}(m,n) \leq 7mn + 4m - 3.$$

Intuitivement, l'algorithme 3 fonctionne donc en temps $\approx m \cdot n$, c'est-à-dire que peu importe l'entrée, le temps d'exécution est d'environ $m \cdot n$ à quelques termes négligeables près. En contraste avec l'exemple précédent, ici les « termes négligeables » ne sont pas tous des constantes. La section suivante nous permettra de définir ce que nous entendons par « négligeable ».

1.2 Notation asymptotique

1.2.1 Notation $\mathcal{O}$

Nous formalisons les notions de la section précédente en introduisant une notation qui permet de comparer des fonctions *asymptotiquement*, c'est-à-dire lorsque la taille de leur entrée tend vers l'infini. Nous nous concentrons sur les fonctions à un seul paramètre. Plus précisément, nous considérons les fonctions de $\mathbb{N}$ vers $\mathbb{R}$ qui sont éventuellement positives, c'est-à-dire les fonctions appartenant à:

$$\mathcal{F} \coloneqq \{f : \mathbb{N} \to \mathbb{R} : \exists m \in \mathbb{N}\ \forall n \geq m\ f(n) > 0\}.$$

Définition 1. Soit $g \in \mathcal{F}$. L'ensemble $\mathcal{O}(g)$ est défini par:

$$\mathcal{O}(g) \coloneqq \{f \in \mathcal{F} : \exists c \in \mathbb{R}_{>0}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\ f(n) \leq c \cdot g(n)\}.$$

Nous appelons les valeurs c et n_0 une *constante multiplicative* et un *seuil*.

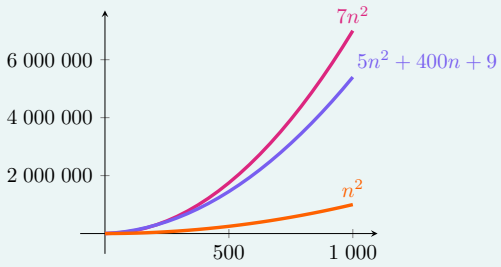
Intuitivement, $f \in \mathcal{O}(g)$ indique que f croît *aussi ou moins* rapidement que la fonction g. Reconsidérons l'algorithme 2. Nous avons déjà établi que son temps d'exécution est d'au plus $7n - 4$ pour tout $n \geq 1$. Nous avons donc:

$$\begin{aligned} t_{\max}(n) &\leq 7n - 4 && \text{pour tout } n \geq 1 \\ &\leq 7n && \text{pour tout } n \geq 0. \end{aligned}$$

Ainsi, en prenant $c := 7$ comme constante multiplicative et $n_0 := 1$ comme seuil, nous concluons que l'algorithme 2 fonctionne en temps $\mathcal{O}(n)$.

Exemples.

Considérons d'autres exemples de fonctions. Posons $f(n) := 5n^2+400n+9$. La fonction f est supérieure à n^2:



Cependant, nous pouvons montrer que $f \in \mathcal{O}(n^2)$. Nous avons:

$$\begin{aligned} f(n) &= 5n^2 + 400n + 9 \\ &\leq 5n^2 + 400n + n \cdot n && \text{pour tout } n \geq 3 \\ &\leq 5n^2 + n \cdot n + n \cdot n && \text{pour tout } n \geq 400 \\ &= 7n^2. \end{aligned}$$

Ainsi, en prenant $c := 7$ comme constante multiplicative et $n_0 := 400$ comme seuil, nous concluons que $f \in \mathcal{O}(n^2)$.

Comme autre exemple, montrons que $n^2 \in \mathcal{O}(n!)$. Pour tout $n \geq 2$:

$$\begin{aligned} n^2 &= (n-1)n + n \\ &\leq n! + n && (\text{car } n \geq 2 \text{ et } n! = 1 \cdots (n-1)n) \\ &\leq n! + n! \\ &= 2n!. \end{aligned}$$

Ainsi, en prenant 2 à la fois comme constante multiplicative et comme seuil, nous concluons que $n^2 \in \mathcal{O}(n!)$.

Dans l'exemple ci-dessus, nous avons vu que f croît au plus aussi rapidement que n^2 asymptotiquement, et pareillement pour n^2 par rapport à $n!$. Nous nous

attendons donc intuitivement à ce que $f \in \mathcal{O}(n!)$. Cela est bien le cas, puisque la relation induite par $\mathcal{O}$ est transitive:

Proposition 6. *Pour toutes $f, g, h \in \mathcal{F}$, si $f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(h)$, alors $f \in \mathcal{O}(h)$.*

Démonstration. Soient c', n_0' et c'', n_0'' les constantes multiplicative et les seuils qui montrent respectivement que $f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(h)$. Nous montrons que $f \in \mathcal{O}(h)$ en prenant $c \coloneqq c' \cdot c''$ comme constante multiplicative et $n_0 \coloneqq \max(n_0', n_0'')$ comme seuil. Pour tout $n \geq n_0$, nous avons:

$$\begin{aligned} f(n) &\leq c' \cdot g(n) && \text{(car } n \geq n_0 \geq n_0'\text{)} \\ &\leq c' \cdot c'' \cdot h(n) && \text{(car } n \geq n_0 \geq n_0''\text{)} \\ &= c \cdot h(n) && \text{(par définition de } c\text{).} \end{aligned}$$

□

Rappelons que nous avons vu que $5n^2 + 400n + 9 \in \mathcal{O}(n^2)$. Cela est plutôt intuitif puisque $5n^2$ est le terme dominant et puisque les constantes « n'importent pas » asymptotiquement. Nous formalisons ce raisonnement. Pour toutes fonctions $f, g \in \mathcal{F}$ et tout coefficient $c \in \mathbb{R}_{>0}$, les fonctions $f + g$, $c \cdot f$ et $\max(f, g)$ sont définies par:

$$\begin{aligned} (f + g)(n) &\coloneqq f(n) + g(n), \\ (c \cdot f)(n) &\coloneqq c \cdot f(n), \\ (\max(f, g))(n) &\coloneqq \max(f(n), g(n)). \end{aligned}$$

Les deux propositions suivantes montrent que les constantes qui apparaissent dans une somme de fonctions n'importent pas asymptotiquement, et qu'une somme de fonctions se comporte asymptotiquement comme leur maximum.

Proposition 7. *Nous avons $f_1 + \ldots + f_k \in \mathcal{O}(c_1 \cdot f_1 + \ldots + c_k \cdot f_k)$ pour toutes fonctions $f_1, \ldots, f_k \in \mathcal{F}$ et tous coefficients $c_1, \ldots, c_k \in \mathbb{R}_{>0}$.*

Proposition 8. *$f_1 + \ldots + f_k \in \mathcal{O}(\max(f_1, \ldots, f_k))$ pour toutes $f_1, \ldots, f_k \in \mathcal{F}$.*

Exemples.

Reconsidérons l'exemple $f(n) = 5n^2 + 400n + 9$. Montrons à nouveau que $f \in \mathcal{O}(n^2)$, cette fois plus brièvement à l'aide de nos observations.

Par la proposition 7, $f \in \mathcal{O}(n^2 + n + 1)$. Ainsi, puisque $\max(n^2, n, 1) = n^2$, la proposition 8 et la proposition 6 impliquent $f \in \mathcal{O}(n^2)$.

Voyons un autre exemple. Posons $g(n) \coloneqq 9000n^2 + 3^n + 8$. Par la proposition 7, nous avons $g \in O(n^2 + 3^n + 1)$. Observons que $\max(n^2, 3^n, 1) = 3^n$. Ainsi, par la proposition 8 et la proposition 6, nous avons $g \in \mathcal{O}(3^n)$.

Lors de l'analyse d'algorithmes, nous obtiendrons souvent des fonctions qui sont des *polynômes*. Les observations précédentes suggèrent que tout polynôme

de degré d appartient à $\mathcal{O}(n^d)$ car cela correspond à son terme dominant. Cela s'avère vrai. Cependant, nous ne pouvons pas simplement combiner les propositions précédentes pour s'en convaincre. En effet, un polynôme peut contenir des termes *négatifs*, alors que nous n'avons raisonné jusqu'ici que sur des sommes de fonctions *éventuellement positives*. Démontrons donc cette observation:

Proposition 9. *$f \in \mathcal{O}(n^d)$ pour tout polynôme $f \in \mathcal{F}$ de degré d.*

Exemples.

Par la proposition précédente, nous pouvons directement déterminer, par exemple, que $2n^3 - 5n^2 + 3n - 100 \in \mathcal{O}(n^3)$ et $n^2/8 - 9000n \in \mathcal{O}(n^2)$.

1.2.2 Notation Ω

La notation $\mathcal{O}$ nous permet de borner des fonctions supérieurement. Afin d'analyser des algorithmes, il s'avère également utile de borner des fonctions *inférieurement*. Dans ce but, nous introduisons la notation Ω.

Définition 2. Soit $g \in \mathcal{F}$. L'ensemble $\Omega(g)$ est défini par:

$$\Omega(g) \coloneqq \{f \in \mathcal{F} : \exists c \in \mathbb{R}_{>0}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\ f(n) \geq c \cdot g(n)\}\,.$$

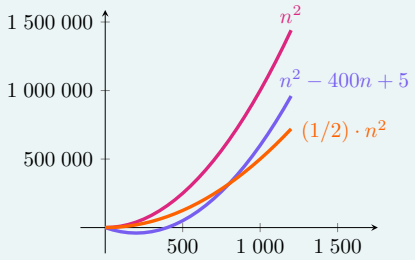
Nous appelons les valeurs c et n_0 une *constante multiplicative* et un *seuil*.

Intuitivement, $f \in \Omega(g)$ indique que f croît *au moins aussi* rapidement que la fonction g. Remarquons que la définition de $\Omega(g)$ ne diffère de celle de $\mathcal{O}(g)$ que par la comparaison « $\leq$ » qui est remplacée par « $\geq$ ». Ainsi, nous pouvons en déduire que

$$f \in \Omega(g) \iff g \in \mathcal{O}(f).$$

Exemple.

Voyons un exemple. Soit $f(n) \coloneqq n^2 - 400n + 5$ la fonction suivante:



Bien que f soit inférieure à n^2, nous avons $f \in \Omega(n^2)$. En effet:

$$\begin{aligned} f(n) &= n^2 - 400n + 5 \\ &\geq n^2 - 400n \\ &\geq n^2 - \frac{n}{2} \cdot n && \text{pour tout } n \geq 800 \\ &= \frac{1}{2} \cdot n^2. \end{aligned}$$

Ainsi, en prenant $c \coloneqq 1/2$ comme constante multiplicative et $n_0 \coloneqq 800$ comme seuil, nous concluons que $f \in \Omega(n^2)$.

De façon similaire à la notation $\mathcal{O}$, il est possible de démontrer que:

Proposition 10. *$f \in \Omega(n^d)$ pour tout polynôme $f \in \mathcal{F}$ de degré d.*

1.2.3 Notation Θ

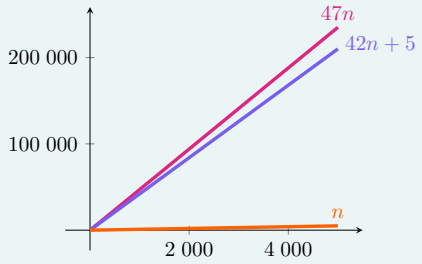
Nous introduisons une troisième et dernière notation asymptotique qui permet de borner une fonction à la fois inférieurement *et* supérieurement. Pour toute fonction $g \in \mathcal{F}$, nous définissons:

$$\Theta(g) \coloneqq \mathcal{O}(g) \cap \Omega(g).$$

Autrement dit, $f \in \Theta(g)$ lorsque $f \in \mathcal{O}(g)$ et $f \in \Omega(g)$. Intuitivement, $\Theta(g)$ décrit donc l'ensemble des fonctions qui croissent *aussi* rapidement que g.

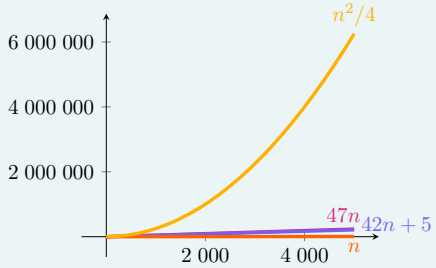
Exemples.

Par exemple, considérons la fonction $f(n) \coloneqq 42n + 5$ suivante:



Puisque f est de degré 1, nous avons $f \in \mathcal{O}(n)$. De plus, $f(n) = 42n+5 \geq 42n$ pour tout $n \in \mathbb{N}$. Ainsi, en prenant 42 comme constante et 0 comme seuil, nous concluons que $f \in \Omega(n)$ et ainsi que $f \in \Theta(n)$.

Informellement, nous pouvons voir graphiquement que les constantes multiplicatives et additives jouent un rôle négligeable lorsqu'on compare à une fonction quadratique comme $n^2/4$:



Voyons un autre exemple. Soit $g(n) := 9000n$. Nous avons $g \in \mathcal{O}(n^2)$ puisque g est un polynôme de degré 1, alors que n^2 est un polynôme de degré 2. Cependant, $g \notin \Omega(n^2)$ puisque g croît moins rapidement que n^2. Cet argument demeure intuitif et ne démontre pas cette affirmation.

Prouvons-la par contradiction. Supposons que $g \in \Omega(n^2)$. Il existe $c \in \mathbb{R}_{>0}$ et $n_0 \in \mathbb{N}$ tels que $9000n \geq c \cdot n^2$ pour tout $n \geq n_0$. En supposant que $n \geq 1$, nous pouvons diviser des deux côtés par c et n afin d'obtenir:

$$\frac{9000}{c} \geq n \qquad \text{pour tout } n \geq \max(n_0, 1).$$

Nous avons donc une constante du côté gauche qui borne supérieurement une valeur arbitrairement grande du côté droit. Il y a contradiction, ce qui démontre $g \notin \Omega(n^2)$. Nous en concluons donc que $g \notin \Theta(n^2)$.

Par la proposition 9 et la proposition 10, nous avons:

Proposition 11. *$f \in \Theta(n^d)$ pour tout polynôme $f \in \mathcal{F}$ de degré d.*

1.3 Étude de cas: vote à majorité absolue

Considérons le problème suivant: étant donnée une séquence T de n éléments, nous cherchons à déterminer si T contient une *valeur majoritaire*, c'est-à-dire une valeur qui apparaît plus de $n/2$ fois dans T. Par exemple, la séquence $[\mathsf{a}, \mathsf{b}, \mathsf{b}, \mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{c}, \mathsf{a}]$ contient la valeur majoritaire a qui apparaît 5 fois parmi 8 éléments; alors que la séquence $[\mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{b}, \mathsf{c}]$ ne possède pas de valeur majoritaire. Ce problème peut être vu comme un scénario où des personnes votent pour une option à une élection, et où nous cherchons à déterminer si une option a remportée la majorité absolue des voix. Un algorithme simple qui détermine si c'est le cas (et qui retourne l'option gagnante le cas échéant) consiste à compter le nombre d'occurrences de chaque valeur $T[i]$ et à vérifier chaque fois si ce décompte excède $n/2$. Cette approche est décrite sous forme de pseudocode à l'algorithme 4.

Analysons le temps de calcul t de l'algorithme 4. Nous considérons toutes les opérations comme élémentaires, c'est-à-dire l'affectation, la comparaison, l'addition, la division et l'accès à un élément de T. Remarquons que les deux

Algorithme 4 : Recherche d'une valeur majoritaire.

Entrées : séquence T de $n \in \mathbb{N}$ éléments comparables
Résultat : une valeur x t.q. T contient plus de $n/2$ occurrences de x s'il en existe une, et « *aucune* » sinon

```
1 pour i ∈ [1..n]
2 |  c ← 0
3 |  pour j ∈ [1..n]                    // Compter # d'occur. de T[i]
4 |  |  si T[j] = T[i] alors c ← c + 1
5 |  si c > n ÷ 2 alors retourner T[i]
6 retourner aucune
```

boucles sont exécutées exactement n fois, et que c est incrémenté entre 0 à n fois à la ligne 4. Ainsi, nous obtenons:

$$t(n) \leq n \cdot (1 + 5n + 3).$$

Nous avons donc $t \in \mathcal{O}(n^2)$ puisque:

$$\begin{aligned} t(n) &\leq 5n^2 + 4n && \text{pour tout } n \geq 0 \\ &\leq 5n^2 + 4n^2 && \text{pour tout } n \geq 0 \\ &= 9n^2. \end{aligned}$$

Cherchons maintenant à borner t inférieurement. Si T ne contient pas de valeur majoritaire, alors la condition « $c > n \div 2$ » n'est jamais satisfaite. Nous avons donc $t(n) \geq n \cdot (1 + 3n + 2) = 3n^2 + 3n \geq 3n^2$ et ainsi $t \in \Omega(n^2)$. Par conséquent, $t \in \Theta(n^2)$ et ainsi l'algorithme 4 fonctionne en temps quadratique. Notons que l'analyse aurait pu être simplifiée en gardant le terme dominant de chaque expression, puisqu'il s'agit de polynômes.

L'algorithme 4 peut être amélioré. En effet, si la séquence contient plusieurs copies d'une même valeur non majoritaire, alors celle-ci risque d'être reconsidérée plusieurs fois, ce qui augmente inutilement le temps d'exécution. L'algorithme 5 décrit une version modifiée où nous considérons chaque valeur au plus une fois.

Analysons le temps d'exécution t de l'algorithme 5. En supposant que les

Algorithme 5 : Recherche d'une valeur majoritaire, sans considérer une valeur plus d'une fois.

Entrées : séquence T de $n \in \mathbb{N}$ éléments comparables
Résultat : une valeur x t.q. T contient plus de $n/2$ occurrences de x s'il en existe une, et « *aucune* » sinon

```
1 pour i ∈ [1..n]
2     si T[i] ≠ ⊥ alors
3         c ← 0; x ← T[i]
4         pour j ∈ [i..n]                    // Compter # d'occur. de T[i]
5             si T[j] = x alors
6                 c ← c + 1
7                 T[j] ← ⊥                   // «Retirer» en remplaçant par ⊥
8         si c > n ÷ 2 alors retourner x
9 retourner aucune
```

conditions des trois **si** sont toujours satisfaites, nous obtenons:

$$\begin{aligned} t(n) &\leq \sum_{i=1}^{n} \left(5 + \sum_{j=i}^{n} 5 + 2\right) \\ &= \sum_{i=1}^{n} (5 + 5(n-i+1) + 2) \\ &= \sum_{i=1}^{n} (5n - 5i + 12) \\ &= 5n^2 + 12n - 5\sum_{i=1}^{n} i \\ &= 5n^2 + 12n - 5n(n+1)/2 \\ &= \frac{5}{2} \cdot n^2 + \frac{19}{2} \cdot n. \end{aligned}$$

Ainsi, t est borné supérieurement par un polyôme de degré 2, ce qui implique que $t \in \mathcal{O}(n^2)$.

Cherchons maintenant à borner t inférieurement. Considérons le cas où T contient des éléments qui sont tous distincts. Une telle séquence ne contient pas d'élément majoritaire, et ainsi la condition « $c > n \div 2$ » n'est jamais satisfaite. De plus, la condition « $T[i] \neq \bot$ » est toujours satisfaite car tous les éléments sont distincts. Les deux boucles sont donc exécutées un nombre maximal de

fois. Ainsi:

$$\begin{aligned} t(n) &\geq \sum_{i=1}^{n} \sum_{j=i}^{n} 1 \\ &= \sum_{i=1}^{n} (n - i + 1) \\ &= n^2 + n - \sum_{i=1}^{n} i \\ &= n^2 + n - n(n+1)/2 \\ &= \frac{n^2}{2} + \frac{n}{2}. \end{aligned}$$

Nous obtenons donc $t \in \Omega(n^2)$ et par conséquent $t \in \Theta(n^2)$. Les deux algorithmes ont donc un temps d'exécution quadratique (dans le pire cas). Ainsi, bien que notre deuxième algorithme puisse être légèrement plus efficace, l'amélioration du temps d'exécution est négligeable.

Nous verrons qu'il existe des algorithmes plus efficaces pour ce problème: $\Theta(n \log n)$ (laissé en exercice) et $\Theta(n)$ (présenté à la section 1.8).

1.4 Simplification du décompte des opérations

Une ligne de code d'un algorithme est dite *élémentaire* si son exécution engendre un nombre d'opérations élémentaires indépendant de la taille de l'entrée. Par exemple, ces lignes de l'algorithme 5 sont toutes élémentaires:

```
1 si T[j] = x alors
2 |   c ← c + 1
3 |   T[j] ← ⊥
```

En général, toutes les lignes d'un algorithme sont élémentaires à l'exception des appels de sous-fonctions complexes.

Le bloc de code ci-dessus exécute 2 ou 5 opérations élémentaires. Lors de l'analyse asymptotique de l'algorithme, cette constante précise n'importe pas. Nous pouvons donc simplement considérer ce bloc comme une seule « grande » opération élémentaire. En général, si le nombre de fois qu'un algorithme atteint certaines lignes élémentaires $j_2, \ldots, j_\ell$ est borné par le nombre de fois qu'il atteint une ligne élémentaire j_1, alors il suffit de compter le nombre d'occurrences de j_1 et d'ignorer les lignes $j_2, \ldots, j_\ell$. Par exemple, dans le code ci-dessus, le nombre d'exécutions de la ligne $j_1 \coloneqq 1$ borne celle des lignes $j_2 \coloneqq 2$ et $j_3 \coloneqq 3$. La preuve formelle de cette observation est donnée en annexe.

Appliquons cette observation à l'algorithme 5. Ses lignes 3 et 8 sont exécutées au plus autant de fois que la ligne 2, et ses lignes 6 et 7 sont exécutées au plus autant de fois que sa ligne 5. Ainsi, nous en dérivons un « squelette » tel

que décrit à l'algorithme 6. Cela facilite l'analyse de l'algorithme puisque nous pouvons en dériver une expression plus simple:

$$t'_{\max}(n) \leq \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} i.$$

Nous appliquerons de plus en plus ce type de simplification afin de faciliter l'analyse d'algorithmes plus complexes.

Algorithme 6 : Squelette de l'algorithme 5.

```
pour i ∈ [1..n]
    si /* élémentaire */ alors
        pour j ∈ [i..n]
            /* élémentaire                    */
retourner
```

1.5 Règle de la limite

Afin d'analyser des fonctions plus complexes, il s'avère parfois pratique d'utiliser la *règle de la limite*:

Proposition 12. *Soient $f, g \in \mathcal{F}$ et $\ell := \lim_{n\to+\infty} f(n)/g(n)$ une limite qui existe.*

Nous avons:

- *si $\ell \in \mathbb{R}_{>0}$, alors $f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(f)$;*
- *si $\ell = 0$, alors $f \in \mathcal{O}(g)$ et $g \notin \mathcal{O}(f)$;*
- *si $\ell = +\infty$, alors $f \notin \mathcal{O}(g)$ et $g \in \mathcal{O}(f)$.*

Exemple.

Afin d'illustrer cette règle, considérons un exemple tiré de [BB96, p. 84].

Posons $f(n) \coloneqq \log n$ et $g(n) \coloneqq \sqrt{n}$. Nous avons:

$$\begin{aligned}
\lim_{n\to+\infty} \frac{f(n)}{g(n)} &= \lim_{n\to+\infty} \frac{f'(n)}{g'(n)} && \text{(par la règle de L'Hôpital)} \\
&= \lim_{n\to+\infty} \frac{1/(\log_e 2 \cdot n)}{1/(2 \cdot \sqrt{n})} \\
&= \lim_{n\to+\infty} \frac{2 \cdot \sqrt{n}}{\log_e 2 \cdot n} \\
&= \frac{2}{\log_e 2} \cdot \lim_{n\to+\infty} \frac{\sqrt{n}}{n} \\
&= \frac{2}{\log_e 2} \cdot \lim_{n\to+\infty} \frac{1}{\sqrt{n}} \\
&= 0.
\end{aligned}$$

Par la proposition 12, nous obtenons donc $f \in \mathcal{O}(g)$ et $g \notin \mathcal{O}(f)$, ou autrement dit $\log n \in \mathcal{O}(\sqrt{n})$ et $\sqrt{n} \notin \mathcal{O}(\log n)$.

1.6 Notation asymptotique à plusieurs paramètres

Dans le cas d'un algorithme dont la taille des entrées dépend de plusieurs paramètres, nous pouvons utiliser les notations asymptotiques étendues naturellement à des fonctions multivariées. Nous présentons brièvement le cas à deux paramètres qui sera le plus fréquent. Soit $\mathcal{F}_2$ l'ensemble des fonctions éventuellement non négatives à deux entrées:

$$\mathcal{F}_2 \coloneqq \left\{ f \colon \mathbb{N}^2 \to \mathbb{R} : \exists k, k' \in \mathbb{N}\ \forall m \geq k\ \forall n \geq k'\ f(m,n) > 0 \right\}.$$

Définition 3. Soit $g \in \mathcal{F}_2$. L'ensemble $\mathcal{O}(g)$ est défini par:

$$\mathcal{O}(g) \coloneqq \{ f \in \mathcal{F}_2 : \exists c \in \mathbb{R}_{>0}\ \exists m_0, n_0 \in \mathbb{N}\ \forall m \geq m_0\ \forall n \geq n_0 \\ f(m,n) \leq c \cdot g(m,n) \}.$$

Les ensembles $\Omega(g)$ et $\Theta(g)$ sont définis par:

$$\begin{aligned}
\Omega(g) &\coloneqq \{ f \in \mathcal{F}_2 : g \in \mathcal{O}(f) \}, \\
\Theta(g) &\coloneqq \mathcal{O}(f) \cap \Omega(g).
\end{aligned}$$

Reconsidérons l'algorithme 3 qui calcule la valeur maximale apparaissant dans une matrice de taille $m \times n$. Nous avons déjà montré que son temps d'exécution t satisfait:

$$5mn + 4m - 1 \leq t(m,n) \leq 7mn + 4m - 3 \qquad \text{pour tous } m, n \geq 1.$$

Montrons que $t \in \Theta(mn)$. Nous avons:

$$\begin{aligned} t(m,n) &\leq 7mn + 4m - 3 && \text{pour tous } m, n \geq 1 \\ &\leq 7mn + 4m \\ &\leq 7mn + 4mn && \text{pour tout } n \geq 1 \\ &= 11mn. \end{aligned}$$

Ainsi en prenant $m_0 = n_0 \coloneqq 1$ comme seuils, et $c \coloneqq 11$ comme constante multiplicative, nous obtenons $t \in \mathcal{O}(m \cdot n)$. Nous pouvons dériver $t \in \Omega(m \cdot n)$ similairement et en conclure que $t \in \Theta(m \cdot n)$.

1.7 Correction et terminaison

1.7.1 Correction

Nous nous sommes intéressé·e·s au temps d'exécution d'algorithmes, en prenant pour acquis que ceux-ci fonctionnent. Cependant, il n'est pas toujours simple de se convaincre qu'un algorithme fonctionne correctement. Formalisons le concept de *correction*.

Un algorithme possède un domaine d'entrée $\mathbb{D}$, par ex. l'ensemble des entiers, des séquences, des matrices, des arbres binaires, des paires d'entiers, etc. Sur toute entrée $x \in \mathbb{D}$, on s'attend à ce que l'algorithme retourne une sortie y telle qu'une propriété $\varphi(x, y)$ soit satisfaite. Par exemple, reconsidérons l'algorithme 2 qui cherche à calculer la valeur maximale apparaissant dans une séquence non vide. Dans le cas de cet algorithme, nous avons:

$$\begin{aligned} \mathbb{D} &\coloneqq \{x : x \text{ est une séquence de } n \in \mathbb{N}_{>0} \text{ éléments}\}, \\ \varphi(x, y) &\coloneqq (y = \max\{x[i] : 1 \leq i \leq n\}). \end{aligned}$$

Définition 4. Nous disons qu'un algorithme est *correct* si pour toute entrée $x \in \mathbb{D}$, la sortie y de l'algorithme sur entrée x est telle que $\varphi(x, y)$ soit vraie.

L'appartenance d'une entrée à $\mathbb{D}$ s'appelle la *pré-condition*, et la propriété φ s'appelle la *post-condition*. En mots, un algorithme est donc correct si sur chaque entrée qui satisfait la pré-condition, la sortie satisfait la post-condition.

Afin de démontrer qu'un algorithme est correct, nous avons souvent recours à un *invariant*, c'est-à-dire une propriété qui demeure vraie à chaque fois qu'une ou certaines lignes de code sont atteintes. Par exemple, dans le cas de l'algorithme 2, il est possible d'établir l'invariant suivant par induction sur i:

Proposition 13. *À chaque fois que l'algorithme 2 atteint le* ***tant que****, l'égalité suivante est satisfaite:* $max = \max\{s[j] : 1 \leq j \leq i - 1\}$.

Puisque l'algorithme termine avec $i = n + 1$, l'invariant nous indique que l'algorithme retourne $max = \max\{s[j] : 1 \leq j \leq n\}$, ce qui satisfait la post-condition. Ainsi, l'algorithme 2 est correct.

1.7.2 Bon fonctionnement

Pour la plupart des algorithmes que nous considérerons, il sera évident que ceux-ci terminent, c'est-à-dire qu'une instruction **retourner** est exécutée sur toute entrée. Toutefois, cela n'est pas toujours aussi simple. Par exemple, à ce jour personne ne sait si l'algorithme 7 termine sur toute entrée.

Algorithme 7 : Calcul de la séquence de Collatz.

Entrées : $n \in \mathbb{N}$
Sorties : 1

tant que $n \neq 1$
  si n *est pair* **alors**
    $n \leftarrow n \div 2$
  sinon
    $n \leftarrow 3n + 1$
retourner n

Remarques.

— Techniquement, si un algorithme n'est pas correct ou ne termine pas, nous devrions plutôt parler de « procédure ». Ainsi, un algorithme est une procédure qui est correcte et qui termine sur toute entrée. Pour être précis, il faudrait donc dire « la procédure est correcte » et « la procédure termine » plutôt que « l'algorithme est correct » et « l'algorithme termine ». Nous nous permettons toutefois cet abus de langage.

— Certains ouvrages nomment *correction partielle* ce que nous appelons ici la correction. Dans ces ouvrages, « correction » réfère à « correction partielle + terminaison ».

1.8 Étude de cas: vote à majorité absolue (suite)

Revisitons le problème de la section 1.3, c'est-à-dire la recherche d'une valeur majoritaire dans une séquence de n éléments. Les deux algorithmes que nous avons présentés fonctionnaient en temps $\Theta(n^2)$. L'algorithme 8 décrit une procédure élégante, conçue par Robert S. Boyer et J. Strother Moore [BM91], qui recherche une valeur majoritaire en temps $\Theta(n)$. La correction de cet algorithme n'est pas évidente. Ainsi, nous expliquons le fonctionnement de l'algorithme et démontrons qu'il est correct.

L'algorithme 8 fonctionne en deux phases:

(a) on trouve une valeur x qui est majoritaire si T en contient une; et

(b) on vérifie que x est bien majoritaire.

Algorithme 8 : Recherche de Boyer–Moore d'une valeur majoritaire.

Entrées : séquence T de $n \in \mathbb{N}$ éléments comparables
Résultat : une valeur x t.q. T contient plus de $n/2$ occurrences de x s'il en existe une, et « *aucune* » sinon

```
1  x ← aucune; c ← 0          // Chercher une valeur majoritaire x
2  pour i ∈ [1..n]
3  |  si c = 0 alors x ← T[i]; c ← 1
4  |  sinon si T[i] = x alors c ← c + 1
5  |  sinon c ← c − 1
6  c ← 0                      // Vérifier que x est bien majoritaire
7  pour i ∈ [1..n]
8  |  si T[i] = x alors c ← c + 1
9  si c > n ÷ 2 alors
10 |  retourner x
11 sinon
12 |  retourner aucune
```

La première phase est plutôt subtile. Celle-ci garde un compteur c à jour et parcourt tous les éléments de T. Initialement, on suppose que $x = T[1]$ est majoritaire et on pose $c = 1$. Par la suite, on incrémente c pour chaque occurrence de x, et on décrémente c pour chaque non occurrence de x. Si c atteint 0 à l'élément $T[i]$, alors x n'était pas majoritaire puisqu'il y avait « plus de votes en défaveur de x qu'en faveur de x ». On continue donc le processus, mais cette fois en prenant $x = T[i]$ comme nouvelle candidate. Intuitivement, si T possède une valeur majoritaire, celle-ci « survivra aux incrémentations et décrémentations ». La seconde phase confirme simplement que x est bien majoritaire.

Sur la séquence $T = [\mathtt{a}, \mathtt{b}, \mathtt{b}, \mathtt{a}, \mathtt{a}, \mathtt{a}, \mathtt{c}, \mathtt{a}]$, la première phase de l'algorithme termine avec $x = \mathtt{a}$ et $c = 2$ comme en témoigne sa trace:

i		1	2	3	4	5	6	7	8
$T[i]$		a	b	b	a	a	a	c	a
x	–	a	a	b	b	a	a	a	a
c	0	1	0	1	0	1	2	1	2

La deuxième phase confirme que a est bel et bien majoritaire.

Comme second exemple, considérons la séquence $T' = [\mathtt{a}, \mathtt{b}, \mathtt{a}, \mathtt{b}, \mathtt{c}]$. La première phase termine avec $x = \mathtt{c}$ et $c = 1$:

i		1	2	3	4	5
$T'[i]$		a	b	a	b	c
x	–	a	a	a	a	c
c	0	1	0	1	0	1

La deuxième phase vérifie si x apparaît au moins 3 fois, ce qui n'est pas le cas, et conclut donc que T' ne possède pas de valeur majoritaire.

Afin de démontrer que l'algorithme 8 est correct, nous établissons d'abord un invariant satisfait par sa première boucle. Intuitivement, celui-ci affirme qu'après la $i^{\text{ème}}$ itération de la première boucle, x est l'unique valeur *possiblement* majoritaire dans la sous-séquence $T[1:i] = [T[1], T[2], \ldots, T[i]]$.

Proposition 14. *Après l'exécution du corps de la première boucle de l'algorithme 8, l'invariant suivant est satisfait:*

(a) $c \geq 0$*;*

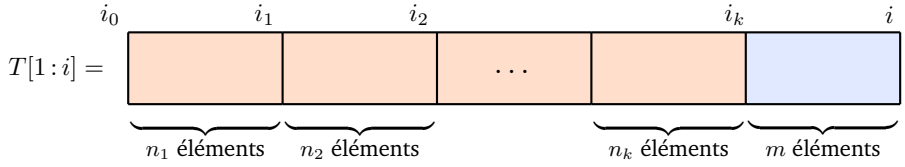
(b) x *apparaît au plus* $\frac{i+c}{2}$ *fois dans* $T[1:i]$*;*

(c) y *apparaît au plus* $\frac{i-c}{2}$ *fois dans* $T[1:i]$*, pour tout* $y \neq x$*.*

Démonstration semi-formelle. Le compteur c ne devient jamais négatif. En effet, il est ou bien nul, auquel cas il est remis à 1; ou bien positif auquel cas il est incrémenté ou décrémenté. Concentrons-nous donc sur les propriétés (b) et (c).

Supposons que la boucle de la première phase de l'algorithme ait été exécutée i fois. Le compteur c a pris la valeur zéro à k reprises durant ces itérations, pour un certain $k \in \mathbb{N}$. Soient $i_0 < i_1 < i_2 < \cdots < i_k$ les itérations auxquelles cela s'est produit, où $i_0 \coloneqq 0$ (le moment avant d'entrer dans la boucle).

Posons $n_j \coloneqq i_j - i_{j-1}$ pour tout $j \in [1..k]$, et $m \coloneqq i - i_k$. Soit $y \in T$ une valeur différente de x. En mots, x est la valeur que nous considérons actuellement majoritaire, et y est une autre valeur qui n'est *pas* actuellement considérée majoritaire. Schématiquement, nous avons:



Remarquons que le $j^{\text{ème}}$ bloc contient exactement $n_j/2$ occurrences d'une certaine valeur et exactement $n_j/2$ occurrences d'autres valeurs. Intuitivement, il s'agit d'un « match nul » entre une certaine valeur et les autres. Ainsi, le $j^{\text{ème}}$ bloc contient au plus $n_j/2$ ocurrences de x, et au plus $n_j/2$ occurrences de y.

Examinons le dernier bloc de m éléments qui correspond au « duel en cours » entre x et les autres. Ce « duel » peut être vu comme un « match nul », plus c voix supplémentaires pour x. Ce bloc contient donc c occurrences de x, $(m-c)/2$ *autres* occurrences de x, et $(m-c)/2$ occurrences d'autres valeurs. Ainsi, x apparaît exactement $c + (m-c)/2 = 2c/2 + (m-c)/2 = (m+c)/2$ fois dans le dernier bloc, et y apparaît au plus $(m-c)/2$ fois dans ce même bloc.

Globalement, en cumulant ces décomptes, nous obtenons:

$$\begin{aligned}
\#\text{ d'occurrences de } x \text{ dans } T[1:i] &\leq \sum_{j=1}^{k}(n_j/2) + (m+c)/2 \\
&= (1/2)\cdot\left(\sum_{j=1}^{k} n_j + m + c\right) \\
&= (1/2)\cdot(i+c) \\
&= \frac{i+c}{2} \\
\#\text{ d'occurrences de } y \text{ dans } T[1:i] &\leq \sum_{j=1}^{k}(n_j/2) + (m-c)/2 \\
&= (1/2)\cdot\left(\sum_{j=1}^{k} n_j + m - c\right) \\
&= (1/2)\cdot(i-c) \\
&= \frac{i-c}{2}.
\end{aligned}$$

□

Théorème 1. *L'algorithme 8 est correct.*

Démonstration. Si l'algorithme retourne une valeur différente de « *aucune* », alors clairement celle-ci est majoritaire, car la deuxième phase s'assure qu'on ne peut retourner qu'une valeur majoritaire. Il suffit donc de prouver que si T possède une valeur majoritaire, alors la première phase se termine avec celle-ci.

Nous prouvons cette affirmation par contradiction. Supposons que T possède une valeur majoritaire y et que la première phase se termine avec $x \neq y$. Par la proposition 14 (c), y apparaît au plus $\frac{n-c}{2}$ fois dans $T[1:n] = T$, où $c \geq 0$ par la proposition 14 (a). La valeur y apparaît donc au plus $\frac{n-c}{2} \leq \frac{n}{2}$ fois dans T, ce qui contredit le fait qu'elle soit majoritaire. □

Observation.

La première phase de l'algorithme de Boyer et Moore considère chaque élément de T une et une seule fois, et les consomme du début vers la fin de la séquence. De plus, la seconde phase n'est pas nécessaire si nous savons à coup sûr que l'entrée possède une valeur majoritaire. L'algorithme peut donc s'avérer particulièrement efficace avec les entrées sous forme de *flux*, c.-à-d. où l'algorithme reçoit son entrée progressivement sans que sa taille ne soit connue à priori.

1.9 Exercices

1.1) Montrez que si $f \in \mathcal{O}(g)$, alors $\mathcal{O}(f) \subseteq \mathcal{O}(g)$. Déduisez-en l'équivalence suivante: $\mathcal{O}(f) = \mathcal{O}(g) \iff f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(f)$. Est-ce aussi le cas si l'on remplace $\mathcal{O}$ par Ω? Et par Θ?

1.2) Montrez que pour toutes fonctions $f, g, h \in \mathcal{F}$, si $f \in \Omega(g)$ et $g \in \Omega(h)$, alors $f \in \Omega(h)$. Est-ce aussi le cas si l'on remplace Ω par Θ?

1.3) Montrez que $f \in \Omega(g) \iff g \in \mathcal{O}(f)$.

1.4) Ordonnez les fonctions suivantes selon la notation $\mathcal{O}$:

$$5n^2 - n,\ 3n^2,\ 4^n,\ 8(n+2) - 1 + 9n,\ n!,\ n^3 - n^2 + 7,\ n \log n,\ 1000000,\ 2^n.$$

1.5) Dites si $3^n \in \Theta(2^n)$.

1.6) Montrez que la relation $R \subseteq \mathcal{F} \times \mathcal{F}$ définie par $R(f, g) \iff f \in \Theta(g)$ est une relation d'équivalence. Est-ce aussi le cas si l'on remplace Θ par $\mathcal{O}$ ou Ω?

1.7) Comparez $f(n) \coloneqq n^2$ et $g(n) \coloneqq 2^n$ à l'aide de la règle de la limite.

1.8) Montrez que $\log_a n \in \Theta(\log_b n)$ pour tous $a, b \in \mathbb{N}_{\geq 2}$.

1.9) Montrez que $\binom{n}{d} \in \Theta(n^d)$ pour tout $d \in \mathbb{N}$.

1.10) Montrez que $\sum_{i=1}^{n} i^d \in \Theta(n^{d+1})$ pour tout $d \in \mathbb{N}$.

1.11) Montrez que $n^d \in \mathcal{O}(n!)$ pour tout $d \in \mathbb{N}_{\geq 2}$.

1.12) Montrez que $\log n \in \mathcal{O}(\sqrt[d]{n})$ et $(\log n)^d \in \mathcal{O}(n)$ pour tout $d \in \mathbb{N}_{>0}$.

1.13) ★ Montrez que $(\log n)^c \in \mathcal{O}(\sqrt[d]{n})$ et $\sqrt[d]{n} \notin \mathcal{O}((\log n)^c)$ pour $c, d \in \mathbb{N}_{>0}$.

1.14) ★ Soient $f(n) \coloneqq n$ et $g(n) \coloneqq 2^{\lfloor \log n \rfloor}$. Montrez que $f \in \Theta(g)$, mais que la limite $\lim_{n \to \infty} f(n)/g(n)$ n'existe pas. Autrement dit, la règle de la limite ne peut pas toujours être appliquée. (*tiré de [BB96, p. 85]*)

1.15) Soit $f(m, n) = \frac{mn}{2} + 3m \log(n \cdot 2^n) + 7n$. Montrez que $f \in \mathcal{O}(mn)$.

1.16) Quel est le temps d'exécution de l'algorithme 4 dans le *meilleur cas*?

1.17) Nous avons vu qu'il est possible de déterminer si une séquence possède une valeur majoritaire en temps $\Theta(n^2)$ et $\Theta(n)$. Donnez un algorithme intermédiaire qui résout ce problème en temps $\mathcal{O}(n \log n)$, en supposant que vous ayez accès à une primitive qui permet de trier une séquence en temps $\mathcal{O}(n \log n)$.

1.18) Démontrez la proposition 13 par induction sur i. Comme cas de base, considérez la première fois que le **tant que** est atteint. Pour l'étape d'induction, supposez que l'invariant soit vrai et montrez qu'il est préservé après l'exécution du corps de la boucle.

1.19) Argumentez que l'algorithme 8 fonctionne en temps $\Theta(n)$.

1.20) ★ Montrez que l'algorithme ci-dessous termine (sur toute entrée). 
(*basé sur une question d'Etienne D. Massé, A19*)

Entrées : $n \in \mathbb{N}_{>0}$
Sorties : `vrai`

tant que n *est pair*
  $n \leftarrow n + (n \div 2)$
retourner `vrai`

1.21) Considérons le problème 3SUM défini ainsi: 

ENTRÉE:	une séquence s de n nombres entiers
QUESTION:	est-ce qu'il existe trois indices distincts $i, j, k \in [1..n]$ tels que $s[i] + s[j] + s[k] = 0$?

(a) Ce problème peut être résolu en considérant tous les sous-ensembles de trois indices distincts. Combien y a-t-il de tels sous-ensembles?

(b) Donnez et analysez un algorithme qui résout 3SUM en temps $\mathcal{O}(n^2)$.

Remarque.

Le problème 3SUM est encore activement étudié en algorithmique, notamment dû à sa relation avec différents problèmes de géométrie computationnelle. Le problème peut être résolu en temps

$$\mathcal{O}\left((n^2/\log(n)^2) \cdot (\log\log n)^c\right)$$

pour une certaine constante $c > 0$ [Cha20]. Il semble impossible de résoudre 3SUM en temps $\mathcal{O}(n^{2-\varepsilon})$, et ce peu importe la constante $\varepsilon > 0$, mais cela demeure une conjecture.

1.22) Quel est le temps d'exécution dans le pire cas de l'algorithme 5 lorsque le nombre de valeurs distinctes en entrée est *constant*? Obtenez-vous une meilleure complexité que pour l'algorithme 4?

1.23) L'indice de Jaccard, défini par $J(X, Y) := |X \cap Y|/|X \cup Y|$ est une mesure de similitude entre deux ensembles non vides X et Y. Elle peut être utile, 

par exemple, pous identifier des similitudes dans une base de données. Considérons une implémentation d'ensembles qui offre ces opérations:

Opération	**Temps d'exécution**
Union $X \cup Y$	$\mathcal{O}(\|X\| + \|Y\|)$
Intersection $X \cap Y$	$\mathcal{O}(\|X\| \cdot \|Y\|)$
Taille $\|X\|$	$\mathcal{O}(1)$

Avec une telle implémentation, le calcul naïf de $J(X, Y)$ prend un temps de $\mathcal{O}(|X|\cdot|Y|+|X|+|Y|) = \mathcal{O}(|X|\cdot|Y|)$. Expliquez comment faire mieux.

(*inspiré d'un projet encadré à la session H21*)

2

Tri

Nous traitons maintenant d'un problème fondamental en algorithmique: le tri. Celui-ci consiste à ordonner les éléments d'une séquence donnée selon un certain ordre. Plus formellement, nous dirons que des éléments sont *comparables* s'ils proviennent d'un ensemble muni d'un ordre total, par ex. $\mathbb{N}$ sous l'ordre usuel $\leq$, ou l'ensemble des chaînes de caractères ordonnées sous l'ordre lexicographique. Nous disons qu'une séquence s de n éléments comparables est *triée* si $s[1] \leq s[2] \leq \cdots \leq s[n]$. De façon générale, un *algorithme de tri* est un algorithme dont la pré-condition et la post-condition correspondent à:

$$\begin{aligned} \mathbb{D} &= \{s : s \text{ est une séquence d'éléments comparables}\}, \\ \varphi(s,t) &= t \text{ est triée et } t \text{ est une permutation de } s. \end{aligned}$$

Autrement dit, un algorithme de tri prend une séquence s d'éléments comparables en entrée, et produit une séquence triée t dont les éléments sont ceux de s, mais dans un ordre (possiblement) différent. Dans le reste du chapitre, nous présentons quelques algorithmes de tri répandus.

2.1 Approche générique

Nous débutons par une approche générique qui permet de trier une séquence à l'aide d'une seule opération. Celle-ci est utilisée implicitement ou explicitement par essentiellement *tous* les algorithmes de tri[1]. Pour toute séquence s de n éléments comparables, posons:

$$\text{inv}(s) \coloneqq \{(i,j) \in [1..n]^2 : i < j \text{ et } s[i] > s[j]\}.$$

Autrement dit, $\text{inv}(s)$ est l'ensemble des *inversions* de s, c'est-à-dire les paires d'indices dont le contenu est mal ordonné. Nous présentons un algorithme de tri simple: tant que s possède une inversion (i,j), les éléments $s[i]$ et $s[j]$ sont intervertis. Cette procédure est décrite à l'algorithme 9.

1. Sauf pour quelques exceptions, comme l'algorithme de l'exercice 2.14) et « *bogosort* ».

Algorithme 9 : Algorithme générique de tri.

Entrées : séquence s d'éléments comparables
Résultat : s est triée

```
1 tant que ∃(i, j) ∈ inv(s)
2 |  s[i] ↔ s[j]                    // inverser s[i] et s[j]
```

Exemple.

Considérons la séquence $s = [30, 50, 10, 40, 20]$. Ses inversions sont

$$\text{inv}(s) = \{(1,3), (1,5), (2,3), (2,4), (2,5), (4,5)\}.$$

En corrigeant l'inversion $(1,3)$, nous obtenons la séquence $s' = [10, 50, 30, 40, 20]$ dont les inversions sont:

$$\text{inv}(s') = \{(2,3), (2,4), (2,5), (3,5), (4,5)\}.$$

Notons que bien que des inversions aient disparu, une nouvelle inversion est aussi apparue. Toutefois, le *nombre* d'inversions a diminué.

En corrigeant l'inversion $(2,5)$ de s', nous obtenons la séquence $s'' = [10, 20, 30, 40, 50]$ qui ne possède aucune inversion, c.-à-d. $\text{inv}(s'') = \emptyset$. Observons que s'' est à la fois triée et une permutation de s. Nous avons donc bel et bien trié s.

Intuitivement, l'algorithme 9 fonctionne car il y a une forme de progrès à chaque itération: la séquence est « de plus en plus triée ». En effet, bien que des inversions puissent être créées, le nombre d'inversions diminue à chaque itération, et ce *peu importe l'ordre* dans lequel les inversions sont corrigées. Plus formellement:

Proposition 15. *Soit s' la séquence obtenue à partir d'une séquence s après l'exécution de la ligne 2 de l'algorithme 9. Nous avons* $|\text{inv}(s)| > |\text{inv}(s')|$.

Démonstration. Soit n la taille de s et soit f la fonction telle que:

$$f(x,y) \coloneqq \begin{cases} 1 & \text{si } (x,y) \in \text{inv}(s), \\ 0 & \text{sinon.} \end{cases}$$

Nous définissons f' de la même façon pour s'. Autrement dit, $f(x,y)$ indique si (x,y) est une inversion dans s, et $f'(x,y)$ indique la même chose pour s'.

Nous devons vérifier que le nombre d'inversions a diminué en passant de s vers s'. Puisque l'inversion (i,j) a été corrigée, nous avons:

$$f(i,j) = 1 > 0 = f'(i,j). \tag{2.1}$$

Posons $X \coloneqq [1..n] \setminus \{i, j\}$. Puisque le contenu de s aux positions X n'a pas changé, nous avons:

$$f(x, y) = f'(x, y) \qquad \text{pour tous } x, y \in X. \tag{2.2}$$

Il suffit donc de vérifier que le nombre d'inversions entre X et $\{i, j\}$ n'a pas augmenté. Pour chaque position de X, il y a trois cas possibles: elle est *plus petite* que i et j, *comprise entre* i et j, ou *plus grande* que i et j. Plus informellement: elle se situe à « gauche », au « centre » ou à « droite », que nous dénotons:

$$\begin{aligned} G &\coloneqq \{x \in [1..n] : x < i\}, \\ C &\coloneqq \{x \in [1..n] : i < x < j\}, \\ D &\coloneqq \{x \in [1..n] : j < x\}. \end{aligned}$$

Gauche. Soit $x \in G$. Informellement, puisque les éléments à droite de la position x sont demeurés les mêmes (dans un ordre différent), le nombre d'inversions avec i et j n'a pas changé. Autrement dit, nous avons $(x, i) \in \text{inv}(s) \iff (x, j) \in \text{inv}(s')$ et $(x, j) \in \text{inv}(s) \iff (x, i) \in \text{inv}(s')$. Ainsi:

$$f(x, i) + f(x, j) = f'(x, i) + f'(x, j) \qquad \text{pour tout } x \in G. \tag{2.3}$$

Droite. Par un raisonnement symétrique au cas précédent, nous obtenons:

$$f(i, x) + f(j, x) = f'(i, x) + f'(j, x) \qquad \text{pour tout } x \in D. \tag{2.4}$$

Centre. Soit $x \in C$. Observons d'abord qu'il est impossible que $s[i] \leq s[x] \leq s[j]$ puisque $s[i] > s[j]$. Il y a donc trois ordonnancements possibles:

Avant		**Après**	
Ord. dans s	$f(i, x) + f(x, j)$	Ord. dans s'	$f'(i, x) + f'(x, j)$
$s[i] > s[x] \leq s[j]$	1	$s'[i] \geq s[x] < s[j]$	1 ou 0
$s[i] \leq s[x] > s[j]$	1	$s[i] < s[x] \geq s[j]$	1 ou 0
$s[i] > s[x] > s[j]$	2	$s[i] < s[x] < s[j]$	0

Cela démontre que:

$$f(i, x) + f(x, j) \geq f'(i, x) + f'(x, j) \qquad \text{pour tout } x \in C. \tag{2.5}$$

Fin de la preuve. Nous avons montré que le nombre d'inversions a strictement diminué. Les détails formels se trouvent en annexe. □

Théorème 2. *L'algorithme 9 est correct et termine sur toute entrée. De plus, le nombre d'itérations effectuées par l'algorithme appartient à* $\mathcal{O}(n^2)$.

Démonstration. Soit s une séquence de n éléments comparables. Démontrons d'abord que l'algorithme termine sur entrée s. Soit t_i la séquence obtenue après i exécutions de la boucle en débutant à partir de s. Supposons que l'algorithme ne termine pas. Par la proposition 15, nous avons $|\text{inv}(t_0)| > |\text{inv}(t_1)| > \cdots$ ce qui est impossible puisque $|\text{inv}(t_i)|$ ne peut pas décroître sous 0. Ainsi, il y a contradiction et l'algorithme termine après k itérations pour un certain $k \in \mathbb{N}$.

Remarquons que les éléments de t_k et s sont les mêmes (dans un ordre possiblement différent) puisque t_k a été obtenue en permutant les éléments de s. Ainsi, afin de montrer que l'algorithme est correct, il suffit de montrer que t_k est ordonnée. Supposons que ce ne soit pas le cas. Il existe donc des indices $i, j \in [1..n]$ tels que $i < j$ et $t_k[i] > t_k[j]$. La condition de la boucle est donc satisfaite et ainsi l'algorithme ne termine pas à l'itération k. Nous obtenons donc une contradiction, ce qui implique que t_k est bien triée.

Puisque le nombre d'inversions diminue à chaque itération et que s possède au plus $(n-1)+\ldots+1+0 = n(n-1)/2$ inversions, l'algorithme effectue $\mathcal{O}(n^2)$ itérations. □

2.2 Tri par insertion

Nous étudions maintenant plusieurs algorithmes de tri concrets, en débutant par le *tri par insertion*. Celui-ci trie une séquence s en construisant un préfixe trié de s de plus en plus grand:

— on débute avec $i = 1$;

— on considère $s[1:i-1]$ comme étant triée (trivialement vrai au départ);

— on insère $s[i]$ à l'endroit qui rend $s[1:i]$ triée;

— on répète le processus en incrémentant i jusqu'à n.

Cette procédure est décrite sous forme de pseudocode à l'algorithme 10

Algorithme 10 : Algorithme de tri par insertion.

Entrées : séquence s de n éléments comparables
Sorties : séquence s triée

$i \leftarrow 1$
tant que $i \leq n$
  $j \leftarrow i$ `// insérer` $s[i]$ `dans` $s[1:i]$ `en corrigeant`
  tant que $j > 1 \wedge s[j-1] > s[j]$ `// les inversions`
    $s[j-1] \leftrightarrow s[j]$
    $j \leftarrow j - 1$
  $i \leftarrow i + 1$
retourner s

On peut démontrer que le tri par insertion est correct en observant que l'invariant « $s[1:i-1]$ est triée » est satisfait chaque fois que l'on atteint la boucle

principale, ce qui résulte en « $s[1:(n+1)-1] = s[1:n] = s$ est triée » à la sortie de la boucle principale.

Analysons le temps de calcul de l'algorithme. Le temps d'exécution dans le meilleur cas appartient à $\Omega(n)$ car il faut exécuter la boucle principale n fois. Lorsque s est déjà triée, la boucle interne n'est jamais exécutée. Ainsi, le temps d'exécution dans le meilleur cas appartient à $\mathcal{O}(n)$, et par conséquent à $\Theta(n)$. Si la condition de la boucle interne est satisfaite un nombre maximal de fois, c.-à-d. $i-1$ fois, alors l'algorithme fonctionne en temps:

$$\mathcal{O}\left(\sum_{i=1}^{n}(i-1)\right) = \mathcal{O}\left(\sum_{i=1}^{n-1} i\right) = \mathcal{O}(n(n-1)/2) = \mathcal{O}(n^2/2 - n/2) = \mathcal{O}(n^2).$$

Cela se produit lorsque les éléments de s sont en ordre décroissant. En effet, l'insertion de $s[i]$ dans $s[1:i]$ requiert $i-1$ tours de la boucle interne, puisque tous les éléments à sa gauche lui sont supérieurs. Ainsi, le temps d'exécution dans le pire cas appartient à $\Theta(n^2)$. Cette analyse se raffine de la façon suivante:

Proposition 16. *Le tri par insertion fonctionne en temps $\Theta(n+k)$ où k est le nombre d'inversions de la séquence en entrée.*

Puisqu'il n'y a aucune inversion dans une séquence triée, et $n(n-1)/2$ inversions dans une séquence dont les éléments apparaissent en ordre décroissant, la proposition 16 donne bien $\Theta(n)$ et $\Theta(n^2)$ pour ces deux cas.

La proposition 16 montre notamment que le tri par insertion est un bon choix pour les séquences quasi-triées. De plus, le tri par insertion performe généralement bien en pratique sur les séquences de petite taille.

2.3 Tri par monceau

Le *tri par monceau* s'appuie sur la structure de données connue sous le nom de monceau (ou de tas), qui offre les opérations suivantes:

— transformer une séquence en un monceau;

— retirer un élément de valeur minimale;

— insérer un élément.

Une implémentation décente de ces opérations offre, respectivement, une complexité de $\Theta(n)$, $\Theta(\log n)$ et $\Theta(\log n)$ dans le pire cas, où n dénote le nombre d'éléments.

Grâce à ces opérations, il est possible de trier une séquence s en la transformant en monceau, puis en retirant ses éléments itérativement. Puisque le retrait donne toujours un élément minimal, les éléments de s sont retirés en ordre croissant, ce qui permet de construire une séquence triée t. Cette procédure est décrite sous forme de pseudocode à l'algorithme 11.

La correction du tri par monceau est immédiate (en supposant que le monceau est lui-même bien implémenté). Observons que la boucle est itérée précisément n fois. Ainsi, le temps d'exécution dans le pire cas appartient à

$$\Theta(1+n+n\cdot(\log n+1)) = \Theta(1+n+n\log n+n) = \Theta(n\log n).$$

Algorithme 11 : Algorithme de tri par monceau.

Entrées : séquence s d'éléments comparables
Sorties : séquence s triée

$t \leftarrow [\,]$
transformer s **en monceau**
tant que $|s| > 0$
  retirer $x \in s$
  ajouter x **à** t
retourner t

Cet algorithme s'avère donc plus rapide que le tri par insertion sur les séquences de grande taille.

Remarque.

Selon l'implémentation du monceau, le temps dans le meilleur cas peut appartenir à $\Theta(n)$. En effet, si tous les éléments de s sont égaux, alors le retrait d'un élément se fait en temps $\Theta(1)$, ce qui donne $\Theta(n)$ au total.

2.4 Tri par fusion

Le *tri par fusion* (ou *tri fusion*) trie une séquence s grâce aux règles suivantes:

— on découpe s en sous-séquences x et y, c.-à-d. $s = x + y$;

— on trie x et y séparément;

— on fusionne les deux sous-séquences triées afin qu'elles soient collectivement triées.

On découpe généralement s en deux: $x \coloneqq s[1\,{:}\,m]$ et $y \coloneqq s[m+1\,{:}\,n]$, où n est la taille de s et $m = n \div 2$. Cette approche peut sembler circulaire puisqu'il faut savoir trier afin de trier. Cependant, puisque x et y sont de *taille inférieure* à n, il suffit de les trier récursivement jusqu'à l'obtention de séquences de taille ≤ 1, qui sont trivialement triées. La fusion s'effectue en sélectionnant itérativement le plus petit élément de x et y, en gardant des pointeurs i et j vers leur plus petit élément respectif. Cette procédure est décrite sous forme de pseudocode à l'algorithme 12.

La sous-routine `fusion` fonctionne en temps $\Theta(|x| + |y|)$ puisqu'elle itère exactement une fois sur chacun des éléments de x et y. Le temps d'exécution de `trier` s'avère difficile à analyser puisque s n'est généralement pas découpée parfaitement en deux. Afin de simplifier l'analyse, supposons que `fusion` effectue au plus $c \cdot (|x| + |y|)$ opérations pour une certaine constante $c \in \mathbb{R}_{>0}$, et que

Algorithme 12 : Algorithme de tri par fusion.

Entrées : séquence s d'éléments comparables
Sorties : séquence s triée

```
trier(s):
    fusion(x, y):                    // fusionne deux séq. triées
        i ← 1; j ← 1; z ← []
        tant que i ≤ |x| ∧ j ≤ |y|
            si x[i] ≤ y[j] alors
                ajouter x[i] à z
                i ← i + 1
            sinon
                ajouter y[j] à z
                j ← j + 1
        retourner z + x[i : |x|] + y[j : |y|]
    si |s| ≤ 1 alors retourner s
    sinon
        m ← |s| ÷ 2
        retourner fusion(trier(s[1 : m]), trier(s[m + 1 : |s|]))
```

$|s|$ soit une puissance de 2. Nous avons:

$$t(2^k) \leq \begin{cases} 1 & \text{si } k = 0, \\ 2 \cdot t(2^{k-1}) + c \cdot 2^k & \text{si } k > 0. \end{cases}$$

Proposition 17. *$t(2^k) \leq 2^k(ck+1)$ pour tout $k \in \mathbb{N}_{>0}$.*

Par conséquent, lorsque n est de la forme $n = 2^k$, nous avons $t(n) \leq 2^k(ck+1) = n(c \log n + 1) = c \cdot n \log n + n$. Informellement, nous concluons donc que:

$$t \in \mathcal{O}(n \log n : n \text{ est une puissance de } 2).$$

Nous verrons plus tard au chapitre 5 que $t \in \mathcal{O}(n \log n)$, sans se restreindre aux puissances de 2.

2.5 Tri rapide

Le *tri rapide* (ou « *quicksort* » en anglais) trie une séquence s à l'aide des règles suivantes:

— on choisit un élément pivot $s[i]$;

— on divise s en une séquence t qui contient tous les éléments inférieurs au pivot, et une séquence u qui contient tous les éléments supérieurs ou égaux au pivot;

— on trie t et u récursivement.

Il existe plusieurs façons de choisir le pivot, et d'autres variantes du découpage de s. Par exemple, une implémentation simple choisit le pivot arbitrairement, et découpe s en trois parties tel que décrit à l'algorithme 13.

Algorithme 13 : Algorithme de tri rapide (implémentation simple).

Entrées : séquence s d'éléments comparables
Sorties : séquence s triée

```
trier(s):
    si |s| = 0 alors retourner s
    sinon
        choisir pivot ∈ s
        gauche ← [x ∈ s : x < pivot]
        milieu ← [x ∈ s : x = pivot]
        droite ← [x ∈ s : x > pivot]
        retourner trier(gauche) + milieu + trier(droite)
```

La description du tri rapide donnée à l'algorithme 13 nécessite une quantité linéaire de mémoire auxiliaire. Il est possible d'utiliser une quantité *constante* de mémoire en modifiant directement s comme à l'algorithme 14.

Dans le pire cas, le temps d'exécution du tri rapide appartient à $\Theta(n^2)$. Toutefois, en choisissant un pivot aléatoirement ou avec une bonne heuristique, le temps d'exécution est réduit à $\Theta(n \log n)$ en moyenne (avec une faible constante multiplicative), ce qui le rend attrayant en pratique.

Remarque.

En théorie, le tri rapide fonctionne en temps $\mathcal{O}(n \log n)$ dans le pire cas, lorsque le pivot est choisi comme étant la médiane. Cela repose sur la possibilité théorique d'identifier la médiane d'une séquence en temps linéaire. Cependant, ce n'est généralement pas efficace en pratique.

2.6 Propriétés intéressantes

Nous introduisons deux notions intéressantes concernant les algorithmes de tri. Nous disons qu'un algorithme de tri fonctionne *sur place* s'il n'utilise pas de séquence auxiliaire, et nous disons qu'il est stable s'il préserve l'ordre relatif des éléments égaux. Plus formellement, considérons un algorithme de tri $\mathcal{A}$. Nous écrivons σ_s afin de dénoter la permutation telle que $\sigma_s(i)$ est la position de $s[i]$ dans la sortie de $\mathcal{A}$ sur entrée s. Nous disons que $\mathcal{A}$ est *stable* s'il satisfait:

$$\forall s \; \forall i < j \quad (s[i] = s[j]) \rightarrow (\sigma_s(i) < \sigma_s(j)).$$

Algorithme 14 : Algorithme de tri rapide (implémentation sur place avec partition de Lomuto).

Entrées : séquence s d'éléments comparables
Sorties : séquence s triée

```
trier(s):
    partition(lo, hi):      // partionne s autour de x = s[hi] et
        x ← s[hi]; i ← lo     // retourne le nouvel index i de x
        pour j ∈ [lo..hi]
            si s[j] < x alors
                s[i] ↔ s[j]
                i ← i + 1
        s[i] ↔ s[hi]
        retourner i
    trier'(lo, hi):
        si lo < hi alors
            pivot ← partition(lo, hi)
            trier'(lo, pivot − 1)          // Trier le côté gauche
            trier'(pivot + 1, hi)           // Trier le côté droit
    trier'(1, |s|)
    retourner s
```

Par exemple, considérons la séquence $s = [⑤, 2, 1, ②]$ dont nous cherchons à ordonner les éléments selon leur valeur numérique. Un algorithme qui produirait la sortie $[1, ②, 2, ⑤]$ ne serait *pas* stable puisque l'ordre relatif des éléments 2 et ② a été altéré.

Les propriétés satisfaites par un algorithme de tri dépendent de son implémentation. Voici un sommaire des propriétés typiquement satisfaites par les algorithmes présentés jusqu'ici. Notons qu'elles ne sont pas nécessairement satisfaites simultanément par une seule implémentation; et qu'il faut donc considérer ce sommaire avec précaution:

Algorithme	**Complexité** (par cas) meilleur	moyen	pire	**Sur place**	**Stable**
insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
monceau	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
fusion	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
rapide	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	✓	✗

Remarque.

La plupart des langages de programmation utilisent le tri par insertion pour les séquences de petite taille; le tri rapide ou le tri par monceau pour les séquences de grande taille; et le tri par fusion pour les séquences de grande taille lorsque la stabilité est requise. Souvent, une *combinaison* de ceux-ci est utilisée, par ex. « *Timsort* » qui combine tris par fusion et par insertion, et « *introsort* » qui combine tris rapide et par monceau.

2.7 Tri sans comparaison

Tous les algorithmes de tri présentés jusqu'ici trient en comparant les éléments de la séquence en entrée. Il est possible de démontrer que tout algorithme de ce type fonctionne en temps $\Omega(n \log n)$ dans le pire cas. Il existe cependant des algorithmes de tri qui fonctionnent plus rapidement pour certains types de données. Par exemple, considérons la séquence $[7, 3, 5, 1, 2]$. En représentation binaire, celle-ci correspond à:

$$s = [\texttt{111}, \texttt{011}, \texttt{101}, \texttt{001}, \texttt{010}].$$

Réorganisons s sous la forme $s = lo + hi$, où lo contient les séquences qui terminent par 0, et hi celles qui terminent par 1 (en préservant l'ordre relatif des éléments à l'intérieur de lo et hi). Nous obtenons:

$$s' = [\underbrace{\texttt{010}}_{lo}, \underbrace{\texttt{111}, \texttt{011}, \texttt{101}, \texttt{001}}_{hi}].$$

En répétant ce processus, mais maintenant en considérant l'avant-dernier bit, nous obtenons:

$$s'' = [\underbrace{\texttt{101}, \texttt{001}}_{lo}, \underbrace{\texttt{010}, \texttt{111}, \texttt{011}}_{hi}].$$

En répétant une dernière fois, en considérant le premier bit, nous obtenons:

$$s''' = [\underbrace{\texttt{001}, \texttt{010}, \texttt{011}}_{lo}, \underbrace{\texttt{101}, \texttt{111}}_{hi}].$$

Remarquons que s''' correspond numériquement à la séquence triée $[1, 2, 3, 5, 7]$.

Cette procédure, décrite sous forme de pseudocode à l'algorithme 15, est une variante binaire du *tri radix*. Son temps d'exécution appartient à $\Theta(mn)$ dans le meilleur et pire cas. Lorsque n est grand et que m est une constante, cela donne un tri qui fonctionne en temps $\Theta(n)$. Cela surpasse la barrière du $\Omega(n \log n)$, ce qui n'est pas contradictoire, car cet algorithme n'est pas basé sur la comparaison d'éléments et se limite à un type *spécifique* de données.

Algorithme 15 : Algorithme de tri sans comparaison

Entrées : séquence s de n séquences de m bits
Résultat : s est triée

```
1 pour j de m à 1 (à rebours)
2 |   lo ← [ ]; hi ← [ ]
3 |   pour i ∈ [1..n]
4 |   |   si s[i][j] = 0 alors
5 |   |   |   ajouter s[i] à lo
6 |   |   sinon
7 |   |   |   ajouter s[i] à hi
8 |   s ← lo + hi
9 retourner s
```

2.8 Exercices

2.1) Donnez un algorithme qui détermine si une séquence s possède au moins une inversion, et en retourne une le cas échéant. Votre algorithme doit fonctionner en temps $\mathcal{O}(|s|)$.

2.2) Une *séquence binaire* est une séquence dont chaque élément vaut 0 ou 1, par ex. $s = [0, 1, 0, 0, 1, 0, 1]$. Donnez un algorithme qui trie des séquences binaires en temps linéaire avec une quantité constante de mémoire auxiliaire. Autrement dit, donnez un algorithme sur place. Votre algorithme est-il stable?

Répétez l'exercice en supposant que s est une liste chaînée. Tentez d'obtenir un algorithme sur place et stable.

2.3) ★ Donnez un algorithme *itératif* pour le problème précédent qui n'utilise *aucun branchement* (explicite ou implicite), à l'exception de la boucle principale (donc pas de **si**, **min**, **max**, etc. et au plus une boucle **pour**, **tant que**, **faire ... tant que**, etc.) Votre algorithme doit fonctionner en *temps linéaire* et *sur place*.

2.4) Donnez un algorithme qui regroupe les éléments égaux d'une séquence s de façon contigüe. Par exemple, $[\mathsf{a}, \mathsf{a}, \mathsf{c}, \mathsf{b}, \mathsf{b}]$ et $[\mathsf{c}, \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{b}]$ sont des *regroupements* valides de $s = [\mathsf{b}, \mathsf{a}, \mathsf{a}, \mathsf{c}, \mathsf{b}]$. Votre algorithme doit fonctionner en temps $\mathcal{O}(|s|^2)$. L'algorithme peut utiliser les comparaisons $\{=, \neq\}$, mais pas $\{<, \leq, \geq, >\}$.

2.5) Donnez un algorithme qui détermine si une séquence s est un regroupement (au sens de la question précédente). Est-ce que la possibilité d'utiliser $\{<, \leq, \geq, >\}$ fait une différence?

2.6) Considérez une pile de n crêpes de diamètres différents, où les crêpes sont numérotées de 1 à n du haut vers le bas. Vous pouvez réorganiser la pile à l'aide d'une spatule en l'insérant sous une crêpe k, puis en renversant l'ordre des crêpes 1 à k. Donnez un algorithme qui trie les crêpes avec $\mathcal{O}(n)$ renversements. Vous avez accès au diamètre de chaque crêpe.

2.7) Décrivez une façon de rendre tout algorithme de tri stable.

2.8) Décrivez une version améliorée du tri par insertion qui utilise la recherche dichotomique afin d'accélérer l'insertion. Le temps d'exécution dans le pire cas appartient-il toujours à $\Theta(n^2)$?

2.9) Déterminez pourquoi le temps d'exécution du tri rapide appartient à $\Theta(n^2)$ dans le pire cas

2.10) ★ Implémentez le tri par fusion de façon purement itérative; donc sans récursion et sans émulation de la récursion à l'aide d'une pile.

2.11) ★ Démontrez la proposition 16.

2.12) Décrivez une implémentation du tri par fusion qui fonctionne sur les listes chaînées, donc où on ne peut *pas* accéder au $i^{\text{ème}}$ élément en temps constant. Le temps d'exécution demeure-t-il de $\mathcal{O}(n \log n)$? Êtes-vous en mesure d'adapter le tri rapide aux listes chaînées?

2.13) Le tri comptage permet de trier des entiers compris entre 1 et k. Comprenez d'abord cette implémentation, puis analysez sa complexité:

Entrées : $k \in \mathbb{N}_{\geq 1}$, séquence s de n entiers appartenant à $[1..k]$
Sorties : séquence s triée

$c \leftarrow \overbrace{[0, 0, \ldots, 0]}^{k \text{ fois}}$
pour $x \in s$
  $c[x] \leftarrow c[x] + 1$
$i \leftarrow 0$
pour $x \in [1..k]$
  faire $c[x]$ **fois**
    $s[i] \leftarrow x$
    $i \leftarrow i + 1$
retourner s

Si chaque élément de s avait une identité, ce tri serait-il stable? Si ce n'est pas le cas, adaptez l'approche pour obtenir la stabilité.

2.14) ★ Considérons cet algorithme de tri:

Entrées : séquence s de n éléments comparables
Sorties : séquence s triée

pour $i \in [1..n]$
  pour $j \in [1..n]$
    si $s[i] < s[j]$ **alors**
      $s[i] \leftrightarrow s[j]$ `// inverser` $s[i]$ `et` $s[j]$
retourner s

Remarquez que l'algorithme augmente parfois la quantité d'inversions! Montrez qu'il est tout de même correct en démontrant cet invariant:

> Après les $i^{\text{ème}}$ et $j^{\text{ème}}$ itérations des boucles **pour**, nous avons $s[1] \leq s[2] \leq \cdots \leq s[i-1]$ et $s[i] \geq \max s[1:j]$.

2.15) Nous disons qu'une inversion (i, j) est *adjacente* si $j = i + 1$. Considérons un algorithme de tri qui fonctionne par correction d'inversions adjacentes (uniquement). Expliquez pourquoi l'algorithme est forcément stable.

2.16) Considérons la notion d'inversion adjacente introduite à l'exercice 2.15). Montrez l'affirmation suivante: une séquence s qui possède k inversions et dont toutes les inversions sont adjacentes nécessite exactement k corrections d'inversions.

2.17) ★ Considérons une approche « gloutonne » au tri d'une séquence s. Informellement, on considère toutes les inversions de s et on corrige celle qui permet de réduire la quantité d'inversions le plus possible.

Formellement, soit s une séquence d'éléments comparables. Pour chaque inversion (i, j) de s, on définit $g(i, j)$ comme étant le nombre d'inversions de la séquence s après la correction de (i, j). On dit qu'une inversion (i, j) est *prometteuse* si $g(i, j)$ est minimal parmi toutes les inversions de s. L'approche gloutonne consiste à corriger une inversion prometteuse à chaque itération.

Montrez que l'approche gloutonne ne fonctionne pas.

Cet exercice répond à une question posée en classe sur l'algorithme 9 (A23).

3

Graphes

Dans ce chapitre, nous introduisons formellement les graphes: une structure de données qui généralise les arbres (couverts par ex. dans un cours de structures de données comme IFT339). Nous discutons de leur représentation, leurs propriétés, ainsi que certains algorithmes fondamentaux (parcours, détection de cycle, tri topologique, etc.)

3.1 Graphes non dirigés

Un *graphe non dirigé* est une paire $\mathcal{G} = (V, E)$ d'ensembles tels que $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$. Nous appelons V et E respectivement l'ensemble des *sommets* et des *arêtes* de $\mathcal{G}$. Si V est fini, nous disons que $\mathcal{G}$ est *fini*, et sinon qu'il est *infini*. Par défaut, nous supposerons qu'un graphe est fini.

Nous écrivons $u \to v$ afin d'indiquer que $\{u, v\} \in E$. Observons que $u \to v \iff v \to u$. Deux sommets u et v sont *adjacents* si $\{u, v\} \in E$. Nous disons que v est un *voisin* de u, si u et v sont adjacents.

Le *degré* d'un sommet u, dénoté $\deg(u)$, correspond à son nombre de voisins. Remarquons qu'un graphe non dirigé possède entre 0 et $\binom{|V|}{2} \in \Theta(|V|^2)$ arêtes.

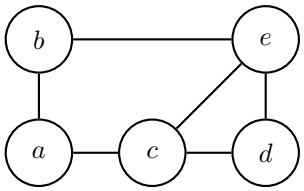


FIGURE 3.1 – Exemple de graphe (fini) non dirigé.

Exemple.

Considérons le graphe non dirigé

$$\mathcal{G} = (\{a, b, c, d, e\}, \{\{a, b\}, \{a, c\}, \{b, e\}, \{c, d\}, \{c, e\}, \{d, e\}\})$$

représenté graphiquement à la figure 3.1. Ce graphe possède 5 sommets et 6 arêtes. Les voisins du sommet c sont $\{a, e, d\}$. Nous avons $\deg(a) = \deg(b) = \deg(d) = 2$ et $\deg(c) = \deg(e) = 3$.

3.2 Graphes dirigés

Un *graphe dirigé* est une paire $\mathcal{G} = (V, E)$ d'ensembles tels que $E \subseteq \{(u, v) \in V \times V : u \neq v\}$. Comme pour les graphes non dirigés, nous appelons V et E respectivement l'ensemble des *sommets* et des *arêtes* de $\mathcal{G}$, et nous considérerons les graphes finis par défaut.

Nous écrivons $u \to v$ afin d'indiquer que $(u, v) \in E$. Lorsque $u \to v$, nous disons que u est un *prédecesseur* de v, et que v est un *successeur* de u.

Le *degré entrant* d'un sommet u, dénoté $\deg^-(u)$, correspond à son nombre de prédecesseurs, et le *degré sortant* d'un sommet u, dénoté $\deg^+(u)$, correspond à son nombre de successeurs. Remarquons qu'un graphe dirigé possède entre 0 et $|V| \cdot (|V| - 1) \in \Theta(|V|^2)$ arêtes. De plus, nous avons:

$$|E| = \sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v).$$

Exemple.

Considérons le graphe dirigé

$$\mathcal{G} = (\{a, b, c, d, e\}, \{(a, b), (b, c), (c, a), (c, d), (d, b), (d, c), (d, e)\})$$

représenté graphiquement à la figure 3.2. Ce graphe possède 5 sommets et 7 arêtes. Nous avons $a \to b$, mais pas $b \to a$. De plus, nous avons:

$$\begin{aligned} \deg^-(a) = \deg^-(d) = \deg^-(e) = 1, \quad & \deg^+(a) = \deg^+(b) = 1, \\ \deg^-(b) = \deg^-(c) = 2, \quad & \deg^+(c) = 2, \\ & \deg^+(d) = 3, \\ & \deg^+(e) = 0. \end{aligned}$$

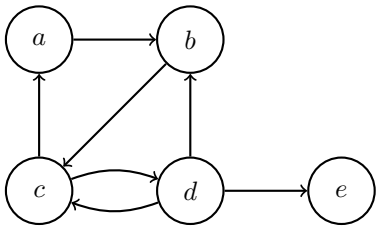


FIGURE 3.2 – Exemple de graphe fini dirigé.

Remarque.

Certains ouvrages font la distinction entre *sommets* et *arêtes* pour les graphes non dirigés, et *noeuds* et *arcs* pour les graphes dirigés. Nous ne ferons pas cette distinction.

3.3 Représentation

3.3.1 Matrice d'adjacence

Les graphes sont généralement représentés sous forme de matrice ou de liste d'adjacence. La *matrice d'adjacence* d'un graphe $\mathcal{G}$ est la matrice $\mathbf{A}$ définie par:

$$\mathbf{A}[u,v] \coloneqq \begin{cases} 1 & \text{si } u \to v, \\ 0 & \text{sinon.} \end{cases}$$

Ainsi, $\mathbf{A}[u,v]$ indique si $\mathcal{G}$ possède une arête du sommet u vers le sommet v. Observons que dans le cas des graphes non dirigés, nous avons $\mathbf{A}[u,v] = \mathbf{A}[v,u]$ pour tous sommets u et v.

Exemple.

Les graphes de la figure 3.1 et de la figure 3.2 sont représentés respectivement par ces matrices d'adjacence:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array} \quad \text{et} \quad \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}.$$

3.3.2 Liste d'adjacence

Une *liste d'adjacence* d'un graphe $\mathcal{G} = (V, E)$ est une séquence adj telle que $adj[u] \coloneqq [v \in V : u \to v]$ pour tout sommet u. Autrement dit, $adj[u]$ dénote l'ensemble des voisins de u si $\mathcal{G}$ est non dirigé, ou l'ensemble des successeurs de u si $\mathcal{G}$ est dirigé. Une liste d'adjacence s'implémente généralement sous forme de tableau lorsque $V = \{1, 2, \ldots, n\}$, et de tableau associatif sinon. À l'interne, chaque entrée $adj[u]$ se représente sous forme de liste.

Exemple.

Les graphes de la figure 3.1 et de la figure 3.2 sont représentés respectivement par les listes d'adjacence:

$$\begin{aligned} adj[a] &= [b, c] & adj'[a] &= [b] \\ adj[b] &= [a, e] & adj'[b] &= [c] \\ adj[c] &= [a, d, e] & adj'[c] &= [a, d] \\ adj[d] &= [c, e] & adj'[d] &= [b, c, e] \\ adj[e] &= [b, c, d], \text{ et} & adj'[e] &= [\,]. \end{aligned}$$

3.3.3 Complexité des représentations

Le tableau suivant dresse un sommaire de la complexité de l'identification des voisins, successeurs et prédecesseurs d'un sommet, l'ajout et le retrait d'arêtes, ainsi que de la mémoire utilisée par les deux représentations [1]. Puisque nous utiliserons principalement les graphes comme structures de données statiques, nous n'entrons pas ici dans le détail des opérations dynamiques comme l'ajout et le retrait de sommets. Notons également que certaines des complexités ci-dessous peuvent être réduites à l'aide d'implémentations plus sophistiquées.

	Matrice d'adj.	**Liste d'adjacence**	
		graphe non dirigé	*graphe dirigé*
$u \to v$?	$\Theta(1)$	$\mathcal{O}(1 + \min(\deg(u), \deg(v)))$	$\mathcal{O}(1 + \deg^+(u))$
$\{v : u \to v\}$	$\Theta(\lvert V\rvert)$	$\mathcal{O}(1 + \deg(u))$	$\mathcal{O}(1 + \deg^+(u))$
$\{u : u \to v\}$	$\Theta(\lvert V\rvert)$	$\mathcal{O}(1 + \deg(v))$	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$
Ajouter $u \to v$	$\Theta(1)$	$\mathcal{O}(1 + \deg(u) + \deg(v))$	$\mathcal{O}(1 + \deg^+(u))$
Retirer $u \to v$	$\Theta(1)$	$\mathcal{O}(1 + \deg(u) + \deg(v))$	$\mathcal{O}(1 + \deg^+(u))$
Mémoire	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$

1. Nous supposons que l'accès à $adj[u]$ se fait en temps constant dans le pire cas, mais en pratique ce n'est pas nécessairement le cas lorsque $V \neq [1..n]$. En effet, si les sommets proviennent d'un domaine plus complexe, alors adj risque d'être implémenté par une table de hachage qui offre généralement un temps constant *amorti*.

3.4 Accessibilité

Un *chemin* C, allant d'un sommet u vers un sommet v, est une séquence de sommets $C = [v_0, v_1, \ldots, v_k]$ telle que $v_0 = u$, $v_k = v$, et $v_{i-1} \to v_i$ pour tout $i \in [1..k]$. Nous disons qu'un chemin est *simple* s'il ne répète aucun sommet. La *longueur* d'un chemin correspond à $|C| \coloneqq k$. Nous écrivons $u \xrightarrow{*} v$ afin d'indiquer qu'il existe un chemin de u vers v. En particulier, comme la séquence vide est un chemin, nous avons $u \xrightarrow{*} u$ pour tout sommet u.

Exemple.

Le chemin $a \to b \to e \to d \to c \to e \to b$ du graphe de la figure 3.1 est un chemin non simple de a vers b. Le chemin $a \to b \to c \to d \to e$ du graphe de la figure 3.2 est un chemin simple de longueur 4 qu'on ne peut pas étendre.

Nous considérons deux façons de parcourir un graphe. Plus précisément, étant donné un sommet de départ u, nous présentons deux approches qui calculent l'ensemble des sommets accessibles par u, c.-à-d. l'ensemble $\{v : u \xrightarrow{*} v\}$. Cela permet notamment de déterminer si un sommet cible est accessible à partir d'un sommet de départ.

3.4.1 Parcours en profondeur

La première approche consiste à débuter au sommet u, de considérer l'un de ses successeurs, puis de visiter le graphe récursivement le plus profondément possible, et finalement de rebrousser chemin et répéter avec un autre successeur de u. Afin d'éviter l'exploration d'un sommet plus d'une fois, nous marquons progressivement chaque sommet visité. Cette approche est décrite sous forme de pseudocode à l'algorithme 16.

Algorithme 16 : Parcours en profondeur (version récursive).

Entrées : graphe $\mathcal{G} = (V, E)$ et sommet $u \in V$
Résultat : une séquence $s = [v \in V : u \xrightarrow{*} v]$

```
1 s ← [ ]
2 parcours(x):
3 |  si x n'est pas marqué alors
4 |  |  marquer x
5 |  |  pour y ∈ V : x → y                    // explorer voisins/succ.
6 |  |  |  parcours(y)
7 |  |  ajouter x à s                  // ajouter aux sommets accessibles
8 parcours(u)
9 retourner s
```

Observons que l'algorithme 16 n'explore un sommet qu'au plus une fois: une fois s'il est accessible à partir du sommet de départ, et aucune autrement. De plus, chaque sommet visité lance une exploration pour chacun de ses successeurs. Ainsi, l'algorithme fonctionne en temps $\mathcal{O}(|V| + |E|)$ dans le pire cas.

Comme l'algorithme cherche à explorer le graphe le plus profondément possible, son implémentation récursive peut consommer une quantité non négligeable de mémoire en pratique, selon l'architecture. L'algorithme 17 donne une description itérative de la même approche, cette fois en utilisant explicitement une pile. Remarquons que la taille de la pile ne peut pas excéder $\mathcal{O}(|E|)$ puisque chaque arête est explorée au plus une fois. Une légère modification permettrait de borner sa taille par $\mathcal{O}(\min(|V|, |E|))$.

Algorithme 17 : Parcours en profondeur (version itérative).

Entrées : graphe $\mathcal{G} = (V, E)$ et sommet $u \in V$
Résultat : une séquence $s = [v \in V : u \xrightarrow{*} v]$

```
1 initialiser une pile P ← [u]; s ← [ ]
2 tant que P n'est pas vide
3 |  dépiler x de P
4 |  si x n'est pas marqué alors
5 |  |  marquer x
6 |  |  pour y ∈ V : x → y              // explorer voisins/succ.
7 |  |  |  empiler y dans P
8 |  |  ajouter x à s          // ajouter aux sommets accessibles
9 retourner s
```

3.4.2 Parcours en largeur

Une alternative au parcours en profondeur consiste à explorer le graphe en largeur. Autrement dit, nous débutons au sommet u, puis explorons chacun de ses successeurs, puis chacun de leurs successeurs, et ainsi de suite. Cette approche s'apparente à la version itérative du parcours en profondeur: on remplace simplement la pile par une file. Cette procédure est décrite à l'algorithme 18.

3.5 Cycles et ordres topologiques

Un *cycle* est un chemin non vide d'un sommet vers lui-même. Nous disons qu'un cycle est *simple* s'il ne répète aucun sommet, à l'exception du sommet de départ qui apparaît au début et à la fin, et n'utilise pas deux fois la même arête [2]. Autrement dit, un cycle est simple s'il ne contient pas d'autres cycles. Nous disons qu'un graphe est *acyclique* s'il ne possède aucun cycle simple.

2. Cette deuxième contrainte est redondante pour les graphes dirigés, mais elle empêche de considérer $u \to v \to u$ comme étant un cycle simple pour les graphes non dirigés.

Algorithme 18 : Parcours en largeur.

Entrées : graphe $\mathcal{G} = (V, E)$ et sommet $u \in V$
Résultat : une séquence $s = [v \in V : u \xrightarrow{*} v]$

```
1 initialiser une file F ← [u]; s ← [ ]
2 tant que F n'est pas vide
3 |  retirer x de F
4 |  si x n'est pas marqué alors
5 |  |  marquer x
6 |  |  pour y ∈ V : x → y                    // explorer voisins/succ.
7 |  |  |  ajouter y à F
8 |  |  ajouter x à s                  // ajouter aux sommets accessibles
9 retourner s
```

Exemple.

Le chemin $a \to b \to e \to d \to c \to a$ du graphe de la figure 3.1 est un cycle simple de a vers a. Le chemin $a \to b \to c \to d \to c \to a$ du graphe de la figure 3.2 est un cycle non simple car il contient le cycle $c \to d \to c$.

La notion de cycle est liée à la notion d'ordre topologique. Un *ordre topologique* d'un graphe dirigé $\mathcal{G} = (V, E)$ est une séquence de sommets $v_1, v_2, \ldots, v_k$ qui satisfait:

— $V = \{v_1, v_2, \ldots, v_k\}$, et
— $\forall i, j \in [1..k] : (v_i, v_j) \in E \implies i < j$.

Par exemple, considérons le graphe $\mathcal{G} = (V, E)$ où V dénote l'ensemble des cours obligatoires qu'une personne doit suivre dans son parcours universitaire, et où $(u, v) \in E$ indique que le cours u est préalable au cours v. Un ordre topologique de $\mathcal{G}$ décrit un ordre dans lequel cette personne peut s'inscrire à ses cours. S'il n'existe aucun ordre topologique, alors cela signifie que cette personne ne pourra jamais graduer! Ce scénario indiquerait qu'il existe un cycle dans $\mathcal{G}$:

Proposition 18. *Un graphe possède un ordre topologique ssi il est acyclique.*

La recherche d'un ordre topologique $\preceq$ dans un graphe acyclique s'apparente en quelque sorte à un problème de tri, mais où nous n'avons qu'une information partielle sur l'ordre $\preceq$; celui-ci doit satisfaire la contrainte $u \xrightarrow{*} v \implies u \preceq v$. Nous ne pouvons donc pas utiliser les algorithmes de tri du chapitre 2. [3]

Nous présentons donc, à l'algorithme 19, une procédure qui calcule un ordre topologique d'un graphe en temps linéaire, s'il en existe un, et qui détecte la présence d'un cycle autrement.

3. Le calcul d'un ordre toplogique correspond précisément à la construction d'un ordre total compatible avec $\xrightarrow{*}$, c.-à-d. une extension linéaire de $\xrightarrow{*}$.

Algorithme 19 : Algorithme de Kahn: tri topologique.

Entrées : graphe dirigé $\mathcal{G} = (V, E)$
Résultat : un ordre topologique de $\mathcal{G}$ s'il en existe un

```
// Calcul des degrés entrants
d ← [v ↦ 0 : v ∈ V]
pour (u, v) ∈ E
    d[v] ← d[v] + 1
// Calcul des sommets de degré entrant 0
initialiser une file F ← [ ]
pour v ∈ V
    si d[v] = 0 alors ajouter v à F
// Tri topologique
ordre ← [ ]
tant que F n'est pas vide
    retirer u de F
    ajouter u à ordre
    pour v ∈ V : u → v                    // explorer successeurs
        d[v] ← d[v] − 1
        si d[v] = 0 alors ajouter v à F
si |ordre| = |V| alors retourner ordre
sinon retourner cycle détecté
```

Cette procédure utilise l'approche suivante:

— on identifie les sommets qui ne dépendent d'aucun sommet, c.-à-d. les sommets de degré entrant 0;

— on ajoute ces sommets (arbitrairement) à l'ordre topologique;

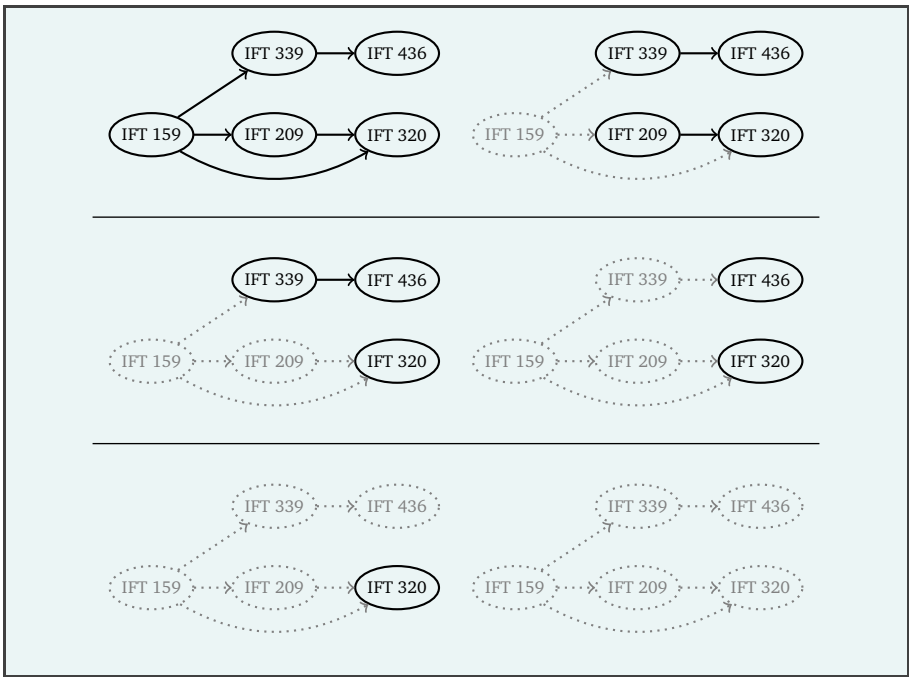
— on retire implicitement ces sommets du graphe en mettant à jour le degré entrant de leurs successeurs;

— on recommence le processus tant que possible.

S'il existe un sommet dont le degré entrant ne décroît pas à 0, alors cela signifie qu'il apparaît sur un cycle.

Exemple.

Voici une exécution de l'algorithme 19 qui retourne l'ordre topologique [IFT 159, IFT 209, IFT 339, IFT 436, IFT 320]:



Analysons le temps d'exécution de l'algorithme 19. Le calcul des degrés entrants s'effectue en temps $\Theta(|V|+|E|)$, et l'identification des sommets de degré entrant 0 s'effectue en temps $\Theta(|V|)$. L'exploration d'un sommet u requiert un temps appartenant à $\mathcal{O}(1 + \deg^+(u))$. Puisque tous les sommets peuvent être explorés, le temps total de l'exploration du graphe appartient donc à

$$\mathcal{O}\left(\sum_{u\in V}(1+\deg^+(u))\right) = \mathcal{O}\left(\sum_{u\in V}1 + \sum_{u\in V}\deg^+(u)\right) = \mathcal{O}(|V|+|E|).$$

Ainsi, le temps total de l'algorithme appartient à $\mathcal{O}(|V| + |E|)$.

3.6 Connexité et arbres couvrants

Un *sous-graphe* d'un graphe $\mathcal{G} = (V, E)$ est un graphe $\mathcal{G}' = (V', E')$ tel que $V' \subseteq V$, $E' \subseteq E$ et E' n'utilise que des sommets de V'. Le *sous-graphe induit* par un ensemble de sommets $V' \subseteq V$ est le sous-graphe $G[V'] \coloneqq (V', E')$ où E' contient *toutes* les arêtes de E dont les sommets appartiennent à V'.

Si $u \xrightarrow{*} v$ pour tous $u, v \in V$, nous disons que $\mathcal{G}$ est *connexe* s'il est non dirigé, et *fortement connexe* s'il est dirigé. Une *composante connexe* d'un graphe non dirigé est un sous-graphe connexe maximal. Une *composante fortement connexe* d'un graphe dirigé est un sous-graphe fortement connexe maximal.

Exemple.

Le graphe $\mathcal{G}$ de la figure 3.1 est connexe et ne possède donc qu'une seule composante connexe, c.-à-d. $\mathcal{G}$ lui-même. Le graphe $\mathcal{G}'$ de la figure 3.2 n'est pas fortement connexe puisqu'il n'existe aucun chemin du sommet e vers les autres sommets. Les composantes fortement connexes de $\mathcal{G}'$ sont $\mathcal{G}'[\{a, b, c, d\}]$ et $\mathcal{G}'[\{e\}]$.

Informellement, il est clair que la notion d'arbre est un cas particulier de la notion de graphe. Formellement, elle se définit de la façon suivante. Une *forêt* est un graphe non dirigé acyclique et un *arbre* est une forêt connexe. Un sommet v d'une forêt est une *feuille* si $\deg(v) = 1$, et un *sommet interne* sinon. Nous disons qu'un arbre est une *arborescence* s'il possède un sommet spécial r appelé sa *racine*. Dans ce cas, nous ne considérons pas r comme étant une feuille, même lorsque $\deg(r) = 1$.

Alternativement, les arbres se caractérisent de plusieurs façons équivalentes:

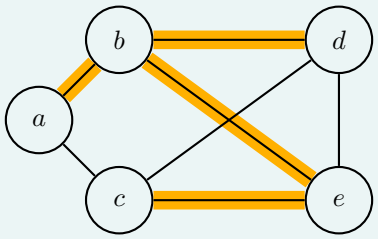
Proposition 19. *Soit $\mathcal{G} = (V, E)$ un graphe non dirigé. Les propriétés suivantes sont toutes équivalentes:*

- *$\mathcal{G}$ est connexe et acyclique;*
- *$\mathcal{G}$ est connexe et $|E| = |V| - 1$;*
- *$\mathcal{G}$ est acyclique et $|E| = |V| - 1$.*

Bien que les graphes non dirigés sont plus généraux que les arbres, nous allons voir qu'il est toujours possible d'identifier des arbres à l'intérieur de leurs composantes connexes. Formellement, un *arbre couvrant* d'un graphe non dirigé $\mathcal{G}$ est un sous-graphe de $\mathcal{G}$ qui contient tous ses sommets et qui est un arbre. Un tel arbre correspond à une façon de relier tous les sommets de $\mathcal{G}$ avec le moins d'arêtes possibles.

Exemple.

Les traits gras colorés ci-dessous correspondent à un arbre couvrant:



Afin de montrer l'existence d'arbres couvrants, nous revisitons le parcours en profondeur. Considérons la variante décrite à l'algorithme 20. Nous disons qu'un sommet v exploré via la ligne 7 est une *source*. Cette procédure visite *tous* les sommets du graphe, en lançant un parcours en profondeur par source.

Algorithme 20 : Parcours en profondeur qui visite tous les sommets.

Entrées : graphe $\mathcal{G} = (V, E)$
Résultat : —

```
1 parcours(x):
2 |  si x n'est pas marqué alors
3 |  |  marquer x
4 |  |  pour y ∈ V : x → y            // explorer voisins/succ.
5 |  |  |  parcours(y)
6 pour v ∈ V
7 |  si v n'est pas marqué alors parcours(v)
```

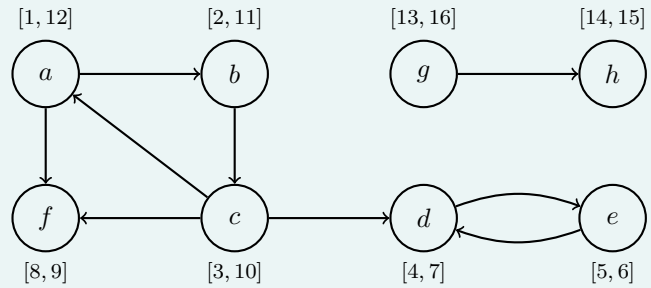
Soit $\mathcal{G} = (V, E)$ un graphe et $n \coloneqq |V|$ son nombre de sommets. Lors d'un parcours en profondeur de $\mathcal{G}$, nous assignons à chaque sommet $v \in V$ un temps de *découverte* $d[v]$, et de *fin de traitement* $f[v]$ de la façon suivante. Nous initialisons un compteur t à 0. Lorsqu'un sommet v est découvert, t est incrémenté et v reçoit le temps de découverte $d[v] = t$. Lorsque tous les voisins/successeurs d'un sommet v ont été complètement traités, t est incrémenté et v reçoit le temps de fin de traitement $f[v] = t$. Ainsi, à la fin de l'algorithme, les temps $[1..2n]$ ont tous été assignés. Nous associons à chaque sommet v l'intervalle $I_v \coloneqq [d[v], f[v]]$. Informellement, les intervalles obtenus lors d'un parcours en profondeur sont « bien parenthésés ». Plus formellement:

Théorème 3. *Soient u et v des sommets distincts d'un graphe $\mathcal{G}$. Exactement une de ces affirmations est vraie:*

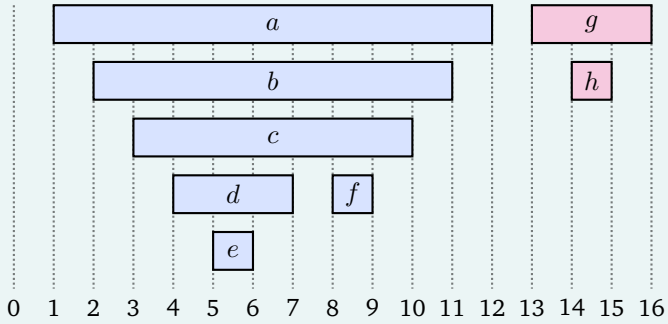
- *I_u contient strictement I_v,*
- *I_v contient strictement I_u,*
- *I_u et I_v sont disjoints.*

Exemple.

Considérons un parcours en profondeur sur le graphe dirigé ci-dessous dans lequel les sommets sont visités en ordre alphabétique. Chaque sommet v est étiqueté par son intervalle I_v.



Graphiquement, nous observons que les intervalles sont « bien parenthésés » au sens du théorème 3, où a et g sont des sources:

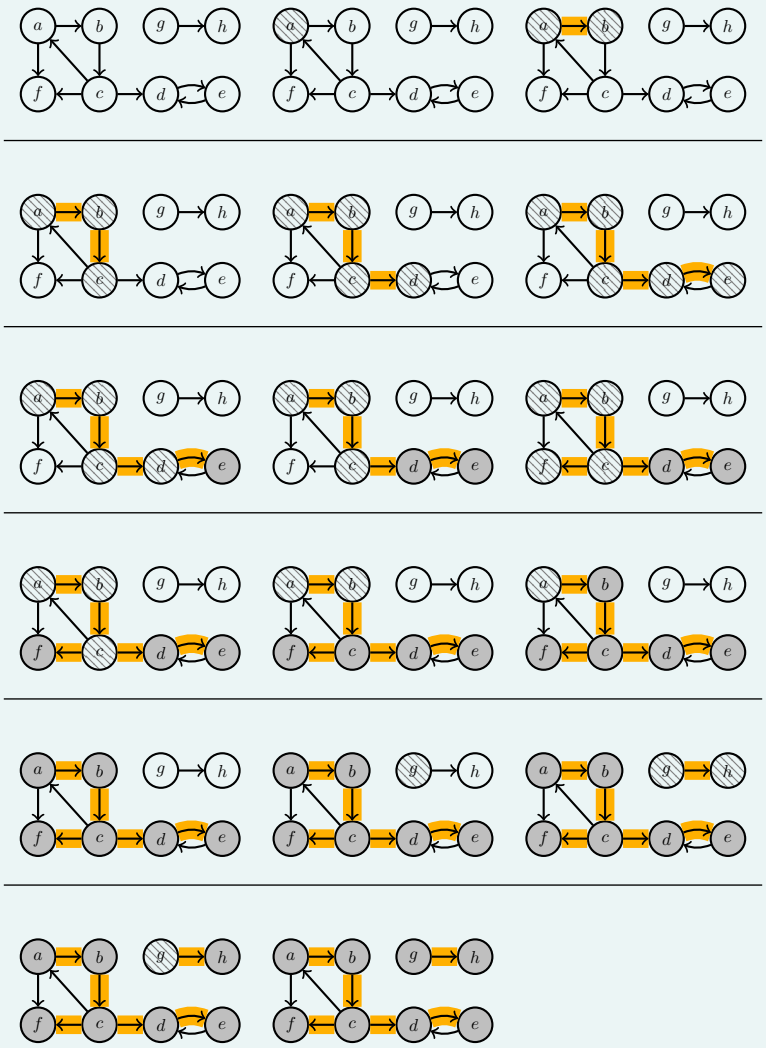


L'exécution d'un parcours en profondeur fait croître des arboresences via un chemin correspondant à la pile d'appel. Afin de s'en convaincre, assignons des couleurs aux sommets. À tout temps t, nous disons qu'un sommet v est: *blanc* s'il n'a pas été découvert ($t < d[v]$); *gris* s'il a été découvert mais est toujours en traitement ($d[v] \leq t < f[v]$); et *noir* s'il a été découvert et traité ($f[v] \leq t$).

Lors de la découverte d'un sommet v via $u \to v$, nous assignons $\text{pred}[v] := u$ comme prédecesseur de v.

Exemple.

Reconsidérons le graphe de l'exemple précédent à chaque étape du parcours (temps 0 à 16). Nous représentons les sommets blancs, gris et noirs respectivement par des sommets vides, rayés et pleins; et les arêtes qui découvrent des sommets par des traits gras de couleur.



À la fin de l'exécution, le graphe est marqué par une forêt qui contient deux arboresences: une enracinée en a qui contient $\{a, b, c, d, e, f\}$ et une enracinée en g qui contient $\{g, h\}$.

Proposition 20. *À chaque temps d'un parcours en profondeur d'un graphe:*

— *les sommets gris forment un chemin simple;*

— *les sommets colorés forment une forêt, avec une arboresence par source.*

L'observation précédente montre comment détecter si un graphe non dirigé est connexe, et, le cas échéant, construire un arbre couvrant. En effet, il suffit

de lancer un parcours en profondeur et de stocker le prédecesseur de chaque sommet. Si une seule source est identifiée, alors le graphe est connexe et la forêt obtenue est un arbre couvrant. Formellement:

Corollaire 1. *Un graphe non dirigé $\mathcal{G} = (V, E)$ possède un arbre couvrant ssi $\mathcal{G}$ est connexe. De plus, un tel arbre se calcule en temps linéaire, c.-à-d. $\mathcal{O}(|V| + |E|)$.*

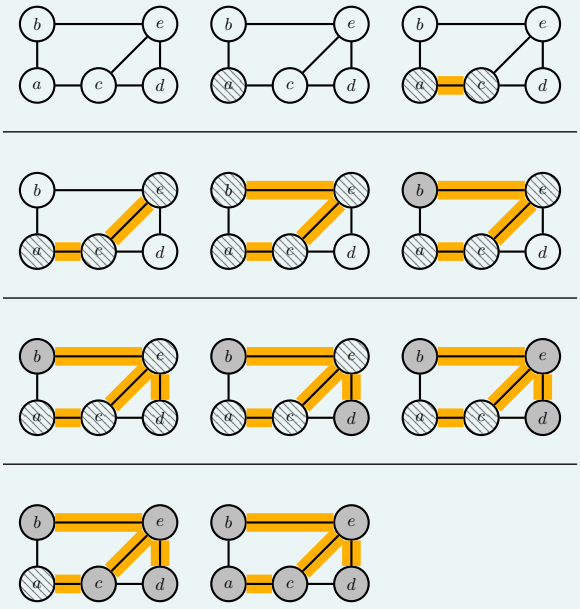
Démonstration. Considérons la fin d'un parcours en profondeur de $\mathcal{G}$. Tous les sommets de $\mathcal{G}$ sont noirs. Ainsi, par la proposition 20, tous les sommets de $\mathcal{G}$ font partie d'une forêt $\mathcal{F}$, avec une arboresence par source. De plus, $\mathcal{F}$ se calcule en temps linéaire comme le parcours est linéaire.

$\Rightarrow$) Puisque $\mathcal{G}$ possède un arbre couvrant, chaque sommet peut atteindre chaque sommet dans cet arbre, ce qui signifie que $\mathcal{G}$ est connexe.

$\Leftarrow$) Soit u le premier sommet de $\mathcal{G}$ sur lequel le parcours est lancé. Comme $\mathcal{G}$ est connexe, u peut atteindre tous les sommets, c.-à-d. $\{v \in V : u \xrightarrow{*} v\} = V$. Ainsi, au temps $f[u]$, tous les sommets ont été visités, ce qui fait de u l'unique source du parcours. La forêt $\mathcal{F}$ contient donc une seule arboresence. Comme elle contient tous les sommets de $\mathcal{G}$, il s'agit d'un arbre couvrant. □

Exemple.

Rappelons que le graphe de la figure 3.1 est connexe. Il possède donc un arbre couvrant. Nous pouvons en obtenir un en lançant un parcours en profondeur, par ex. à partir du sommet a:



Des adaptations plus sophistiquées de l'algorithme de parcours en profondeur, qui s'appuient sur les observations précédentes, permettent d'identifier les composantes fortement connexes d'un graphe *dirigé* en temps linéaire, par ex. l'algorithme de Kosaraju-Sharir et l'algorithme de Tarjan.

3.7 Calcul de plus court chemin

Les algorithmes de parcours en profondeur et largeur permettent notamment de déterminer si un sommet u peut atteindre un sommet v. Comme nous l'avons vu pour le parcours en profondeur, il est possible d'adapter légèrement ces algorithmes afin qu'ils construisent explicitement un chemin simple de u vers v (en stockant une forêt de prédecesseurs). Le parcours en profondeur a tendance à construire des chemins longs, parfois même de longueur maximale. Cependant, le parcours en largeur construit toujours un chemin de longueur *minimale* puiqu'il explore progressivement les sommets à distance $1, 2, \ldots, |V|-1$ du sommet de départ.

L'algorithme 21 présente une adaptation du parcours en largeur qui construit un chemin de longueur minimale (s'il en existe un). Celle-ci débute au sommet de départ u et parcourt le graphe en largeur. Lors de l'exploration d'un nouveau sommet y, on stocke le prédecesseur $pred[y]$ qui a mené à son exploration. Si le sommet cible v est atteint, alors on arrête l'exploration. On reconstruit le chemin qui mène de u vers v en calculant la séquence $[v, pred[v], pred[pred[v]], \ldots, u]$ qui décrit le chemin inverse. Il suffit donc de renverser cette séquence afin d'obtenir le chemin minimal. Notons qu'on peut éviter ce renversement en ajoutant $v, pred[v], pred[pred[v]], \ldots$ directement dans une file.

Algorithme 21 : Calcul de plus court chemin.

Entrées : graphe $\mathcal{G} = (V, E)$ et sommet $u \in V$
Résultat : un chemin de longueur minimale tel que $u \xrightarrow{*} v$ s'il en existe un, ou « *aucun* » sinon

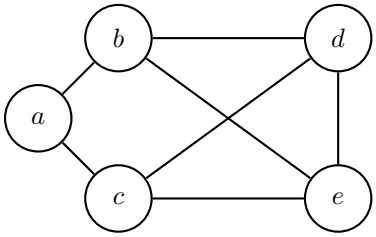
```
1  si u = v alors retourner [ ]                          // chemin trivial?
   // Parcourir le graphe en largeur
2  initialiser une file F ← [u]
3  trouvé ← faux
4  pred   ← [ ]
5  tant que ¬trouvé et F n'est pas vide
6  |  retirer x de F
7  |  marquer x
8  |  pour y ∈ V : x → y                    // explorer voisins/succ.
9  |  |  si y n'est pas marqué alors
10 |  |  |  pred[y] ← x                 // se souvenir d'où on arrive
11 |  |  |  si y = v alors                        // cible v atteinte?
12 |  |  |  |  trouvé ← vrai
13 |  |  |  sinon
14 |  |  |  |  ajouter y à F
   // Construire le chemin de u vers v
15 si ¬trouvé alors
16 |  retourner aucun
17 sinon
18 |  s ← [ ]
19 |  x ← v
20 |  tant que x ≠ u                  // rebrousser chemin de v vers u
21 |  |  ajouter x à s
22 |  |  x ← pred[x]
23 |  renverser s
24 |  retourner s
```

3.8 Exercices

3.1) Nous avons vu comment détecter la présence de cycle dans un graphe dirigé à l'aide du tri topologique. Donnez un autre algorithme de détection de cycle basé sur le parcours en profondeur.

3.2) Considérez le scénario suivant: n personnes participent à une fête où les rafraîchissements sont gratuits. Une personne qui a vu une annonce de l'événement sur Internet s'est invitée à la fête. Cette personne connaît tout le monde (via la liste d'invité·e·s affichée en ligne), mais personne ne la connaît. Aidez les personnes participant à la fête à sauver les stocks de rafraîchissements en donnant un algorithme qui identifie l'intrus en temps $\mathcal{O}(n)$, étant donné une matrice d'adjacence $\mathbf{A}$ de taille $n \times n$, où $\mathbf{A}[i, j]$ indique si la personne i connaît la personne j.

3.3) Identifiez d'autres arbres couvrants du graphe présenté en exemple:



3.4) Un graphe non dirigé $\mathcal{G} = (V, E)$ est dit *biparti* s'il existe $X, Y \subseteq V$ tels que $X \cap Y = \emptyset$, $X \cup Y = V$, et $\{u, v\} \in E \implies (u \in X \iff v \in Y)$. Autrement dit, $\mathcal{G}$ est biparti si on peut partitionner ses sommets en deux ensembles X et Y de telle sorte que toute arête possède un sommet dans X et un sommet dans Y. Donnez un algorithme qui détermine efficacement si un graphe est biparti. Analysez sa complexité. Adaptez votre algorithme afin qu'il calcule une partition X, Y si le graphe est biparti.

3.5) Montrez qu'un graphe dirigé possède un cycle si, et seulement si, il possède un cycle simple.

3.6) ★ Un *cycle eulérien* est un cycle qui visite *toutes* les arêtes d'un graphe *exactement une* fois.

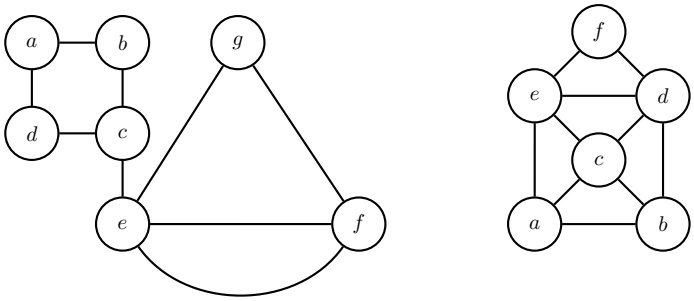
a) Montrez que si un graphe non dirigé $\mathcal{G}$ possède un cycle eulérien, alors $\mathcal{G}$ possède au plus une composante connexe non triviale et tous ses sommets soqnt de degré pair.

Indice: pensez à construire un cycle à partir d'un sommet arbitraire.

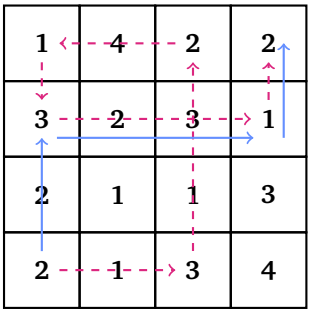
b) Donnez un algorithme qui détermine si un graphe non dirigé possède un cycle eulérien, et en identifie un le cas échéant.

c) Un *chemin eulérien* est un chemin qui visite toutes les arêtes d'un graphe exactement une fois, sans nécessairement revenir au point de départ. Donnez un algorithme qui détermine si un graphe non dirigé possède un chemin eulérien, et en identifie un le cas échéant.

d) Exécutez votre dernier algorithme sur ces graphes:



3.7) Un *labyrinthe numérique* est une grille g de taille $n \times n$ où $g[i,j] \in \mathbb{N}$. À partir d'une case (i,j) de g, on peut se déplacer de $g[i,j]$ positions vers le haut, la droite, le bas ou la gauche, pourvu qu'on demeure dans la grille. Nous disons qu'il est possible de *traverser* un labyrinthe numérique s'il existe une façon de se rendre de sa case $(1,1)$ à sa case (n,n). Par exemple, il est notamment possible de traverser ce labyrinthe numérique en six étapes (lignes tiretées) et trois étapes (lignes pleines):



a) Donnez un algorithme qui détermine s'il est possible de traverser un labyrinthe numérique. Adaptez-le pour qu'il identifie une solution minimale (lorsqu'il en existe une).

b) Donnez un algorithme qui détermine s'il existe une case accessible à partir de laquelle il est impossible de se déplacer.

(*tiré de [Eri19, chap. 5, ex. 11]* ©)

3.8) Montrez que si un graphe dirigé possède un ordre topologique, alors il est acyclique.

3.9) Considérons ce jeu d'instructions très limité:

— `cbz xi, j`: saute à la ligne j si $x_i = 0$, et à la ligne suivante sinon;

— `cbnz xi, j`: saute à la ligne j si $x_i \neq 0$, et à la ligne suivante sinon;

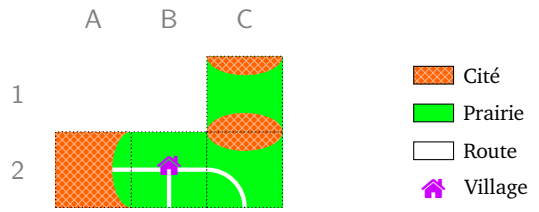
— `b j`: saute à la ligne j;

— `ret`: termine le programme.

Donnez un algorithme qui, étant donné les valeurs des registres, détermine si un programme termine. On suppose que la dernière ligne contient toujours `ret`. Par exemple, le programme ci-dessous termine sur $\langle x_1 = 1, x_4 = 0, x_5 = 1, \ldots\rangle$ mais ne termine pas sur $\langle x_1 = 1, x_4 = 1, x_5 = 1, \ldots\rangle$.

```
1: cbnz x5, 5
2: ret
3: cbz  x4, 6
4: b    3
5: cbnz x1, 4
6: ret
```

3.10) Dans le jeu de société Carcassonne, une carte est construite itérativement à l'aide de tuiles carrées. Une tuile s'organise en deux types de zones: les *prairies* et les *cités*. De plus, une tuile peut être traversée par des *routes* aux extrémités centrales. Ces routes se terminent aux cités ou à des points de jonction nommés *villages*. L'article anglophone Wikipédia illustre l'ensemble des tuiles du jeu. À titre d'exemple, considérons cette carte constituée de quatre tuiles:



La carte contient

— une route complète de taille 2 entre une cité et un village (A2 – B2);

— une route incomplète de taille 1 à partir d'un village (B2);

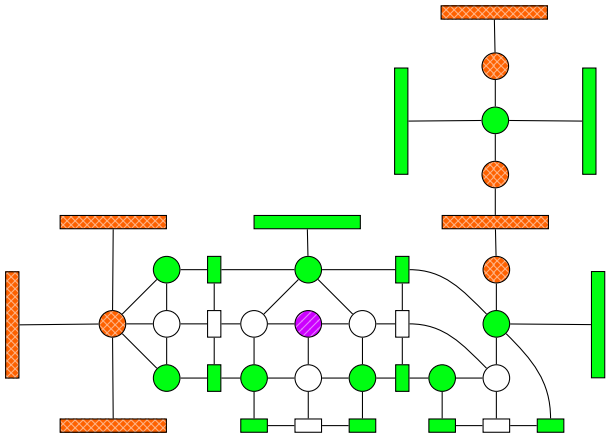
— une route incomplète de taille 2 à partir d'un village (B2 – C2);

— une cité incomplète de taille 1 (A2);

— une cité complète de taille 2 (C1 – C2);

— une cité incomplète de taille 1 (C1).

La carte se modélise à l'aide d'un graphe non dirigé où l'on représente les frontières d'une tuile par des sommets rectangulaires, les attributs d'une tuile par des sommets circulaires, et leurs liens par des arêtes:

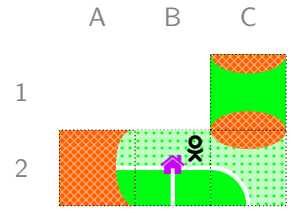


En supposant qu'une carte est représentée sous forme d'un tel graphe:

a) Donnez un algorithme qui identifie l'ensemble des routes et des cités, ainsi que leur taille respective et leur état (complète ou non);

b) Donnez un algorithme qui, étant donné un sommet x de type prairie, retourne la quantité de cités complètes adjacentes à la prairie à laquelle x appartient.

   Par exemple, la prairie qui contient $x =$ 웃 sur la carte ci-dessous est adjacente à une seule cité complète (ainsi qu'une cité incomplète):



c) Donnez un algorithme qui, étant donné un sommet x de type prairie, retourne la quantité de cités complètes entièrement entourées par la prairie à laquelle x appartient.

d) ★ Consultez les règlements complets du jeu et donnez un algorithme qui calcule le score de chaque personne, étant donné une carte et la disposition des « meeples ».

e) ★ Consultez l'ensemble des tuiles du jeu (en ignorant les abbayes et les blasons). Expliquez pourquoi chaque tuile peut être représentée par une séquence de quatre caractères parmi $\{\texttt{C}, \texttt{P}, \texttt{V}, \texttt{R}\}$. Expliquez comment convertir une telle séquence vers un graphe.

   (exercice inspiré de multiples défaites contre Raphaëlle H. Corbeil, E23)

3.11) Un *chemin hamiltonien* est un chemin qui passe par chaque sommet d'un graphe *exactement une* fois. Donnez un algorithme qui, étant donné un graphe dirigé acyclique, détermine s'il existe un chemin hamiltonien. Pensez au nombre d'ordres topologiques.

Paradigmes

4

Algorithmes gloutons

Les algorithmes gloutons sont des procédures qui, de manière générale, tentent de construire une solution (optimale) en considérant progressivement des candidats et en ne reconsidérant jamais leurs choix. Afin d'introduire ce paradigme, nous considérons le problème de calcul d'arbre couvrant minimal et présentons deux algorithmes gloutons pour résoudre ce problème.

4.1 Arbres couvrants minimaux

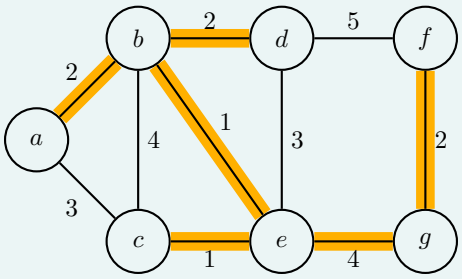
Un *graphe pondéré* est un graphe $\mathcal{G} = (V, E)$ dont chaque arête $e \in E$ est étiquetée par un poids $p[e]$, souvent un nombre entier, naturel ou rationnel. Le *poids* d'un graphe correspond à

$$p(\mathcal{G}) \coloneqq \sum_{e \in E} p[e].$$

Un *arbre couvrant (de poids) minimal* d'un graphe pondéré $\mathcal{G}$ est un arbre couvrant de $\mathcal{G}$ dont le poids est minimal parmi tous les arbres couvrants de $\mathcal{G}$.

Exemple.

Le graphe ci-dessous contient un arbre couvrant de poids 12, ce qui correspond au poids minimal parmi tous ses arbres couvrants.



4.1.1 Algorithme de Prim–Jarník

L'algorithme de Prim–Jarník suit l'approche suivante:

— on débute par $E' := \emptyset$ et $V' := \{v\}$ où v est un sommet arbitraire;

— on choisit une arête e de plus petit poids qui possède *exactement un* sommet appartenant à V';

— on ajoute e à E' et son nouveau sommet à V';

— on répète jusqu'à ce que V' contienne tous les sommets.

Autrement dit, l'algorithme débute par un arbre constitué d'un seul sommet et le fait croître d'un sommet à la fois en choisissant toujours la plus petite arête qui ne crée pas de cycle. La figure 4.1 illustre l'exécution de l'algorithme.

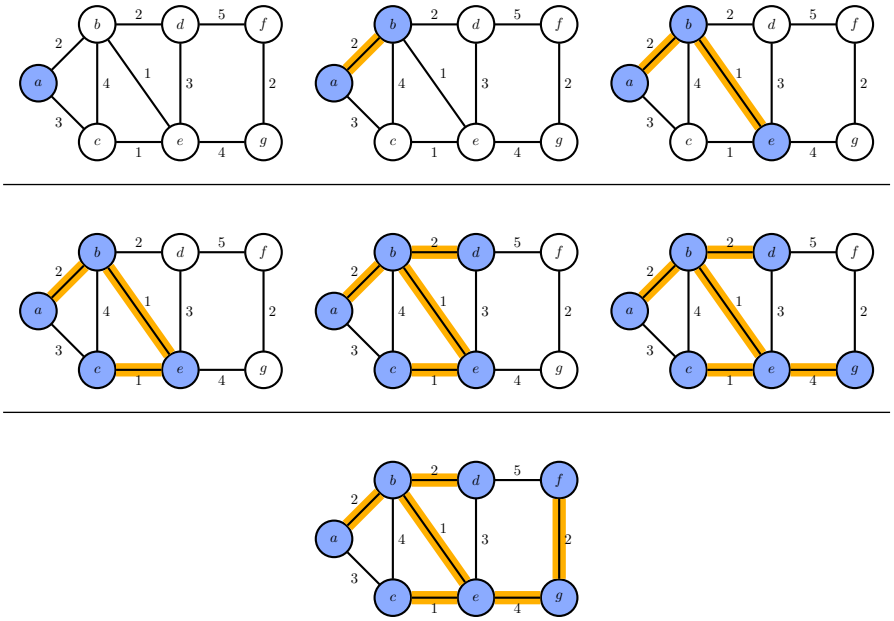


FIGURE 4.1 – Arbre couvrant minimal obtenu par l'algorithme de Prim–Jarník.

Remarquons que l'algorithme doit déterminer à répétition la plus petite arête disponible. Afin d'implémenter efficacement cette opération, nous ajoutons les arêtes découvertes dans un monceau ordonné par leur poids. Cette approche est décrite sous forme de pseudocode à l'algorithme 22.

Analysons l'algorithme de Prim–Jarník, en supposant l'utilisation d'un monceau binaire. L'initialisation du monceau se fait en temps $\mathcal{O}(|E|)$. Puisque chaque arête ne peut être considérée que deux fois (via chacun de ses sommets), le temps total des retraits de candidats s'effectue en temps $\mathcal{O}(|E| \log |E|)$. Remarquons que le corps du bloc **si** est exécuté au plus une fois par sommet. Le temps

Algorithme 22 : Algorithme de Prim–Jarník.

Entrées : graphe non dirigé connexe $\mathcal{G} = (V, E)$ pondéré par p
Résultat : un arbre couvrant minimal de $\mathcal{G}$

$E' \leftarrow \emptyset$
marquer un sommet $v \in V$
transformer $candidats \leftarrow [e \in E : v \in e]$ en monceau ordonné par p
tant que $candidats$ *est non vide*
  retirer e **des** $candidats$
  si *exactement un sommet de* e *est marqué* **alors**
    ajouter e **à** E'
    $x \leftarrow$ sommet non marqué de e
    marquer x
    pour *tout voisin* y *de* x
      ajouter $\{x, y\}$ **aux** $candidats$
retourner (V, E')

total de l'exécution du bloc **si** est donc de:

$$\begin{aligned} f(|V|, |E|) &\in \mathcal{O}\left(\overbrace{|V| \cdot \log|E|}^{\text{ajouts}} + \overbrace{|V|}^{\text{marquage}} + \overbrace{\sum_{x \in V} \deg(x) \cdot \log|E|}^{\text{ajout des voisins}}\right) \\ &= \mathcal{O}\left(|V| \cdot \log|E| + \log|E| \cdot \sum_{x \in V} \deg(x)\right) \\ &= \mathcal{O}(|V| \cdot \log|E| + \log|E| \cdot 2|E|) \\ &\subseteq \mathcal{O}(|E| \log|E|), \end{aligned}$$

où nous avons utilisé $|V| \in \mathcal{O}(|E|)$ car $|E| \geq |V| - 1$ par connexité de $\mathcal{G}$.

En sommant les trois analyses, nous obtenons un temps total de l'algorithme appartenant à:

$$\mathcal{O}(|E| + |E| \log|E| + f) \subseteq \mathcal{O}(|E| \log|E|).$$

Puisque $\log|E| \leq \log(|V|^2) = 2\log|V| \in \mathcal{O}(\log|V|)$, nous concluons donc que l'algorithme fonctionne en temps $\mathcal{O}(|E| \log|V|)$.

Démontrons maintenant que l'algorithme de Prim–Jarník calcule bel et bien un arbre couvrant minimal.

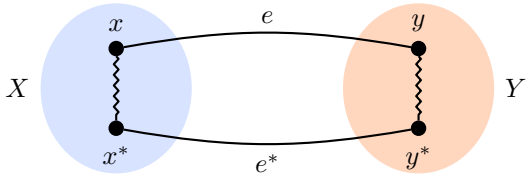
Théorème 4. *L'algorithme de Prim–Jarník est correct.*

Démonstration. Soit $\mathcal{G} = (V, E)$ un graphe non dirigé connexe pondéré par p. Soit $E' \subseteq E$ l'ensemble d'arêtes retourné par l'algorithme de Prim–Jarník sur

entrée $(\mathcal{G}, p)$. Il est assez simple de démontrer que E' forme un arbre couvrant $\mathcal{T} \coloneqq (V, E')$. Montrons qu'il s'agit d'un arbre couvrant *minimal*.

Soit $E^* \subseteq E$ un ensemble d'arêtes tel que $\mathcal{T}^* \coloneqq (V, E^*)$ est un arbre couvrant minimal. Si $E^* = E'$, alors nous avons terminé. Supposons que $E^* \neq E'$. Remarquons que $E' \setminus E^* \neq \emptyset$ puisque E' et E sont de même taille mais diffèrent. Soit $e = \{x, y\} \in E' \setminus E^*$. Lorsque l'algorithme a sélectionné e, il s'agissait d'une arête *minimale* parmi celles avec un sommet dans l'ensemble X des sommets marqués et un sommet dans l'ensemble $Y \coloneqq V \setminus X$ des sommets non marqués.

Puisque $\mathcal{T}^*$ est un arbre couvrant, il contient un chemin entre x et y. Puisque $e \notin E^*$, ce chemin contient une arête $e^* = \{x^*, y^*\} \in E^* \setminus E'$ qui traverse X et Y, c.-à-d. telle que $x^* \in X$ et $y^* \in Y$. Schématiquement, nous avons:



Posons $\mathcal{T}^{**} \coloneqq (V, E^* \setminus \{e^*\} \cup \{e\})$. Autrement dit, retirons e^* de $\mathcal{T}^*$ et ajoutons-lui e. Puisque $\mathcal{T}^{**}$ est connexe et a $|V| - 1$ arêtes, il s'agit d'un arbre couvrant. Par minimalité de $\mathcal{T}^*$, nous avons $p(\mathcal{T}^{**}) \geq p(\mathcal{T}^*)$. De plus, par minimalité de e, nous avons $p[e^*] \geq p[e]$. Ainsi,

$$p(\mathcal{T}^{**}) = p(\mathcal{T}^*) - p[e^*] + p[e] \leq p(\mathcal{T}^*).$$

Par conséquent, $\mathcal{T}^{**}$ est minimal. De plus, $\mathcal{T}^{**}$ possède une arête de plus en commun avec $\mathcal{T}$. Ainsi, en répétant ce processus jusqu'à avoir remplacé toutes les arêtes qui diffèrent, nous en concluons que $\mathcal{T}$ est minimal. □

Une conséquence de cette preuve est que l'algorithme de Prim–Jarník peut identifier tous les arbres couvrants minimaux en variant les bris d'égalité.

Corollaire 2. *L'algorithme 22 peut identifier* tous *les arbres couvrants minimaux.*

4.1.2 Algorithme de Kruskal

L'algorithme de Kruskal suit l'approche suivante:

- on débute avec $E' \coloneqq \emptyset$ et on considère chaque sommet comme un arbre;
- on choisit une arête e de plus petit poids qui connecte deux arbres;
- on ajoute e à E';
- on répète jusqu'à ce qu'il ne reste qu'un seul arbre.

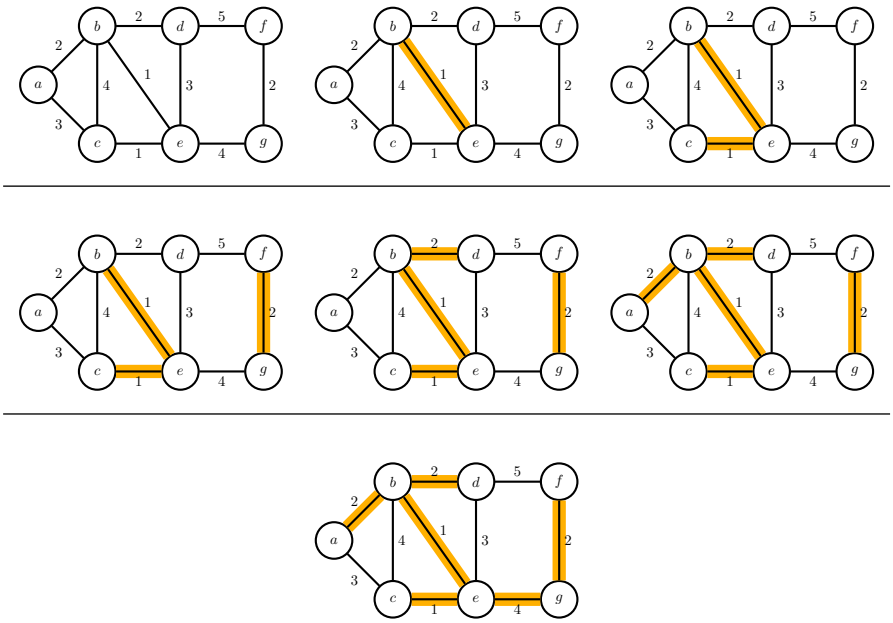


FIGURE 4.2 – Arbre couvrant minimal obtenu par l'algorithme de Kruskal.

Autrement dit, l'algorithme débute par une forêt de $|V|$ arbres et réduit progressivement le nombre d'arbres en choisissant la plus petite arête disponible.

La figure 4.2 illustre l'exécution de l'algorithme de Kruskal, et l'algorithme 23 décrit l'algorithme sous forme de pseudocode. On peut démontrer que l'algorithme de Kruskal est correct en adaptant l'argument établi pour l'algorithme de Prim–Jarník.

Théorème 5. *L'algorithme de Kruskal est correct. De plus, il peut identifier* tous *les arbres couvrants minimaux.*

Algorithme 23 : Algorithme de Kruskal.

Entrées : graphe non dirigé connexe $\mathcal{G} = (V, E)$ pondéré par p
Résultat : un arbre couvrant minimal de $\mathcal{G}$

$E' \leftarrow \emptyset$

considérer chaque sommet $v \in V$ comme un arbre

pour $\{u, v\} \in E$ *en ordre croissant selon* p
  si u *et* v *n'appartiennent pas au même arbre* **alors**
    fusionner les arbres contenant u et v
    ajouter $\{u, v\}$ **à** E'

retourner (V, E')

Ensembles disjoints. Une implémentation efficace de l'algorithme de Kruskal requiert un mécanisme afin d'identifier si une arête relie deux arbres distincts ou non. Autrement dit, nous devons être en mesure de répondre rapidement à des questions de la forme « est-ce que u et v appartiennent au même arbre? »

Pour ce faire, nous représentons chaque arbre par l'ensemble des sommets qui le constituent. Par exemple, l'exécution illustrée à la figure 4.2 fait évoluer ces ensembles ainsi:

$$
\begin{array}{llllll}
\{a\} & \{b\} & \{c\} & \{d\} & \{e\} & \{f\}\ \{g\} \\
\{a\} & \{b,e\} & \{c\} & \{d\} & \{f\} & \{g\} \\
\{a\} & \{b,c,e\} & \{d\} & \{f\} & \{g\} & \\
\{a\} & \{b,c,e\} & \{d\} & \{f,g\} & & \\
\{a\} & \{b,c,e,d\} & \{f,g\} & & & \\
\{a,b,c,e,d\} & \{f,g\} & & & & \\
\{a,b,c,e,d,f,g\} & & & & &
\end{array}
$$

À tout moment, ces ensembles sont disjoints puisqu'un sommet ne peut pas appartenir simultanément à deux arbres. Nous décrivons donc une structure de données qui permet de manipuler une collection d'*ensembles disjoints*.

Nous représentons chaque ensemble par une arborescence. Par exemple, la figure 4.3 illustre une représentation possible de la partition $\{a\}, \{b,c,e,d\}, \{f,g\}$.

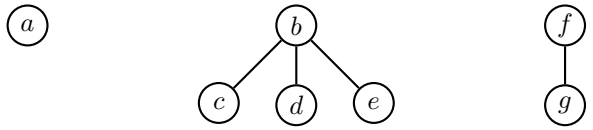


FIGURE 4.3 – Représentation de $\{a\}, \{b,c,e,d\}, \{f,g\}$ sous arborescences.

Notre structure de données implémentera ces opérations:

— `init`(X): crée la collection $\{\{x\} : x \in X\}$;

— `trouver`(x): retourne un représentant de l'ensemble auquel x appartient;

— `union`(x,y): fusionne les ensembles auxquels x et y appartiennent.

Initialement, chaque élément $x \in X$ appartient à une arborescence dont x est l'unique sommet. Pour chaque élément $x \in X$, nous stockons $parent[x]$, qui correspond au parent de x dans son arborescence, et $hauteur[x]$, qui correspond à la hauteur de cette arborescence. Si x forme la racine de son arborescence, alors nous stockons $parent[x] = x$ afin de le représenter.

La recherche d'un élément x remonte son arborescence, puis retourne l'élément à la racine comme représentant de l'ensemble. L'union de deux sommets x et y considère l'arborescence qui contient x et celle qui contient y, puis adjoint l'arbre de plus petite hauteur à l'autre arbre. Une telle implémentation est décrite sous forme de pseudocode à l'algorithme 24.

Algorithme 24 : Implémentation d'ensembles disjoints.

```
init(X):
    // Représenter chaque élément par une arborescence
    parent  ← [x ↦ x : x ∈ X]
    hauteur ← [x ↦ 0 : x ∈ X]

trouver(x):
    // Remonter l'arborescence jusqu'à sa racine
    tant que x ≠ parent[x]
        x ← parent[x]
    retourner x

union(x, y):
    x ← trouver(x)
    y ← trouver(y)
    // Adjoindre l'arborescence de hauteur min. à l'autre
    si hauteur[x] > hauteur[y] alors
        parent[y] ← x
    sinon si hauteur[y] > hauteur[x] alors
        parent[x] ← y
    sinon
        parent[y] ← x
        hauteur[x] ← hauteur[x] + 1
```

Puisque l'union minimise la hauteur des arborescences, celles-ci demeurent toujours de hauteur au plus logarithmique par rapport au nombre de sommets qu'elles contiennent [1]. Ainsi, nous obtenons ces complexités dans le pire cas:

Opération	`init`(X)	`trouver`(x)	`union`(x, y)
Complexité	$\Theta(\|X\|)$	$\mathcal{O}(\log \|X\|)$	$\mathcal{O}(\log \|X\|)$

Analyse de l'algorithme de Kruskal. Analysons l'algorithme de Kruskal implémenté à l'aide d'ensembles disjoints, tel que décrit à l'algorithme 25.

L'initialisation de E et D se fait respectivement en temps $\Theta(1)$ et $\Theta(|V|)$. Le tri implicite dans la boucle principale requiert un temps de $\mathcal{O}(|E| \log |E|)$. La boucle principale est exécutée précisément $|E|$ fois. Les opérations `union` et `trouver` se font en temps $\mathcal{O}(\log |V|)$, et l'ajout à E' en temps constant. Ainsi, nous obtenons une complexité totale de:

$$\mathcal{O}(1 + |V| + |E| \log |E| + |E| \log |V|) = \mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|).$$

1. On peut démontrer que chaque arborescence de $k \in \mathbb{N}_{\geq 1}$ sommets possède une hauteur d'au plus $\lfloor \log k \rfloor$ (possible de s'inspirer de l'exercice 0.21)).

Algorithme 25 : Algorithme de Kruskal avec ensembles disjoints.

Entrées : graphe non dirigé connexe $\mathcal{G} = (V, E)$ pondéré par p
Résultat : un arbre couvrant minimal de $\mathcal{G}$

$E' \leftarrow \emptyset$
$D \leftarrow$ `init(`V`)`
pour $\{u, v\} \in E$ *en ordre croissant selon* p
  si D.`trouver(`u`)` $\neq D$.`trouver(`v`)` **alors**
    D.`union(`u`, `v`)`
    ajouter $\{u, v\}$ **à** E'
retourner (V, E')

Amélioration. L'algorithme de Kruskal peut être rendu plus efficace en améliorant l'implémentation d'ensembles disjoints. L'idée consiste à modifier la fonction de recherche afin de « compresser » le chemin exploré. Après avoir remonté d'un élément x vers sa racine r, on réexplore le chemin à nouveau en rattachant tous les sommets du chemin directement à r. Cela produit des arborescences de hauteurs quasi constantes. Cette approche est implémentée à l'algorithme 26.

Cette légère modification réduit la complexité de l'algorithme de Kruskal à $\mathcal{O}(\alpha(|V|) \cdot |E|)$, où α est l'inverse de la fonction d'Ackermann. Bien que $\alpha(|V|)$ ne soit pas une constante, la fonction α croît si lentement que sa valeur demeure inférieure à 5 pour toute entrée envisageable en pratique.

Algorithme 26 : Recherche avec compression de chemin.

`trouver(`x`):`
  $r \leftarrow x$
  `// Remonter l'arborescence jusqu'à sa racine`
  tant que $r \neq parent[r]$
    $r \leftarrow parent[r]$
  `// Compression du chemin de` x `vers` r
  tant que $x \neq r$
    $x, parent[x] \leftarrow parent[x], r$
  retourner r

4.2 Approche générique

Les algorithmes de Prim–Jarník et de Kruskal partagent un certain nombre de caractéristiques communes. Ils considèrent chaque arête *une seule* fois, ils vérifient si l'ajout d'une arête est admissible, et ils l'ajoutent le cas échéant. Ce type d'algorithme est dit *glouton* (ou *vorace*) puisqu'on construit une solution partielle en ajoutant le plus de candidats possibles jusqu'à l'obtention d'une so-

lution complète, et ce sans jamais reconsidérer nos choix. Un patron générique d'algorithme glouton est décrit à l'algorithme 27.

Les algorithmes de Prim–Jarník et de Kruskal implémentent ce patron ainsi:

Composante	Prim–Jarník	Kruskal
« $candidats$ »	ensemble des arêtes	ensemble des arêtes
« **sélectionner** c »	plus petite arête e avec un sommet non exploré	plus petite arête e non considérée
« `admissible(`S`,` c`)` »	l'ajout de e à E' ne crée pas de cycle?	l'ajout de e à E' connecte deux arbres?
« S est une solution? »	l'ensemble E' touche à tous les sommets?	l'ensemble E' ne forme qu'un seul arbre?

Algorithme 27 : Patron générique d'algorithme glouton.

Entrées : Représentation d'un ensemble de candidats
Résultat : Une solution formée de candidats

$S \leftarrow \emptyset$

tant que *l'ens. des candidats est non vide et* S *n'est pas une solution*
  sélectionner et retirer c **des** candidats
  si `admissible(`S`,` c`)` **alors**
    ajouter c à S

si S *est une solution* **alors retourner** S
sinon retourner *impossible*

4.3 Problème du sac à dos

Soit $c \in \mathbb{N}_{>0}$, soit v une séquence de n éléments appartenant à $\mathbb{N}_{>0}$, et soit p une séquence de n éléments appartenant à $\{1, 2, \ldots, c\}$. Nous appelons c la *capacité*, v la séquence de *valeurs* et p la séquence de *poids*. Informellement, le problème du sac à dos consiste à mettre des objets de poids p, dans un sac à dos qui peut contenir un poids maximal de c, de telle sorte que la valeur des objets choisis est maximisée. Plus formellement, le *problème du sac dos* consiste à maximiser

$$\text{val}(x) := \sum_{i=1}^{n} x[i] \cdot v[i]$$

sous la contrainte $c \geq \sum_{i=1}^{n} x[i] \cdot p[i]$, où $x \in \{0, 1\}^n$ indique les objets choisis.

Exemple.

Considérons les objets suivants et un sac de capacité $c = 900$:

Objets	1	2	3	4	5	6
Valeurs v	50	5	65	10	12	20
Poids p	700	320	845	70	420	180

En choisissant les objets 4, 5 et 6, nous obtenons une valeur de 42 et un poids de 670. En choisissant les objets 1 et 4, nous obtenons une valeur de 60 et un poids de 770. En choisissant les objets 1 et 6, nous obtenons une valeur de 70 et un poids de 880. Cette dernière solution est optimale. En effet, en explorant toutes les combinaisons, nous remarquerions qu'il est impossible d'obtenir une valeur supérieure à 70 sans excéder la capacité.

4.3.1 Approche gloutonne

Une approche simple et naturelle afin de résoudre le problème du sac à dos consiste à considérer le ratio entre la valeur et le poids de chaque objet. Dans l'exemple précédent, le quatrième objet possède un ratio de $1/7$, ce qui correspond à la valeur qu'offre l'objet pour chaque unité de poids. Une procédure gloutonne trie simplement les objets en ordre décroissant de ce ratio et ajoute les objets au sac tant que sa capacité n'est pas excédée.

Exemple.

Reconsidérons les objets précédents en les ordonnant par leur ratio:

Objets	4	6	3	1	5	2
Valeurs v	10	20	65	50	12	5
Poids p	70	180	845	700	420	320
Ratio	1/7	1/9	1/13	1/14	1/35	1/64

En prenant les objets 4 et 6, nous atteignons une valeur de 30 et un poids de 250. Il est impossible d'ajouter l'objet suivant, c'est-à-dire l'objet 3, puisque le poids du sac excéderait sa capacité de $c = 900$.

L'exemple ci-dessus montre que l'approche gloutonne ne fonctionne pas toujours puisque la solution obtenue n'est pas maximale. En fait:

Proposition 21. *La solution retournée par la procédure gloutonne pour le problème du sac à dos peut être arbitrairement mauvaise.*

Démonstration. Soit $c \geq 3$. Considérons l'entrée $v = [2, c]$ et $p = [1, c]$. La procédure gloutonne calcule les ratios $[2/1, c/c] = [2, 1]$. Ainsi, elle choisit le premier objet, ce qui donne une valeur de 2 et un poids de 1. Il est impossible d'ajouter le deuxième objet puisque cela mènerait à un poids de $c+1 > c$. Cette solution n'est pas maximale puisque choisir le deuxième objet donne une valeur et un poids de c. Remarquons que plus c est grand, plus l'écart se creuse entre la solution retournée et la solution maximale. □

4.3.2 Variante fractionnelle

L'approche gloutonne permet néanmoins de résoudre une variante du problème du sac à dos où on peut choisir une *fraction* d'un objet. Plus formellement, dans la variante fractionnelle, on cherche à maximiser

$$\text{val}(x) = \sum_{i=1}^{n} x[i] \cdot v[i]$$

sous la contrainte $c \geq \sum_{i=1}^{n} x[i] \cdot p[i]$, où $x \in [0, 1]^n$ indique quelle fraction de chaque objet est choisie. Remarquons que la seule différence entre les deux problèmes est le passage de l'intervalle discret $\{0, 1\}$ à l'intervalle continu $[0, 1]$. Dans le contexte des objets d'un sac, cette variante fait du sens si on peut les découper, par ex. comme la pomme des exemples précédents.

Exemple.

Reconsidérons à nouveau le même exemple:

Objets	4	6	3	1	5	2
Valeurs v	10	20	65	50	12	5
Poids p	70	180	845	700	420	320
Ratio	1/7	1/9	1/13	1/14	1/35	1/64

En prenant entièrement les objets 4 et 6, nous atteignons une valeur de 30 et un poids de 250. En prenant $10/13$ de l'objet 3, nous obtenons une valeur de $30+(10/13)\cdot 65 = 80$ avec un poids de $250+(10/13)\cdot 845 = 900$.

L'exemple précédent suggère donc une légère modification de la procédure gloutonne: on prend entièrement les objets en ordre décroissant de ratio, puis on ajoute la plus grande fraction du prochain objet qui n'entre pas dans le sac. Cette approche est décrite sous forme de pseudocode à l'algorithme 28. Contrairement à sa variante discrète, cet algorithme retourne toujours une solution maximale pour le problème du sac à dos fractionnel:

Proposition 22. *L'algorithme 28 résout correctement la variante fractionnelle du problème du sac à dos.*

Algorithme 28 : Algorithme pour le problème du sac à dos fractionnel.

Entrées : capacité $c \in \mathbb{N}_{>0}$, valeurs $v \in \mathbb{N}_{>0}^n$ et poids $p = \{1, 2, \ldots, c\}^n$
Résultat : solution (maximale) au problème du sac à dos fractionnel

$indices \leftarrow [1, 2, \ldots, n]$
trier $indices$ en ordre décroissant selon $v[i]$ / $p[i]$
$valeur \leftarrow 0$
$poids \leftarrow 0$
$x \leftarrow [0, 0, \ldots, 0]$
pour $i \in indices$
  si $poids + p[i] \leq c$ **alors**
    $valeur \leftarrow valeur + v[i]$
    $poids \leftarrow poids + p[i]$
    $x[i] \leftarrow 1$
  sinon
    $\lambda \leftarrow (c - poids)$ / $p[i]$
    $valeur \leftarrow valeur + \lambda \cdot v[i]$
    $poids \leftarrow poids + \lambda \cdot p[i]$
    $x[i] \leftarrow \lambda$
retourner x

4.3.3 Approximation

Bien que l'approche gloutonne ne fonctionne pas pour le problème du sac à dos (discret), il est possible d'approximer une solution optimale en s'inspirant de la variante fractionnelle du problème:

— on calcule une solution x selon l'approche gloutonne;

— on choisit la meilleure solution entre x et la solution y constituée uniquement du prochain objet qui n'entre pas dans le sac (s'il en reste un).

Exemple.

Reconsidérons les objets précédents en les ordonnant par leur ratio:

Objets	4	6	3	1	5	2
Valeurs v	10	20	65	50	12	5
Poids p	70	180	845	700	420	320
Ratio	1/7	1/9	1/13	1/14	1/35	1/64

L'approche gloutonne choisit les objets 4 et 6, et obtient une valeur de 30 et un poids de 250. L'objet suivant, c'est-à-dire l'objet 3, possède une valeur de $65 > 30$. On choisit donc simplement l'objet 3.

Cette procédure, décrite à l'algorithme 29, ne retourne pas nécessairement une solution maximale, mais l'approxime toujours:

Proposition 23. *La solution retournée par l'algorithme 29 possède une valeur d'au moins $\frac{1}{2}$ de la valeur optimale.*

Démonstration. Si la solution x obtenue par l'algorithme contient tous les objets, alors x est forcément optimale. Nous considérons donc une entrée où x ne contient pas tous les objets. Soient v^* et v^*_{frac} respectivement les valeurs maximales de la variante discrète et fractionnelle du problème du sac à dos sur l'entrée de l'algorithme. À la sortie de la boucle principale, nous avons:

$$\begin{aligned} valeur + v[indices[i]] &\geq v^*_{\text{frac}} && \text{(car } v^*_{\text{frac}} = valeur + \lambda v[indices[i]] \text{ où } \lambda \in [0,1]\text{)} \\ &\geq v^* && \text{(on ne peut pas faire pire dans la variante frac.)} \end{aligned}$$

Ainsi, $\max(valeur, v[indices[i]]) \geq v^*/2$, ce qui implique que la solution retournée par l'algorithme possède une valeur d'au moins $\frac{1}{2} \cdot v^*$. □

Algorithme 29 : Approximation pour le problème du sac à dos.

Entrées : valeurs $v \in \mathbb{N}_{>0}^n$, poids $\mathbb{N}_{>0}^n$ et capacité $c \in \mathbb{N}_{>0}$
Résultat : solution (maximale) au problème du sac à dos fractionnel

$indices \leftarrow [1, 2, \ldots, n]$
trier $indices$ en ordre décroissant selon $v[i]$ / $p[i]$

$valeur \leftarrow 0$
$poids \leftarrow 0$
$x \leftarrow [0, 0, \ldots, 0]$
$i \leftarrow 1$

tant que $(i \leq n) \wedge (poids + p[indices[i]] \leq c)$
    $j \leftarrow indices[i]$
    $valeur \leftarrow valeur + v[j]$
    $poids \leftarrow poids + p[j]$
    $x[j] \leftarrow 1$
    $i \leftarrow i + 1$

`// Retourner meilleure solution entre` x `et prochain objet`
si $(i > n) \vee (valeur > v[indices[i]])$ **alors** `// (s'il en reste un)`
    retourner x
sinon
    $y \leftarrow [0, 0, \ldots, 0]$
    $y[indices[i]] \leftarrow 1$
    retourner y

Remarque.

Pour tout $\varepsilon \in (0, 1]$, il existe un algorithme qui approxime une solution maximale du problème du sac à dos à un facteur d'au moins $1 - \varepsilon$. Autrement dit, il est possible d'approximer le problème du sac à dos de façon arbitrairement précise. De plus, cette approximation se calcule en temps polynomial.

4.4 Exercices

4.1) Expliquez comment calculer un arbre couvrant de poids minimal en présence de poids négatifs (et positifs).

4.2) Expliquez comment calculer un arbre couvrant de poids *maximal*.

4.3) Supposons que le poids d'un graphe soit défini par le *produit* de ses poids plutôt que la somme. Expliquez comment calculer un arbre de poids minimal sous cette nouvelle définition.

4.4) Lorsqu'un graphe non dirigé $\mathcal{G}$ n'est pas connexe, il est impossible d'obtenir un arbre couvrant de $\mathcal{G}$. Toutefois, nous pouvons relaxer cette notion et considérer une *forêt couvrante*, c'est-à-dire un sous-graphe de $\mathcal{G}$ qui est une forêt telle que chacun de ses arbres est un arbre couvrant d'une composante connexe de $\mathcal{G}$. Une forêt couvrante correspond à un arbre couvrant standard pour les graphes connexes. Les algorithmes de Prim–Jarník et de Kruskal permettent-ils de calculer une forêt couvrante de poids minimal? Si ce n'est pas le cas, est-il possible de les adapter?

4.5) Donnez un algorithme simple qui calcule les composantes connexes d'un graphe non dirigé grâce à une structure d'ensembles disjoints.

4.6) Considérons le problème de remplissage de bacs suivant. Nous avons:

- un nombre arbitraire de bacs, chacun de capacité $c \in \mathbb{N}_{\geq 1}$;
- n objets à ranger dans des bacs, où l'objet i est de taille $p[i] \in [1..c]$.

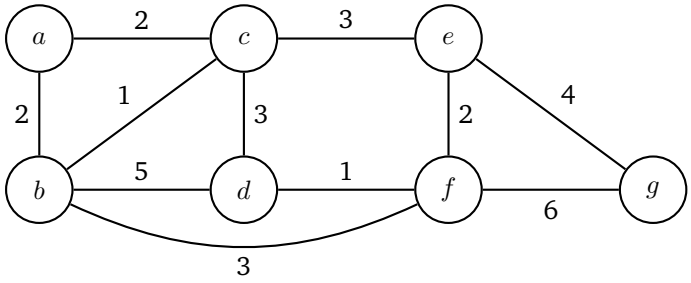
Nous devons *minimiser* le nombre de bacs afin de ranger tous les objets. Montrez que cette approche gloutonne ne résout pas le problème:

- trier les objets en ordre décroissant;
- assigner chaque objet au premier bac qui peut le contenir sans excéder la capacité, ou en utiliser un nouveau le cas échéant.

Remarque.

La procédure ci-dessus approxime la valeur optimale. Plus précisément, si l'algorithme retourne la valeur m, et que la solution minimale est m^*, alors on a la garantie que $m \leq 1{,}\overline{2} \cdot m^* + 0{,}\overline{6}$.

4.7) Identifiez *deux* arbres couvrants minimaux du graphe pondéré suivant et donnez leur poids:



4.8) Soit $\mathcal{G} = (V, E)$ un graphe non dirigé connexe et pondéré par une séquence p. Soit $f \in E$ une arête de poids minimal, c.-à-d. une arête telle que $p[f] = \min\{p[e] : e \in E\}$. Est-ce qu'il *existe forcément* un arbre couvrant minimal de $\mathcal{G}$ qui contient f? Justifiez.

4.9) ★ L'approche gloutonne pour le problème du sac à dos fonctionne en temps $\mathcal{O}(n \log n)$ dû au tri. Donnez une implémentation de cette approche qui fonctionne en temps $\mathcal{O}(n)$. Vous pouvez supposer l'existence d'une procédure qui retourne la médiane d'une séquence en temps linéaire.
(*Indice: pensez récursivement et exploitez* $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$)

4.10) ★★ Un *monoïde abélien totalement ordonné* est un tuple $(\mathbb{D}, 0_{\mathbb{D}}, \oplus, \preceq)$ tel que:

- $\preceq$ est un ordre total sur l'ensemble $\mathbb{D}$;
- $\oplus\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ est une opération qui satisfait ces propriétés:
  - $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ (associativité);
  - $x \oplus y = y \oplus x$ (commutativité);
  - $x \oplus 0_{\mathbb{D}} = x = 0_{\mathbb{D}} \oplus x$ (neutre);
- $x \preceq y$ implique $x \oplus z \preceq y \oplus z$ (compatibilité).

Les arêtes d'un graphe peuvent être pondérées par des valeurs de $\mathbb{D}$, c.-à-d. que $p[e] \in \mathbb{D}$ pour chaque arête e. Dans ce contexte:

- le poids d'un arbre trivial (sans arête) est $0_{\mathbb{D}}$;
- le poids d'un arbre couvrant constitué des arêtes $\{e_1, \ldots, e_n\}$ est $p[e_1] \oplus \ldots \oplus p[e_n]$;
- un arbre couvrant est *minimal* si son poids est minimal sous $\preceq$.

(a) Montrez que l'algorithme de Prim–Jarník est correct, même dans le contexte d'un monoïde abélien totalement ordonné.

(b) Remarquez que ces tuples sont tous des monoïdes abéliens totalement ordonnés:

- $(\mathbb{N}, 0, +, \leq)$
- $(\mathbb{Z}, 0, +, \leq)$

— $(\mathbb{N}, 1, \cdot, \leq)$

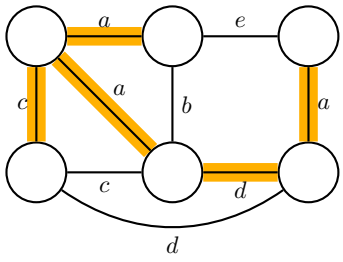
Ce n'est pas le cas de $(\mathbb{Z}, 1, \cdot, \leq)$. Pourquoi?

(c) Montrez que pour $(\mathbb{Z}, 1, \cdot, \leq)$ l'algorithme de Prim–Jarník n'est pas correct. Autrement dit, donnez un graphe sur lequel l'algorithme ne retourne pas un arbre couvrant minimal.

(d) Dans un graphe pondéré par des entiers positifs, un arbre couvrant est minimal sous l'addition ssi il est minimal sous la multiplication. Pourquoi?

(e) L'algorithme de Prim–Jarník peut être utilisé pour trouver des arbres couvrants maximaux, plutôt que minimaux. Pourquoi?

(f) Considérons des graphes non dirigés dont les arêtes sont étiquetées par les lettres $\{a, b, c, \ldots\}$. Dans ce contexte, le poids d'un arbre est le multi-ensemble (c.-à-d. un ensemble avec répétitions) des lettres qui apparaissent sur ses arêtes. Un arbre couvrant $\mathcal{T}$ est meilleur qu'un arbre couvrant $\mathcal{T}'$ si: $\mathcal{T}$ possède plus d'occurences de a que $\mathcal{T}'$; ou il en possède un nombre égal, et $\mathcal{T}$ possède plus d'occurences de b que $\mathcal{T}'$; ou etc. Par exemple, l'arbre couvrant ci-dessous donne lieu au multi-ensemble $\wr a, a, a, c, d \wr$, et est donc meilleur qu'un arbre couvrant dont le multi-ensemble est $\wr a, a, a, d, d \wr$.



Il est possible d'identifier un arbre couvrant qui soit le meilleur possible. Pourquoi?

(*Cet exercice émane de questions sur les exercices 4.1), 4.2), et 4.3) (A22).*)

5

Algorithmes récursifs et approche diviser-pour-régner

Ce chapitre traite de la conception et de l'analyse d'algorithmes récursifs, plus particulièrement de l'approche diviser-pour-régner. Celle-ci consiste essentiellement à résoudre un problème en le découpant en sous-problèmes plus simples qu'on résout récursivement afin d'obtenir une solution globale. Par exemple, le tri par fusion suit cette approche: afin de trier une séquence de $2k$ éléments, on trie deux sous-séquences de k éléments, puis on les fusionne.

Dû à leur nature récursive, la correction de ces algorithmes se démontre généralement par induction, et leur complexité s'analyse à l'aide de récurrences. Nous nous pencherons principalement sur l'analyse de leur complexité et sur certains problèmes où cette approche mène à des algorithmes efficaces.

5.1 Tours de Hanoï

Considérons le problème classique des *tours de Hanoï*:

- n disques de diamètre $n, \ldots, 2, 1$ sont empilés sur une première pile;
- deux autres piles sont vides;
- on peut déplacer le disque x du dessus d'une pile i vers le dessus d'une pile j si le disque y au-dessus de la pile j possède un diamètre supérieur au disque x;
- on doit déplacer l'entièreté de première pile sur la deuxième pile.

Par exemple, la figure 5.1 illustre une solution, pour le cas $n = 4$, qui effectue 15 déplacements de disques.

Ce problème peut être résolu récursivement en suivant cette approche:

- on cherche à déplacer le contenu d'une pile *source* vers une pile *destination* à l'aide d'une pile *temporaire*;
- si la source possède k disques, on déplace (récursivement) ses $k-1$ disques du dessus vers la pile temporaire;

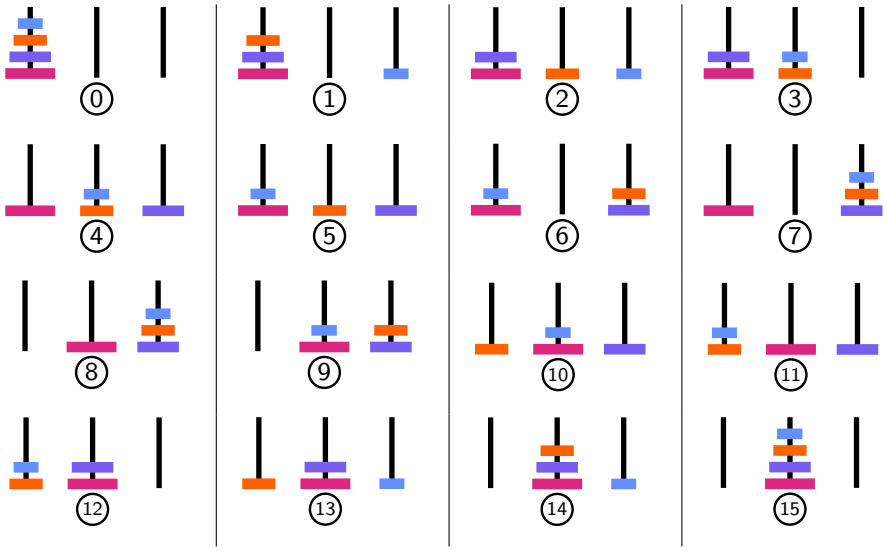


FIGURE 5.1 – Solution au problème des tours de Hanoï pour $n = 4$ disques.

— on déplace le disque restant de la source vers la destination;

— on déplace (récursivement) les $k - 1$ disques de la pile temporaire vers la destination.

Cette procédure est décrite sous forme de pseudocode à l'algorithme 30.

Algorithme 30 : Algorithme pour le problème des tours de Hanoï.

Entrées : $n \in \mathbb{N}$
Résultat : séquence de déplacements résolvant le problème des tours de Hanoï pour n disques

```
hanoi(n):
    deplacements ← []
    hanoi'(k, src, dst, tmp):
        si k > 0 alors
            // Déplacer k − 1 disques de pile src vers pile tmp
            hanoi'(k − 1, src, tmp, dst)
            // Déplacer un disque de pile src vers pile dst
            ajouter (src, dst) à deplacements
            // Déplacer k − 1 disques de pile tmp vers pile dst
            hanoi'(k − 1, tmp, dst, src)
    hanoi'(n, 1, 2, 3)   // Déplacer première pile vers deuxième
    retourner deplacements
```

Cherchons à déterminer la taille de la solution calculée par l'algorithme 30 en examinant la sous-procédure `hanoi'`. Lorsque $k = 0$, la solution est vide (cela correspond au fait qu'il n'y a aucun disque à déplacer). Lorsque $k > 0$, la solution possède un déplacement, plus les déplacements obtenus par les deux appels récursifs. Ainsi, en définissant $t(k)$ comme étant la taille de la solution, nous obtenons:

$$t(k) = \begin{cases} 0 & \text{si } k = 0, \\ 2 \cdot t(k-1) + 1 & \text{si } k > 0. \end{cases}$$

Nous appelons ce type de relation une *relation de récurrence* puisque t dépend d'elle-même. À priori, identifier la complexité asymptotique d'une telle relation s'avère ardu. Il existe plusieurs façons d'y arriver. L'une des plus élémentaires consiste à identifier une *forme close* en substitutant la récurrence à répétition:

$$\begin{aligned}
t(k) &= 2 \cdot t(k-1) + 1 \\
&= 2 \cdot (2 \cdot t(k-2) + 1) + 1 && \text{(par la relation de récurrence)} \\
&= 4 \cdot t(k-2) + 3 \\
&= 4 \cdot (2 \cdot t(k-3) + 1) + 3 && \text{(par la relation de récurrence)} \\
&= 8 \cdot t(k-3) + 7 \\
&\vdots \\
&= 2^i \cdot t(k-i) + (2^i - 1) && \text{(en répétant } i \text{ fois)} \\
&\vdots \\
&= 2^k \cdot t(0) + (2^k - 1) && \text{(en répétant } k \text{ fois)} \\
&= 2^k - 1 && \text{(car } t(0) = 0\text{)}.
\end{aligned}$$

Cette approche suggère donc que $t(k) = 2^k - 1$. Afin de s'en convaincre formellement, on pourrait prouver par induction que c'est bien le cas (ce l'est!)

Comme la taille de la solution est une borne inférieure sur le temps d'exécution de l'algorithme et que celui-ci débute par un appel à `hanoi'` avec $k = n$, nous en concluons qu'il fonctionne en temps $\Omega(2^n)$. Une analyse plus fine montrerait que son temps d'exécution appartient à $\Theta(2^n)$. En fait, il est impossible de résoudre le problème en moins de $2^n - 1$ déplacements, donc aucun algorithme ne peut faire mieux.

Remarque.

Une autre façon de se convaincre que $t(k) = 2^k - 1$ consiste à imaginer k sous représentation binaire et t comme une opération de décalage à gauche suivi d'une incrémentation. Puisqu'on applique cette opération k fois à partir de `0`, on obtient $\texttt{11}\cdots\texttt{1}_2 = 2^k - 1$.

5.2 Récurrences linéaires

Afin d'illustrer une récurrence plus complexe, considérons le problème qui consiste à identifier tous les pavages d'une grille de $3 \times n$ cases à l'aide de tuiles de cette forme (sans rotations):

L'algorithme 31 décrit une procédure récursive qui calcule tous les pavages en observant que tout pavage débute forcément par l'un de ces trois motifs:

Algorithme 31 : Algorithme récursif de pavage d'une grille $3 \times n$.

Entrées : $n \in \mathbb{N}_{\geq 1}$
Résultat : séquence de tous les pavages d'une grille $3 \times n$

`pavages(`n`):`
- **si** $n = 1$ **alors**
  - **retourner** $\left[\right]$
- **si** $n = 2$ **alors**
  - **retourner** $\left[\right]$
- **sinon**
  - $P \leftarrow$ `pavages(`$n-1$`)`
  - $Q \leftarrow$ `pavages(`$n-2$`)`
  - **retourner** $\left[\right] + p : p \in P\Big] + \Big[\right] + q : q \in Q\Big] + \Big[\right] + q : q \in Q\Big]$

5.2.1 Cas homogène

Afin d'estimer le temps d'exécution de l'algorithme 31, évaluons le nombre de pavages $t(n)$ d'une grille $3 \times n$. En examinant l'algorithme, nous remarquons que dans le cas général la séquence de retour possède une taille de $|P| + 2 \cdot |Q|$. Ainsi, nous obtenons cette récurrence:

$$t(n) = \begin{cases} 1 & \text{si } n = 1, \\ 3 & \text{si } n = 2, \\ t(n-1) + 2 \cdot t(n-2) & \text{sinon.} \end{cases}$$

La méthode de substitution mène plus difficilement à une forme close. Nous empruntons donc une autre approche. En déplaçant tous les termes à gauche de

l'égalité, la relation se réécrit sous la forme $t(n)-t(n-1)-2\cdot t(n-2)=0$. Nous disons que cette récurrence est *linéaire*, car son côté gauche est une combinaison linéaire, et *homogène*, car son côté droit vaut 0. Il existe une méthode mécanique afin de résoudre les récurrences linéaires homogènes. Nous la décrivons en trois étapes en l'illustrant sur notre problème de pavage. [1]

A) Polynôme caractéristique. Étant donné une récurrence linéaire homogène $a_0 \cdot t(n) + a_1 \cdot t(n-1) + \ldots + a_d \cdot t(n-d) = 0$, nous considérons son *polynôme caractéristique*:

$$a_0 \cdot x^d + a_1 \cdot x^{d-1} + \ldots + a_d \cdot x^0.$$

Dans notre problème de pavage, nous obtenons donc le polynôme:

$$p(x) = x^2 - x - 2 = (x-2)(x+1).$$

B) Forme close. Nous identifions ensuite les racines $\lambda_1, \lambda_2, \ldots, \lambda_d$ du polynôme caractéristique. Si toutes les racines sont distinctes [2], la récurrence possède cette forme close:

$$c_1 \cdot \lambda_1^n + c_2 \cdot \lambda_2^n + \ldots + c_d \cdot \lambda_d^n,$$

où $c_1, c_2, \ldots, c_d$ sont des constantes à identifier. Dans notre cas, nous avons:

$$t(n) = c_1 \cdot 2^n + c_2 \cdot (-1)^n.$$

C) Identification des constantes. Afin d'identifier la valeur des constantes $c_1, c_2, \ldots, c_d$, nous construisons un système d'équations linéaires en évaluant d valeurs de t. Par exemple, dans notre cas nous avons:

$$t(1) = c_1 \cdot 2^1 + c_2 \cdot (-1)^1,$$
$$t(2) = c_1 \cdot 2^2 + c_2 \cdot (-1)^2.$$

Puisque $t(1) = 1$ et $t(2) = 3$, nous obtenons le système:

$$1 = 2c_1 - c_2,$$
$$3 = 4c_1 + c_2.$$

En résolvant le système, nous obtenons $c_1 = 2/3$ et $c_2 = 1/3$. Ainsi, nous avons:

$$t(n) = \frac{2}{3} \cdot 2^n + \frac{1}{3} \cdot (-1)^n.$$

Puisque $(1/3) \cdot (-1)^n$ vaut toujours $-1/3$ ou $1/3$, nous concluons que $t \in \Theta(2^n)$ et par conséquent qu'il y a un nombre exponentiel de pavages.

1. L'annexe « Récurrences linéaires » démontre que cette méthode fonctionne en s'appuyant sur l'algèbre linéaire. Cette annexe est technique et dépasse *largement* le cadre de ce cours.
2. Le cas où certaines racines apparaissent plusieurs fois est légèrement plus compliqué.

Exemple.

Examinons la célèbre *suite de Fibonacci* $\mathcal{F}$. Ses deux premiers termes sont 0 et 1, et chacun de ses termes subséquents correspond à la somme des deux termes précédents:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Nous avons:

$$\mathcal{F}_n = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ \mathcal{F}_{n-1} + \mathcal{F}_{n-2} & \text{sinon.} \end{cases}$$

En déplaçant tous les termes à gauche de l'égalité, la récurrence se réécrit $\mathcal{F}_n - \mathcal{F}_{n-1} - \mathcal{F}_{n-2} = 0$. Son polynôme caractéristique est donc:

$$p(x) = x^2 - x - 1 = (x - \lambda_1)(x - \lambda_2),$$

où $\lambda_1 \coloneqq (1 + \sqrt{5})/2 \approx 1{,}618$ et $\lambda_2 \coloneqq (1 - \sqrt{5})/2 = 1 - \lambda_1 \approx -0{,}618$. Ainsi, $\mathcal{F}_n = c_1 \cdot \lambda_1^n + c_2 \cdot \lambda_2^n$ pour certaines constantes c_1 et c_2.

Afin d'identifier la valeur des constantes, on utilise $t(0) = 0$ et $t(1) = 1$ afin de construire ce système d'équations:

$$\begin{aligned} 0 &= \quad c_1 + \quad c_2, \\ 1 &= \lambda_1 \cdot c_1 + \lambda_2 \cdot c_2. \end{aligned}$$

En résolvant le système, nous obtenons $c_1 = 1/\sqrt{5}$ et $c_2 = -1/\sqrt{5}$. Ainsi:

$$\begin{aligned} \mathcal{F}_n &= \frac{1}{\sqrt{5}} \cdot \lambda_1^n - \frac{1}{\sqrt{5}} \cdot \lambda_2^n \\ &= \frac{1}{\sqrt{5}} \cdot (\lambda_1^n - \lambda_2^n) \\ &\approx 0{,}447 \cdot (1{,}618^n - (-0{,}618)^n). \end{aligned}$$

Nous en concluons que $\mathcal{F}_n \in \Theta(\lambda_1^n)$.

Remarque.

Le nombre $\lambda_1 = \frac{1+\sqrt{5}}{2} \approx 1{,}618$, souvent dénoté φ, est le célèbre *nombre d'or* qui se manifeste par ex. dans la nature et en art.

5.2.2 Cas non homogène

Reconsidérons l'algorithme 30 pour le problème des tours de Hanoï. Nous avons déjà analysé le nombre de déplacements calculés par l'algorithme. Analysons maintenant le nombre d'appels récursifs $r(k)$ effectués par `hanoi'` sur entrée k. Lorsque $k = 0$, aucun appel n'est effectué. Autrement, deux appels récursifs sont effectués avec $k - 1$ comme entrée. Nous avons donc:

$$r(k) = \begin{cases} 0 & \text{si } k = 0, \\ 2 \cdot r(k-1) + 2 & \text{sinon.} \end{cases}$$

En réécrivant le cas général, nous obtenons $r(k) - 2 \cdot r(k-1) = 2$. Nous avons donc une récurrence linéaire, mais *non homogène* puisque le terme de droite ne vaut pas 0. Bien que notre approche ne s'applique pas ici, une méthode similaire permet de résoudre une telle récurrence.

Étant donné une récurrence $a_0 \cdot t(n) + a_1 \cdot t(n-1) + \ldots + a_d \cdot t(n-d) = c \cdot b^n$, nous construisons le polynôme caractéristique usuel et le multiplions par $(x-b)$. Dans notre cas, nous avons $c = 2$ et $b = 1$. Ainsi, nous multiplions le polynôme caractéristique par $x - 1$ afin d'obtenir:

$$q(x) = (x-2)(x-1).$$

À partir d'ici, la méthode demeure la même. Nous savons que $r(k)$ peut s'écrire sous la forme

$$r(k) = c_1 \cdot 2^k + c_2 \cdot 1^k.$$

Afin d'identifier les valeurs de c_1 et c_2, nous évaluons $r(0) = 0$ et $r(1) = 2$, et construisons le système d'équations:

$$\begin{aligned} 0 &= \quad c_1 + c_2, \\ 2 &= 2 \cdot c_1 + c_2. \end{aligned}$$

En résolvant le système, nous obtenons $c_1 = 2$ et $c_2 = -2$. Ainsi:

$$r(k) = 2 \cdot 2^k - 2 = 2(2^k - 1).$$

Le nombre d'appels récursifs effectués par `hanoi'` appartient donc à $\Theta(2^k)$.

5.3 Exponentiation rapide

Considérons le problème où nous cherchons à calculer b^n étant donnés $b, n \in \mathbb{N}$. Une approche simple, décrite à l'algorithme 32, consiste à utiliser la définition de l'exponentiation et ainsi de multiplier n fois b avec lui-même.

Il est possible de calculer b^n avec bien moins de multiplications en utilisant une approche diviser-pour-régner:

— si n est pair, on calcule $b^{n \div 2}$ et on le multiplie avec lui-même;

— sinon, on calcule $b^{n \div 2}$ et on le multiplie avec lui-même ainsi que b.

Algorithme 32 : Exponentiation naïve.

Entrées : $b, n \in \mathbb{N}$
Résultat : b^n

```
exp(b, n):
    si n = 0 alors
        retourner 1
    sinon
        retourner b · exp(b, n − 1)
```

Algorithme 33 : Exponentiation rapide.

Entrées : $b, n \in \mathbb{N}$
Résultat : b^n

```
exp-rapide(b, n):
    si n = 0 alors
        retourner 1
    sinon
        m ← exp-rapide(b, n ÷ 2)
        k ← 1
        si n est impair alors k ← b
        retourner m · m · k
```

Cette procédure est décrite à l'algorithme 33. Sa correction découle du fait que l'ordre des multiplications n'a aucune importance; ou en termes plus techniques, par l'associativité de la multiplication.

Analysons le nombre de multiplications $t(n)$ de `exp-rapide` par rapport à n. Comme la procédure effectue deux multiplications et un appel récursif avec l'exposant $n \div 2$, nous obtenons la récurrence:

$$t(n) = \begin{cases} 0 & \text{si } n = 0, \\ t(n \div 2) + 2 & \text{sinon.} \end{cases}$$

Comme cette récurrence n'est pas linéaire, nous ne pouvons pas la résoudre avec la méthode décrite précédemment. Cependant, comme on divise n par deux à répétition, on peut imaginer que le nombre d'appels récursifs est d'au plus $\log(n) + 1$. De plus, à chaque appel, on effectue précisément 2 multiplications. Nous pouvons donc naturellement conjecturer que $t(n) \leq 2 \cdot \log n + 2$ pour tout $n \geq 1$. Cela se vérifie par induction généralisée sur n. Pour $n = 1$,

nous avons $t(1) = 2 = 2 \cdot \log(1) + 2$. De plus:

$$\begin{aligned} t(n) &= t(n \div 2) + 2 && \text{(par définition de } t\text{)} \\ &\leq 2\log(n \div 2) + 4 && \text{(par hypothèse d'induction)} \\ &\leq 2\log(n/2) + 4 && \text{(car } n \div 2 \leq n/2\text{)} \\ &= 2\log n - 2\log 2 + 4 \\ &= 2\log n + 2. \end{aligned}$$

□

L'algorithme 33 effectue donc $t \in \Theta(\log n)$ multiplications, ce qui offre un gain exponentiel par rapport à l'algorithme 32.

5.4 Multiplication rapide

Dans la plupart des algorithmes présentés jusqu'ici, nous avons supposé que les opérations arithmétiques (addition, soustraction, multiplication, etc.) s'effectuent en temps constant. Cela est relativement réaliste si on les imagine comme faisant partie du jeu d'instruction d'une architecture, où les nombres sont généralement représentés sur un nombre fixe de bits, par ex. 64 bits. Cependant, cela dissimule la réelle complexité des opérations arithmétiques qui n'est pas constante lorsqu'on permet un nombre arbitraire de bits (ce que nous avons permis à plusieurs reprises!) On ignore souvent ce détail car il ne s'agit pas du « coeur du problème » ou puisque le coût s'avère « marginal » pour plusieurs applications. Toutefois, on peut difficilement l'ignorer pour des applications où l'on doit manipuler de très grands nombres à répétition, par ex. en calcul numérique ou en cryptographie.

En fait, avec des algorithmes élémentaires, l'addition et la multiplication de deux nombres de n chiffres prend un temps de $\mathcal{O}(n)$ et $\mathcal{O}(n^2)$ respectivement. Nous présentons l'*algorithme de Karatsuba* qui permet de multiplier deux entiers naturels de n chiffres plus rapidement qu'en temps quadratique. Plutôt que de considérer la base 2, considérons la base 10. Remarquons d'abord qu'un nombre de n chiffres peut être décomposé en deux nombres de $k = \lceil n/2 \rceil$ chiffres, en ajoutant des zéros non significatifs au besoin. Par exemple, $6789 = 10^2 \cdot 67 + 89$ et $345 = 10^2 \cdot 03 + 45$. En général, étant donnés deux nombres x et y de n chiffres, ceux-ci s'écrivent sous la forme $x = 10^k \cdot a + b$ et $y = 10^k \cdot c + d$. Nous avons donc:

$$\begin{aligned} x \cdot y &= (10^k \cdot a + b)(10^k \cdot c + d) \\ &= 10^{2k} \cdot ac + 10^k \cdot (ad + bc) + bd. \end{aligned}$$

Ainsi, la multiplication de deux nombres de $\approx 2k$ chiffres est équivalente à quatre multiplications de nombres de k chiffres (si on ignore décalages et additions). Cela ne semble pas nous aider. Toutefois, nous pouvons nous ramener à *trois* multiplications. En effet, observons que

$$ad + bc = ac + bd - (a - b)(c - d).$$

Ainsi, si nous connaissons la valeur de ac, bd et $(a-b)(c-d)$, nous pouvons reconstruire $x \cdot y$, tel que décrit à l'algorithme 34.

Algorithme 34 : Algorithme de multiplication rapide de Karatsuba.

Entrées : $x, y \in \mathbb{N}$ représentés sous $n \in \mathbb{N}_{\geq 1}$ chiffres en base 10
Résultat : $x \cdot y$

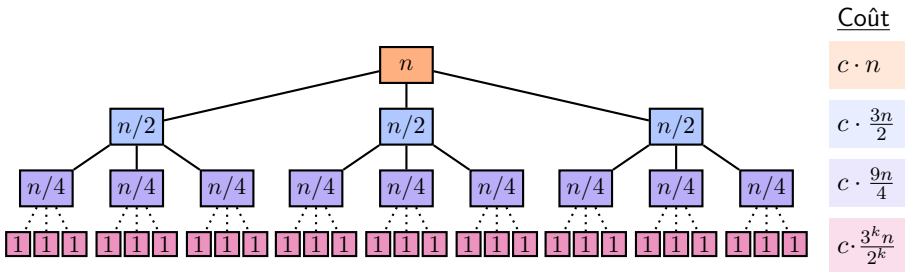
`mult`(n, x, y)**:**
  si $n = 1$ **alors**
    retourner $x \cdot y$
  sinon
    $k \leftarrow \lceil n/2 \rceil$
    $a, b \leftarrow x \div 10^k,\ x \bmod 10^k$
    $c, d \leftarrow y \div 10^k,\ y \bmod 10^k$
    $e \leftarrow$ `mult`(k, a, c)
    $f \leftarrow$ `mult`(k, b, d)
    $g \leftarrow$ `mult`$(k, a-b, c-d)$
    retourner $10^{2k} \cdot e + 10^k \cdot (e + f - g) + f$

Analysons l'algorithme 34 par rapport à n. Lorsque $n = 1$, le temps d'exécution est constant. Lorsque $n > 1$, nous multiplions trois nombres d'au plus n chiffres. Les additions et soustractions prennent un temps de $\mathcal{O}(n)$ si l'on utilise une implémentation standard. Les divisions entières et modulos par 10^k correspondent respectivement à des décalages et des troncations. Ces opérations prennent donc aussi un temps de $\mathcal{O}(n)$. Soit $t(n)$ le temps d'exécution de `mult` pour n chiffres. Nous avons:

$$t(n) \leq \begin{cases} c & \text{si } n = 1, \\ 3 \cdot t(\lceil n/2 \rceil) + c \cdot n & \text{sinon,} \end{cases}$$

où c est une certaine constante.

Estimons la complexité asymptotique de t à l'aide d'un *arbre de récursion*. Posons $k := \log n$. En supposant que n se divise toujours par deux, nous obtenons l'arbre de récursion suivant où chaque sommet correspond à un appel récursif de `mult` et où le coût indiqué à droite correspond au coût total de tous les appels d'un même niveau:



Ainsi, nous avons:

$$
\begin{aligned}
t(n) &\approx cn \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \\
&= cn \cdot \frac{(3/2)^{k+1} - 1}{3/2 - 1} && \text{(série géométrique de raison } 3/2) \\
&\leq cn \cdot \frac{(3/2)^{k+1}}{3/2 - 1} \\
&= 3cn \cdot (3/2)^k \\
&= 3cn \cdot (3/2)^{\log n} && \text{(par définition de } k) \\
&= 3cn \cdot n^{\log(3/2)} \\
&= 3c \cdot n^{1+\log(3/2)} \\
&= 3c \cdot n^{\log 3} && \text{(car } 1 + \log(3/2) = \log(2) + \log(3/2) = \log(2 \cdot 3/2)).
\end{aligned}
$$

Nous nous sommes donc convaincus semi-formellement que le temps d'exécution de l'algorithme 34 appartient à $\mathcal{O}(n^{\log 3})$. Puisque $\log 3 \leq 1{,}585$, cela offre un gain asymptotique considérable sur un algorithme quadratique. Par exemple, $1000^2 = 1\,000\,000$ alors que $1000^{\log 3} \leq 56\,871$.

Remarque.

Il existe des approches plus efficaces que l'algorithme de Karatsuba, par ex. les algorithmes de Toom–Cook, Schönhage–Strassen ($\mathcal{O}(n \cdot \log n \cdot \log\log n)$) et de Fürer ($\mathcal{O}(n \log n \cdot 2^{c \cdot \log^* n})$). En 2019, Harvey et van der Hoeven ont annoncé l'existence d'un algorithme fonctionnant en temps $\mathcal{O}(n \log n)$, ce qui est conjecturé comme optimal (asymptotiquement).

Remarque.

La librairie C++ Boost.Multiprecision utilise l'algorithme de Karatsuba. La librairie C GNU MP utilise des instances de l'algorithme de Toom–Cook (dont Toom-2 qui est l'algorithme de Karatsuba).

5.5 Théorème maître

La plupart des algorithmes qui empruntent l'approche diviser-pour-régner donnent lieu à des récurrences de la forme $t(n) = a \cdot t(\lceil n/b \rceil) + a' \cdot t(\lfloor n/b \rfloor) + f(n)$. En général, nous pouvons ignorer les plafonds et planchers [3] et plutôt considérer

$$t(n) = c \cdot t(n \div b) + f(n) \text{ où } c \coloneqq a + a'.$$

Comme il peut s'avérer fastidieux de résoudre une telle récurrence t, il existe une caractérisation générique de la complexité asymptotique de t par rapport à c, b et $f(n)$. Celle-ci est connue sous le nom de *théorème maître*. Nous présentons ici une version allégée de ce théorème suffisant pour l'analyse de la plupart des algorithmes:

Théorème 6. *Soient $t, f \in \mathcal{F}$, $b \in \mathbb{N}_{\geq 2}$, $c \in \mathbb{N}_{\geq 1}$ et $d \in \mathbb{R}_{>0}$ où $f \in \mathcal{O}(n^d)$ et*

$$t(n) = c \cdot t(n \div b) + f(n) \text{ pour tout } n \text{ suffisamment grand.}$$

La fonction t appartient à:

- $\mathcal{O}(n^d)$ *si* $c < b^d$,
- $\mathcal{O}(n^d \cdot \log n)$ *si* $c = b^d$,
- $\mathcal{O}(n^{\log_b c})$ *si* $c > b^d$.

Exemple.

Appliquons ce théorème à certains algorithmes.

Tri par fusion. Le tri par fusion découpe une séquence en deux, fait un appel récursif sur chacune des deux sous-séquences, puis effectue une fusion en temps linéaire. Nous obtenons donc une récurrence:

$$t(n) = 2 \cdot t(n \div 2) + f(n) \text{ où } f \in \mathcal{O}(n).$$

Nous avons $b = c = 2$, $d = 1$ et ainsi $c = b^d$. Par conséquent, le deuxième cas du théorème maître s'applique, ce qui implique que $t \in \mathcal{O}(n \log n)$.

Exponentiation rapide. Nous avons déjà établi que l'algorithme d'exponentiation rapide donne lieu à la récurrence $t(n) = t(n \div 2) + 2$. Nous avons $b = 2$, $c = 1$ et $d = 0$. Le deuxième cas du théorème maître s'applique car $c = b^d$. Nous obtenons donc $t \in \mathcal{O}(\log n)$.

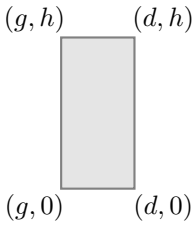
Multiplication rapide. Nous avons déjà établi que l'algorithme de Karatsuba donne lieu à la récurrence $t(n) = 3 \cdot t(n \div 2) + f(n)$ où $f \in \mathcal{O}(n)$. Nous avons $b = 2$, $c = 3$ et $d = 1$. Le troisième cas du théorème maître s'applique car $c = 3 > 2 = b^d$. Nous obtenons donc $t \in \mathcal{O}(n^{\log 3})$.

3. Si les détails techniques vous intéressent, voir, par ex., [Eri19, chapitre 1.7 (*Ignoring Floors and Ceilings Is Okay, Honest*)].

5.6 Problème de la ligne d'horizon

Considérons maintenant un problème plus complexe qui consiste à calculer la surface d'un ensemble de rectangles posés sur une même ligne d'horizon. Ce problème se résout notamment à l'aide de l'approche diviser-pour-régner.

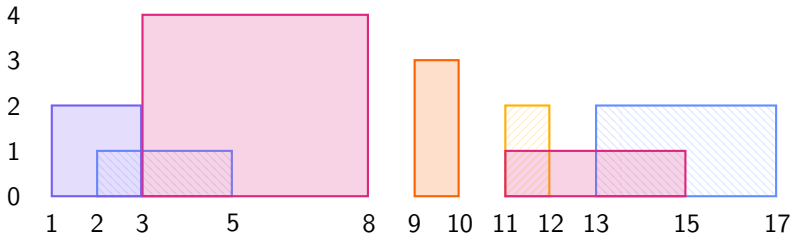
Un *bloc* est un triplet $(g, h, d) \in \mathbb{N}^3_{\geq 1}$ qui décrit ce rectangle dans le plan:



Un *paysage* est une séquence de blocs. Par exemple, le paysage

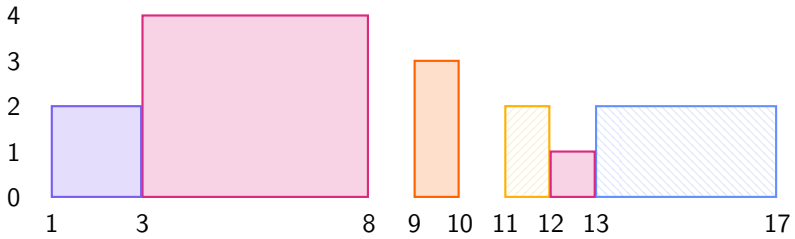
$$[(1,2,3),(2,1,5),(3,4,8),(9,3,10),(11,2,12),(11,1,15),(13,2,17)]$$

correspond graphiquement à:



Puisque plusieurs blocs peuvent se chevaucher, le calcul de la surface d'un paysage n'est pas immédiat. Par exemple, celui du paysage ci-dessus est de $2\cdot(3-1)+4\cdot(8-3)+3\cdot(10-9)+2\cdot(12-11)+1\cdot(13-12)+2\cdot(17-13) = 38$.

Une approche possible afin de calculer la surface d'un paysage consiste à le découper en blocs disjoints possédant la même surface, par exemple:



Nous présentons un algorithme qui découpe un paysage en blocs disjoints à la manière du tri par fusion:

— on divise le paysage en deux sous-séquences de blocs;

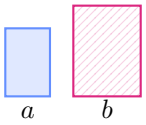
— on découpe chaque sous-séquence (récursivement);

— on fusionne les deux sous-paysages découpés.

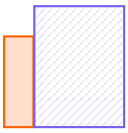
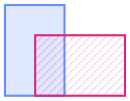
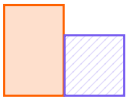
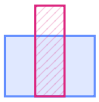
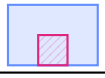
Les deux premières étapes s'implémentent exactement comme celles du tri par fusion. Voyons donc comment implémenter la troisième, c.-à-d. la fusion.

Nous devons fusionner deux paysages découpés x et y dans une nouvelle séquence z. Nous allons consommer les blocs de x et y de la gauche vers la droite, et les ajouter à la fin de z. De plus, nous allons parfois remettre temporairement des blocs dans x et y à partir de leur gauche. Ainsi, nous voyons x et y comme des piles dont les éléments sont empilés/dépilés à partir de la gauche, et z comme une séquence dont les éléments sont ajoutés à droite.

Soient a et b les premiers blocs de x et y respectivement. Supposons, sans perte de généralité, que $gauche(a) \leq gauche(b)$ (autrement on peut intervertir x et y). Si $droite(a) \leq gauche(b)$, on peut simplement retirer a de x et l'ajouter à z puisque les deux blocs sont disjoints:



Si $droite(a) > gauche(b)$, cela signifie que les deux blocs se chevauchent. On découpe donc a et b de cette façon:

Cas	**Avant**	**Après**
$droite(a) \leq droite(b)$ $hauteur(a) \leq hauteur(b)$		
$droite(a) \leq droite(b)$ $hauteur(a) > hauteur(b)$		
$droite(a) > droite(b)$ $hauteur(a) \leq hauteur(b)$		
$droite(a) > droite(b)$ $hauteur(a) > hauteur(b)$		

Pour effectuer le découpage ci-dessus:

— on retire a et b de x et y, respectivement;

— on identifie le cas qui s'applique (parmi les quatre possibles);
— on découpe a et b en un, deux, ou trois blocs selon le cas;
— on remet les blocs découpés dans x et y (selon leur origine).

Algorithme 35 : Découpage récursif d'un paysage en blocs disjoints.

Entrées : un paysage P de n blocs
Résultat : découpage de P en blocs disjoints

```
découper(P):
    si n ≤ 1 alors
        retourner P
    sinon
        x ← découper(P[1 : n ÷ 2])
        y ← découper(P[n ÷ 2 + 1 : n])
        z ← [ ]
        // Fusionner blocs de x et y dans z
        tant que |x| > 0 ∧ |y| > 0
            dépiler a de x
            dépiler b de y
            si gauche(b) < gauche(a) alors                    // intervertir?
                a, b ↔ b, a
                x, y ↔ y, x
            si droite(a) ≤ gauche(b) alors                    // sans chevauchement?
                empiler b sur y
                ajouter a à z
            sinon                                             // découper a et b
                si droite(a) ≤ droite(b) alors
                    si hauteur(a) ≤ hauteur(b) alors
                        empiler (gauche(a), hauteur(a), gauche(b)) sur x
                        empiler b sur y
                    sinon
                        empiler a sur x
                        empiler (droite(a), hauteur(b), droite(b)) sur y
                sinon
                    si hauteur(a) ≤ hauteur(b) alors
                        empiler (droite(b), hauteur(a), droite(a)) sur x
                        empiler b sur y
                        empiler (gauche(a), hauteur(a), gauche(b)) sur x
                    sinon
                        empiler a sur x
        // Retourner z avec les blocs restants de x ou y
        retourner z + x + y
```

Lorsque la fusion de x et y est complétée, l'un des deux peut encore contenir des blocs (comme pour le tri par fusion); on les ajoute simplement à z. L'algorithme 35 décrit cette approche sous forme de pseudocode. Notons qu'afin d'éviter des cas limites embêtants, il faudrait remplacer chaque instruction de la forme « **empiler** c **sur** w » par

$$\textbf{si } \mathit{gauche}(c) \neq \mathit{droite}(c) \textbf{ alors empiler } c \textbf{ sur } w.$$

Autrement dit, on se débarasse des blocs dégénérés avec une surface de 0.

Analysons le temps d'exécution t de l'algorithme 35. On découpe les n blocs de P en deux sous-séquences de taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ respectivement. Remarquons que les appels récursifs et la fusion peuvent créer de nouveaux blocs. Toutefois, on ne peut pas obtenir plus de blocs qu'il y a de lignes verticales, donc au plus $2n$. Ainsi, nous avons:

$$t(n) = \begin{cases} \alpha & \text{si } n \leq 1, \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \alpha \cdot 2n & \text{sinon,} \end{cases}$$

où α est une certaine constante. Dans le jargon du théorème maître, on obtient $b = c = 2$ et $d = 1$. Puisque $c = b^d$, nous obtenons $t \in \mathcal{O}(n \log n)$.

5.7 Racines multiples et changement de domaine

Lors de l'identification de la forme close d'une récurrence linéaire, nous avons supposé que les racines de son polynôme caractéristique étaient distinctes. Cela n'est pas toujours le cas. Si une racine λ apparaît k fois, alors le terme correspondant est de la forme:

$$d_0 \cdot n^0 \cdot \lambda^n + d_1 \cdot n^1 \cdot \lambda^n + \ldots + d_{k-1} \cdot n^{k-1} \cdot \lambda^n.$$

Par exemple, considérons une récurrence t dont le polynôme est $(x-3)(x-2)(x-2)$. Nous obtenons:

$$t(n) = c_1 \cdot 3^n + c_2 \cdot 2^n + c_3 \cdot n \cdot 2^n.$$

Reconsidérons maintenant la récurrence suivante qui capture essentiellement le temps d'exécution du tri par fusion et de l'algorithme du problème de la ligne d'horizon:

$$t(n) = \begin{cases} 1 & \text{si } n = 0, \\ 2 \cdot t(n \div 2) + n & \text{sinon.} \end{cases}$$

Cette récurrence n'est pas linéaire, on ne peut donc pas appliquer directement notre méthode. Cependant, nous pouvons analyser t pour les valeurs de n qui sont des puissances de deux en effectuant un changement de domaine. Posons $s(k) \coloneqq t(2^k)$. Remarquons que $s(0) = t(1) = 3$ et que $s(k) = 2 \cdot t(2^{k-1}) + 2^k$

pour $k > 0$. Ainsi:

$$s(k) = \begin{cases} 3 & \text{si } k = 0, \\ 2 \cdot s(k-1) + 2^k & \text{sinon.} \end{cases}$$

Nous obtenons donc la récurrence linéaire non homogène $s(k) - 2 \cdot s(k-1) = 2^k$, ce qui mène au polynôme:

$$\underbrace{(x-2)}_{\text{poly. car.}} \cdot \overbrace{(x-2)}^{\text{obtenu de } 2^k}$$

et ainsi la forme close:

$$s(k) = c_1 \cdot 2^k + c_2 \cdot k \cdot 2^k.$$

En résolvant le système:

$$\begin{aligned} s(0) = 3 &= \quad c_1 \\ s(1) = 8 &= 2c_1 + 2c_2 \end{aligned}$$

nous obtenons $c_1 = 3$ et $c_2 = 1$. Ainsi:

$$s(k) = 3 \cdot 2^k + k \cdot 2^k.$$

Pour une valeur $n = 2^k$, on obtient donc $t(n) = 3n + n \log n$. Informellement, on conclut donc que $t \in \mathcal{O}(n \log n : n \text{ est une puissance de deux})$. Il existe des notions relativement simples [a] (que nous ne couvrirons pas) qui permettent de lever la condition et d'en conclure que $t \in \mathcal{O}(n \log n)$.

a. Par exemple, voir la notion de « *b-smoothness* » dans [BB96, chap. 3.4].

5.8 Exercices

5.1) Identifiez une forme close pour la récurrence suivante:

$$t(n) = \begin{cases} 1 & \text{si } n = 0, \\ 2 & \text{si } n = 1, \\ t(n-1) + 6 \cdot t(n-2) & \text{sinon.} \end{cases}$$

Identifiez sa complexité asymptotique (aussi précisément que possible).

5.2) Identifiez une forme close pour la récurrence suivante:

$$t(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ t(n-1) + 2 \cdot t(n-2) + 3 & \text{sinon.} \end{cases}$$

Identifiez sa complexité asymptotique (aussi précisément que possible).

5.3) Identifiez une forme close pour la récurrence suivante:

$$t(n) = \begin{cases} 1 & \text{si } n = 0, \\ 0 & \text{si } n = 1, \\ t(n-2) & \text{sinon.} \end{cases}$$

Identifiez sa complexité asymptotique (aussi précisément que possible).

5.4) Identifiez une forme close pour la récurrence suivante:

$$t(n) = \begin{cases} 0 & \text{si } n \leq 2, \\ t(n-1) + 10 \cdot t(n-2) + 8 \cdot t(n-3) + 1 & \text{sinon.} \end{cases}$$

Identifiez sa complexité asymptotique (aussi précisément que possible).

5.5) Soient $c, d \in \mathbb{R}_{\geq 1}$. Identifiez une forme close pour la récurrence suivante:

$$t(n) = \begin{cases} 0 & \text{si } n = 0, \\ d \cdot t(n-1) + c & \text{sinon.} \end{cases}$$

Identifiez sa complexité asymptotique (aussi précisément que possible).

5.6) Nous avons vu que la récurrence définie par $t(0) := 0$ et $t(k) := 2 \cdot t(k-1) + 1$ possède la forme close $t(k) = 2^k - 1$. Montrez que c'est bien le cas par induction sur k.

5.7) Donnez un algorithme qui calcule le nombre d'inversions d'une séquence de n éléments comparables en temps $\mathcal{O}(n \log n)$.

5.8) Donnez un algorithme qui calcule tous les pavages d'une grille $2 \times n$ à l'aide de tuiles de cette forme:

Donnez une récurrence indiquant le nombre de pavages obtenus par l'algorithme par rapport à n.

5.9) Combien y a-t-il de séquences binaires de n bits qui ne contiennent pas deux occurrences consécutives de `1`?

5.10) Adaptez l'algorithme d'exponentiation rapide afin qu'il calcule b^n mod m où $m \in \mathbb{N}_{\geq 2}$.

5.11) Afin d'élaborer l'algorithme de Karatsuba, nous avons d'abord considéré l'identité $(10^k \cdot a + b)(10^k \cdot c + d) = 10^{2k} \cdot ac + 10^k \cdot (ad + bc) + bd$. Nous aurions donc pu implémenter la multiplication de cette façon:

Entrées : $x, y \in \mathbb{N}$ représentés sous $n \in \mathbb{N}_{\geq 1}$ chiffres en base 10
Résultat : $x \cdot y$

`mult`(n, x, y)**:**
  si $n = 1$ **alors**
    retourner $x \cdot y$
  sinon
    $k \leftarrow \lceil n/2 \rceil$

    $a, b \leftarrow x \div 10^k,\ x \bmod 10^k$
    $c, d \leftarrow y \div 10^k,\ y \bmod 10^k$

    $e \leftarrow$ `mult`(k, a, c)
    $f \leftarrow$ `mult`(k, a, d)
    $g \leftarrow$ `mult`(k, b, c)
    $h \leftarrow$ `mult`(k, b, d)

    retourner $10^{2k} \cdot e + 10^k \cdot (f + g) + h$

Quelle est la complexité de cet algorithme? Est-il aussi efficace que l'algorithme de Karatsuba?

5.12) Donnez un algorithme qui reçoit une séquence binaire non décroissante et retourne le nombre de bits égaux à `1` en temps $\mathcal{O}(\log n)$. Par exemple, votre algorithme doit retourner 5 sur entrée $[0, 0, 0, 1, 1, 1, 1, 1]$.

5.13) Un *terrain* est une matrice $\mathbf{A} \in \mathbb{N}^{m \times n}$ où $m, n \geq 3$. Nous disons qu'une paire $(i, j) \in [n] \times [n]$ est un *sommet* de $\mathbf{A}$ si $1 < i < m$, $1 < j < n$ et ces

inégalités sont satisfaites:

$$\begin{array}{ccccc} & & \mathbf{A}[i-1,j] & & \\ & & \wedge & & \\ \mathbf{A}[i,j-1] & < & \mathbf{A}[i,j] & > & \mathbf{A}[i,j+1] \\ & & \vee & & \\ & & \mathbf{A}[i+1,j] & & \end{array}$$

Donnez un algorithme qui détermine si un terrain possède un sommet en temps $\mathcal{O}(m \cdot \log n)$.

5.14) Nous disons qu'une séquence s de $n \in \mathbb{N}_{\geq 1}$ éléments comparables est *ordonnée circulairement* s'il existe $i \in [n]$ tel que

$$s[i] \leq s[i+1] \leq \ldots \leq s[n] \leq \ldots \leq s[i-1].$$

En particulier, si $i = 1$, alors s est ordonnée au sens usuel.

a) Donnez un algorithme qui reçoit en entrée une séquence s ordonnée circulairement et dont tous les éléments sont distincts, et qui retourne le plus grand élément de s en temps $\mathcal{O}(\log n)$. Par exemple, votre algorithme devrait retourner 19 sur entrée:

$$s = [10, 12, 19, 1, 3, 4, 7].$$

b) ★ Montrez que votre algorithme est correct même si s possède des doublons, pourvu que $s[1] \neq s[n]$ où n est la taille de s. S'il n'est pas correct, adaptez-le pour que ce soit le cas.

c) ★★ Montrez qu'*aucun* algorithme ne peut résoudre le problème en moins de n requêtes à s si l'on permet des doublons arbitraires.

5.15) Donnez un algorithme qui reçoit une séquence s de n entiers et qui retourne la plus grande somme contigüe en temps $\mathcal{O}(n \log n)$, c'est-à-dire:

$$\max\{s[i] + \ldots + s[j] : 1 \leq i \leq j \leq n\}.$$

Par exemple, votre algorithme doit retourner 9 sur entrée

$$s = [3, 1, -5, \underline{4, -2, 1, 6}, -3].$$

5.16) Imaginons une implémentation du tri rapide (*quicksort*) où le choix du pivot partitionne la séquence environ au tiers de sa taille. Autrement dit, la séquence est découpée en deux sous-séquences: l'une de taille $n \div 3$ et l'autre de taille $n - (n \div 3)$. Analysez semi-formellement le temps d'exécution asymptotique de l'algorithme à l'aide d'un arbre de récursion.

(*basé sur un passage de [Eri19, chap. 1.7]* ©)

5.17) Rappelons le problème de vote à majorité absolue de la section 1.3: étant donné une séquence s de n éléments, on cherche à identifier une valeur qui apparaît plus de $n/2$ fois dans s (s'il en existe une). Donnez un algorithme diviser-pour-régner qui résout ce problème en temps $\mathcal{O}(n \log n)$. Tentez de concevoir un algorithme qui n'utilise *pas* les comparaisons $\{<, \leq, \geq, >\}$.

5.18) Imaginons un scénario où on cherche à élire une personne parmi n personnes regroupées circulairement. Le processus d'élection consiste à éliminer une personne sur deux à partir de la première personne. Par exemple, s'il y a cinq personnes, les personnes 2 et 4 sont d'abord éliminées, et les personnes 1, 3 et 5 passent à la ronde suivante. La dernière personne non éliminée est élue.

a) Soit t la fonction telle que $t(n)$ est la position de la personne élue lorsqu'il y a initialement n personnes. Exprimez t en tant que récurrence. Cette récurrence est-elle linéaire?

b) Donnez un algorithme qui identifie la position où se placer pour être élu.

c) ★ Montrez que $t(n) = 2 \cdot (n - 2^{\lceil \log n \rceil}) + 1$ par induction généralisée.

d) ★ Donnez un algorithme qui identifie la position où se placer pour être élu, en inspectant les bits de la représentation binaire de n.

5.19) En classe (session A20), une personne a suggéré l'algorithme récursif ci-dessous afin de calculer b^n. Analysez le nombre de multiplications qu'il effectue. Vous pouvez d'abord supposer que n est une puissance de deux.

Entrées : $b, n \in \mathbb{N}$
Résultat : b^n

```
exp'(b, n):
    si n = 0 alors
        retourner 1
    sinon si n = 1 alors
        retourner b
    sinon
        a  ← n ÷ 2
        a' ← n − a
        retourner exp'(b, a) · exp'(b, a')
```

5.20) ★★ *(requiert une connaissance des nombres complexes)*
Identifiez une forme close pour la récurrence suivante:

$$t(n) = \begin{cases} 1 - n & \text{si } n \in \{0, 1\}, \\ -t(n-2) & \text{sinon.} \end{cases}$$

Identifiez sa complexité asymptotique (aussi précisément que possible).

5.21) ★★ *(dépasse légèrement le cadre du cours)*

Soit S un ensemble muni d'un élément neutre e et d'une opération binaire associative $\oplus\colon S \times S \to S$, c.-à-d. que pour tous $a, b, c \in S$:

— $a \oplus e = a = e \oplus a$,

— $a \oplus (b \oplus c) = (a \oplus b) \oplus c$,

Définissons $b^0 \coloneqq e$ et $b^n \coloneqq b^{n-1} \oplus b$ pour tous $b \in S$ et $n \in \mathbb{N}_{\geq 1}$.

(a) Adaptez l'algorithme d'exponentiation rapide afin de calculer b^n avec $\mathcal{O}(\log n)$ applications de $\oplus$.

(b) Montrez que l'algorithme est correct par induction généralisée.

(c) Montrez que chacun de ces triplets $(S, e, \oplus)$ satisfait la précondition:

(i) $S \coloneqq \mathbb{N}$, $e \coloneqq 1$ et $a \oplus b \coloneqq a \cdot b$;

(ii) $S \coloneqq \mathbb{N}$, $e \coloneqq 0$ et $a \oplus b \coloneqq a + b$;

(iii) $S \coloneqq \{0, 1, \ldots, m-1\}$, $e \coloneqq 0$ et $a \oplus b \coloneqq (a + b) \bmod m$;

(iv) $S \coloneqq \{0, 1, \ldots, m-1\}$, $e \coloneqq 1$ et $a \oplus b \coloneqq (a \cdot b) \bmod m$;

(v) $S \coloneqq \mathbb{N}^{k \times k}$, $e \coloneqq \mathbf{I}$ et $\mathbf{A} \oplus \mathbf{B} \coloneqq \mathbf{AB}$.

Une ancienne version de l'exercice supposait que $\oplus$ était commutative. Merci à Cristopher Cruz (A20) qui a remarqué que cette hypothèse est inutile.

Remarque.

Cet exercice montre que l'algorithme d'exponentiation rapide s'applique plus généralement à tout *monoïde*. En particulier, le monoïde décrit en (iv) s'avère utile en cryptographie.

5.22) ★ Soit t la relation de récurrence définie par:

$$t(n) \coloneqq \begin{cases} 0 & \text{si } n = 0, \\ c \cdot t(n \div b) + n^d & \text{sinon.} \end{cases}$$

En vous limitant aux valeurs de n qui sont des puissances de b, montrez semi-formellement, comme nous l'avons fait pour l'algorithme de Karatsuba, que la fonction t appartient à:

— $\mathcal{O}(n^d)$ si $c < b^d$,

— $\mathcal{O}(n^d \cdot \log n)$ si $c = b^d$,

— $\mathcal{O}(n^{\log_b c})$ si $c > b^d$.

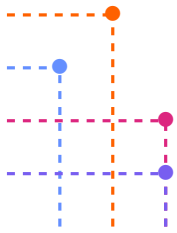
Indice: considérez une série géométrique de raison $r < 1$, $r = 1$ ou $r > 1$.

Observation.

Cet exercice explique en bonne partie la validité du théorème maître.

5.23) Cet exercice porte sur l'identification des éléments maximaux d'un ensemble partiellement ordonné.

a) Nous disons qu'une paire d'entiers $(a', b') \in \mathbb{N}^2$ *domine* une paire $(a, b) \in \mathbb{N}^2$, dénoté $(a, b) \preceq (a', b')$, si $a \leq a'$ et $b \leq b'$. Étant donné un ensemble fini de paires $X \subseteq \mathbb{N}^2$, nous disons qu'une paire $(a, b) \in X$ est *maximale* si aucune autre paire de X la domine. Par exemple, les paires maximales de $\{(1,3), (3,2), (2,4), (3,1)\}$ sont $(3,2)$ et $(2,4)$:



Donnez un algorithme qui résout ce problème en temps $\mathcal{O}(n \log n)$:

ENTRÉE: une séquence s de n éléments de $\mathbb{N}^2$ sans doublons
SORTIE: les éléments maximaux de s

b) ★ Donnez un algorithme qui fonctionne en $\mathcal{O}(n \log n)$ pour le même problème sur $\mathbb{N}^3$ avec l'ordre $\preceq$ défini par $(a, b, c) \preceq (a', b', c')$ si $a \leq a'$, $b \leq b'$ et $c \leq c'$.

c) Rappelons qu'un *ordre partiel* sur un ensemble $\mathbb{D}$ est une relation $\preceq \subseteq \mathbb{D} \times \mathbb{D}$ qui satisfait ces propriétés pour tous $x, y, z \in \mathbb{D}$:

— *réfléxivité*: $x \preceq x$;
— *antisymétrie*: $(x \preceq y \wedge y \preceq x) \implies x = y$;
— *transitivité*: $(x \preceq y \wedge y \preceq z) \implies x \preceq z$.

Un *ordre total* est un ordre partiel où tous les éléments sont comparables: pour tous $x, y \in \mathbb{D}$, nous avons $x \preceq y$ ou $y \preceq x$.

Considérons ce problème:

ENTRÉE: une séquence s de n éléments de $\mathbb{D}$ partiellement ordonnés par $\preceq$ et sans doublons
SORTIE: les éléments maximaux de s sous $\preceq$,
c.-à-d. $\{x \in s : \forall y \in s\,(x = y) \vee \neg(x \preceq y)\}$

— Donnez un algorithme qui résout le problème avec $\mathcal{O}(n^2)$ comparaisons.

— ★★ Montrez que pour l'ensemble $\mathcal{P}(\mathbb{N})$, avec l'ordre partiel $\subseteq$, le problème ne peut pas être résolu avec moins de $n(n-1)/2$ comparaisons.

Remarque.

Soient $\mathbb{D}_1, \mathbb{D}_2, \ldots \mathbb{D}_k$ des ensembles et $\preceq_1, \preceq_2, \ldots, \preceq_k$ des ordres totaux sur ces ensembles respectifs. Considérons la relation $\preceq$, sur l'ensemble $\mathbb{D} := \mathbb{D}_1 \times \mathbb{D}_2 \times \cdots \times \mathbb{D}_k$, définie par $x \preceq y$ ssi $x \preceq_1 y \wedge x \preceq_2 y \wedge \cdots \wedge x \preceq_k y$. Il est possible de calculer les éléments maximaux en mieux que $\mathcal{O}(n^2)$. Par [KLP75], le problème est soluble en $\mathcal{O}(n \log n)$ si $k = 2$, et $\mathcal{O}(n(\log n)^{k-2})$ si $k \geq 3$. Cela généralise les sous-questions (a) et (b).

6

Force brute

Ce chapitre traite de la *force brute*, une approche simple qui consiste à explorer *exhaustivement* un ensemble de solutions candidates jusqu'à l'identification d'une véritable solution. Par exemple, afin de trier une séquence s de n éléments comparables, on pourrait naïvement énumérer toutes les permutations de s jusqu'à l'identification de sa forme triée. Cela nécessiterait $n!$ itérations dans le pire cas, ce qui est impraticable. En général, on nomme « *explosion combinatoire* » le phénomène où l'espace de recherche croît rapidement par rapport à la taille des entrées. Malgré cette limitation générale, la force brute possède certains avantages. Premièrement, elle permet d'établir rapidement un algorithme de référence contre lequel on peut tester nos algorithmes plus efficaces. Deuxièmement, pour plusieurs problèmes, on ne connaît simplement aucune autre approche algorithmique, par ex. plusieurs problèmes dits *NP-complets*. Nous présentons quelques-uns de ces problèmes.

6.1 Problème des n dames

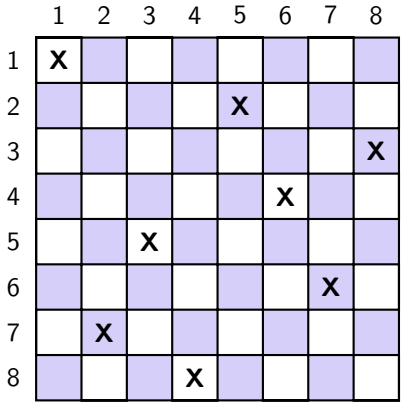
Le célèbre *problème des huit dames* consiste à placer huit dames sur un échiquier sans qu'elles puissent s'attaquer. De façon plus générale, ce problème consiste à placer n pièces sur une grille $n \times n$ sans qu'il y ait plus d'une pièce par ligne, par colonne et par diagonale. Cherchons à résoudre ce problème algorithmiquement. Nous pouvons représenter une solution par une matrice $\mathbf{A} \in \{0,1\}^{n \times n}$ où $\mathbf{A}[i,j]$ indique si une dame apparaît ou non à la position (i,j). Il y a 2^{n^2} telles matrices. Pour $n = 8$, on obtient

$$18\,446\,744\,073\,709\,551\,616 \text{ possibilités.}$$

On peut réduire significativement le nombre de possibilités en observant que $\mathbf{A}$ doit contenir *exactement* n occurrences de 1. Il y a $\binom{n^2}{n}$ telles matrices. Pour $n = 8$, on obtient

$$4\,426\,165\,368 \text{ possibilités.}$$

Ces matrices sont éparses, c-à-d. qu'il y a peu d'occurrences de 1 par rapport aux occurrences de 0. Nous pouvons donc simplifier leur représentation. Remarquons qu'une solution assigne nécessairement une dame à chaque ligne et à chaque colonne. Ainsi, nous pouvons décrire une solution par une séquence sol de taille n où $sol[i]$ indique la colonne de la dame sur la ligne i. Par exemple, pour $n = 8$, la solution $[1, 5, 8, 6, 3, 7, 2, 4]$ correspond graphiquement à:



Il y a $n \cdot (n-1) \cdots 1 = n!$ telles séquences. Par exemple, $8! = 40320$. Peu des $n!$ possibilités forment une solution. Par exemple, il existe 92 solutions pour le cas $n = 8$. Nous avons donc intérêt à ne pas explorer toutes les possibilités.

Algorithme 36 : Force brute pour le problème des huit dames.

Entrées : $n \in \mathbb{N}$
Résultat : solution au problème des n dames (s'il en existe une)

```
dames(n):
    dames'(sol):
        si |sol| = n alors
            retourner sol
        sinon
            pour j ∈ {1, ..., n} \ sol      // essayer colonnes dispo.
                sol' ← sol + [j]            // sol' = sol étendue avec j
                si sol' respecte les contraintes de diagonales alors
                    r ← dames'(sol')
                    si r ≠ aucune alors
                        retourner r
            retourner aucune
    retourner dames'([])
```

Nous allons débuter avec la solution partielle triviale $[\,]$ et progressivement:

— assigner une colonne non utilisée à la dame de la prochaine ligne;

— poursuivre l'assignation récursivement s'il y a au plus une dame par diagonale;

— si l'assignation ne mène pas à une solution, on répète avec les autres colonnes non utilisées.

Cette approche s'implémente de façon récursive tel que décrit à l'algorithme 36. Cette stratégie générale se nomme *retour arrière* puisqu'on revient sur nos décisions lorsqu'on n'identifie aucune solution.

L'algorithme 37 implémente le test des contraintes de diagonales en se basant sur les observations suivantes. Soit (i, j) une position de la grille. Remarquons que la diagonale « sud-ouest $\rightarrow$ nord-est » est caractérisée par $i + j$, alors que la diagonale « nord-ouest $\rightarrow$ sud-est » est caractérisée par $i - j$. Par exemple, pour $n = 8$, nous avons:

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

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

Algorithme 37 : Test des contraintes de diagonales.

Entrées : assignation partielle des dames sol
Résultat : sol respecte les contraintes de diagonales?

```
diag(sol):
    nord ← ∅; sud ← ∅
    pour i ∈ [1..|sol|]
        j ← sol[i]
        si i + j ∈ nord alors retourner faux
        sinon ajouter i + j à nord
        si i − j ∈ sud alors retourner faux
        sinon ajouter i − j à sud
    retourner vrai
```

6.2 Problème du sac à dos

Reconsidérons le problème du sac à dos introduit à la section 4.3. Nous avons présenté un algorithme d'approximation pour ce problème, mais aucun algorithme qui permette de le résoudre exactement. Un algorithme de force brute pour ce problème consiste à explorer les 2^n possibilités. L'algorithme 38 implémente cette approche de façon récursive: si nous sommes rendus à l'objet i, on explore d'une part l'assignation où on ne choisit pas i, puis l'assignation où on choisit i, ce qui augmente la valeur et le poids du sac de $v[i]$ et $p[i]$ respectivement.

Algorithme 38 : Force brute pour le problème du sac à dos.

Entrées : valeurs $v \in \mathbb{N}^n$, poids $p \in \mathbb{N}^n$ et capacité $c \in \mathbb{N}$
Résultat : valeur maximale du sac à dos

```
sac-à-dos-naïf(v, p, c):
    remplir(i, valeur, poids):
        si i > n alors
            si poids ≤ c alors retourner valeur
            sinon              retourner 0
        sinon
            valeur' ← valeur + v[i]
            poids'  ← poids  + p[i]
            retourner max(remplir(i + 1, valeur,  poids),
                          remplir(i + 1, valeur', poids'))
    retourner remplir(1, 0, 0)
```

L'algorithme 38 fonctionne en temps $\Omega(2^n)$, même dans le meilleur cas, puisqu'il explore systématiquement les 2^n assignations possibles sans s'arrêter prématurément. Nous pouvons améliorer l'algorithme en n'explorant pas une assignation si elle excède la capacité du sac. Si nous imaginons la recherche d'une solution comme l'exploration de l'arbre des assignations, nous *élaguons* [1] donc certaines de ses branches. L'algorithme 39 décrit cette modification.

Reconsidérons l'exemple de la section 4.3, c'est-à-dire l'instance:

$$v = [50, 5, 65, 10, 12, 20],$$
$$p = [700, 320, 845, 70, 420, 180],$$
$$c = 900.$$

L'algorithme 38 teste $2^6 = 64$ combinaisons, alors que l'algorithme 39 n'en teste que 18.

Nous pouvons améliorer l'algorithme 39 à l'aide d'un élagage plus agressif:

— on stocke la meilleure solution identifiée jusqu'ici;

1. On parle de « *pruning* » en anglais.

Algorithme 39 : Force brute avec élagage basé sur la capacité.

Entrées : valeurs $v \in \mathbb{N}^n$, poids $p \in \mathbb{N}^n$ et capacité $c \in \mathbb{N}$
Résultat : valeur maximale du sac à dos

```
sac-à-dos-élagage(v, p, c):
    remplir(i, valeur, poids):
        si i > n alors
            retourner valeur
        sinon
            valeur' ← valeur + v[i]
            poids'  ← poids  + p[i]
            sol ← remplir(i + 1, valeur, poids)        // sans objet i
            si poids' ≤ c alors                        // avec objet i
                sol ← max(sol, remplir(i + 1, valeur', poids'))
            retourner sol
    retourner remplir(1, 0, 0)
```

— si une branche ne permet pas d'excéder la meilleure solution, on l'ignore.

Afin d'implémenter la deuxième étape, nous utilisons l'observation suivante: si l'ajout de *tous* les objets restants (en ignorant la capacité du sac) n'excède pas la meilleure solution découverte, alors il est inutile de poursuivre.

Afin d'amorcer la procédure, nous pourrions considérer 0 comme la meilleure solution connue. Cependant, nous pouvons faire mieux en débutant par une solution prometteuse. Par exemple, nous pouvons débuter avec la valeur du meilleur objet ou avec la solution obtenue par l'algorithme glouton (c.-à-d. l'algorithme 29). L'algorithme 40 implémente cette procédure.

Par exemple, considérons cette instance:

$$\begin{aligned} v &= [1, 2, \ldots, 200], \\ p &= [200, \ldots, 2, 1], \\ c &= 150. \end{aligned}$$

L'algorithme 39 effectue $605\,701\,807$ appels récursifs et explore $278\,031\,704$ assignations complètes, alors que l'algorithme 40 effectue $4\,325\,153$ appels récursifs et n'explore *aucune* assignation complète. De plus, une implémentation directe en PYTHON3 donne un temps d'exécution d'approximativement 5 min. 51 sec. et 2,45 sec., respectivement, sur cette instance (sur ma machine).

Ce type d'approche s'inscrit plus généralement dans le cadre du « *branch and bound* », où l'on guide l'exploration d'un arbre de possibilités grâce à des bornes pouvant être identifiées efficacement.

Algorithme 40 : Force brute sans exploration des branches qui ne permettent pas d'excéder la meilleure solution découverte.

Entrées : valeurs $v \in \mathbb{N}^n$, poids $p \in \mathbb{N}^n$ et capacité $c \in \mathbb{N}$
Résultat : valeur maximale du sac à dos

```
sac-à-dos-turbo(v, p, c):
    meilleure ← sol-prometteuse(v, p, c)
    potentiel ← [v[i] + ... + v[n] : i ∈ [1..n]]
    remplir(i, valeur, poids):
        meilleure ← max(meilleure, valeur)
        si (i < n) ∧ (valeur + potentiel[i] > meilleure) alors
            valeur' ← valeur + v[i]
            poids'  ← poids  + p[i]
            remplir(i + 1, valeur, poids)          // sans objet i
            si poids' ≤ c alors                    // avec objet i
                remplir(i + 1, valeur', poids')
    remplir(1, 0, 0)
    retourner meilleure
```

6.3 Problème du retour de monnaie

Le *problème du retour de monnaie* consiste à identifier la plus petite quantité de pièces permettant de rendre la monnaie sur un certain montant:

ENTRÉE:	un montant $m \in \mathbb{N}$ et une séquence s de $n \in \mathbb{N}$ nombres naturels représentant un système monétaire
SORTIE:	plus petit nombre de pièces du système s permettant de former m

Pour le dollar canadien, l'euro, le dinar et le dirham, par exemple, on peut simplement rendre les pièces en ordre décroissant de leur valeur. Cependant, cette approche gloutonne ne fonctionne pas en général. Par exemple, si $m = 10$ et $s = [1, 5, 7]$, alors cette procédure retourne 4 pièces ($10 = 7 + 1 + 1 + 1$), bien que la solution optimale soit constituée de 2 pièces ($10 = 5 + 5$).

Nous pouvons résoudre ce problème par force brute:

— on considère les pièces itérativement;

— si la pièce actuelle $s[i]$ est supérieure au montant à rendre, alors on passe à la prochaine pièce;

— sinon on choisit la meilleure solution entre celle qui prend $s[i]$ et celle qui ne la prend pas.

L'algorithme 41 présente cette procédure sous forme de pseudocode.

Algorithme 41 : Force brute pour retour de monnaie.

Entrées : montant $m \in \mathbb{N}$, séquence s de $n \in \mathbb{N}$ pièces
Résultat : nombre minimal de pièces afin de rendre m

```
monnaie-brute(m, s):
    // k:   montant qu'on doit encore rendre
    // num: nombre de pièces utilisées jusqu'ici
    // i:   indice de la pièce considérée
    aux(k, num, i):
        si i = n + 1 alors
            si k = 0 alors retourner num
            sinon          retourner ∞
        sinon
            sans ← aux(k, num, i + 1)              // sol. sans s[i]
            avec ← ∞
            si k ≥ s[i] alors
                avec ← aux(k − s[i], num + 1, i)   // sol. avec s[i]
            retourner min(sans, avec)
    retourner aux(m, 0, 1)
```

6.4 Satisfaction de formules de logique propositionnelle

Il existe un certain nombre de problèmes qu'on ne sait pas résoudre autrement que par force brute[2]. L'un des plus importants, nommé SAT, consiste à déterminer si une formule de logique propositionnelle est satisfaisable. Par exemple, la formule

$$(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

est satisfaite par l'assignation $x =$ `vrai`, $y =$ `vrai` et $z =$ `faux`. Comme ce problème trouve des applications dans une foule de domaines, dont l'intelligence artificielle, la vérification formelle et l'élaboration de circuits, il existe un large éventail d'algorithmes de force brute sophistiqués qui n'explorent pas nécessairement les 2^n assignations possibles des n variables booléennes.

Ainsi, afin de résoudre un problème difficile, on peut le traduire vers une formule de logique propositionnelle qu'on envoie à un solveur SAT. Par exemple, considérons le problème des n dames. Pour chaque paire (i, j), on introduit une variable booléenne $x_{i,j}$ qui indique si une dame apparaît ou non à la case (i, j) de l'échiquier. On formule ensuite le problème avec les contraintes suivantes:

— il y a au moins une dame par ligne:

$$\bigwedge_{1 \le i \le n} \bigvee_{1 \le j \le n} x_{i,j}$$

2. Ou par programmation dynamique, que nous verrons au chapitre suivant.

— il ne peut y avoir deux dames (ou plus) sur une même ligne:

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n}\bigwedge_{j<k\leq n}(\neg x_{i,j}\vee\neg x_{i,k})$$

— il ne peut y avoir deux dames (ou plus) sur une même colonne:

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq n}\bigwedge_{j<k\leq n}(\neg x_{j,i}\vee\neg x_{k,i})$$

— il ne peut y avoir deux dames (ou plus) sur une même diagonale:

$$\bigwedge_{\substack{i,i',j,j'\in[1..n]\\(i,j)\neq(i',j')\\i+j=i'+j'}}(\neg x_{i,j}\vee\neg x_{i',j'})\ \wedge\ \bigwedge_{\substack{i,i',j,j'\in[1..n]\\(i,j)\neq(i',j')\\i-j=i'-j'}}(\neg x_{i,j}\vee\neg x_{i',j'})$$

Un solveur SAT de pointe risque d'identifier une solution d'une telle formule bien plus rapidement qu'un algorithme par force brute « maison » (cela dépend des problèmes, des formules et de leur taille). Par exemple, voici la formulation du problème pour $n=2$ avec le solveur z3 de *Microsoft Research*.

6.5 Programmation linéaire entière

Pour les problèmes où l'on doit raisonner sur des variables entières non bornées, on peut difficilement exploiter un solveur SAT. Alternativement, nous pouvons tenter de traduire notre problème en *programmation linéaire entière*, un paradigme important notamment en *recherche opérationnelle*. Dans sa forme canonique, on cherche à identifier un vecteur $\boldsymbol{x}\in\mathbb{N}^n$ qui maximise ou minimise une *fonction objectif* linéaire sujet à des contraintes linéaires de cette forme:

$$\begin{aligned}\textbf{optimiser}\quad & \boldsymbol{c}^\mathsf{T}\cdot\boldsymbol{x},\\ \textbf{sujet à}\quad & \mathbf{A}\cdot\boldsymbol{x}\leq\boldsymbol{b}\text{ et }\boldsymbol{x}\in\mathbb{N}^n.\end{aligned}$$

Par exemple, l'instance $m=10$ et $s=[1,5,7]$ du problème du retour de monnaie s'exprime par:

$$\textbf{minimiser } x+y+z \textbf{ sujet à } x+5y+7z=10.$$

L'instance $v=[50,5,65,10,12,20]$, $p=[700,320,845,70,420,180]$, $c=900$ du problème du sac à dos s'exprime quant à elle par:

$$\begin{aligned}\textbf{maximiser}\quad & 50\,x_1+\ \ 5\,x_2+\ \ 65\,x_3+10\,x_4+\ \ 12\,x_5+\ \ 20\,x_6\\ \textbf{sujet à}\quad & 700\,x_1+320\,x_2+845\,x_3+70\,x_4+420\,x_5+180\,x_6\leq 900.\end{aligned}$$

Ces programmes peuvent être résolus efficacement en pratique par des solveurs, comme CPLEX et Gurobi, qui exploitent la force brute combinée à des heuristiques. Par exemple, voici la formulation d'une instance du problème du sac à dos et la formulation d'une instance du problème de retour de monnaie avec le solveur z3.

6.6 Exercices

6.1) La *distance de Levenshtein* entre deux chaînes de caractères u et v, dénotée dist(u, v), est définie comme étant la plus petite quantité d'ajouts, de retraits et de modifications de lettres qui transforment u en v. Nous écrivons ε afin de dénoter la chaîne vide. Par exemple: dist$(ab, ac) = 1$, dist$(abc, ba) = 2$ et dist$(\varepsilon, ab) = 2$. Donnez un algorithme de force brute qui calcule la distance entre deux chaînes données. Pensez d'abord à une borne supérieure sur dist(u, v).

6.2) Donnez un algorithme qui identifie la plus plus longue sous-chaîne contiguë commune entre deux chaînes de caractères u et v. Par exemple, si $u = abcaba$ et $v = abaccab$, alors votre algorithme devrait retourner cab. Analysez sa complexité.

6.3) Revisitons le problème de l'exercice 6.2). Donnez cette fois un algorithme pour la variante du problème où la sous-chaîne commune n'a *pas* à être contiguë. Par exemple, si $u = abcaba$ et $v = abaccab$, votre algorithme devrait retourner $abcab$. Fonctionne-t-il en temps polynomial?

6.4) Un *chemin hamiltonien* est un chemin qui passe par chaque sommet d'un graphe *exactement une* fois. Donnez un algorithme qui détermine s'il existe un chemin hamiltonien entre deux sommets s et t d'un graphe $\mathcal{G}$.

6.5) Donnez un algorithme qui complète une grille partielle de *sudoku*.

6.6) Donnez un algorithme de force brute qui résout le problème de remplissage de bacs défini à l'exercice 4.6).

6.7) Le problème de factorisation consiste à identifier la décomposition d'un entier $m \geq 2$ en nombres premiers. On ne sait pas à ce jour s'il est possible de résoudre ce problème en temps polynomial. L'algorithme de force brute ci-dessous factorise pourtant en temps $\Theta(m)$ dans le pire cas. Expliquez pourquoi c'est le cas et pourquoi cela ne contredit pas le problème ouvert.

Entrées : nombre $m \in \mathbb{N}_{\geq 2}$
Sorties : nombres premiers $p_1, \ldots, p_k$ tels que $m = p_1 \cdots p_k$

$p \leftarrow [\,]$; $d \leftarrow 2$
tant que $m > 1$
  si m mod $d = 0$ **alors**
    $m \leftarrow m \div d$
    ajouter d **à** p
  sinon
    $d \leftarrow d + 1$
retourner p

Remarque.

L'existence d'un algorithme de factorisation polynomial pourrait avoir de sérieuses conséquences sur la sécurité de certains protocoles cryptographiques. L'algorithme de Shor permet de factoriser en temps polynomial, mais sur un ordinateur quantique.

7

Programmation dynamique

La *programmation dynamique* constitue une approche algorithmique qui s'apparente à l'approche diviser-pour-régner et qui surmonte certaines limitations des stratégies gloutonnes et par force brute. Elle repose sur le principe d'optimalité de Bellman qui affirme essentiellement qu'une solution de certains problèmes s'exprime en fonction de celles de sous-problèmes. Nous présentons la programmation dynamique en revisitant certains problèmes et en considérant des problèmes de plus courts chemins dans les graphes.

7.1 Approche descendante

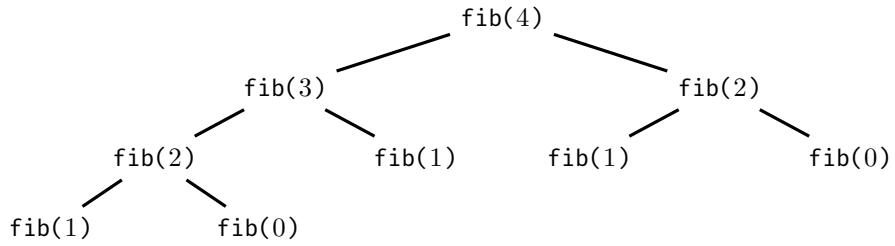
Plusieurs algorithmes récursifs, tels que ceux issus de la force brute, recalculent des solutions intermédiaires à répétition. L'algorithme 42 de calcul de la suite de Fibonacci forme un exemple classique de ce phénomène.

Algorithme 42 : Calcul d'un élément de la suite de Fibonacci.

Entrées : $n \in \mathbb{N}$
Résultat : $n^{\text{ème}}$ terme de la suite de Fibonacci

```
fib(n):
    si n ≤ 1 alors
        retourner n
    sinon
        retourner fib(n − 1) + fib(n − 2)
```

Par exemple, l'arbre des appels récursifs de `fib(4)` montre que les valeurs de `fib(2)`, `fib(1)` et `fib(0)` sont recalculées plusieurs fois:



Nous pouvons éviter ces calculs redondants à l'aide de la *mémoïsation*: on stocke chaque valeur nouvellement calculée, par ex. à l'aide d'un tableau associatif, et on retourne cette valeur lorsqu'elle est requise. Par exemple, l'algorithme 43 utilise la mémoïsation pour le calcul de la suite de Fibonacci. Notons que le gain en temps offert par la mémoïsation augmente l'usage de mémoire. Cependant, le gain en temps est souvent exponentiel alors que l'augmentation en mémoire peut être polynomiale voire linéaire.

Algorithme 43 : Calcul de la suite de Fibonacci avec mémoïsation.

Entrées : $n \in \mathbb{N}$
Résultat : $n^{\text{ème}}$ terme de la suite de Fibonacci

$mem \leftarrow [\,]$
`fib(`n`):`
  si mem *ne contient pas* n **alors**
    si $n \leq 1$ **alors**
      $mem[n] \leftarrow n$
    sinon
      $mem[n] \leftarrow$ `fib(`$n-1$`)` $+$ `fib(`$n-2$`)`
  retourner $mem[n]$

7.2 Approche ascendante

La programmation dynamique est probablement mieux connue sous sa forme *ascendante*: on résout les sous-problèmes itérativement des plus petites instances aux plus grandes, généralement en stockant les résultats intermédiaires dans un tableau.

7.2.1 Problème du retour de monnaie

Afin d'illustrer l'approche ascendante, reconsidérons le problème du retour de monnaie. Supposons que nous cherchions à rendre le montant $m = 10$ dans le système monétaire $s = [1, 5, 7]$. Nous considérons les sous-problèmes suivants: « quelle est la plus petite quantité de pièces afin de rendre le montant j avec

les i premières pièces du système? » Nous inscrivons la réponse à ces questions dans un tableau T:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0 (sans pièce)	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1 (avec pièce 1)	0	1	2	3	4	5	6	7	8	9	10
2 (avec pièce 5)	0	1	2	3	4	1	2	3	4	5	2
3 (avec pièce 7)	0	1	2	3	4	1	2	1	2	3	2

Par exemple, nous avons $T[2, 7] = 3$ car on peut retourner le montant 7 avec trois pièces: $1 + 1 + 5$. L'entrée $T[3, 7] = 1$ raffine cette solution car on peut rendre la pièce 7 maintenant qu'elle est permise. La solution au problème de départ apparaît dans le coin inférieur droit: $T[3, 10] = 2$.

Algorithme 44 : Programmation dynamique pour retour de monnaie.

Entrées : montant $m \in \mathbb{N}$, séquence s de $n \in \mathbb{N}$ pièces
Résultat : nombre minimal de pièces afin de rendre m

```
monnaie-dyn(m, s):
    // T[i, j] = # pièces pour rendre j avec s[1 : i]
    initialiser tableau T[0..n, 0..m] avec ∞
    // Aucun pièce pour rendre 0
    T[0, 0] ← 0
    // Calculer # pièces progressivement
    pour i ∈ [1..n]
        pour j ∈ [0..m]
            sans ← T[i − 1, j]                    // sol. sans s[i]
            avec ← ∞
            si j ≥ s[i] alors avec ← T[i, j − s[i]] + 1  // sol. avec s[i]
            T[i, j] ← min(sans, avec)
    retourner T[n, m]
```

Afin de remplir le tableau algorithmiquement, nous exploitons l'identité:

$$\begin{aligned}
T[i, 0] &= 0, && \text{si } 0 \leq i \leq n,\\
T[0, j] &= \infty && \text{si } 1 \leq j \leq n,\\
T[i, j] &= \min(T[i-1, j],\ T[i, j - s[i]] + 1) && \text{sinon,}
\end{aligned}$$

où nous considérons $T[i, j - s[i]] = \infty$ lorsque $j < s[i]$. En mots:

— on peut toujours rendre le montant 0 avec aucune pièce,

— on ne peut pas rendre d'autre montant sans pièce,

— pour rendre le montant j avec les i premières pièces: ou bien on n'utilise pas la $i^{\text{ème}}$ pièce; ou bien on l'utilise au moins une fois, ce qui réduit le montant à rendre de $s[i]$ et augmente le nombre de pièces utilisées de 1.

L'algorithme 44 implémente cette procédure en remplissant le tableau de la gauche vers la droite et du haut vers le bas.

7.2.2 Problème du sac à dos

Utilisons maintenant la programmation dynamique afin de résoudre le problème du sac à dos. Posons cette instance:

$$\begin{aligned} v &:= [5, 3, 4] \quad \text{(valeurs)}, \\ p &:= [4, 2, 3] \quad \text{(poids)}, \\ c &:= 8 \qquad\quad \text{(capacité)}. \end{aligned}$$

Nous considérons les sous-problèmes suivants: « quelle est la valeur maximale d'un sac à dos de capacité j qu'on peut remplir avec les i premiers objets? » Nous inscrivons la réponse à ces questions dans un tableau T:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0 (sans objet)	0	0	0	0	0	0	0	0	0
1 (avec objet 1)	0	0	0	0	5	5	5	5	5
2 (avec objet 2)	0	0	3	3	5	5	8	8	8
3 (avec objet 3)	0	0	3	4	5	7	8	9	9

Par exemple, nous avons $T[2, 5] = 5$ puisqu'on ne peut pas prendre les deux premiers objets simultanément et puisque le premier objet possède la valeur maximale entre ces deux objets. L'entrée $T[3, 5] = 7$ raffine cette solution car on peut prendre les objets 2 et 3 pour une valeur combinée de 7. La solution au problème de départ apparaît dans le coin inférieur droit: $T[3, 8] = 9$.

Afin de remplir le tableau algorithmiquement, nous exploitons l'identité:

$$\begin{aligned} &T[i, 0] = 0, && \text{si } 0 \leq i \leq n, \\ &T[0, j] = 0 && \text{si } 0 \leq j \leq n, \\ &T[i, j] = \max(T[i-1, j],\ T[i-1, j-p[i]] + v[i]) && \text{sinon}, \end{aligned}$$

où nous considérons $T[i, j - p[i]] = 0$ lorsque $j < p[i]$. En mots:

— on ne peut rien mettre dans un sac de capacité 0,

— on ne peut rien mettre dans un sac s'il n'y a aucun objet,

— pour remplir un sac de capacité j avec les i premiers objets: ou bien on n'utilise pas le $i^{\text{ème}}$ objet; ou bien on l'utilise une unique fois, ce qui réduit le poids du sac de $p[i]$ et augmente sa valeur de $v[i]$.

L'algorithme 45 implémente cette procédure en remplissant une ligne à la fois.

Algorithme 45 : Prog. dynamique pour le problème du sac à dos.

Entrées : valeurs $v \in \mathbb{N}^n$, poids $p \in \mathbb{N}^n$ et capacité $c \in \mathbb{N}$
Résultat : valeur maximale du sac à dos

```
sac-à-dos-dyn(v, p, c):
    // T[i, j] = valeur max. avec objets s[1 : i] et capacité j
    initialiser tableau T[0..n, 0..c] avec 0
    // Calculer valeur du sac à dos progressivement
    pour i ∈ [1..n]
        pour j ∈ [1..c]
            sans ← T[i − 1, j]                          // sans obj. i
            avec ← 0
            si j ≥ p[i] alors avec ← T[i − 1, j − p[i]] + v[i]  // + obj. i
            T[i, j] ← max(sans, avec)
    retourner T[n, c]
```

7.3 Plus courts chemins

Soit $\mathcal{G}$ un graphe pondéré par une séquence de poids entiers p. Le *poids* d'un chemin $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \cdots \xrightarrow{e_n} v_n$ correspond à $p[e_1] + p[e_2] + \ldots + p[e_n]$. Autrement dit, le poids d'un chemin allant de v_0 vers v_n, en empruntant les arêtes $e_1, e_2, \ldots, e_n$, correspond à la somme des poids des arêtes traversées. Dans ce contexte, nous parlons de *distance* ou de *longueur* plutôt que de poids. Nous nous intéressons au calcul de *plus courts chemins*, c'est-à-dire de chemins qui minimisent la distance entre deux sommets. Par exemple, il y a deux plus courts chemins de a vers e à la figure 7.1: $a \to b \to d \to e$ et $a \to b \to e$ qui sont de longueur 6. Remarquons que la notion de plus court chemin n'est bien définie que s'il n'existe aucun cycle de longueur négative. Ainsi, nous considérons les plus courts chemins comme étant *simples*.

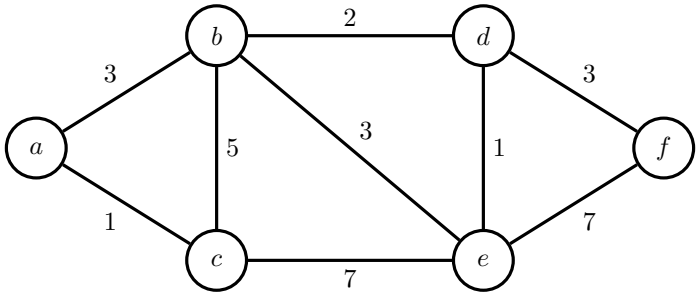


FIGURE 7.1 – Exemple de graphe (non dirigé) pondéré.

7.3.1 Algorithme de Dijkstra

Nous présentons l'*algorithme de Dijkstra* qui calcule la distance du plus court chemin d'un sommet de départ s vers tous les autres sommets du graphe. Cet algorithme fonctionne en attribuant une distance partielle à chaque sommet qu'on raffine itérativement jusqu'à l'obtention de la distance minimale. Plus précisément:

— on attribue une distance partielle $d[v]$ à chaque sommet v: 0 à s et ∞ aux autres sommets. La distance partielle d'un sommet v indique la distance minimale de s vers v identifiée jusqu'ici;

— on choisit un sommet u non marqué dont la distance partielle est minimale parmi tous les sommets, puis on marque ce sommet;

— pour chaque voisin/successeur v de u, on raffine $d[v]$ par $d[u] + p[u, v]$ si cette valeur est inférieure à $d[v]$. Autrement dit, on considère le plus court chemin de s vers u suivi de l'arête de u vers v. Si ce chemin est plus court que le meilleur chemin connu de s vers v, alors on a découvert un chemin plus court;

— On répète tant qu'il existe au moins un sommet non marqué.

L'algorithme 46 décrit cette procédure sous forme de pseudocode de haut niveau.

Algorithme 46 : Algorithme de Dijkstra.

Entrées : graphe $\mathcal{G} = (V, E)$ pondéré par une séquence p de poids non négatifs, et un sommet de départ $s \in V$
Résultat : séquence d t.q. $d[v]$ indique la longueur d'un plus court chemin de s vers v

$d \leftarrow [v \mapsto \infty : v \in V]$
$d[s] \leftarrow 0$
tant que $\exists$ *un sommet* u *non marqué t.q.* $d[u] \neq \infty$
  choisir $u \in V$ t.q. $d[u]$ est min. parmi les sommets non marqués
  marquer u
  pour $v : u \to v$
    $d[v] \leftarrow \min(d[v], d[u] + p[u, v])$
retourner d

Remarquons que si certains sommets sont inaccessibles à partir de s, alors leur distance demeure à ∞ jusqu'à la terminaison. Ainsi, l'algorithme de Dijkstra permet aussi d'identifier l'ensemble d'accessibilité de s.

Identification des chemins. La procédure telle que décrite à l'algorithme 46 ne construit pas les plus courts chemins. Afin de les construire, on peut stocker le

prédecesseur de chaque sommet qui a mené à sa distance (partielle). Autrement dit, à chaque évaluation de

$$\min(d[v], d[u] + p[u, v]),$$

si la valeur est réduite, alors le prédecesseur de v devient u. La figure 7.2 donne une trace de l'exécution de l'algorithme de Dijkstra sur le graphe de la figure 7.1, incluant la construction des prédecesseurs.

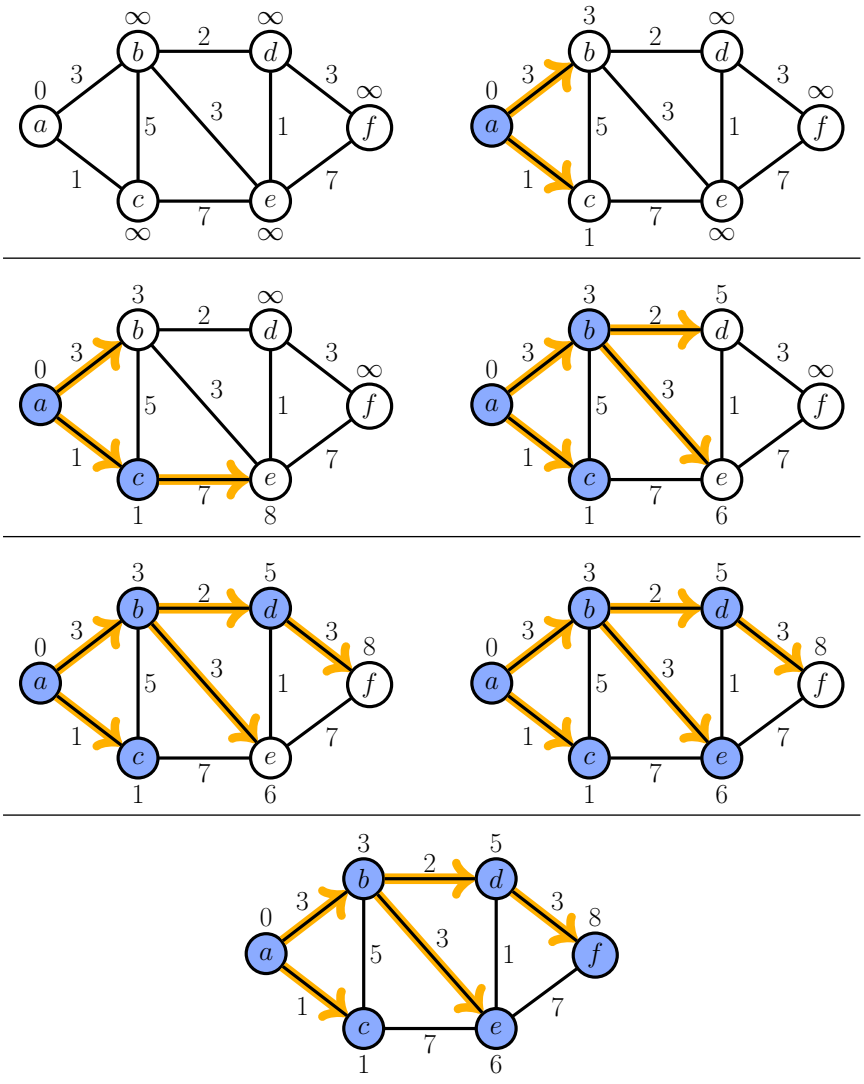


FIGURE 7.2 – Exemple d'exécution de l'algorithme de Dijkstra.

Correction. On peut montrer que l'algorithme de Dijkstra est correct en démontrant la proposition suivante par induction:

Proposition 24. *La boucle principale de l'algorithme de Dijkstra satisfait cet invariant: pour tout sommet $v \in V$,*

- *si v est marqué, alors $d[v]$ est la longueur minimale de s vers v;*
- *si v n'est pas marqué, alors $d[v]$ est la longueur minimale de s vers v parmi les chemins qui n'utilisent que les sommets marqués et v.*

Complexité. La boucle principale de l'algorithme 46 est exécutée au plus $|V|$ fois puisque les sommets marqués ne sont plus considérés. L'identification d'un sommet non marqué u qui minimise $d[u]$ peut se faire en temps $\mathcal{O}(|V|)$ avec une implémentation naïve où on itère sur tous les sommets. Le voisinage de chaque sommet u est exploré au plus une fois. Au total, nous obtenons donc un temps d'exécution de:

$$\mathcal{O}\left(|V| \cdot |V| + \sum_{u \in V} \deg^+(u)\right) = \mathcal{O}\big(|V|^2 + |E|\big)\,.$$

Le goulot d'étranglement de l'algorithme se situe à l'identification du sommet non marqué. On peut améliorer le temps d'exécution à l'aide d'une structure de données plus sophistiquée: un *monceau de Fibonacci*. Ce type de monceau offre notamment ces opérations:

- ajout d'un élément (en temps constant),
- retrait du plus petit élément (en temps logarithmique amorti),
- diminution d'une clé (en temps constant amorti).

Nous pouvons donc initialiser, en temps $\mathcal{O}(|V|)$, un monceau tel que la clé de chaque sommet v est $d[v]$. L'instruction **choisir** peut ainsi être implémentée en retirant le plus petit élément du monceau. Lorsqu'on met $d[v]$ à jour, on met également la clé à jour dans le monceau (qui ne peut que décroître). Comme le retrait et la mise à jour prennent un temps constant et logarithmique amorti, cela raffine le temps d'exécution total à:

$$\mathcal{O}\left(|V| \cdot \log|V| + \sum_{u \in V} \deg^+(u)\right) = \mathcal{O}(|V| \log|V| + |E|).$$

Poids négatifs. L'algorithme de Dijkstra suppose que le poids des arêtes sont non négatifs, autrement l'algorithme peut échouer. Par exemple, considérons le graphe de la figure 7.3 à partir du sommet a.

L'algorithme marque les sommets dans l'ordre $[a, e, b, d, c]$ et retourne notamment la distance $d[e] = 2$, alors que le plus court chemin de a vers e est de longueur $3 + 2 - 2 - 2 = 1$.

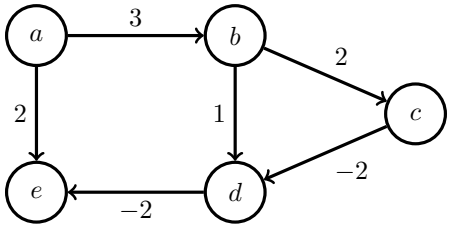


FIGURE 7.3 – Graphe avec poids négatifs sur lequel l'algo. de Dijkstra échoue.

7.3.2 Algorithme de Floyd-Warshall

Nous présentons maintenant l'algorithme de *Floyd-Warshall* qui permet d'identifier les plus courts chemins entre *toutes* les paires de sommets d'un graphe, et ce sans restriction de non négativité sur les poids. Cet algorithme exploite l'observation suivante: si on découpe un plus court chemin de v_i vers v_j, alors on obtient un plus court chemin de v_i vers un sommet intermédiaire v_k, ainsi qu'un plus court chemin de v_k vers v_j.

Algorithme 47 : Algorithme de Floyd-Warshall.

Entrées : graphe $\mathcal{G} = (V, E)$ pondéré par une séquence p de poids entiers (sans cycle négatif), où $V = \{v_1, v_2, \ldots, v_n\}$
Résultat : matrice d t.q. $d[u, v]$ indique la longueur d'un plus court chemin de u vers v

$d \leftarrow [(u, v) \mapsto \infty : u, v \in V]$
pour $v \in V$ `// chemins vides`
  $d[v, v] \leftarrow 0$
pour $(u, v) \in E$ `// chemins directs`
  $d[u, v] \leftarrow p[u, v]$
pour $k \in [1..n]$ `// autres chemins`
  pour $i \in [1..n]$
    pour $j \in [1..n]$
      $d[v_i, v_j] \leftarrow \min(d[v_i, v_j], d[v_i, v_k] + d[v_k, v_j])$
retourner d

Exemple.

Considérons le graphe illustré à la figure 7.4. En exécutant l'algorithme de Floyd-Warshall, nous obtenons la trace suivante:

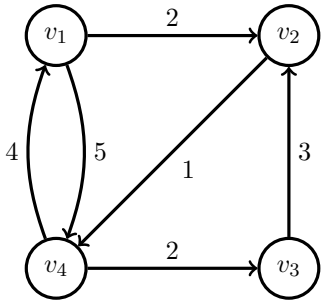


FIGURE 7.4 – Exemple de graphe (dirigé) pondéré.

—	v_1	v_2	v_3	v_4
v_1	0	2	∞	5
v_2	∞	0	∞	1
v_3	∞	3	0	∞
v_4	4	∞	2	0

$k=1$	v_1	v_2	v_3	v_4
v_1	0	2	∞	5
v_2	∞	0	∞	1
v_3	∞	3	0	∞
v_4	4	6	2	0

$k=2$	v_1	v_2	v_3	v_4
v_1	0	2	∞	3
v_2	∞	0	∞	1
v_3	∞	3	0	4
v_4	4	6	2	0

$k=3$	v_1	v_2	v_3	v_4
v_1	0	2	∞	3
v_2	∞	0	∞	1
v_3	∞	3	0	4
v_4	4	5	2	0

$k=4$	v_1	v_2	v_3	v_4
v_1	0	2	5	3
v_2	5	0	3	1
v_3	8	3	0	4
v_4	4	5	2	0

Ainsi, par exemple, la distance minimale de v_4 vers v_2 est de 5, et la distance minimale de v_3 vers v_1 est de 8.

Identification des chemins. La procédure telle que décrite à l'algorithme 47 ne construit pas les plus courts chemins. Afin de les construire, on utilise la même approche que pour l'algorithme de Dijkstra: on construit simultanément une matrice $pred$ où $pred[u, v]$ indique le sommet qui a mené à la valeur $d[u, v]$.

Complexité. La construction de la matrice d requiert un temps de:

$$\Theta(|V|^2 + |V| + |E| + |V|^3).$$

Puisque $|E| \in \mathcal{O}(|V|^2)$, l'algorithme fonctionne donc en temps $\Theta(|V|^3)$.

Correction. Pour tous sommets $u, v \in V$, définissons $\delta_k(u, v)$ comme étant la longueur d'un plus court chemin de u vers v dont les sommets intermédiaires appartiennent à $\{v_1, v_2, \ldots, v_k\}$. Nous avons:

Proposition 25. *La boucle principale de l'algorithme de Floyd-Warshall satisfait cet invariant:* $d[u, v] = \delta_{k-1}(u, v)$ *pour tous* $u, v \in V$.

Corollaire 3. *L'algorithme de Floyd-Warshall est correct.*

Démonstration. Soient $u, v \in V$. À la sortie de la boucle principale, k vaut $n+1$. Par la proposition 25, nous avons donc $d[u, v] = \delta_{n+1-1}(u, v) = \delta_n(u, v)$. Ainsi, par définition de $\delta_n(u, v)$, $d[u, v]$ dénote la longueur d'un plus court chemin de u vers v dont les sommets intermédiaires appartiennent à V. □

Détection de cycle négatif. On peut adapter l'algorithme de Floyd-Warshall afin de détecter la présence d'un cycle négatif:

- on exécute l'algorithme (tel quel, sans modification),
- on vérifie s'il existe un sommet $v \in V$ tel que $d[v, v] < 0$,
- si c'est le cas, il existe un cycle négatif qui passe par v, autrement, il n'existe aucun cycle négatif.

Accessibilité. On peut adapter l'algorithme de Floyd-Warshall afin de calculer la relation d'accessibilité [1] $\xrightarrow{*}$ d'un graphe (non pondéré), en remplaçant:

- le poids $p[u, v]$ par `vrai` si $u = v$ ou $u \to v$, et `faux` sinon,
- l'addition par l'opération $\wedge$,
- le minimum par l'opération $\vee$.

À la sortie, on obtient une matrice booléenne d telle que $d[u, v] =$ `vrai` ssi $u \xrightarrow{*} v$. Cette modification est décrite à l'algorithme 48.

Remarque.

L'algorithme 47 (avec poids) est attribué à *Robert W. Floyd*, alors que l'algorithme 48 (sans poids) est attribué à *Stephen Warshall*.

Algorithme 48 : Algorithme de calcul de relation d'accessibilité.

Entrées : graphe $\mathcal{G} = (V, E)$ où $V = \{v_1, v_2, \ldots, v_n\}$
Résultat : matrice d t.q. $d[u, v]$ indique si $u \xrightarrow{*} v$

$d \leftarrow [(u, v) \mapsto \texttt{faux} : u, v \in V]$
pour $v \in V$ `// chemins vides`
  $d[v, v] \leftarrow \texttt{vrai}$
pour $(u, v) \in E$ `// chemins directs`
  $d[u, v] \leftarrow \texttt{vrai}$
pour $k \in [1..n]$ `// autres chemins`
  pour $i \in [1..n]$
    pour $j \in [1..n]$
      $d[v_i, v_j] \leftarrow d[v_i, v_j] \vee (d[v_i, v_k] \wedge d[v_k, v_j])$
retourner d

Algorithme 49 : Algorithme de Bellman-Ford.

Entrées : graphe $\mathcal{G} = (V, E)$ pondéré par une séquence p de poids entiers (sans cycle négatif), et un sommet de départ $s \in V$
Résultat : séquence d t.q. $d[v]$ indique la longueur d'un plus court chemin de s vers v

$d \leftarrow [v \mapsto \infty : v \in V]$
$d[s] \leftarrow 0$
faire $|V| - 1$ **fois**
  pour *chaque arête* $u \to v$
    $d[v] \leftarrow \min(d[v], d[u] + p[u, v])$
retourner d

7.3.3 Algorithme de Bellman-Ford

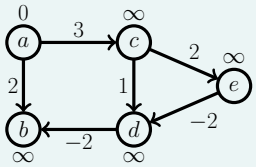
Nous présentons une alternative à l'algorithme de Dijkstra qui permet la présence de poids négatifs: l'*algorithme de Bellman-Ford*.
Cet algorithme associe une distance partielle à chaque sommet: 0 pour le sommet de départ et ∞ pour les autres sommets. Il raffine ensuite itérativement ces distances en explorant les chemins d'au moins une arête, deux arêtes, trois arêtes, etc. Puisque tout chemin simple est de longueur au plus $|V| - 1$, on cesse de raffiner après $|V| - 1$ itérations. L'algorithme 49 décrit cette procédure.

L'identification des chemins requiert simplement l'ajout d'une séquence de prédecesseurs comme pour les autres algorithmes.

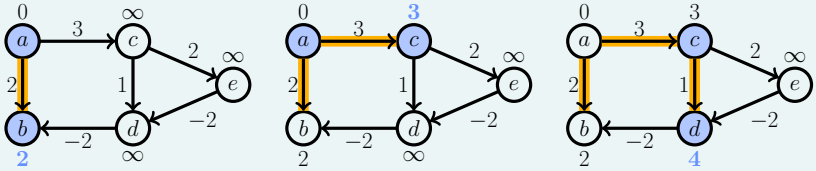
1. Aussi connue sous le nom de *clôture réflexive transitive* de la relation $\to$.

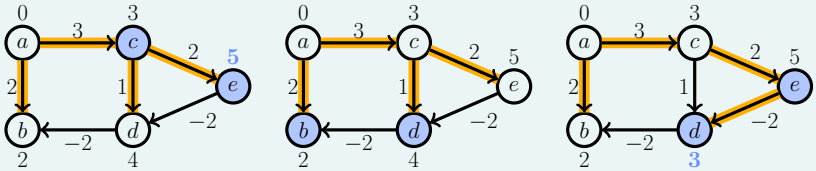
Exemple.

Exécutons l'algorithme de Bellman-Ford sur le graphe dirigé de la figure 7.3, à partir du sommet a, en choisissant les arêtes en ordre alphabétique: $[(a,b),(a,c),(c,d),(c,e),(d,b),(e,d)]$. Dans la visualisation ci-dessous, l'arête actuellement considérée a ses deux sommets colorés; la valeur $d[v]$ annote chaque sommet v; et l'arête qui a mené à $d[v]$ est surlignée.

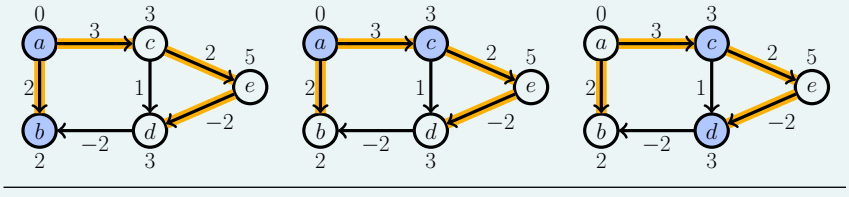


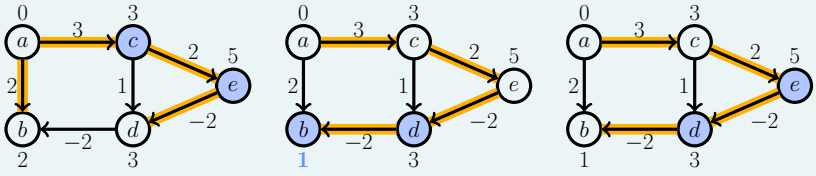
Itération 1





Itération 2

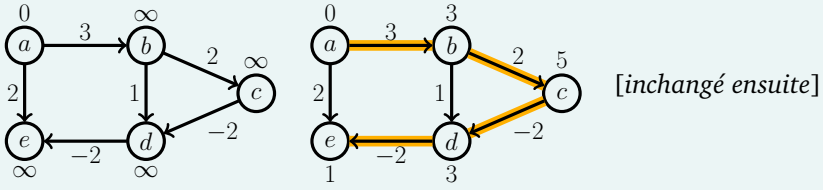




Inchangé ensuite

Remarquons que l'ordre dans lequel les arêtes sont considérées a une incidence sur l'exécution de l'algorithme. Par exemple, reconsidé-

rons le graphe précédent en choisissant les arêtes dans cet ordre: $[(a,b), (b,c), (c,d), (d,e), (a,e), (b,d)]$. Après considération des quatre premières arêtes, les plus courts chemins sont déjà découverts. Ainsi, les valeurs ne peuvent plus être améliorées après la première itération.



Complexité. Le temps d'exécution de l'algorithme appartient à

$$\Theta(|V| + 1 + (|V| - 1) \cdot |E|) = \Theta(|V| \cdot |E|).$$

Correction. Soit $\delta_i(v)$ la longueur d'un plus court chemin de s vers v utilisant au plus i arêtes. On peut montrer que l'algorithme de Bellman-Ford est correct en démontrant la proposition suivante par induction:

Proposition 26. *Après i exécutions de la boucle principale de l'algorithme de Bellman-Ford, pour tout sommet $v \in V$,*

- *si $d[v] \neq \infty$, alors $d[v]$ correspond à la longueur d'un chemin de s vers v,*
- $d[v] \leq \delta_i(v)$.

Détection de cycle négatif. On peut adapter l'algorithme afin d'identifier la présence d'un cycle négatif en lui ajoutant le pseudocode suivant:

```
pour chaque arête u → v
    si d[u] + p[u, v] < d[v] alors
        retourner cycle négatif détecté
```

En effet, si $d[u] + p[u,v] < d[v]$, alors une itération supplémentaire aurait diminué $d[v]$. Or, cela est impossible en l'absence d'un cycle négatif.

7.3.4 Sommaire

Voici un sommaire des trois algorithmes de plus courts chemins introduits:

	Dijkstra	**Bellman-Ford**	**Floyd-Warshall**
Types de chemins	d'un sommet vers les autres		paires de sommets
Poids négatifs?	✗	✓	✓
Temps d'exécution	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$	$\Theta(\|V\| \cdot \|E\|)$	$\Theta(\|V\|^3)$
Temps (si $\|E\| \in \Theta(1)$)	$\mathcal{O}(\|V\| \log \|V\|)$	$\Theta(\|V\|)$	$\Theta(\|V\|^3)$
Temps (si $\|E\| \in \Theta(\|V\|)$)	$\mathcal{O}(\|V\| \log \|V\|)$	$\Theta(\|V\|^2)$	$\Theta(\|V\|^3)$
Temps (si $\|E\| \in \Theta(\|V\|^2)$)	$\mathcal{O}(\|V\|^2)$	$\Theta(\|V\|^3)$	$\Theta(\|V\|^3)$

7.4 Exercices

7.1) La quantité de mémoire utilisée par l'algorithme 44 appartient à $\mathcal{O}(m \cdot n)$. Adaptez l'algorithme afin de la réduire à $\mathcal{O}(m)$.

7.2) L'algorithme 44 et l'algorithme 45 retournent la valeur de la solution optimale mais pas les pièces/objets choisis. Adaptez ces algorithmes afin qu'ils retournent l'ensemble des pièces/objets.

7.3) Donnez un algorithme de programmation dynamique qui calcule la distance entre deux chaînes données, telle que définie à l'exercice 6.1).

7.4) Donnez un algorithme de programmation dynamique qui identifie une sous-chaîne contiguë de longueur maximale entre deux chaînes données; voir l'exercice 6.2) pour un exemple. Analysez sa complexité.

7.5) Donnez un algorithme de programmation dynamique qui identifie une sous-chaîne (pas nécessairement contiguë) de longueur maximale entre deux chaînes données; voir l'exercice 6.3) pour un exemple. Analysez sa complexité.

7.6) Identifiez une famille de graphes pour laquelle il existe un nombre exponentiel de plus courts chemins entre deux des sommets.

7.7) Montrez que si un graphe ne possède pas de cycle négatif et qu'il existe un plus court chemin entre deux sommets, alors il en existe un *simple*.

7.8) Adaptez le pseudocode des algorithmes de Dijkstra, Floyd-Warshall et Bellman-Ford afin de construire des plus courts chemins (et non seulement les distances).

7.9) Comment pourrait-on adapter l'algorithme de Floyd-Warshall afin de calculer la relation $\xrightarrow{+}$ d'un graphe? Cette relation est définie par:

$$u \xrightarrow{+} v \iff \text{il existe un chemin } \textit{non vide} \text{ de } u \text{ vers } v.$$

7.10) Modifiez l'algorithme de Bellman-Ford afin qu'il puisse parfois terminer plus rapidement qu'en $|V| - 1$ itérations.

7.11) Si nous voulons seulement identifier un plus court chemin d'un sommet s vers un sommet t, à quel moment pouvons-nous arrêter l'exécution de l'algorithme de Dijkstra?

7.12) Pouvons-nous adapter l'algorithme de Dijkstra afin de déterminer la distance minimale entre chaque paire de sommets? Si c'est le cas, y a-t-il un avantage en comparaison à l'algorithme de Floyd-Warshall? Sinon, pourquoi?

7.13) L'algorithme de Dijkstra ne fonctionne pas sur les graphes avec des poids négatifs. Peut-on le faire fonctionner avec le prétraitement suivant?

Prétraitement: on remplace le poids $p[e]$ de chaque arête e par le nouveau poids $p[e] + |d|$, où d est le plus petit poids négatif du graphe?

7.14) Deux personnes situées dans des villes distinctes veulent se rencontrer. Elles désirent le faire le plus rapidement possible et décident donc de se rejoindre dans une ville intermédiaire. Expliquez comment identifier cette ville algorithmiquement. Considérez un graphe pondéré $\mathcal{G} = (V, E)$ où V est l'ensemble des villes, et où chaque arête $u \to v$ de poids d représente une route directe de u vers v dont le temps (idéalisé) pour la franchir est de d minutes.

(*tiré de [Eri19, chap. 8, ex. 14]* 🅭)

7.15) Nous pouvons raffiner la notion de plus court chemin en minimisant d'abord le poids d'un chemin, puis son nombre d'arêtes. Par exemple, il n'y a qu'*un seul* plus court chemin de a vers e à la figure 7.1, puisque le chemin $a \to b \to e$ possède moins d'arêtes que le chemin $a \to b \to d \to e$, bien qu'ils soient tous deux de poids minimal. Donnez un algorithme qui identifie des plus courts chemins (sous la nouvelle définition) d'un sommet de départ s vers tous les autres sommets.

(*tiré de [Eri19, chap. 8, ex. 12]* 🅭)

7.16) Un *ensemble indépendant* d'un graphe non dirigé $\mathcal{G} = (V, E)$ est un ensemble $U \subseteq V$ tel que $u, v \in U$ implique $\{u, v\} \notin E$. Autrement dit, un ensemble indépendant est un ensemble de sommets qui ne sont pas reliés. Donnez un algorithme qui identifie, en temps polynomial, un ensemble indépendant de *taille* maximale lorsque $\mathcal{G}$ est un arbre.

(*tiré des notes de cours de Manuel Lafond*)

7.17) Considérons un graphe non dirigé $\mathcal{G} = (V, E)$ pondéré par p. Il existe un plus plus court chemin de $s \in V$ vers $t \in V$ ssi s et t appartiennent à la même composante connexe C, et C ne contient aucun poids négatif. Pourquoi?

7.18) Considérons un graphe $\mathcal{G} = (V, E)$ dirigé et pondéré, où toute arête possède le même poids $c \in \mathbb{Z}$. Il est possible de calculer la distance (minimale) d'un sommet $s \in V$ vers tous les autres en temps $\mathcal{O}(|V| + |E|)$. Pourquoi?

7.19) Considérons ce problème:

ENTRÉE:	un graphe $\mathcal{G} = (V, E)$ pondéré par p, deux sommets $s, t \in V$
SORTIE:	longueur maximale parmi les chemins simples de s vers t dans $\mathcal{G}$

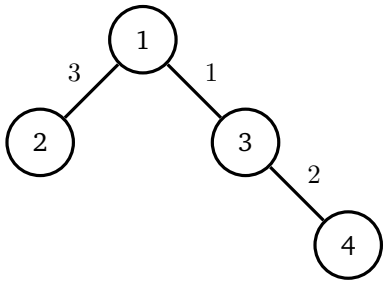
Pour le résoudre, on pourrait être tenté de multiplier chaque poids $p[e]$

par -1, puis d'identifier un plus court chemin de s vers t. Pourquoi est-ce que cette approche ne fonctionne pas?

Remarque.

Ce problème est NP-complet; on ne connaît donc aucun algorithme qui le résout en temps polynomial.

7.20) ★ Nous définissons la *mesure de centralité* d'un sommet u d'un graphe (V, E) par $\sum_{v \in V} \text{dist}(u, v)$, où dist dénote la longueur d'un plus court chemin. Donnez un algorithme qui reçoit un arbre (V, E) et qui retourne la mesure de centralité de chacun de ses sommets en temps $\mathcal{O}(|V| + |E|)$. Par exemple, il devrait retourner $[7, 13, 7, 11]$ sur cet arbre:



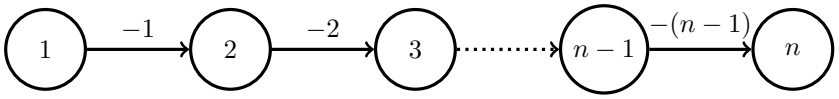
7.21) En classe virtuelle (A20), une personne m'a demandé si l'algorithme de Floyd-Warshall résout le problème du calcul de plus courts chemins d'un sommet vers tous les autres (comme Bellman-Ford) lorsqu'on retire sa seconde boucle. Une implémentation m'a presque faire croire que c'était le cas, mais ce ne l'est pas car l'algorithme raisonne trop localement. Montrez que l'approche ne fonctionne pas à l'aide d'un contre-exemple. Autrement dit, montrez que cet algorithme n'est pas correct:

Entrées : graphe $\mathcal{G} = (V, E)$ pondéré par une séquence p de poids entiers (sans cycle négatif), et un sommet de départ $s \in V$
Résultat : séquence d t.q. $d[v]$ indique la longueur d'un plus court chemin de s vers v

$d \leftarrow [(u, v) \mapsto \infty : u, v \in V]$
pour $v \in V$
  $d[v, v] \leftarrow 0$
pour $(u, v) \in E$
  $d[u, v] \leftarrow p[u, v]$
pour $k \in [1..n]$
  pour $j \in [1..n]$
    $d[s, v_j] \leftarrow \min(d[s, v_j], d[s, v_k] + d[v_k, v_j])$
retourner d

★ Montrez que l'algorithme est toutefois correct si $\mathcal{G}$ est un graphe dirigé acyclique et que V est ordonné topologiquement. Proposez également une simplification du pseudocode qui, dans ce cas, réduit la complexité de $\mathcal{O}(|V|^2)$ à $\mathcal{O}(|V| + |E|)$.

7.22) Considérons cette famille d'entrées:



Que se produit-il si l'on exécute l'algorithme de Bellman-Ford en considérant les arêtes dans ces ordres:

a) $(1, 2), (2, 3), \ldots, (n-1, n)$ (gauche à droite),

b) $(n-1, n), \ldots, (2, 3), (1, 2)$ (droite à gauche),

c) $(1, 2), (3, 4), \ldots$, puis $(2, 3), (4, 5), \ldots$ (impair puis pair)

7.23) Donnez un algorithme qui résout le problème de l'exercice 5.15) en temps linéaire, c.-à-d. qui reçoit une séquence s de n entiers et qui retourne la plus grande somme contigüe en temps $\mathcal{O}(n)$.

7.24) Donnez un algorithme qui résout le problème de l'exercice 5.15) à l'aide d'un algorithme de plus courts chemins.

8

Algorithmes et analyse probabilistes

Dans l'ensemble des chapitres précédents, nous avons étudié les algorithmes dits *déterministes*: les algorithmes dont la valeur de retour et le temps d'exécution ne diffèrent jamais sur une même entrée. Dans ce chapitre, nous considérons les *algorithmes probabilistes* ayant accès à une source d'aléa (idéalisée).

8.1 Nombres aléatoires

Considérons le scénario suivant: vous désirez jouer à un jeu de société qui requiert un dé à six faces, mais vous n'avez qu'une pièce de monnaie (non biaisée). Comment pouvez-vous simuler un dé à l'aide de votre pièce?

Une solution algorithmique simple consiste à:

— choisir trois bits aléatoires $y_2\, y_1\, y_0$ avec trois tirs à pile ou face;

— retourner x si le nombre binaire $y_2\, y_1\, y_0$ vaut $x \in [1, 6]$, et recommencer sinon, c.-à-d. lorsque $y_2 = y_1 = y_0 = 0$ ou $y_2 = y_1 = y_0 = 1$.

Cette procédure, décrite à l'algorithme 50, génère un nombre aléatoire $x \in [1, 6]$ de façon uniforme. Cependant, elle peut en théorie effectuer un nombre arbitraire d'itérations. Cherchons à identifier le « nombre moyen » de tirs à pile ou face effectués par l'algorithme.

Rappelons que l'*espérance* d'une variable aléatoire X, dont l'image est $\mathbb{N}$, est définie par

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} \Pr(X = i) \cdot i.$$

Celle-ci correspond intuitivement à la moyenne pondérée d'un grand nombre de résultats d'une expérience aléatoire. Dans notre cas, nous devons donc identifier la valeur $\mathbb{E}[X]$ où X est la variable aléatoire qui dénote le nombre d'itérations effectuées par la boucle principale.

Remarquons que chaque itération de la procédure est indépendante de la précédente. De plus, la probabilité de quitter la boucle à une itération donnée

Algorithme 50 : Lancer de dé à l'aide d'une pièce.

Entrées : —
Résultat : nombre $x \in [1, 6]$ choisi de façon aléatoire et uniforme

```
lancer-dé():
    faire
        choisir un bit y_0 à pile ou face
        choisir un bit y_1 à pile ou face
        choisir un bit y_2 à pile ou face
    tant que y_2 = y_1 = y_0
    retourner 4 · y_2 + 2 · y_1 + y_0
```

est de $p := 6/8 = 3/4$. Ainsi, avec probabilité p, on effectue un seul tour de boucle, et avec probabilité $(1 - p)$, on effectue un tour de boucle, plus $\mathbb{E}[X]$ tours supplémentaires. Nous avons donc:

$$\begin{aligned}\mathbb{E}[X] &= p \cdot 1 + (1 - p) \cdot (1 + \mathbb{E}[X]) \\ &= p + 1 + \mathbb{E}[X] - p - p \cdot \mathbb{E}[X] \\ &= (1 - p) \cdot \mathbb{E}[X] + 1.\end{aligned}$$

Ainsi, $p \cdot \mathbb{E}[X] = 1$ et par conséquent $\mathbb{E}[X] = 1/p$. Nous concluons donc que $\mathbb{E}[X] = 4/3$ et ainsi que le nombre espéré de tirs à pile ou face est de $3 \cdot (4/3) = 4$.

Observation.

Nous aurions pu obtenir l'espérance en observant que X suit une *loi géométrique* de paramètre $p = 3/4$ et ainsi que $\mathbb{E}[X] = 1/p = 4/3$.

Nous pouvons généraliser cette approche afin de générer un nombre $x \in [a, b]$ de façon uniforme. Remarquons d'abord que ce problème se réduit à générer un nombre appartenant à $[0, b - a]$, qu'on peut ensuite additionner à a. Nous supposons donc sans perte de généralité que l'intervalle débute à 0. Pour générer un nombre $x \in [0, c - 1]$:

— on choisit k bits, où k est assez grand pour représenter 0 à $c - 1$;

— on retourne la valeur x du nombre binaire si $x < c$, et recommence sinon.

Cette procédure est décrite sous forme de pseudocode à l'algorithme 51.

Soit X la variable aléatoire qui dénote le nombre d'itérations effectuées par la boucle principale pour une entrée c fixée. Comme pour l'algorithme précédent, chaque itération est indépendante des précédentes. De plus, la probabilité

Algorithme 51 : Génération de nombre aléatoire à l'aide d'une pièce.

Entrées : $c \in \mathbb{N}_{\geq 1}$
Résultat : nombre $x \in [0, c-1]$ choisi de façon aléatoire et uniforme

```
uniforme(c):
    k ← ⌈log c⌉
    faire
        x ← 0
        pour i ∈ [0..k − 1]
            choisir un bit y à pile ou face
            x ← x + 2^i · y
    tant que x ≥ c
    retourner x
```

de quitter la boucle pour une itération donnée est de $p \coloneqq c/2^k$. Ainsi:

$$\begin{aligned}
\mathbb{E}[X] &= 1/p && \text{(car } X \text{ suit une loi géométrique de paramètre } p\text{)} \\
&= 2^k/c && \text{(par définition de } p\text{)} \\
&= 2^{\lceil \log c \rceil}/2^{\log c} && \text{(par définition de } k \text{ et par } c = 2^{\log c}\text{)} \\
&= 2^{\lceil \log c \rceil - \log c} && \\
&< 2 && \text{(car } \lceil \log c \rceil - \log c < 1\text{).}
\end{aligned}$$

Puisque chaque itération effectue k tirs à pile ou face, le nombre espéré de tirs est de $\mathbb{E}[X] \cdot k < 2k = 2\lceil \log c \rceil$. Informellement, cela signifie qu'en « moyenne » l'algorithme lance $\mathcal{O}(\log c)$ fois une pièce pour générer un nombre.

Remarque.

En pratique, les ordinateurs n'ont généralement pas accès à une source d'aléa parfaite et utilisent plutôt des générateurs de nombres pseudo-aléatoires comme « *Mersenne Twister* ».

8.2 Paradigmes probabilistes

Afin d'illustrer deux paradigmes probabilistes, considérons ce problème:

ENTRÉE: une séquence s de taille paire dont la moitié des éléments sont égaux à $a \in \mathbb{N}$ et l'autre moitié à $b \in \mathbb{N}$, où $a \neq b$
SORTIE: $\max(s)$

Intuitivement, tout algorithme déterministe doit itérer sur au moins la moitié des éléments de s afin de retourner le maximum. Autrement dit, un algorithme

déterministe résout forcément ce problème en temps $\Omega(n)$ dans le pire cas. Nous présentons deux algorithmes probabilistes qui surmontent cette barrière.

8.2.1 Algorithmes de Las Vegas et temps espéré

Un *algorithme de Las Vegas* est un algorithme probabiliste qui retourne toujours le bon résultat, mais dont le temps d'exécution dépend des choix probabilistes. Par exemple, considérons l'algorithme 52. Celui-ci:

— choisit un élément aléatoire $s[i]$;
— retourne $\max(s[i], s[1])$ si $s[i] \neq s[1]$, et recommence sinon.

Algorithme 52 : Maximum probabiliste: Las Vegas.

Entrées : séquence s de taille n paire dont la moitié des éléments sont égaux à $a \in \mathbb{N}$ et l'autre moitié à $b \in \mathbb{N}$ où $a \neq b$
Résultat : $\max(s)$

```
max-las-vegas(s):
    boucler
        choisir i ∈ [1..n] de façon uniforme
        si s[i] > s[1] alors
            retourner s[i]
        sinon si s[i] < s[1] alors
            retourner s[1]
```

Ainsi, lorsque l'algorithme termine, la valeur retournée est forcément $\max(s)$. Toutefois, le temps d'exécution varie selon l'ordonnancement de s et les choix probabilistes de i. Nous pouvons néanmoins borner son « temps espéré ».

Soient $\mathcal{A}$ un algorithme probabiliste et Y_x la variable aléatoire qui dénote le nombre d'opérations élémentaires exécutées par $\mathcal{A}$ sur entrée x. Le *temps espéré de $\mathcal{A}$ (dans le pire cas)* est la fonction $t_{\text{esp}} : \mathbb{N} \to \mathbb{N}$ telle que:

$$t_{\text{esp}}(n) \coloneqq \max \{\mathbb{E}[Y_x] : \text{entrée } x \text{ de taille } n\}.$$

Soit X_s la variable aléatoire qui dénote le nombre d'itérations effectuées par la boucle principale de l'algorithme 52. À une itération donnée, la probabilité de choisir $s[i] \neq s[1]$ est de $p \coloneqq 1/2$, puisque la moitié des éléments sont égaux à $s[1]$. Comme chaque itération est indépendante de la précédente, nous avons à nouveau une loi géométrique de paramètre p, ce qui mène à $\mathbb{E}[X_s] = 1/p = 2$. Ainsi, le temps espéré de l'algorithme appartient à $\mathcal{O}(2) = \mathcal{O}(1)$, en supposant toutes les opérations élémentaires.

Remarquons que bien que le temps espéré soit constant, l'exécution de l'algorithme 52 peut être d'une durée arbitrairement grande avec faible probabilité, et infinie avec probabilité 0. De façon générale, on peut remédier à ce problème en arrêtant l'exécution d'un algorithme de Las Vegas après un certain nombre d'itérations. En contrepartie, l'algorithme indique alors qu'aucune valeur de retour n'a été identifiée.

8.2.2 Algorithmes de Monte Carlo et probabilité d'erreur

Un *algorithme de Monte Carlo* est un algorithme probabiliste dont le temps d'exécution peut être borné indépendamment des choix aléatoires, mais qui peut retourner des valeurs erronées. Par exemple, considérons l'algorithme 53 qui:

— choisit un élément aléatoire $s[i]$;

— retourne $s[i]$ si $s[i] > s[1]$, et recommence au plus 275 fois sinon.

L'algorithme ne retourne pas nécessairement la bonne valeur, par ex. avec une certaine malchance on pourrait obtenir $s[i] = s[1] = \min(s)$ à chaque itération, auquel cas la valeur de sortie serait le minimum plutôt que le maximum. Cependant, le nombre d'itérations ne peut jamais excéder 275. Ainsi, le temps d'exécution appartient à $\mathcal{O}(1)$, en supposant toutes les opérations comme élémentaires.

Algorithme 53 : Maximum probabiliste: Monte Carlo.

Entrées : séquence s de taille paire dont la moitié des éléments sont égaux à $a \in \mathbb{N}$ et l'autre moitié à $b \in \mathbb{N}$ où $a \neq b$
Résultat : $\max(s)$

```
max-monte-carlo(s):
    faire 275 fois
        choisir i ∈ [1..|s|] de façon uniforme
        si s[i] > s[1] alors
            retourner s[i]
    retourner s[1]
```

Soit $\mathcal{A}$ un algorithme probabiliste et soit Y_x la variable aléatoire qui dénote la valeur de sortie de $\mathcal{A}$ sur entrée x. La *probabilité d'erreur* de $\mathcal{A}$ est la fonction $\text{err}\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ telle que:

$$\text{err}(n) := \max\{\Pr(Y_x \neq \text{bonne sortie sur } x) : \text{entrées } x \text{ de taille } n\}.$$

Analysons la probabilité d'erreur de l'algorithme 53. Nous voulons borner $\Pr(Y_s \neq \max(s))$. Si $s[1] = \max(s)$, alors l'algorithme retourne forcément la bonne valeur. Sinon, il retourne la mauvaise valeur si et seulement si $s[i] = s[1]$ à *chacune* des 275 itérations. Ainsi, $\Pr(Y_s \neq \max(s)) \leq (1/2)^{275} = \frac{1}{2^{275}}$. Comme cette probabilité est indépendante de la taille de s, nous avons $\text{err}(n) \leq 1/2^{275}$.

8.3 Coupe minimum: algorithme de Karger

Nous introduisons un algorithme de Monte Carlo élégant pour un problème plus complexe: la coupe minimum. Nous disons qu'une *coupe* d'un graphe non dirigé $\mathcal{G} = (V, E)$ est une partition non triviale (X, Y) de V, c.-à-d. $X, Y \neq \emptyset$, $X \cup Y = V$ et $X \cap Y = \emptyset$. La *taille* d'une coupe correspond au nombre d'arêtes

de $\mathcal{G}$ qui relient X et Y. Plus formellement:

$$\text{taille}(X, Y) \coloneqq |\{\{x, y\} \in E : x \in X, y \in Y\}|.$$

Le *problème de la coupe minimum* consiste à identifier une coupe de $\mathcal{G}$ qui minimise sa taille. Par exemple, considérons le graphe $\mathcal{G}$ illustré à la figure 8.1. Les coupes $(\{a, b, c\}, \{d, e\})$, $(\{a, b, d, e\}, \{c\})$ et $(\{a, b, c, d\}, \{e\})$ possèdent une taille de 5, 4 et 2 respectivement. En inspectant toutes les coupes, nous conclurions que la coupe minimum possède une taille de 2. Comme il existe $2^{|V|-1} - 1$ coupes en général, la force brute se bute à une complexité exponentielle.

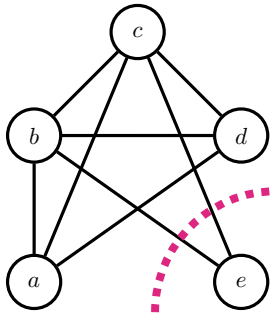


FIGURE 8.1 – Exemple de coupe minimum identifiée par un trait tireté.

L'*algorithme de Karger* est un algorithme de Monte Carlo qui identifie une coupe minimum en:

— choisissant une arête $e = \{u, v\}$ aléatoirement de façon uniforme;

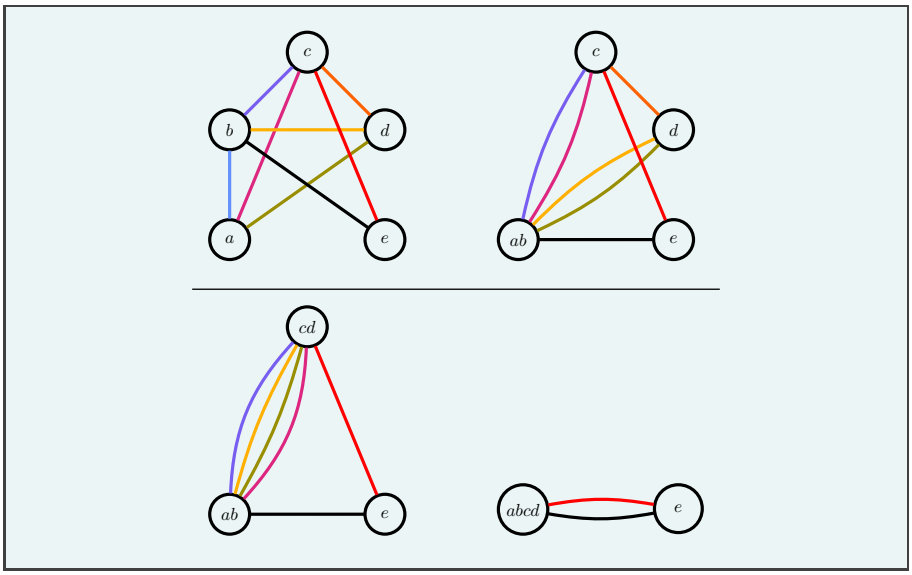
— contractant e par la fusion de u et v;

— répétant tant qu'il existe plus de deux sommets.

L'opération de contraction peut créer des arêtes parallèles et des boucles. À chaque contraction, nous conservons les arêtes parallèles et nous nous débarrassons des boucles. La taille de la coupe obtenue correspond au nombre d'arêtes parallèles entre les deux derniers sommets tel que décrit à l'algorithme 54.

Exemple.

Voici une exécution de l'algorithme de Karger:



Puisque l'algorithme retire un sommet par itération, il effectue toujours $|V|-2$ itérations au total. De plus, l'opération de contraction peut être implémentée en temps $\mathcal{O}(|V|)$, ce qui mène à un temps total de $\mathcal{O}(|V|^2)$. Cependant, la valeur de retour n'est pas nécessairement la bonne. Cherchons donc à borner la probabilité d'erreur.

Algorithme 54 : Algorithme de Karger.

Entrées : graphe non dirigé $\mathcal{G} = (V, E)$
Résultat : taille d'une coupe minimum de $\mathcal{G}$

```
karger(V, E):
    tant que |V| > 2
        choisir {u, v} ∈ E aléatoirement de façon uniforme
        retirer u et v de V
        ajouter uv à V
        pour {x, y} ∈ E
            retirer {x, y} de E
            si x ∈ {u, v} alors x ← uv
            si y ∈ {u, v} alors y ← uv
            si x ≠ y alors ajouter {x, y} à E
    retourner |E|
```

Proposition 27. *Une coupe minimum possède une taille d'au plus* $2|E|/|V|$.

Démonstration. Soit k la taille d'une coupe minimum. La coupe $(\{v\}, V \setminus \{v\})$

possède une taille égale à $\deg(v)$. Donc, $k \leq \deg(v)$ pour tout $v \in V$, et ainsi:

$$\begin{aligned} |V| \cdot k &= \overbrace{k + k + \ldots + k}^{|V| \text{ fois}} \\ &\leq \sum_{v \in V} \deg(v) \\ &= 2|E|. \end{aligned}$$

En divisant les deux côtés par $|V|$, nous obtenons $k \leq 2|E|/|V|$. □

Théorème 7. *La probabilité d'erreur de l'algorithme de Karger est inférieure ou égale à* $1 - 1/|V|^2$.

Démonstration. Fixons une coupe minimum $C = (X, Y)$. Observons d'abord que la probabilité de choisir une arête qui traverse C est d'au plus $2/|V|$. En effet, il y a $|E|$ choix d'arêtes et au plus $2|E|/|V|$ arêtes qui traversent C par la proposition 27. Si l'algorithme contracte une arête qui ne traverse pas C à chaque itération, alors la valeur de retour est correcte. Soit p la probabilité de contracter une arête qui ne traverse pas C à chaque itération. Nous avons:

$$\begin{aligned} p &\geq \left(1 - \frac{2}{|V|}\right)\left(1 - \frac{2}{|V|-1}\right)\left(1 - \frac{2}{|V|-2}\right)\cdots\left(1 - \frac{2}{4}\right)\cdot\left(1 - \frac{2}{3}\right) \\ &= \frac{|V|-2}{|V|} \cdot \frac{|V|-3}{|V|-1} \cdot \frac{|V|-4}{|V|-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \\ &= \frac{2}{|V| \cdot (|V|-1)} \\ &\geq \frac{1}{|V|^2}. \end{aligned}$$

Remarquons que C n'est pas nécessairement l'unique coupe minimum. La probabilité de succès est donc d'au moins p. Ainsi, la probabilité d'erreur est d'au plus $1 - p \leq 1 - 1/|V|^2$. □

8.4 Amplification de probabilité

Le théorème 7 affirme que la probabilité d'erreur de l'algorithme de Karger est d'au plus $q \coloneqq 1 - 1/|V|^2$. Par exemple, $q = 24/25 = 0{,}96$ sur le graphe de la figure 8.1. Nous pouvons réduire cette probabilité en répétant l'algorithme k fois et en conservant la plus petite taille identifiée. Dans ce cas, la probabilité d'erreur est d'au plus $q \cdot q \cdots q = q^k$ car il y a un échec si et seulement si les k itérations échouent. Nous avons:

Proposition 28. $q^k \leq 2^{-k/|V|^2}$.

Ainsi, en prenant $k \coloneqq \varepsilon \cdot |V|^2$, la probabilité d'erreur est réduite à au plus $1/2^{\varepsilon}$.

En général, nous pouvons réduire la probabilité d'erreur d'un algorithme de Monte Carlo (et ainsi *amplifier* sa probabilité de succès) en le répétant k fois, puis en retournant la meilleure valeur ou la valeur majoritaire selon le type de problème.

8.5 Temps moyen

Il ne faut pas confondre le temps espéré avec le temps moyen. Ce-dernier correspond à la moyenne du temps d'exécution parmi toutes les entrées d'une même taille. Plus formellement, soient $\mathcal{A}$ un algorithme (déterministe) et f la fonction telle que $f(x)$ dénote le nombre d'opérations élémentaires exécutées par $\mathcal{A}$ sur entrée x. Le *temps d'exécution moyen de* $\mathcal{A}$ est la fonction $t_{\text{moy}} \colon \mathbb{N} \to \mathbb{N}$ telle que:

$$t_{\text{moy}}(n) \coloneqq \sum_{\substack{\text{entrée } x \\ \text{de taille } n}} f(x) \;/\; (\text{nombre d'entrées de taille } n).$$

Ainsi, l'analyse en temps moyen correspond à faire l'hypothèse que les entrées d'un algorithme sont distribuées uniformément, et à faire la moyenne sur toutes les entrées d'une même taille. La validité de cette hypothèse dépend donc grandement de l'application.

À titre d'exemple, le temps moyen du tri par insertion est quadratique:

Proposition 29. *Le temps moyen du tri par insertion appartient à* $\Theta(n^2)$.

8.6 Exercices

8.1) Montrez que la distribution des nombres générés par l'algorithme 50 est bien uniforme.

8.2) Donnez une procédure afin de générer un nombre aléatoire uniformément parmi $[0, 2^k - 1]$ à l'aide d'une pièce. Votre algorithme doit *toujours* fonctionner en temps $\mathcal{O}(k)$.

8.3) Dites si l'algorithme diviser-pour-régner suivant génère un nombre aléatoire $x \in [a, b]$ de façon uniforme:

Entrées : $a, b \in \mathbb{N}_{\geq 1}$
Résultat : nombre $x \in [a, b]$ choisi de façon aléatoire et uniforme

`uniforme'`(a, b)`:`
  si $a = b$ **alors**
    retourner a
  sinon
    choisir un bit y à pile ou face
    $m \leftarrow (a + b)/2$
    si $y = 0$ **alors**
      retourner `uniforme'`$(a, \lfloor m \rfloor)$
    sinon
      retourner `uniforme'`$(\lceil m \rceil, b)$

8.4) Supposons que vous ayez accès à une pièce de monnaie biaisée: elle retourne pile avec probabilité $0 < p < 1$ et face avec probabilité $q := 1 - p$. Vous ne connaissez pas la valeur de p. Donnez un algorithme qui simule une pièce non biaisée.

8.5) L'*algorithme de Freivalds* permet de tester si $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ où $\mathbf{A}$, $\mathbf{B}$ et $\mathbf{C}$ sont des matrices carrées. Dites s'il est de Las Vegas ou de Monte Carlo. S'il s'agit du premier cas, analysez le temps espéré, sinon, analysez la probabilité d'erreur.

Entrées : $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{Q}^{n \times n}$
Résultat : $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$?

générer un vecteur aléatoire $\boldsymbol{v} \in \{0, 1\}^n$ de façon uniforme
retourner $\mathbf{A} \cdot (\mathbf{B} \cdot \boldsymbol{v}) = \mathbf{C} \cdot \boldsymbol{v}$

Indice: pensez à la probabilité que $\mathbf{D} \cdot \boldsymbol{v} = \mathbf{0}$, où $\mathbf{D} := \mathbf{A} \cdot \mathbf{B} - \mathbf{C}$.

8.6) En supposant qu'on puisse supprimer un élément d'une séquence en temps constant, l'algorithme suivant améliore l'algorithme 52 puisque le nombre

d'itérations ne peut plus être arbitrairement grand:

Entrées : séquence s de taille n paire dont la moitié des éléments sont égaux à $a \in \mathbb{N}$ et l'autre moitié à $b \in \mathbb{N}$ où $a \neq b$
Résultat : $\max(s)$

```
max-las-vegas'(s):
    boucler
        choisir i ∈ [2..|s|] de façon uniforme
        si s[i] > s[1] alors
            retourner s[i]
        sinon si s[i] < s[1] alors
            retourner s[1]
        sinon
            retirer le i^ème élément de s
```

— Dites combien d'itérations peuvent être effectuées au maximum.

— Quelle est la probabilité que l'algorithme termine à la $i^{\text{ème}}$ itération?

— ★ Bornez le temps espéré de l'algorithme.

(*basé sur un algorithme proposé en classe par Etienne D. Massé, A2019*)

8.7) Considérons ce problème:

ENTRÉE: une séquence s telle que $|s| \bmod 3 = 0$, deux tiers de ses éléments sont égaux à $a \in \mathbb{N}$ et l'autre tiers à $b \in \mathbb{N}$, où $a \neq b$
SORTIE: $\max(s)$

Analysez le temps espéré et la probabilité d'erreur des algorithmes 52 et 53 par rapport à ce problème.

Annexes

Fiches récapitulatives

Les fiches des pages suivantes résument le contenu de chacun des chapitres. Elles peuvent être imprimées recto-verso, ou bien au recto seulement afin d'être découpées et pliées en deux. À l'ordinateur, il est possible de cliquer sur la plupart des puces « ▶ » pour accéder à la section du contenu correspondant.

1. Analyse des algorithmes

Temps d'exécution

- *Opérations élémentaires*: dépend du contexte, souvent comparaisons, affectations, arithmétique, accès, etc.
- *Pire cas* $t_{\max}(n)$: nombre maximum d'opérations élémentaires exécutées parmi les entrées de taille n
- *Meilleur cas* $t_{\min}(n)$: même chose avec « minimum »
- $t_{\max}(m,n)$, $t_{\min}(m,n)$: même chose par rapport à m et n

Notation asymptotique

- *Déf.*: $f \in \mathcal{O}(g)$ si $n \geq n_0 \to f(n) \leq c \cdot g(n)$ pour certains c, n_0
- *Signifie*: f croît moins ou aussi rapid. que g pour $n \to \infty$
- *Transitivité*: $f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(h) \to f \in \mathcal{O}(h)$
- *Règle des coeff.*: $f_1 + \ldots + f_k \in \mathcal{O}(c_1 \cdot f_1 + \ldots + c_k \cdot f_k)$
- *Règle du max.*: $f_1 + \ldots + f_k \in \mathcal{O}(\max(f_1, \ldots, f_k))$
- *Déf.*: $f \in \Omega(g) \leftrightarrow g \in \mathcal{O}(f)$; $f \in \Theta(g) \leftrightarrow f \in \mathcal{O}(g) \cap \Omega(g)$
- *Règle des poly.*: f polynôme de degré $d \to f \in \Theta(n^d)$

Notation asymptotique (suite)

- *Simplification*: lignes élem. comptées comme une seule opér.
- *Règle de la limite*:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f \in \mathcal{O}(g) \text{ et } g \notin \mathcal{O}(f) \\ +\infty & f \notin \mathcal{O}(g) \text{ et } g \in \mathcal{O}(f) \\ \text{const.} & \Theta(f) = \Theta(g) \end{cases}$$

- *Multi-params.*: $\mathcal{O}, \Omega, \Theta$ étendues avec plusieurs seuils

Correction et terminaison

- *Correct*: sur *toute* entrée x qui satisfait la pré-condition, x et sa sortie y satisfont la post-condition
- *Termine*: atteint instruction **retourner** sur *toute* entrée
- *Invariant*: propriété qui demeure vraie à chaque fois qu'une ou certaines lignes de code sont atteintes

Exemples de complexité

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^2 \log n) \subset \mathcal{O}(n^3) \subset \mathcal{O}(n^d) \subset \mathcal{O}(2^n) \subset \mathcal{O}(3^n) \subset \mathcal{O}(b^n) \subset \mathcal{O}(n!)$$

2. Tri

Approche générique

- *Inversion*: indices (i,j) t.q. $i < j$ et $s[i] > s[j]$
- *Progrès*: corriger une inversion en diminue la quantité
- *Procédure*: sélectionner et corriger une inversion, jusqu'à ce qu'il n'en reste plus

Algorithmes (par comparaison)

- *Insertion*: considérer $s[1:i-1]$ triée et insérer $s[i]$ dans $s[1:i]$
- *Monceau*: transformer s en monceau et retirer ses éléments
- *Fusion*: découper s en deux, trier chaque côté et fusionner
- *Rapide*: réordonner autour d'un pivot et trier chaque côté

Propriétés

- *Sur place*: n'utilise pas de séquence auxiliaire
- *Stable*: l'ordre relatif des éléments égaux est préservé

Sommaire

Algorithme	Complexité (par cas) meilleur	moyen	pire	Sur place	Stable
insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
monceau	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗
fusion	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
rapide	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	✓	✗

Usage

- *Petite taille*: tri par insertion
- *Grande taille*: tri par monceau ou tri rapide
- *Grande taille + stabilité*: tri par fusion

Tri sans comparaison

- *Par comparaison*: barrière théorique de $\Omega(n \log n)$
- *Sans comparaison*: possible de faire mieux pour certains cas
- *Représentation binaire*: trier (de façon stable) en ordonnant du bit de poids faible vers le bit de poids fort
- *Complexité*: $\Theta(mn)$ où m = nombre de bits et $n = |s|$

3. Graphes

Graphes

- *Graphe*: $\mathcal{G} = (V, E)$ où V = sommets et E = arêtes
- *Dirigé vs. non dirigé*: $\{u,v\} \in E$ vs. $(u,v) \in E$
- *Degré (cas non dirigé)*: $\deg(u)$ = # de voisins
- *Degré (cas dirigé)*: $\deg^-(u)$ = # préd., $\deg^+(u)$ = # succ.
- *Taille*: $|E| \in \Theta(\text{somme des degrés})$ et $|E| \in \mathcal{O}(|V|^2)$
- *Chemin*: séq. $u_0 \to \cdots \to u_k$ (taille = k, simple si sans rép.)
- *Cycle*: chemin de u vers u (simple si sans rép. sauf début/fin)
- *Sous-graphe*: obtenu en retirant sommets et/ou arêtes
- *Composante*: sous-graphe max. où sommets access. entre eux

Parcours

- *Profondeur*: explorer le plus loin possible, puis retour (pile)
- *Largeur*: explorer successeurs, puis leurs succ., etc. (file)
- *Temps d'exécution*: $\mathcal{O}(|V| + |E|)$

Représentation

$$\begin{array}{c} \\ a \\ b \\ c \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array} \quad \begin{array}{l} [a \mapsto [b,c], \\ b \mapsto [a], \\ c \mapsto [b]] \end{array}$$

	Mat.	Liste (non dirigé)	Liste (dir.)
$u \to v$?	$\Theta(1)$	$\mathcal{O}(\min(\deg(u), \deg(v)))$	$\mathcal{O}(\deg^+(u))$
$\{v : u \to v\}$	$\Theta(\lvert V\rvert)$	$\mathcal{O}(\deg(u))$	$\mathcal{O}(\deg^+(u))$
$\{v : u \to v\}$	$\Theta(\lvert V\rvert)$	$\mathcal{O}(\deg(v))$	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$
Modif. $u \to v$	$\Theta(1)$	$\mathcal{O}(\deg(u) + \deg(v))$	$\mathcal{O}(\deg^+(u))$
Mémoire	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	

Propriétés et algorithmes

- *Plus court chemin*: parcours en largeur + stocker préd.
- *Ordre topologique*: $u_1 \preceq \cdots \preceq u_n$ où $i < j \implies (u_j, u_i) \notin E$
- *Tri topologique*: mettre sommets de degré 0 en file, retirer en mettant les degrés à jour, répéter tant que possible
- *Détec. de cycle*: tri topo. + vérifier si contient tous sommets
- *Temps d'exécution*: tous linéaires

Arbres

- *Arbre*: graphe connexe et acyclique (ou prop. équivalentes)
- *Forêt*: graphe constitué de plusieurs arbres
- *Arbre couv.*: arbre avec tous sommets d'un graphe $\mathcal{G}$ non dirigé; possible ssi $\mathcal{G}$ connexe; se trouve avec parcours en prof.

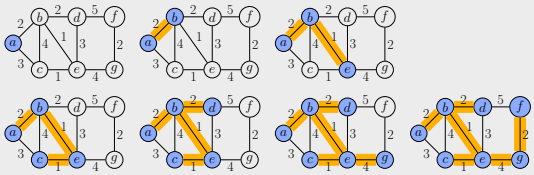
4. Algorithmes gloutons

Arbres couvrants minimaux

- *Graphe pondéré*: $\mathcal{G} = (V, E)$ où $p[e]$ est le poids de l'arête e
- *Poids d'un graphe*: $p(\mathcal{G}) = \sum_{e \in E} p[e]$
- *Arbre couv. min.*: arbre couvrant de $\mathcal{G}$ de poids minimal

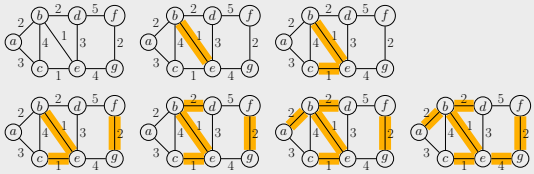
Algorithme de Prim–Jarník

- *Approche*: faire grandir un arbre en prenant l'arête min.
- *Complexité*: $\mathcal{O}(|E| \log |V|)$ avec monceau



Algorithme de Kruskal

- *Approche*: connecter forêt avec l'arête min. jusqu'à un arbre
- *Complexité*: $\mathcal{O}(|E| \log |V|)$ avec ensembles disjoints



Ensembles disjoints

- *But*: manipuler une partition d'un ensemble V
- *Représentation*: chaque ensemble sous une arborescence

Algorithme glouton

1) *Choisir* un candidat c itérativement (*sans reconsidérer*)
2) Ajouter c à solution partielle S si *admissible*
3) Retourner S si *solution (complète)*, « impossible » sinon

Problème du sac à dos

- *But*: choisir objets pour maxim. valeur sans excéder capacité
- *Algo. glouton*: trier en ordre décroissant par $val[i]/poids[i]$
- *Fonctionne* si on peut découper objets (sinon approx. à ½)

5. Algorithmes récursifs et approche diviser-pour-régner

Diviser-pour-régner

- *A)* découper en sous-problèmes disjoints
- *B)* obtenir solutions récursivement
- *C)* s'arrêter aux cas de base (souvent triviaux)
- *D)* combiner solutions pour obtenir solution globale
- *Exemple*: tri par fusion $\mathcal{O}(n \log n)$

Récurrences linéaires

- *Cas homogène:* $\sum_{i=0}^{d} a_i \cdot t(n-i) = 0$
- *Polynôme caractéristique:* $\sum_{i=0}^{d} a_i \cdot x^{d-i}$
- *Forme close:* $t(n) = \sum_{i=1}^{d} c_i \cdot \lambda_i^n$ où les λ_i sont les racines
- *Constantes c_i:* obtenues en résolvant un sys. d'éq. lin.
- *Cas non homo.:* si $= c \cdot b^n$, on multiplie poly. par $(x - b)$
- *Exemple*:
  - Récurrence: $t(n) = 3 \cdot t(n-1) + 4 \cdot t(n-2)$
  - Poly. carac.: $x^2 - 3x - 4 = (x-4)(x+1)$
  - Forme close: $t(n) = c_1 \cdot 4^n + c_2 \cdot (-1)^n$

Autres méthodes

- *Substitution*: remplacer $t(n), t(n-1), t(n-2), \ldots$ par sa déf. jusqu'à deviner la forme close
- *Arbres*: construire un arbre représentant la récursion et identifier le coût de chaque niveau

Quelques algorithmes

- *Hanoï*: $src[1:n-1] \to tmp, src[n] \to dst, tmp[1:n-1] \to dst$ $\mathcal{O}(2^n)$
- *Exp. rapide*: exploiter $b^n = (b^{n \div 2})^2 \cdot b^{n \bmod 2}$ $\mathcal{O}(\log n)$
- *Mult. rapide*: calculer $(a+b)(c+d)$ en 3 mult. $\mathcal{O}(n^{\log 3})$
- *Horizon*: découper blocs comme tri par fusion $\mathcal{O}(n \log n)$

Théorème maître (allégé)

- $t(n) = c \cdot t(n \div b) + f(n)$ où $f \in \mathcal{O}(n^d)$:
  - $\mathcal{O}(n^d)$ si $c < b^d$
  - $\mathcal{O}(n^d \cdot \log n)$ si $c = b^d$
  - $\mathcal{O}(n^{\log_b c})$ si $c > b^d$

6. Force brute

Approche

- *Exhaustif*: essayer toutes les sol. ou candidats récursivement
- *Explosion combinatoire*: souvent # solutions $\geq b^n, n!, n^n$
- *Avantage*: simple, algo. de test, parfois seule option
- *Désavantage*: généralement très lent et/ou avare en mémoire

Techniques pour surmonter explosion

- *Élagage*: ne pas développer branches inutiles
- *Contraintes*: élaguer si contraintes enfreintes
- *Bornes*: élaguer si impossible de faire mieux
- *Approximations*: débuter avec approx. comme meilleure sol.
- *Si tout échoue*: solveurs SAT ou d'optimisation

Problème des n dames

- *But*: placer n dames sur échiquier sans attaques
- *Algo.*: placer une dame par ligne en essayant colonnes dispo.

Sac à dos

- *But*: maximiser valeur sans excéder capacité
- *Algo.*: essayer sans et avec chaque objet
- *Mieux*: élaguer dès qu'il y a excès de capacité
- *Mieux++*: élaguer si aucune amélioration avec somme valeurs

Retour de monnaie

- *But*: rendre montant avec le moins de pièces
- *Algo.*: pour chaque pièce, essayer d'en prendre 0 à # max.

7. Programmation dynamique

Approche

- *Principe d'optimalité*: solution optimale obtenue en combinant solutions de sous-problèmes qui se chevauchent
- *Descendante*: algo. récursif + mémoïsation (ex. Fibonacci)
- *Ascendante*: remplir tableau itér. avec solutions sous-prob.

Retour de monnaie

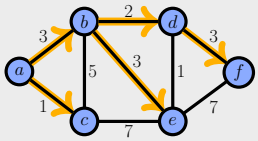
- *Sous-question*: # pièces pour rendre j avec pièces 1 à i?
- *Identité*: $T[i,j] = \min(T[i-1,j],\ T[i,j-s[i]]+1)$
- *Exemple*: montant $m=10$ et pièces $s=[1,5,7]$

	0	1	2	3	4	5	6	7	8	9	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8	9	10
2	0	1	2	3	4	1	2	3	4	5	2
3	0	1	2	3	4	1	2	1	2	3	2

Sac à dos

- *Sous-question*: val. max. avec capacité j et les objets 1 à i?
- *Identité*: $T[i,j] = \max(T[i-1,j],\ T[i-1,j-p[i]]+v[i])$

Plus courts chemins



- *Déf.*: chemin simple de poids minimal
- *Bien défini*: si aucun cycle négatif
- *Approche générale*: raffiner distances partielles itérativement
- *Dijkstra*: raffiner en marquant sommet avec dist. min.
- *Floyd-Warshall*: raffiner via sommet intermédiaire v_k
- *Bellman-Ford*: raffiner avec $\geq 1, 2, \ldots, |V|-1$ arêtes
- *Sommaire*:

	Dijkstra	Bellman-Ford	Floyd-Warshall
Types de chemins	d'un sommet vers les autres		paires de sommets
Poids négatifs?	✗	✓	✓
Temps d'exécution	$\mathcal{O}(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert \cdot \lvert E\rvert)$	$\Theta(\lvert V\rvert^3)$
Temps ($\lvert E\rvert \in \Theta(1)$)	$\mathcal{O}(\lvert V\rvert \log \lvert V\rvert)$	$\Theta(\lvert V\rvert)$	$\Theta(\lvert V\rvert^3)$
Temps ($\lvert E\rvert \in \Theta(\lvert V\rvert)$)	$\mathcal{O}(\lvert V\rvert \log \lvert V\rvert)$	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert^3)$
Temps ($\lvert E\rvert \in \Theta(\lvert V\rvert^2)$)	$\mathcal{O}(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert^3)$	$\Theta(\lvert V\rvert^3)$

8. Algorithmes et analyse probabilistes

Modèle probabiliste

- *Modèle*: on peut tirer à pile ou face (non déterministe)
- *Aléa*: on peut obtenir une loi uniforme avec une pièce
- *Idéalisé*: on suppose avoir accès à une source d'aléa parfaite (en pratique: source plutôt pseudo-aléatoire)

Algorithmes de Las Vegas

- *Temps*: varie selon les choix probabilistes
- *Valeur de retour*: toujours correcte
- *Exemple*: tri rapide avec pivot aléatoire
- *Temps espéré*: dépend de $\mathbb{E}[Y_x]$ où $Y_x = \#$ opér. sur entrée x

Algorithmes de Monte Carlo

- *Temps*: borne ne varie *pas* selon les choix probabilistes
- *Valeur de retour*: pas toujours correcte
- *Exemple*: algorithme de Karger
- *Prob. d'erreur*: dépend de $\Pr(Y_x \neq$ bonne sortie sur $x)$

Coupe minimum: algorithme de Karger

- *Coupe*: partition (X, Y) des sommets d'un graphe non dirigé
- *Taille*: # d'arêtes qui traversent X et Y
- *Coupe min.*: identifier la taille minimale d'une coupe
- *Algorithme*: contracter itérativement une arête aléatoire en gardant les multi-arêtes, mais pas les boucles

- *Prob. d'erreur*: $\leq 1 - 1/|V|^2$ (Monte Carlo)
- *Amplification*: on peut réduire (augmenter) la prob. d'erreur (de succès) arbitrairement (en général: avec min., maj., $\vee$, etc.)

Temps moyen

- *Temps moyen*: $\sum$(temps instances de taille n) / # instances
- *Attention*: *pas* la même chose que le temps espéré
- *Hypothèse*: entrées distribuées uniformément ($\pm$ réaliste)
- *Exemple*: $\Theta(n^2)$ pour le tri par insertion

Solutions des exercices

Cette section présente des solutions à certains des exercices du document. Dans certains cas, il ne s'agit que d'ébauches de solutions.

Chapitre 0

0.1) 3

0.2) $X \cup Y = \{-23, 1, 3, 5, 6, 9, 23, a, c\}$, $X \cap Y = \{5, 9, a\}$, $X \setminus Y = \{1, 6, 23, c\}$, $Y \setminus X = \{-23, 3\}$

0.4) Soient $m, n \in \mathbb{N}$ tels que m est pair et n est impair. Démontrons que $m+n$ est impair. Par définition, il existe $a, b \in \mathbb{N}$ tels que $m = 2a$ et $n = 2b+1$. Nous avons donc $m+n = 2a+2b+1 = 2(a+b)+1$. Ainsi $m+n = 2k+1$ où $k \coloneqq a+b$. Nous concluons donc que $m+n$ est impair. □

0.8) 2^n par l'exercice 0.7).

0.9) Montrons que $n! > 2^n$ pour tout $n \in \mathbb{N}_{\geq 4}$ par induction sur n.

Cas de base ($n = 4$). Nous avons $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24 > 16 = 2^4$.

Étape d'induction. Soit $n \geq 4$. Supposons que $n! > 2^n$. Montrons que $(n+1)! > 2^{n+1}$. Nous avons:

$$\begin{aligned}(n+1)! &= 1 \cdot 2 \cdots n \cdot (n+1) \\ &> 2^n \cdot (n+1) && \text{(par hypothèse d'induction)} \\ &> 2^n \cdot 2 && \text{(car } n+1 \geq 4+1 > 2\text{)} \\ &= 2^{n+1}.\end{aligned}$$ □

0.10) Soient $n, k \in \mathbb{N}$. Si $n < k$, alors les deux côtés de l'équation sont trivialement égaux à 0. Si $n \geq k$, alors nous avons:

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

0.11) Observons que $\binom{0}{0} = 1$ et $\binom{0}{k} = 0$ pour $k > 0$. Ainsi, la formule de Pascal montre que $\binom{n}{k}$ se calcule récursivement comme une grande somme de 1 et 0, ce qui en fait forcément un entier naturel. Nous pourrions démontrer ce fait de façon un peu plus formellement en procédant par induction sur n.

0.17) Observons d'abord qu'il est impossible de payer 1$ puisque toute combinaison de 2$ et 5$ excède forcément ce montant. De plus, il est impossible de payer 3$ puisqu'il faut nécessairement prendre une pièce de 2$ et qu'il est impossible de payer le 1$ restant (par l'argument précédent). Clairement, il est possible de payer 2$. Montrons maintenant qu'il est possible de payer tout montant $n \geq 4$ par induction généralisée.

Cas de base ($n \in \{4, 5\}$). Pour $n = 4$, il suffit de prendre deux pièces de 2$, et pour $n = 5$ il suffit de prendre un billet de 5$.

Étape d'induction. Soit $n \geq 5$. Supposons qu'il soit possible de payer tout montant compris dans l'intervalle $[4..n]$ et cherchons à montrer qu'il est possible de payer $(n+1)$$. Nous prenons une pièce de 2$, puis il reste à payer $(n-1)$$. Observons que $n - 1 \in [4..n]$. Ainsi, par hypothèse d'induction, il est possible de payer le reste du montant. □

0.18) Les cas $n = 1$ et $n \geq 3$ sont tous corrects. Par contre, le cas $n = 2$ est problématique. En effet, lorsqu'on retire Alice du groupe, il ne reste que Bob, et vice-versa. Il n'y a donc pas de chevaux intermédiaires qui partagent la même qu'Alice *et* Bob.

0.19)

(a) Non. Il suffit de donner un contre-exemple. Imaginons que A termine en $7^{\text{ème}}$, $5^{\text{ème}}$ et $1^{\text{ère}}$ positions; et que B termine en $2^{\text{ème}}$, $3^{\text{ème}}$ et $6^{\text{ème}}$ positions. Dans le système de pointage multiplicatif, A obtient 35 points et l'emporte car B obtient 36 points. Dans un système additif, B obtient 11 points et l'emporte car A obtient 13 points. Cela s'est produit aux Jeux olympiques de 2020: Jakob Schubert (A) a obtenu le bronze, alors que Tomoa Narasaki (B) a terminé au pied du podium en quatrième place. □

(b) Non, ce système est équivalent au système multiplicatif et nous avons montré que celui-ci n'est pas équivalent au système additif. En effet, observons que $\log a + \log b + \log c = \log abc$. De plus, $\log x < \log y$ ssi $x < y$. Ainsi, $\log a + \log b + \log c < \log a' + \log b' + \log c'$ ssi $\log(abc) < \log(a'b'c')$ ssi $abc < a'b'c'$. □

(c) Non (sauf si $n = 1$, ce qui ne serait pas une compétition...)

Preuve 1: Il y a au plus $n \cdot n \cdot n = n^3$ pointages différents. Les positions $[1, 1, 2]$ et $[2, 1, 1]$ mènent au même score: 2. Il y a donc au plus $n^3 - 1$ pointages différents. □

Preuve 2: Le plus grand score inférieur à n^3 est $n \cdot n \cdot (n-1) = n^3 - n^2$. Comme $n \geq 2$, cela signifie que les pointages $n^3 - 3$, $n^3 - 2$ et $n^3 - 1$ ne sont pas réalisables. □

Preuve (partielle) 3: La seule façon d'obtenir un score qui soit un nombre premier p est de terminer deux fois en première position, et une fois en position p. Comme $p \leq n$, il est impossible d'obtenir un score premier strictement supérieur à n. □

Ce troisième argument a été proposé par deux personnes étudiantes (A22). Pour être complet, il faut également argumenter qu'il existe toujours un nombre premier dans $[n+1..n^3]$. Cela est vrai. En fait, il en existe toujours un dans $[n+1..2n-1]$ par le postulat de Bertrand.

0.20) Soit un nombre $x \in \mathbb{N}$ dont les chiffres décimaux sont $c_{n-1}, \ldots, c_0$. Nous avons $x = \sum_{i=0}^{n-1} c_i \cdot 10^i$. Posons $s \coloneqq c_{n-1} + \ldots + c_0$, la somme des chiffres de x. Nous avons:

$$\begin{aligned}
x &= \sum_{i=0}^{n-1} c_i \cdot 10^i \\
&= \sum_{i=0}^{n-1} (c_i + c_i \cdot (10^i - 1)) \\
&= \sum_{i=0}^{n-1} c_i + \sum_{i=0}^{n-1} c_i \cdot (10^i - 1) \\
&= s + \sum_{i=0}^{n-1} c_i \cdot (10^i - 1) \\
&= s + \sum_{i=0}^{n-1} c_i \cdot \sum_{j=0}^{i-1} 9 \cdot 10^j \qquad (10-1 = 9, 100-1 = 99, \ldots) \\
&= s + 3 \cdot 3 \cdot \sum_{i=0}^{n-1} \left(c_i \cdot \sum_{j=0}^{i-1} 10^j \right).
\end{aligned}$$

Ainsi, $x = s + 3b$ pour un certain $b \in \mathbb{N}$.

Si x est un multiple de 3, c-à-d. que $x = 3a$ pour $a \in \mathbb{N}$, alors $s = x - 3b = 3a - 3b = 3(a-b)$ et ainsi s est un multiple de 3. Si s est un multiple de 3, c.-à-d. que $s = 3a$ pour $a \in \mathbb{N}$, alors $x = 3a + 3b = 3(a+b)$ et ainsi x est un multiple de 3. □

0.21) Soit T un arbre de n sommets et soit r sa racine. Si $n = 1$, alors l'arbre est de hauteur $0 = \log 1$. Supposons maintenant que $n \geq 2$ et que l'hypothèse est vraie pour tout arbre de moins de n sommets. Il y a deux cas possibles: r a un ou deux enfants.

Cas 1. Le sous-arbre T' enraciné en l'unique enfant x de r possède $n-1$ sommets. Nous avons:

$$\begin{aligned}
\text{hauteur}(T) &= \text{hauteur}(T') + 1 \\
&\geq \log(n-1) + 1 && \text{(par hypothèse d'induction)} \\
&\geq \log(n/2) + 1 && \text{(car } n \geq 2\text{)} \\
&= \log(n) - \log(2) + 1 \\
&= \log(n) - 1 + 1 \\
&= \log n.
\end{aligned}$$

Cas 2. Soient x et y les deux enfants de r. Soient T_x et T_y les sous-arbres enracinés respectivement en x et y. Nous avons:

$$\begin{aligned}
d &= \max(\text{hauteur}(T_x), \text{hauteur}(T_y)) + 1 \\
&\geq \max(\log(\text{sommets}(T_x)), \log(\text{sommets}(T_y))) + 1 \\
&\geq \log(\max(\text{sommets}(T_x), \text{sommets}(T_y))) + 1 \\
&\geq \log(n/2) + 1 \\
&= \log(n) - 1 + 1 \\
&= \log n.
\end{aligned}$$

□

0.22) Lorsque $n = 0$, nous avons bien $\sum_{i=0}^{n} r^i = r^0 = 1 = (1-r^1)/(1-r)$. Soit $n > 0$. Nous avons:

$$\begin{aligned}
\sum_{i=0}^{n} r^i &= r^n + \sum_{i=0}^{n-1} r^i \\
&= r^n + (1-r^n)/(1-r) && \text{(par hypothèse d'induction)} \\
&= [(1-r)r^n + 1 - r^n]/(1-r) \\
&= (r^n - r^{n+1} + 1 - r^n)/(1-r) \\
&= (1-r^{n+1})/(1-r).
\end{aligned}$$

□

0.23) Premier scénario: $10 \cdot 10 = 100$ combinaisons et ainsi $0{,}1$ seconde. Deuxième scénario: $(366 + 365) \cdot 2 \cdot 100 = 146\,200$ combinaisons et ainsi moins de trois minutes (2 minutes, 26 secondes, 2 millisecondes).

0.24) Procédons par induction généralisée. Appelons les deux personnes respectivement *Alice* et *Bob*. Si $n \in [1..3]$, alors Alice gagne en retirant toutes les allumettes. Si $n = 4$, alors Alice laisse forcément 3, 2 ou 1 allumettes, puis Bob vide la pile et gagne. Soit $n > 4$. Nous considérons deux cas.

Cas 1: $n \bmod 4 \neq 0$. Posons $r := n \bmod 4$. Nous avons $r \in \{1, 2, 3\}$. Alice retire r allumettes. Si $n - r = 0$, alors Alice a gagné. Autrement, $n - r \geq 4$. Bob retire donc $k \in \{1, 2, 3\}$ allumettes et il en reste $n - r - k$. Remarquons

que $(n-r-k)$ mod $4 = 4-r \neq 0$. Ainsi, par hypothèse d'induction, Alice a une stratégie gagnante pour gagner le jeu à partir de la pile résultante.

Cas 2: n mod $4 = 0$. Alice laisse forcément $n-1$, $n-2$ ou $n-3$ allumettes. Bob peut donc retirer des allumettes de telle sorte à ce qu'il n'en reste que $n-4$. Par hypothèse d'induction, Bob possède une stratégie gagnante pour $n-4$ allumettes, car $n-4$ est aussi un multiple de quatre. □

0.25) Procédons par induction sur n. Nous avons $4^2 = 16 = 2^4$. Soit $n \geq 4$. Nous avons:

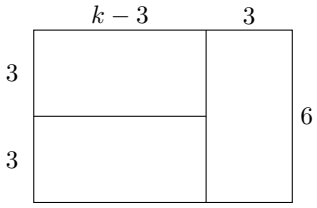
$$\begin{aligned}
(n+1)^2 &= n^2 + 2n + 1 \\
&\leq n^2 + (n-1)n + n && \text{(car } n \geq 4\text{)} \\
&= n^2 + n^2 \\
&\leq 2^n + 2^n && \text{(par hypothèse d'induction)} \\
&= 2^{n+1}.
\end{aligned}$$

□

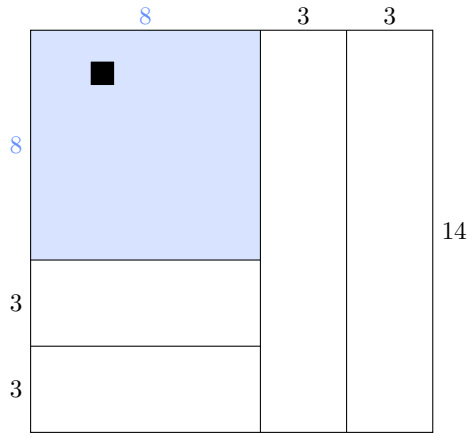
0.27)

(a) Il suffit de répéter ce motif k fois:

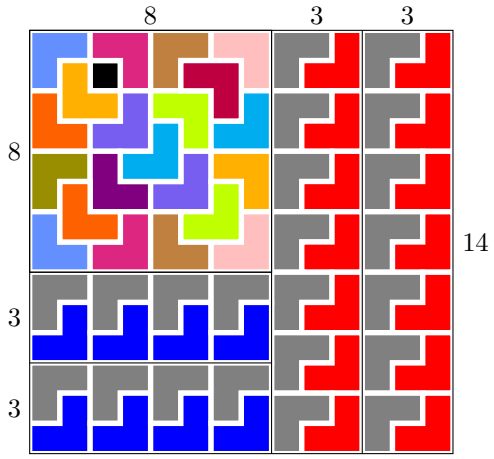
(b) Si k est pair, alors on peut scinder la grille en deux sous-grilles $3 \times k$ et invoquer (a). Si k est impair, alors on peut découper la grille comme suit et invoquer (a) sur chaque sous-grille:



(c) Considérons la case retranchée de la grille. Peu importe sa position, il est possible de l'encadrer d'une sous-grille 8×8 ancrée dans un coin de la grille 14×14. Le reste de la grille se divise en sous-grilles $3 \times k$ où k est pair. Par exemple:

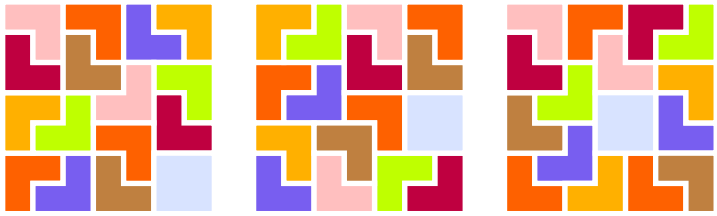


Nous avons vu à la section 0.2.4 que la sous-grille 8×8 est quasi-pavable puisque 8 est une puissance de 2. De plus, par (a), il est possible de paver une sous-grille de taille $3 \times k$ où k pair. Par exemple, sur l'exemple ci-dessus, nous obtenons ce pavage:



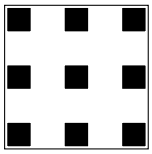
Cette solution a été proposée par Louis Desruisseaux (A23).

(d) Considérons la case retranchée de la grille. Peu importe sa position, on peut l'encadrer d'une sous-grille 4×4 ancrée dans un coin de la grille 7×7. Ensuite, on peut encadrer la case retranchée d'une sous-grille 2×2 ancrée dans un coin de la grille 4×4. Nous savons qu'il est possible de quasi-paver la sous-grille 2×2. De plus, il est possible de paver le reste de la grille. Voici trois cas possibles:



Tous les autres cas s'obtiennent à partir de réflexions. □

(e) Considérons la grille 5×5 et considérons ses cases comme suit:



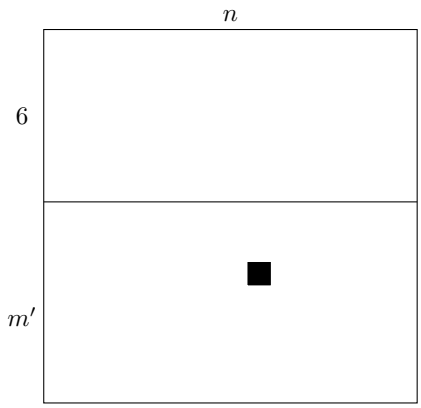
La grille possède 25 cases dont 9 noires. Comme $25 = 3 \cdot 8 + 1$, il faut utiliser 8 triominos pour quasi-paver la grille. Deux cases noires ne peuvent pas être couvertes par un même triomino. Ainsi, on ne peut pas paver la grille à une case blanche près. □

(f) Sans perte de généralité, nous pouvons supposer que $m \geq n$ (l'autre cas est symétrique). Si $m = 4$ et $n = 4$, alors nous avons terminé car 4 est une puissance de 2. Si $m = 7$ et $n = 4$, alors nous pouvons paver une sous-grille 3×4 par (a), puis on obtient une sous-grille 4×4 qui est quasi-pavable. Si $m = 10$ et $n = 4$, alors nous pouvons paver une sous-grille 3×4 par (a), puis on obtient une sous-grille 7×4 qui est quasi-pavable. Si $m = 7$ et $n = 7$, alors cela est couvert par (d). Si $m = 10$ et $n = 7$, alors nous pouvons paver une sous-grille 10×3 par (a), puis on obtient une sous-grille 10×4 qui est quasi-pavable. □

(g) Soient $a, b \geq 1$. Posons $m := (3a + 1)$ et $n := (3b + 1)$. Montrons que la grille $m \times n$ est quasi-pavable. Nous procédons par induction. Si $m, n \in \{4, 7, 10\}$, alors nous avons terminé par (f). Supposons donc que $m \geq 13$. Le cas où $n \geq 13$ est symétrique. Posons $m' := m - 6$. Remarquons que

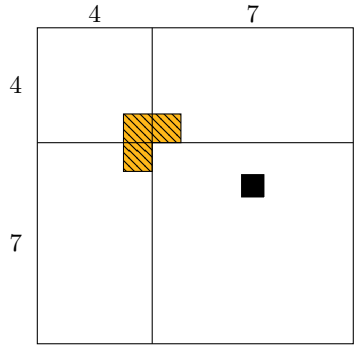
$$\begin{aligned} m' \geq m/2 &\iff m - 6 \geq m/2 \\ &\iff m/2 \geq 6 \\ &\iff m \geq 12. \end{aligned}$$

Puisque $m \geq 13$, nous avons $m' \geq m/2$. Considérons la case retranchée de la grille. Puisque $m' \geq m/2$, peu importe la position de la case, il est possible de l'encadrer d'une sous-grille $m' \times n$ ancrée dans un coin de la grille $m \times n$:



Comme $m' = (3a+1) - 6 = 3(a-2) + 1$, cette sous-grille est quasi-pavable par hypothèse d'induction. L'autre sous-grille est de taille $6 \times n$ et est ainsi pavable par (b). □

(h) Sans perte de généralité, nous pouvons supposer que $m \geq n$ (l'autre cas est symétrique). Si $m = 8$ et $n = 8$, alors nous avons terminé car 8 est une puissance de 2. Si $m = 11$ et $n = 8$, alors nous pouvons paver une sous-grille 3×8 par (a), puis on obtient une sous-grille 8×8 qui est quasi-pavable. Si $m = 11$ et $n = 11$, alors nous pouvons encadrer la case retranchée par une sous-grille 7×7, et retrancher un triomino dans le coin des autres sous-grilles. Par exemple:

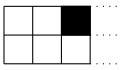


Par (f), les quatre sous-grilles sont quasi-pavables. □

(i) Soient $a, b \geq 2$. Posons $m \coloneqq (3a+2)$ et $n \coloneqq (3b+2)$. Montrons que la grille $m \times n$ est quasi-pavable. Nous procédons par induction. Si $m, n \in \{8, 11\}$, alors nous avons terminé par (h). Supposons donc que $m \geq 14$. Le cas où $n \geq 14$ est symétrique. Posons $m' \coloneqq m - 6$. Considérons la case retranchée de la grille. Puisque $m \geq 14$, nous avons $m' \geq m/2$, et ainsi peu importe la position de la case, il est possible de l'encadrer d'une sous-grille $m' \times n$ ancrée dans un coin

de la grille $m \times n$. Comme $m' = (3a+2)-6 = 3(a-2)+2$, cette sous-grille est quasi-pavable par hypothèse d'induction. L'autre sous-grille est de taille $6 \times n$ et est ainsi pavable par (b). □

(j) Si $m = n = 1$ ou $m = n = 2$, alors la grille est clairement quasi-pavable. Si $m = 1$ et $n > 1$ (ou vice versa), alors la grille n'est clairement pas quasi-pavable. Si $m = 2$ et $n > 2$ (ou vice versa), alors la grille n'est pas quasi-pavable en raison de cette case retranchée:



Pour le reste de la démonstration, nous pouvons donc supposer que $m, n \geq 3$. Remarquons que la grille possède mn cases. Comme un triomino contient 3 cases, et qu'une case est retranchée à la grille, nous devons avoir $mn \bmod 3 = 1$ pour que la grille soit quasi-pavable. Soient $a, b \geq 1$ et $r, s \geq 0$ les entiers tels que $m = 3a+r$ et $n = 3b+s$. Nous avons

$$\begin{aligned} mn \bmod 3 &= (3a + r)(3b + s) \bmod 3 \\ &= 3(3ab + as + br) + rs \bmod 3 \\ &= rs \bmod 3. \end{aligned}$$

Ainsi,

$$\begin{aligned} mn \bmod 3 = 1 &\iff rs \bmod 3 = 1 \\ &\iff (r = s = 1) \vee (r = s = 2) \\ &\iff r = s \in \{1, 2\} \\ &\iff m \bmod 3 = n \bmod 3 \in \{1, 2\}. \qquad (1) \end{aligned}$$

En (e), nous avons montré que la grille 5×5 n'est pas quasi-pavable. Si la contrainte (1) est satisfaite, $m \neq 5$ et $n \neq 5$, alors la grille $m \times n$ est quasi-pavable par (g) ou (i). □

Chapitre 1

1.1) Soient $f, g \in \mathcal{F}$ tels que $f \in \mathcal{O}(g)$. Soit $h \in \mathcal{O}(f)$. Puisque $h \in \mathcal{O}(f)$ et $f \in \mathcal{O}(g)$, nous avons $h \in \mathcal{O}(g)$ par transitivité. Ainsi, $\mathcal{O}(f) \subseteq \mathcal{O}(g)$. □

1.4)

$$\begin{aligned}\mathcal{O}(1000000) &\subset \mathcal{O}(8(n+2)-1+9n) \subset \mathcal{O}(n\log n) \subset \mathcal{O}(5n^2-n)\\ &= \mathcal{O}(3n^2) \subset \mathcal{O}(n^3-n^2+7) \subset \mathcal{O}(2^n) \subset \mathcal{O}(4^n) \subset \mathcal{O}(n!).\end{aligned}$$

1.5) Non. Cela découle, par exemple, de la règle de la limite:

$$\lim_{n\to+\infty} 3^n/2^n = \lim_{n\to+\infty} (3/2)^n = +\infty.$$

1.7) Nous avons:

$$\begin{aligned}\lim_{n\to\infty}\frac{f(n)}{g(n)} &= \lim_{n\to\infty}\frac{f'(n)}{g'(n)} && \text{(par la règle de L'Hôpital)}\\ &= \lim_{n\to\infty}\frac{2n}{\log_e(2)\cdot 2^n}\\ &= \frac{2}{\log_e(2)}\cdot\lim_{n\to\infty}\frac{n}{2^n}\\ &= \frac{2}{\log_e(2)}\cdot\lim_{n\to\infty}\frac{n'}{(2^n)'} && \text{(par la règle de L'Hôpital)}\\ &= \frac{2}{\log_e(2)}\cdot\lim_{n\to\infty}\frac{1}{\log_e(2)\cdot 2^n}\\ &= \frac{2}{\log_e(2)^2}\cdot\lim_{n\to\infty}\frac{1}{2^n}\\ &= 0.\end{aligned}$$

Ainsi, par la règle de la limite, $n^2 \in \mathcal{O}(2^n)$ et $2^n \notin \mathcal{O}(n^2)$.

1.8) Nous avons $\log_a n \in \Theta(\log_b n)$ car $\log_a n = (1/\log_b a)\cdot\log_b n$.

1.9) Nous avons $\binom{n}{d} \in \mathcal{O}(n^d)$ puisque

$$\begin{aligned}\binom{n}{d} &= \frac{n!}{d!\cdot(n-d)!}\\ &= \frac{1}{d!}\cdot(n-d+1)\cdots(n-1)\cdot n\\ &\le \frac{1}{d!}\cdot n\cdots n\cdot n\\ &= \frac{1}{d!}\cdot n^d.\end{aligned}$$

De plus, $\binom{n}{d} \in \Omega(n^d)$ car

$$\begin{aligned}\binom{n}{d} &= \frac{n!}{d! \cdot (n-d)!} \\ &= \frac{1}{d!} \cdot (n-d+1) \cdots (n-1) \cdot n \\ &\geq \frac{1}{d!} \cdot (n/2) \cdots (n/2) \cdot (n/2) \qquad \text{(pour tout } n \geq 2d-2\text{)} \\ &= \frac{1}{d!} \cdot (n/2)^d \\ &= \frac{1}{d! \cdot 2^d} \cdot n^d. \end{aligned}$$

□

1.10) Soit $d \in \mathbb{N}$. Posons $f(n) \coloneqq \sum_{i=1}^{n} i^d$. Montrons d'abord que $f \in \mathcal{O}(n^{d+1})$. Nous avons:

$$\begin{aligned} f(n) &= 1^d + 2^d + \ldots + n^d \\ &\leq n^d + n^d + \ldots + n^d \qquad \text{(pour tout } n \geq 1\text{)} \\ &= n \cdot n^d \\ &= n^{d+1}. \end{aligned}$$

Ainsi, en prenant 1 à la fois comme constante multiplicative et comme seuil, nous obtenons $f \in \mathcal{O}(n^{d+1})$.

Montrons maintenant que $f \in \Omega(n^{d+1})$. Expliquons d'abord l'approche informellement. Nous pouvons scinder la somme en deux avec les « petits » termes d'un côté et les « grands » termes de l'autre:

$$\underbrace{1^d + 2^d + \ldots + (n \div 2)^d}_{n \div 2 \text{ petits termes}} + \underbrace{((n \div 2) + 1)^d + \ldots + (n-1)^d + n^d}_{\geq n/2 \text{ grands termes}}.$$

Un grand terme vaut au moins $(n/2)^d$ et il y a au moins $n/2$ tels termes. Nous avons donc une valeur d'au moins $(n/2)^{d+1} = (1/2^{d+1}) \cdot n^{d+1}$.

Plus formellement, nous avons:

$$\begin{aligned}
f(n) &= \sum_{i=1}^{n} i^d \\
&\geq \sum_{i=(n\div 2)+1}^{n} i^d && \text{(on garde les } i \geq \text{à la médiane)} \\
&\geq \sum_{i=(n\div 2)+1}^{n} ((n\div 2)+1)^d && \text{(car chaque } i \geq (n\div 2)+1) \\
&\geq (n/2)\cdot((n\div 2)+1)^d && \text{(car au moins } n/2 \text{ termes)} \\
&\geq (n/2)\cdot(n/2)^d && \text{(car } (n\div 2)+1 \geq n/2) \\
&= (n/2)^{d+1} \\
&= \frac{1}{2^{d+1}}\cdot n^{d+1}.
\end{aligned}$$

Ainsi, en prenant $1/2^{d+1}$ comme constante multiplicative et 0 comme seuil, nous obtenons $f \in \Omega(n^{d+1})$, et par conséquent $f \in \Theta(n^{d+1})$. □

1.11) Soit $k \in \mathbb{N}_{\geq 2}$. Observons d'abord que $k(n-k) \geq n$ pour tout $n \geq \frac{k^2}{k-1}$. En effet:

$$\begin{aligned}
k(n-k) \geq n &\iff kn - k^2 \geq n \\
&\iff kn - n \geq k^2 \\
&\iff n(k-1) \geq k^2 \\
&\iff n \geq \frac{k^2}{k-1} && \text{(car } k-1 \geq 2-1 > 0).
\end{aligned}$$

En utilisant cette observation, nous obtenons donc:

$$\begin{aligned}
n^d &= n\cdot n^{d-1} \\
&\leq n\cdot\prod_{k=2}^{d} k(n-k) && \text{pour tout } n \geq \frac{d^2}{d-1} \\
&= n\cdot 2\cdot(n-2)\cdots d\cdot(n-d) \\
&= 2\cdot 3\cdots d\cdot(n-d)\cdots(n-3)\cdot(n-2)\cdot n \\
&\leq n! && \text{pour tout } n \geq 2d+1.
\end{aligned}$$

Ainsi, en prenant $\max(d^2/(d-1), 2d+1)$ comme seuil et 1 comme constante multiplicative, nous concluons que $n^d \in \mathcal{O}(n!)$. □

1.12) Soit $d \in \mathbb{N}_{>0}$. Montrons d'abord que $\log n \in \mathcal{O}\left(\sqrt[d]{n}\right)$. Pour tout $n \in \mathbb{N}_{\geq 1}$:

$$\begin{aligned}\log n &= \log\left((\sqrt[d]{n})^d\right)\\ &= d \cdot \log \sqrt[d]{n}\\ &\leq d \cdot \sqrt[d]{n} \qquad \text{(car } \log x \leq x \text{ pour tout } x \in \mathbb{R}_{>0}\text{)}.\end{aligned}$$

Ainsi en prenant d comme constante multiplicative et 1 comme seuil, nous concluons que $\log n \in \mathcal{O}(\sqrt[d]{n})$.

Montrons maintenant que $(\log n)^d \in \mathcal{O}(n)$. Pour tout $n \in \mathbb{N}_{\geq 1}$:

$$\begin{aligned}(\log n)^d &\leq \left(d \cdot \sqrt[d]{n}\right)^d \qquad \text{(par l'observation ci-dessus)}\\ &= d^d \cdot n.\end{aligned}$$

Ainsi en prenant d^d comme constante multiplicative et 1 comme seuil, nous concluons que $(\log n)^d \in \mathcal{O}(n)$. □

1.13) ★ Nous avons:

$$\begin{aligned}&\lim_{n\to+\infty} \frac{(\log n)^c}{\sqrt[d]{n}}\\ &= \lim_{n\to+\infty} \frac{(c\cdot(\log n)^{c-1})/(\log_e(2)\cdot n)}{n^{(1/d)-1}} \qquad \text{(par L'Hôpital)}\\ &= \lim_{n\to+\infty} \frac{(c\cdot(\log n)^{c-1})/\log_e(2)}{n^{1/d}}\\ &= \frac{c}{\log_e(2)} \cdot \lim_{n\to+\infty} \frac{(\log n)^{c-1}}{\sqrt[d]{n}}\\ &= \frac{c}{\log_e(2)} \cdot \lim_{n\to+\infty} \frac{((c-1)\cdot(\log n)^{c-2})/(\log_e(2)\cdot n)}{n^{(1/d)-1}} \qquad \text{(par L'Hôpital)}\\ &= \frac{c}{\log_e(2)} \cdot \lim_{n\to+\infty} \frac{((c-1)\cdot(\log n)^{c-2})/\log_e(2)}{n^{(1/d)}}\\ &= \frac{c(c-1)}{(\log_e(2))^2} \cdot \lim_{n\to+\infty} \frac{(\log n)^{c-2}}{\sqrt[d]{n}}\\ &= \cdots \qquad \text{(en répétant)}\\ &= \frac{c!}{(\log_e(2))^c} \cdot \lim_{n\to+\infty} \frac{1}{\sqrt[d]{n}}\\ &= 0.\end{aligned}$$

Nous pouvons donc conclure par la règle de la limite. □

1.15) Nous avons:

$$\begin{aligned} f(m,n) &= \frac{mn}{2} + 3m\log(n\cdot 2^n) + 7n \\ &\leq mn + 3m\log(n\cdot 2^n) + 7n \\ &= mn + 3m\log(n) + 3m\log(2^n) + 7n \\ &= mn + 3m\log(n) + 3mn + 7n \\ &\leq mn + 3mn + 3mn + 7n && \text{pour tout } n \geq 1 \\ &\leq mn + 3mn + 3mn + mn && \text{pour tout } m \geq 7 \\ &= 8mn. \end{aligned}$$

Ainsi, nous concluons que $f \in \mathcal{O}(mn)$ en prenant $c \coloneqq 8$ comme constante multiplicative, et $m_0 \coloneqq 7$ et $n_0 \coloneqq 1$ comme seuils. □

1.20) ★ Afin d'obtenir une contradiction, supposons qu'il existe un nombre $n \in \mathbb{N}_{>0}$ sur lequel l'algorithme ne termine pas. Soit x_i la valeur de la variable n après la $i^{\text{ème}}$ itération de la boucle **tant que**. Nous avons $x_0 = n$. De plus, comme n demeure toujours pair, nous avons $x_i = 3x_{i-1}/2$ pour tout $i > 0$. Ainsi,

$$x_i = 3x_{i-1}/2 = 3^2 x_{i-2}/2^2 = \cdots = 3^i x_0/2^i = 3^i n/2^i.$$

Soit $k \in \mathbb{N}$ la plus grande valeur telle que 2^k divise n. Rappelons que x_k est un entier pair par hypothèse. Ainsi, $3^k n/2^k$ est pair. Cela implique que n est divisible par 2^{k+1}, ce qui contradit la maximalité de k. □

1.21)

(a) Il y a $\binom{n}{3} = n(n-1)(n-2)/6$ tels sous-ensembles. Cela donnerait donc un algorithme qui fonctionne en temps $\mathcal{O}(n^3)$ dans le pire cas.

(b) Analysons cet algorithme:

Entrées : séquence s de n entiers
Résultat : trois indices distincts $i, j, k \in [1..n]$ tels que $s[i] + s[j] + s[k] = 0$ s'il en existe, *aucun* sinon

trier s en ordre croissant
pour $i \in [1..n-2]$
  $j \leftarrow i + 1$
  $k \leftarrow n$
  tant que $j < k$
    $somme \leftarrow s[i] + s[j] + s[k]$
    si $somme < 0$ **alors**
      $j \leftarrow j + 1$
    sinon si $somme > 0$ **alors**
      $k \leftarrow k - 1$
    sinon
      retourner (i, j, k)
retourner *aucun*

Nous supposons l'usage d'une implémentation efficace de **trier** qui prend un temps de $\Theta(n \log n)$ dans le meilleur et pire cas.

Meilleur cas: Le temps dans le meilleur cas appartient à $\Omega(n \log n)$ en raison du tri. Considérons la séquence $s_n \coloneqq [0, 0, \ldots, 0]$ de taille n. Pour tout $n \geq 3$, après le tri, exactement 10 opérations élémentaires sont exécutées. Ainsi, sur entrée s_n, l'algorithme fonctionne en temps $\mathcal{O}(n \log n + 10) = \mathcal{O}(n \log n)$. Par conséquent, $t_{\min} \in \Theta(n \log n)$.

Pire cas: La boucle **pour** est exécutée au plus $n - 2$ fois. Son corps exécute 3 opérations élémentaires, suivies de la boucle **tant que**. Celle-ci s'exécute au plus $k-j \leq n-(i+1) \leq n-2$ fois, car la distance entre j et k diminue à chaque itération. La boucle **tant que** exécute une comparaison suivie d'au plus 10 opérations élémentaires. Ainsi, le nombre $f(n)$ d'opérations effectuées après **trier** est tel que:

$$\begin{aligned} f(n) &\leq (n-2)(3 + (n-2) \cdot 11) \\ &= (n-2)(3 + 11n - 22) \\ &\leq n \cdot (3n + 11n) \\ &= 14n^2. \end{aligned}$$

Par conséquent, $t_{\max} \in \mathcal{O}(n \log n + f) = \mathcal{O}(n \log n + n^2) = \mathcal{O}(n^2)$.

Montrons que $t_{\max} \in \Omega(n^2)$. Considérons la séquence $u_n \coloneqq [-1, -1, \ldots, -1]$ de taille n. Clairement, aucun sous-ensemble non vide de u_n ne somme à 0. Ainsi, la boucle **pour** est exécutée un nombre maximal de fois. De plus, le compteur j est incrémenté à chaque tour de la

boucle **tant que**, car nous avons toujours $somme = -3 < 0$. Soit $g(n)$ le temps d'exécution après **trier** sur entrée u_n. Nous avons:

$$\begin{aligned}
g(n) &= \sum_{i=1}^{n-2} \left(3 + \sum_{j=i+1}^{n-1} 9 \right) \\
&\geq \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=1}^{n-2} ((n-1) - (i+1) + 1) \\
&= \sum_{i=1}^{n-2} (n - i - 1) \\
&= \sum_{i=1}^{n-2} i \\
&= (n-2)(n-1)/2 \\
&= (n^2 - 3n + 2)/2 \\
&\geq (n^2 - 3n)/2 \\
&\geq (n^2 - n^2/2)/2 \qquad \text{(pour tout } n \geq 6\text{)} \\
&= (1/4) \cdot n^2.
\end{aligned}$$

Par conséquent, $t_{\max} \in \Omega(n \log n + g) = \Omega(n \log n + n^2) = \Omega(n^2)$. Comme $t_{\max} \in \mathcal{O}(n^2)$, cela implique $t_{\max} \in \Theta(n^2)$.

1.23) Remarquons que $|X \cup Y| = |X| + |Y| - |X \cap Y|$. Ainsi, $|X \cap Y| = |X| + |Y| - |X \cup Y|$. On peut donc calculer l'intersection en temps $\mathcal{O}(1 + 1 + |X| + |Y|) = \mathcal{O}(|X| + |Y|)$. L'indice de Jaccard se calcule ainsi en temps linéaire lorsqu'on évite de calculer directement l'intersection.

Chapitre 2

2.2) Nous donnons un algorithme qui maintient deux indices i et j tels que $s[1:i-1]$ ne contient que des 0 et $s[j+1:n]$ ne contient que des 1. L'algorithme se termine avec $i = j$. Ainsi, à la fin, $s[1:i-1]$ ne contient que des 0, $s[i]$ contient une valeur parmi $\{0,1\}$, et $s[i+1:n]$ ne contient que des 1. L'algorithme n'est pas stable en raison de « croisements » lors de la correction d'inversions.

Entrées : séquence binaire s de n éléments
Sorties : séquence s triée

$i \leftarrow 1$; $j \leftarrow n$
tant que $i < j$ // Invar.: $s[1:i-1]$ ne contient que des 0
  si $s[i] = 0$ **alors** // $s[j+1:n]$ ne contient que des 1
    $i \leftarrow i + 1$
  sinon
    $s[i] \leftrightarrow s[j]$
    $j \leftarrow j - 1$
retourner s

Voici un autre algorithme sur place. Il est similaire à l'algorithme précédent, mais j se déplace vers la droite plutôt que la gauche. Il n'est pas stable pour une raison similaire.

Entrées : séquence binaire s de n éléments
Sorties : séquence s triée

$i \leftarrow 1$; $j \leftarrow 2$
tant que $j \leq n$ // Invar.: $s[1:i-1]$ ne contient que des 0
  si $s[i] = 0$ **alors** // $s[i+1:j-1]$ ne contient que des 1
    $i \leftarrow i + 1$
  sinon si $s[j] = 1$ **alors**
    $j \leftarrow j + 1$
  sinon
    $s[i] \leftrightarrow s[j]$
  si $j \leq i$ **alors**
    $j \leftarrow i + 1$
retourner s

Algorithme proposé en classe par un·e étudiant·e (A23).

Voici un autre algorithme sur place. Celui-ci suppose que s est sous forme de liste chaînée. Chaque fois qu'on rencontre un élément dont la valeur est 0, on le fait pointer vers la tête de la liste, et on fait pointer son prédécesseur vers son successeur. Chaque fois qu'on rencontre un élément dont la valeur est 1, on le laisse tel quel. L'algorithme n'est pas stable comme tous les 0 sont renversés.

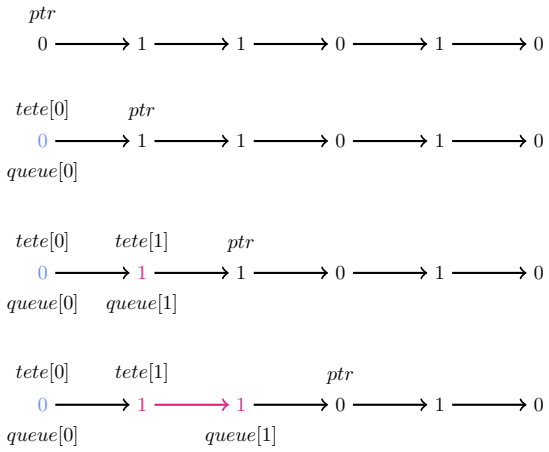
Entrées : séquence binaire s sous forme de liste chaînée
Sorties : séquence s triée

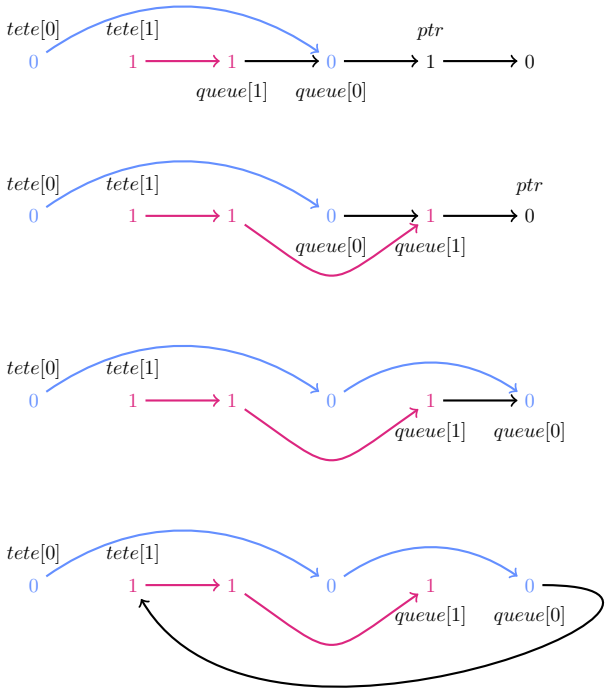
$tete \leftarrow s$
$ptr \leftarrow s$
$pred \leftarrow \bot$
tant que $ptr \neq \bot$
  si $(pred \neq \bot) \wedge (ptr.val = 0)$ **alors**
    $pred.succ \leftarrow ptr.succ$
    $ptr.succ \leftarrow tete$
    $tete \leftarrow ptr$
    $ptr \leftarrow pred.succ$
  sinon
    $pred \leftarrow ptr$
    $ptr \leftarrow ptr.succ$
retourner $tete$

Algorithme proposé en classe par un·e autre étudiant·e (A23).

Nous présentons un autre algorithme sur place. Celui-ci suppose que s est sous forme de liste chaînée. On cherche à maintenir deux listes: celle des 0 et celle des 1. On garde en mémoire la tête et la queue de ces deux listes. À la toute fin, on fait pointer la queue de la liste des 0 vers la tête de la liste des 1. Cet algorithme est stable comme les deux listes sont construites progressivement de gauche à droite.

Voici un exemple de l'exécution de l'algorithme:





Voici le pseudocode de l'algorithme:

Entrées : séquence binaire s sous forme de liste chaînée
Sorties : séquence s triée

$tete \leftarrow [\bot, \bot]$
$queue \leftarrow [\bot, \bot]$
$ptr \leftarrow s$
// Créer deux listes pour les 0 et 1
tant que $ptr \neq \bot$
  $i \leftarrow ptr.valeur$
  si $tete[i] = \bot$ **alors**
    $tete[i] \leftarrow ptr$
  sinon
    $queue[i].succ = ptr$
  $queue[i] \leftarrow ptr$
  $ptr \leftarrow ptr.succ$
// Connecter les deux listes
si $tete[0] = \bot$ **alors**
  retourner $tete[1]$
sinon
  $queue[0].succ \leftarrow tete[1]$
  si $queue[1] \neq \bot$ **alors**
    $queue[1].succ \leftarrow \bot$
  retourner $tete[0]$

Algorithme conçu avec Vincent Beaudoin (A23).

2.6) On insère la spatule sous la plus grande crêpe, puis on renverse. Ensuite, on insère la spatule sous la pile et on l'inverse au complet. La plus grande crêpe est maintenant à la dernière position. En répétant ce processus n fois, on obtient une pile de crêpes triée. Puisque chaque itération nécessite deux renversements, on obtient $\mathcal{O}(n)$ renversements au total. Une analyse légèrement plus détaillée montre que cette procédure effectue $2n - 3$ renversements.

Remarque.

Il est possible d'obtenir une meilleure constante multiplicative:

> *When I was an assistant professor at Harvard, Bill was a junior. My girlfriend back then said that I had told her: "There's this undergrad at school who is the smartest person I've ever met."*
>
> *That semester, Gates was fascinated with a math problem called pancake sorting: How can you sort a list of numbers, say 3-4-2-1-5, by flipping prefixes of the list? You can flip the first two numbers to get 4-3-2-1-5, and the first four to finish it off:*

> *1-2-3-4-5. Just two flips. But for a list of n numbers, nobody knew how to do it with fewer than 2n flips. Bill came to me with an idea for doing it with only 1.67n flips. We proved his algorithm correct, and we proved a lower bound–it cannot be done faster than 1.06n flips. We held the record in pancake sorting for decades. It was a silly problem back then, but it became important, because human chromosomes mutate this way.*
>
> *Two years later, I called to tell him our paper had been accepted to a fine math journal. He sounded eminently disinterested. He had moved to Albuquerque, New Mexico to run a small company writing code for microprocessors, of all things. I remember thinking: "Such a brilliant kid. What a waste."*
>
> — Christos Papadimitriou

2.12) On peut identifier le centre en avançant en alternance un pointeur d'une position et un autre de deux positions. Le reste de l'approche demeure la même:

Entrées : liste chaînée s d'éléments comparables
Sorties : liste chaînée correspondant à s triée

```
fusion(x, y):
    si x = ⊥ alors retourner y
    sinon si y = ⊥ alors retourner x
    sinon
        // Identifier la première valeur
        si x.valeur ≤ y.valeur alors
            z ← x
            x ← x.succ
        sinon
            z ← y
            y ← y.succ
        // Construire le reste de la liste
        debut ← z
        tant que x ≠ ⊥ ou y ≠ ⊥
            si (y = ⊥) ou (x ≠ ⊥ et x.valeur ≤ y.valeur) alors
                z.succ ← x
                x      ← x.succ
            sinon
                z.succ ← y
                y      ← y.succ
            z ← z.succ
        retourner debut

trier(s):
    si s = ⊥ ou s.succ = ⊥ alors
        retourner s
    sinon
        // Trouver le centre
        x ← s
        y ← s.succ
        tant que (y ≠ ⊥) et (y.succ ≠ ⊥)
            x ← x.succ
            y ← y.succ.succ
        // Scinder en deux
        y      ← x.succ
        x.succ ← ⊥
        x      ← s
        // Trier récursivement et fusionner
        retourner fusion(trier(x), trier(y))
```

2.13) L'initialisation de c prend un temps de $\Theta(k)$. Le décompte du nombre d'occurrences de chaque valeur prend un temps de $\Theta(n)$. Puisqu'une valeur ne pas apparaître plus de n fois dans s, nous avons $0 \leq c[x] \leq n$ pour chaque x. Ainsi, le dernier bloc prend un temps de $\mathcal{O}(k \cdot n)$. Cette dernière analyse n'est pas suffisamment fine. En effet, la complexité du dernier bloc est de

$$\Theta\left(\sum_{x=1}^{k} c[x]\right) = \Theta(n),$$

puisque la somme des décomptes correspond au nombre total d'éléments. Ainsi, le temps total appartient à $\Theta(k + n + n) = \Theta(n + k)$.

L'algorithme *détruit* l'identité des éléments lors de la « repopulation » de s, ce qui est « pire » qu'être non stable. L'implémentation suivante préserve leur identité et leur ordre relatif, avec la même complexité algorithmique, en contrepartie de l'usage d'une seconde séquence:

Entrées : $k \in \mathbb{N}_{\geq 1}$, séquence s de n entiers appartenant à $[1..k]$
Sorties : séquence s triée

$c \leftarrow \overbrace{[0, 0, \ldots, 0]}^{k \text{ fois}}$

pour $x \in s$
  $c[x] \leftarrow c[x] + 1$

$total \leftarrow 0$

pour $i \in [1..k]$
  $c[i], total \leftarrow total, total + c[i]$

$t \leftarrow \overbrace{[\bot, \bot, \ldots, \bot]}^{n \text{ fois}}$

pour $x \in s$
  $t[c[x]] \leftarrow x$
  $c[x] \leftarrow c[x] + 1$

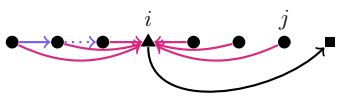
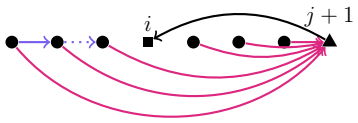
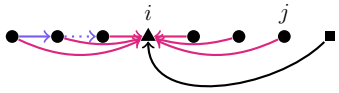
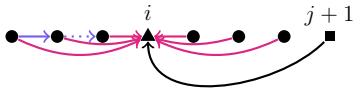
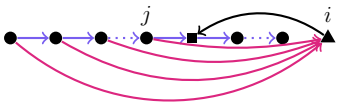
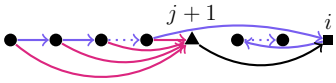
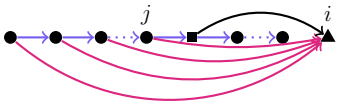
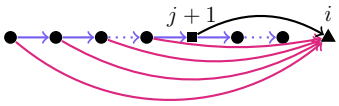
retourner t

2.14) ★ Nous procédons par induction lexicographique sur (i, j). Nous utiliserons des diagrammes afin de représenter s, où chaque lien de la forme « $k \to \ell$ » dénote « $s[k] \leq s[\ell]$ ».

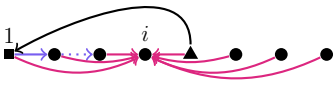
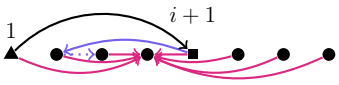
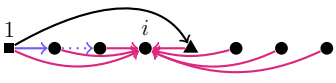
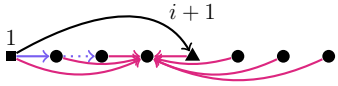
- Lorsque $i = j = 1$, l'invariant est trivialement vrai.
- Supposons l'invariant vrai pour (i, j) où $i \in [1..n]$ et $j \in [1..n-1]$. Montrons qu'il est vrai pour $(i, j+1)$. Nous considérons les quatre cas possibles.

  Avant la $(j+1)^{\text{ème}}$ exécution de la boucle interne, nous avons $s[1] \leq \cdots \leq s[i-1]$ et $s[i] \geq \max s[1:j]$ par hypothèse d'induction. Après la

$(j+1)^{\text{ème}}$ exécution, nous avons bien $s[1] \leq \cdots \leq s[i-1]$ et $s[i] \geq \max s[1:j+1]$:

avant	**après**
	
	
	
	

► Supposons l'invariant vrai pour (i, n) où $i \in [1..n-1]$. Montrons qu'il est vrai pour $(i+1, 1)$. Considérons la $(i+1)^{\text{ème}}$ itération de la boucle externe. Avant la $1^{\text{ère}}$ exécution de la boucle interne, nous avons $s[1] \leq \cdots \leq s[i-1]$ et $s[i] \geq \max s[1:n]$ par hypothèse d'induction. Après la $1^{\text{ère}}$ exécution, nous avons bien $s[1] \leq \cdots \leq s[i]$ et $s[i+1] \geq \max s[1:1]$:

avant	**après**
	
	

2.16) Si l'on corrige successivement les k inversions adjacentes, on obtient une séquence triée.

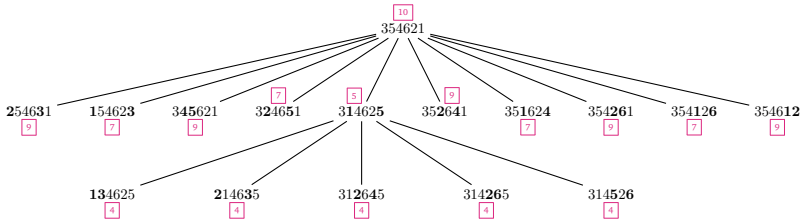
Montrons, par induction, qu'il est impossible de faire mieux. Si $k = 0$, alors s est trivialement déjà triée. Soit $k > 0$. Si l'on corrige une inversion

adjacente de s, alors on obtient une séquence s' avec une inversion en moins et dont toutes les inversions sont encore adjacentes. Par hypothèse d'induction, il est impossible de trier s' en moins de $k-1$ corrections. Il est donc impossible de trier s en moins de $1+(k-1)=k$ corrections. □

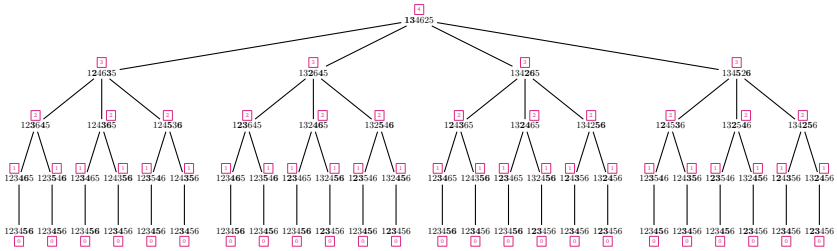
2.17) ★ La séquence $[3,5,4,6,2,1]$ peut être triée à l'aide de quatre corrections d'inversions:

$$\begin{aligned}
&[3,5,4,6,2,1] && (10 \text{ inversions})\\
\rightarrow\ &[3,\mathbf{2},4,6,\mathbf{5},1] && (7 \text{ inversions})\\
\rightarrow\ &[\mathbf{1},2,4,6,5,\mathbf{3}] && (4 \text{ inversions})\\
\rightarrow\ &[1,2,4,\mathbf{3},5,\mathbf{6}] && (1 \text{ inversions})\\
\rightarrow\ &[1,2,\mathbf{3},\mathbf{4},5,6] && (0 \text{ inversions}).
\end{aligned}$$

Toutefois, l'approche gloutonne effectue six corrections. Voyons pourquoi. À la première itération, il y a 10 inversions parmi lesquelles choisir. En corrigeant l'inversion prometteuse, on réduit à 5 inversions. Ces 5 inversions sont toutes prometteuses: elles mènent chacune à 4 inversions:



L'approche gloutonne peut donc mener à n'importe quelle de ces cinq séquences. Considérons la première de celles-ci. Les inversions sont toujours prometteuses; on obtient quatre corrections peu importe ce qui est fait:



Les quatre autres séquences mènent à des arbres similaires.

Chapitre 3

3.1) Un graphe possède un cycle ssi un parcours en profondeur découvre une arête $u \to v$ telle que v est gris. La validité de cette affirmation mène directement à un algorithme: on lance un parcours en profondeur à partir d'un sommet arbitraire et on retourne « cyclique » ssi une telle arête est découverte.

$\Rightarrow$) Soit C un cycle, soit u le sommet de C noirci en premier, et soit v un successeur de u dans C. Par minimalité de u, nous avons $f[u] < f[v]$. Considérons le temps t où $u \to v$ est découverte. À ce moment, u est gris. Nous montrons que v est aussi gris. Si v est blanc, alors $d[u] < t \leq d[v] < f[v]$. Par le théorème des parenthèses (théorème 3), nous avons

$$d[u] < d[v] < f[v] < f[u],$$

ce qui contredit $f[u] < f[v]$. Si v est noir, alors $f[v] < t < f[u]$, ce qui est aussi une contradiction.

$\Leftarrow$) Par la proposition 20, les sommets gris forment un chemin simple. Lors de la découverte de $u \to v$, le sommet u est le dernier sommet de C. Ainsi, $v \xrightarrow{*} u \to v$. □

3.2)

Entrées : matrice $\mathbf{A} \in \{0,1\}^{n \times n}$ d'un groupe qui contient un intrus
Résultat : indice de l'intrus

$i \leftarrow 1$
$j \leftarrow 1$
tant que $j \leq n$
  si $i = j$ **alors**
    $j \leftarrow j + 1$
  sinon si $A[i, j] = 0$ **alors** `// i n'est pas l'intrus`
    $i \leftarrow i + 1$
  sinon `// j n'est pas l'intrus`
    $j \leftarrow j + 1$
retourner i

Afin d'alléger la notation, écrivons $[x..y)$ afin de dénoter $[x..y-1]$. Démontrons que l'algorithme ci-dessus est correct. Montrons d'abord qu'il satisfait ces invariants de la boucle:

- $i \leq j$,
- chaque personne $p \in [1..j) \setminus \{i\}$ n'est pas une intruse.

Au départ, nous avons $i = j = 1$ et $[1..j) \setminus \{i\} = \emptyset$. L'invariant est donc satisfait trivialement avant la première itération. Supposons maintenant que l'invariant soit satisfait et qu'on exécute le corps de la boucle. Dénotons les nouvelles valeurs par i' et j'. Il y a trois cas possibles:

- *Cas* $i = j$. Nous obtenons $j' = i' + 1$ et ainsi $i' < j'$. Par hypothèse, chaque personne $p \in [1..j) \setminus \{i\}$ n'est pas une intruse. Notons que $[1..j') \setminus \{i'\} = [1..j] \setminus \{i\} = [1..j) \setminus \{i\}$, où la dernière égalité découle de $i = j$. Ainsi, l'affirmation demeure la même.
- *Cas* $i \neq j$ *et* $A[i,j] = 0$. Comme $i \neq j$, par hypothèse nous avons $i < j$. Nous obtenons $i' = i + 1 \leq j = j'$. Par hypothèse, chaque personne $p \in [1..j) \setminus \{i\}$ n'est pas une intruse. Puisque $A[i,j] = 0$, la personne i ne connaît pas la personne j. Ainsi, i n'est pas l'intrus. Par conséquent, chaque personne $p \in [1..j)$ n'est pas une intruse. En particulier, chaque personne $p \in [1..j) \setminus \{i'\}$ n'est pas une intruse.
- *Cas* $i \neq j$ *et* $A[i,j] = 1$. Par hypothèse nous avons $i \leq j$, et nous obtenons donc $i' = i \leq j < j+1 = j'$. Par hypothèse, chaque personne $p \in [1..j) \setminus \{i\}$ n'est pas une intruse. Puisque $A[i,j] = 1$, la personne i connaît la personne j. Ainsi, j n'est pas l'intrus. Par conséquent, chaque personne $p \in [1..j] \setminus \{i\}$ n'est pas une intruse. Comme $i' = i$ et $j' = j+1$, nous pouvons affirmer de façon équivalente que chaque personne $p \in [1..j') \setminus \{i'\}$ n'est pas une intruse.

Voyons maintenant pourquoi l'invariant implique la correction de l'algorithme. À la fin de l'exécution, nous savons que $i \leq j = n+1$ et que chaque personne $p \in [1..n] \setminus \{i\}$ n'est pas une intruse. Ainsi, le seul candidat est i. Comme nous avons la promesse que le groupe contient un intrus, il s'agit forcément de i (et en particulier $i \leq n$). □

3.4) On marque les sommets en alternance entre rouge et noir:

Entrées : graphe non dirigé $\mathcal{G} = (V, E)$
Résultat : $\mathcal{G}$ est biparti?

$biparti \leftarrow$ `vrai`
$couleur \leftarrow [v \mapsto$ *aucune* $: v \in V]$

`colorier(`u, v`):`
  si $couleur[u] =$ *aucune* **alors**
    $couleur[u] \leftarrow c$
    pour $v : u \to v$
      `colorier(`v, *couleur inverse de* c`)`
  sinon si $couleur[u] \neq c$ **alors**
    $biparti \leftarrow$ `faux`
pour $v \in V$
  si $couleur[v] =$ *aucune* **alors**
    `colorier(`v, *rouge*`)`
retourner $biparti$

3.5) Soit $\mathcal{G} = (V, E)$ un graphe dirigé.

⇐) Si $\mathcal{G}$ contient un cycle simple, alors clairement il contient un cycle.

⇒) Soit $C = [v_0, v_1, \ldots, v_k]$ un cycle de $\mathcal{G}$. Si C est simple, alors nous avons terminé. Sinon, il existe deux indices $0 \leq i < j < k$ tels que $v_i = v_j$. Posons $C' \coloneqq [v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_k]$. Remarquons que C' est un cycle plus court que C. Ainsi, en répétant ce processus suffisamment de fois, nous devons obtenir un cycle simple. □

3.6) ★

a) Comme les sommets isolés ne jouent aucun rôle, nous pouvons les retirer et considérer que $\mathcal{G}$ est connexe. Prenons un sommet u quelconque de $\mathcal{G}$ et choisissons progressivement des arêtes afin de former un chemin. À un certain point, nous revenons forcément à u. Autrement dit, peu importe la façon de procéder, nous obtenons un chemin C de la forme $u = v_0 \to v_1 \to \cdots \to v_k = u$. En effet, comme chaque sommet est de degré pair, lorsqu'on entre dans un sommet v_i via une arête, il en existe une autre qui permet de quitter v_i. Retirons C de $\mathcal{G}$. Nous obtenons un sous-graphe $\mathcal{G}'$ avec moins d'arêtes et dont tous les sommets sont de degré pair. S'il n'y a plus d'arête, nous avons terminé. Autrement, nous pouvons répéter le processus avec un autre sommet u' adjacent à un sommet de C (il en existe forcément un par connexité), et obtenir un cycle C' qui débute en u'. Nous pouvons donc fusionner C et C' afin d'obtenir un seul cycle. Le processus se répète ainsi jusqu'à ce qu'il ne reste plus aucune arête.

b) La preuve précédente est constructive et clairement implémentable.

Remarque.

Il s'agit de l'algorithme de Hierholzer.

c) Il existe un chemin eulérien ssi le graphe contient au plus une composante connexe non triviale et tous ses sommets sont de degrés pair sauf possiblement deux. En effet, supposons qu'il y ait exactement deux sommets u et w de degré impair. On peut ajouter un nouveau sommet v entre u et w, ainsi que les arêtes $\{u, v\}$ et $\{v, w\}$. Tous les sommets sont maintenant de degré pair. Nous pouvons donc construire un cycle eulérien C pour ce graphe, puis retirer $u \to v \to w$, ce qui donne un chemin eulérien du graphe de départ (dont les extrémités sont u et v).

3.8) Soit $\mathcal{G}$ un graphe dirigé qui possède un ordre topologique T. Afin d'obtenir une contradiction, supposons que $\mathcal{G}$ contienne un cycle $C = [v_0, v_1, \ldots, v_k]$. Écrivons $x \prec y$ pour dénoter « x apparaît avant y dans l'ordre topologique T ». Par définition, nous devons avoir $v_0 \prec v_1 \prec \cdots \prec v_k$, et ainsi $v_0 \prec v_k$ par transitivité. Nous obtenons une contradiction puisque $v_0 = v_k$. □

3.11) Nous allons montrer que ces trois propriétés sont équivalentes:

a) il existe un unique ordre topologique;

b) il existe un ordre topologique $v_1, v_2, \ldots, v_k$ t.q. $v_1 \to v_2 \to \cdots \to v_k$;

c) il existe un chemin hamiltonien.

Avant de démontrer ces équivalences, donnons un algorithme qui répond à l'exercice. Celui-ci identifie un ordre topologique $v_1, v_2, \ldots, v_k$ et retourne `vrai` ssi $v_1 \to v_2 \to \cdots \to v_k$. Cet algorithme est correct par l'équivalence de a), b) et c).

Montrons maintenant les équivalences.

$a) \Rightarrow b)$. Soit $v_1, v_2, \ldots, v_k$ l'unique ordre topologique. Afin d'obtenir une contradiction, supposons qu'il existe $i \in [1..k-1]$ tel que $v_i \not\to v_{i+1}$. En intervertissant v_i et v_{i+1}, nous avons toujours un ordre topologique, ce qui contredit l'unicité de l'ordre.

$b) \Rightarrow c)$. Le chemin $v_1 \to v_2 \to \cdots \to v_k$ visite chaque sommet exactement une fois.

$c) \Rightarrow a)$. Soit $v_1 \to v_2 \to \cdots \to v_k$ un chemin hamiltonien. Comme le graphe est acyclique, $v_1, v_2, \ldots, v_k$ est un ordre topologique. Afin d'obtenir une contradiction, supposons qu'il existe un autre ordre topologique $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$. Soit $j \in [1..k]$ le premier indice tel que $i_j \neq j$. Par minimalité de j, nous avons $i_j > j$. Soit $\ell \in [1..k]$ tel que $i_\ell = i_j - 1$. Nous avons forcément $\ell \geq j$. En effet, autrement, par minimalité de j, nous aurions $j > \ell = i_\ell = i_j - 1 \geq j$. Ainsi, $j < \ell$.Comme $v_{i_\ell} \to v_{i_j}$, cela contredit le fait que $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ soit un ordre topologique. □

Chapitre 4

4.1) Les deux algorithmes présentés fonctionnent directement avec les poids négatifs (aucune modification nécessaire). Si l'on ne croît pas que ces algorithmes fonctionnent avec des poids négatifs (à défaut d'avoir vu une preuve!), on peut argumenter qu'ils peuvent être adaptés. Soit $\mathcal{G} = (V, E)$ un graphe non dirigé pondéré par p et soit c son plus petit poids, c.-à-d. $c = \min\{p[e] : e \in E\}$. On remplace p par p' tel que $p'[e] \coloneqq p[e] + |c| + 1$. Un arbre couvrant de $\mathcal{G}$ est minimal sous p si et seulement si il est minimal sous p'. De plus, p' ne possède que des poids positifs.

4.2) Les deux algorithmes présentés fonctionnent également sur les poids négatifs. Il suffit donc de multiplier chaque poids par -1 et de rechercher un arbre couvrant minimal.

4.3) Minimiser $p[1] \cdot p[2] \cdots p[n]$ est équivalent à minimiser son logarithme: $\log(p[1] \cdot p[2] \cdots p[n]) = \log p[1] + \log p[2] + \ldots + \log p[n]$. Il suffit donc de remplacer chaque poids par son logarithme et de rechercher un arbre couvrant minimal sous la définition standard.

4.5) En $\mathcal{O}(\max(|V|, |E|) \cdot \log |V|)$:

Entrées : graphe non dirigé $\mathcal{G} = (V, E)$
Résultat : composantes connexes de $\mathcal{G}$

`init(`V`)`
pour $u \in V$
  pour $v : u \to v$
    `union(`u, v`)`
$c \leftarrow [v \mapsto [\,] : v \in V]$
pour $v \in V$
  $u \leftarrow$ `trouver(`v`)`
  ajouter v à $c[u]$
composantes $\leftarrow [\,]$
pour $v \in V$
  si $c[v] \neq [\,]$ **alors ajouter** $c[v]$ **à** *composantes*
retourner *composantes*

4.6) La procédure gloutonne remplit trois bacs plutôt que deux pour $c \coloneqq 20$ et $p \coloneqq [10, 6, 6, 6, 5, 5]$.

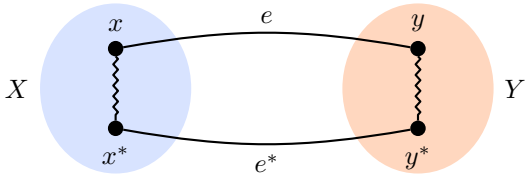
4.10) ★★

(a) Nous adaptons simplement la preuve du théorème 4. Les modifications apparaissent en couleur.

Démonstration. Soit $\mathcal{G} = (V, E)$ un graphe non dirigé connexe pondéré par p (où les poids proviennent de $\mathbb{D}$). Soit $E' \subseteq E$ l'ensemble d'arêtes retourné par l'algorithme de Prim–Jarník sur entrée $(\mathcal{G}, p)$. Il est assez simple de démontrer que E' forme un arbre couvrant $\mathcal{T} \coloneqq (V, E')$. Montrons qu'il s'agit d'un arbre couvrant *minimal* sous $\preceq$.

Soit $E^* \subseteq E$ un ensemble d'arêtes tel que $\mathcal{T}^* \coloneqq (V, E^*)$ est un arbre couvrant minimal. Si $E^* = E'$, alors nous avons terminé. Supposons que $E^* \neq E'$. Remarquons que $E' \setminus E^* \neq \emptyset$ puisque E' et E sont de même taille mais diffèrent. Soit $e = \{x, y\} \in E' \setminus E^*$. Lorsque l'algorithme a sélectionné e, il s'agissait d'une arête *minimale* sous $\preceq$ parmi celles avec un sommet dans l'ensemble X des sommets marqués et un sommet dans l'ensemble $Y \coloneqq V \setminus X$ des sommets non marqués.

Puisque $\mathcal{T}^*$ est un arbre couvrant, il contient un chemin entre x et y. Puisque $e \notin E^*$, ce chemin contient une arête $e^* = \{x^*, y^*\} \in E^* \setminus E'$ qui traverse X et Y, c.-à-d. telle que $x^* \in X$ et $y^* \in Y$. Schématiquement, nous avons:



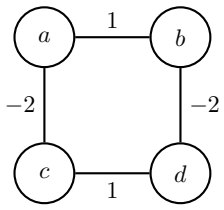
Posons $\mathcal{T}^{**} \coloneqq (V, E^* \setminus \{e^*\} \cup \{e\})$. Autrement dit, retirons e^* de $\mathcal{T}^*$ et ajoutons-lui e. Puisque $\mathcal{T}^{**}$ est connexe et a $|V| - 1$ arêtes, il s'agit d'un arbre couvrant. Par minimalité de $\mathcal{T}^*$, nous avons $p(\mathcal{T}^{**}) \succeq p(\mathcal{T}^*)$. De plus, par minimalité de e, nous avons $p[e^*] \succeq p[e]$. Posons $z = \sum_{f \in E^* \setminus \{e^*\}} p[f]$ (où la somme est sous $\oplus$). Nous avons:

$$\begin{aligned} p(\mathcal{T}^{**}) &= p[e] \oplus z && \text{(par déf. de } \mathcal{T}^{**}) \\ &\preceq p[e^*] \oplus z && \text{(par } p[e] \preceq p[e^*] \text{ et par compatibilité de } \preceq) \\ &= p(\mathcal{T}^*) && \text{(par déf. de } \mathcal{T}^*.) \end{aligned}$$

Par conséquent, $\mathcal{T}^{**}$ est minimal. De plus, $\mathcal{T}^{**}$ possède une arête de plus en commun avec $\mathcal{T}$. Ainsi, en répétant ce processus jusqu'à avoir remplacé toutes les arêtes qui diffèrent, nous en concluons que $\mathcal{T}$ est minimal. □

(b) Toutes les propriétés sont satisfaites à l'exception de la compatibilité. En effet, nous avons, par exemple, $0 \leq 1$ et $0 \cdot (-1) \geq 1 \cdot (-1)$.

(c) Sur le graphe ci-dessous, l'algorithme de Prim–Jarník retourne un arbre couvrant de poids $(-2)\cdot 1\cdot(-2) = 4$, alors que le poids minimal est $1\cdot(-2)\cdot 1 = -2$.



(d) Nous avons vu que l'algorithme de Prim–Jarník est correct sous $(\mathbb{N}, 0, +, \leq)$ et $(\mathbb{N}, 1, \cdot, \leq)$, car ce sont tous deux des monoïdes abéliens totalement ordonnés. De plus, sous $(\mathbb{D}, 0_{\mathbb{D}}, \oplus, \preceq)$, l'algorithme de Prim–Jarník choisit les arêtes en se basant uniquement sur $\preceq$ (jamais sur $0_{\mathbb{D}}$ ou $\oplus$). Ainsi, l'algorithme de Prim–Jarník retourne les mêmes arbres couvrants de poids minimaux sous $(\mathbb{N}, 0, +, \leq)$ et $(\mathbb{N}, 1, \cdot, \leq)$, car le même ordre est utilisé.

(e) Remarquez que si $\preceq$ est un ordre total, alors $\succeq$ en est aussi un. De plus, si $\preceq$ est compatible, alors c'est aussi le cas de $\succeq$. En effet, nous avons

$$\begin{aligned} x \succeq y &\iff y \preceq x \\ &\implies y + z \preceq x + z \qquad \text{(par compatibilité de } \preceq) \\ &\iff x + z \succeq y + z. \end{aligned}$$

Ainsi, si $(\mathbb{D}, 0_{\mathbb{D}}, \oplus, \preceq)$ est un monoïde totalement ordonné, alors c'est aussi le cas de $(\mathbb{D}, 0_{\mathbb{D}}, \oplus, \succeq)$. Par conséquent, afin d'identifier un arbre couvrant *maximal*, par exemple, sous $(\mathbb{N}, 0, +, \leq)$, il suffit d'en obtenir un *minimal* sous $(\mathbb{N}, 0, +, \geq)$.

(f) Soit $\wr\wr$ le multi-ensemble vide, soit $\uplus$ l'opération qui combine deux multi-ensembles, et soit $\preceq$ l'ordre défini par $M \preceq M'$ ssi

$$(M[a] > M'[a]) \vee (M[a] = M'[a] \wedge M[b] > M'[b]) \vee \cdots .$$

Le tuple $(\{a, b, \ldots\}, \wr\wr, \uplus, \preceq)$ est un monoïde abélien totalement ordonné. Ainsi, l'algorithme de Prim–Jarník permet de calculer un arbre couvrant minimal sous l'ordre $\preceq$.

Chapitre 5

5.1) A) Polynôme caractéristique: $x^2 - x - 6 = (x-3)(x+2)$.

B) Forme close: $t(n) = c_1 \cdot 3^n + c_2 \cdot (-2)^n$.

C) Identification des constantes:

$$\begin{array}{rcrcr} 1 & = & c_1 & + & c_2 \\ 2 & = & 3c_1 & - & 2c_2 \end{array}$$

Donc, $c_1 = 4/5$ et $c_2 = 1/5$, et ainsi

$$t(n) = (4/5) \cdot 3^n + (1/5) \cdot (-2)^n \in \Theta(3^n).$$

5.2) A) Polynôme caractéristique: $x^2-x-2 = (x-2)(x+1)$, puis on multiplie par $(x-1)$ car non homogène.

B) Forme close: $t(n) = c_1 \cdot 2^n + c_2 \cdot (-1)^n + c_3 \cdot 1^n$.

C) Identification des constantes:

$$\begin{array}{rcrcrcr} 0 & = & c_1 & + & c_2 & + & c_3 \\ 1 & = & 2c_1 & - & c_2 & + & c_3 \\ 4 & = & 4c_1 & + & c_2 & + & c_3 \end{array}$$

Donc, $c_1 = 4/3$, $c_2 = 1/6$ et $c_3 = -3/2$, et ainsi

$$t(n) = (4/3) \cdot 2^n + (1/6) \cdot (-1)^n - (3/2) \in \Theta(2^n).$$

5.3) A) Polynôme caractéristique: $x^2 - 1 = (x-1)(x+1)$.

B) Forme close: $t(n) = c_1 \cdot 1^n + c_2 \cdot (-1)^n$.

C) Identification des constantes:

$$\begin{array}{rcrcr} 1 & = & c_1 & + & c_2 \\ 0 & = & c_1 & - & c_2 \end{array}$$

Donc, $c_1 = c_2 = 1/2$, et ainsi

$$t(n) = (1/2) \cdot 1^n + (1/2) \cdot (-1)^n = \frac{1 + (-1)^n}{2} \in \Theta(1).$$

5.5) En substitutant la récurrence à répétition, nous obtenons

$$
\begin{aligned}
t(n) &= d \cdot t(n-1) + c \\
&= d \cdot (d \cdot t(n-2) + c) + c && \text{(par la relation de récurrence)} \\
&= d^2 \cdot t(n-2) + dc + c \\
&= d^2 \cdot (d \cdot t(n-3) + c) + dc + c && \text{(par la relation de récurrence)} \\
&= d^3 \cdot t(k-3) + d^2 + dc + c \\
&\vdots \\
&= d^i \cdot t(k-i) + c \sum_{j=0}^{i-1} d^j && \text{(en répétant } i \text{ fois)} \\
&\vdots \\
&= d^n \cdot t(0) + c \sum_{j=0}^{n-1} d^j && \text{(en répétant } n \text{ fois)} \\
&= c \sum_{j=0}^{n-1} d^j && \text{(car } t(0) = 0\text{)}.
\end{aligned}
$$

Si $d = 1$, alors $t \in \Theta(n)$ car $t(n) = cn$. Si $d > 1$, alors $t(n) = c\frac{d^n - 1}{d-1}$ car il s'agit d'une série géométrique de raison d. Ainsi, $t \in \Theta(d^n)$ si $d > 1$.

5.7) On adapte le tri par fusion afin d'identifier les inversions en triant:

Entrées : séquence s d'éléments comparables
Sorties : séquence s triée et nombre d'inversions

```
trier(s):
    fusion(x, y):
        i ← 1; j ← 1; z ← []
        c ← 0          // nombre d'inversions entre x et y
        tant que i ≤ |x| ∧ j ≤ |y|
            si x[i] ≤ y[j] alors
                ajouter x[i] à z
                i ← i + 1
            sinon
                ajouter y[j] à z
                j ← j + 1
                c ← c + (|x| − i + 1)
        retourner (z + x[i : |x|] + y[j : |y|], c)
    si |s| ≤ 1 alors retourner (s, 0)
    sinon
        m ← |s| ÷ 2
        x, a ← trier(s[1 : m])
        y, b ← trier(s[m + 1 : |s|])
        z, c ← fusion(x, y)
        retourner (z, a + b + c)
```

5.8) Cet algorithme mène à la suite de Fibonacci $f(n) = f(n-1) + f(n-2)$:

Entrées : $n \in \mathbb{N}_{\geq 1}$
Résultat : séquence de tous les pavages d'une grille $2 \times n$

```
pavages(n):
    si n = 1 alors
        retourner [▮]
    si n = 2 alors
        retourner [▮▮, ▮▮]
    sinon
        P ← pavages(n − 1)
        Q ← pavages(n − 2)
        retourner [▮+p : p ∈ P] + [▮▮+q : q ∈ Q]
```

5.12) On adapte la recherche dichotomique afin d'identifier la première occurrence de 1:

Entrées : séquence de bits s ordonnée de façon croissante
Sorties : nombre d'occurrences de `1` dans s

```
compter(s):
    premier-un(lo, hi):
        si lo = hi alors
            si s[lo] = 1 alors retourner lo
            sinon retourner aucune
        sinon
            mid ← (lo + hi) ÷ 2
            si s[mid] = 0 alors
                retourner premier-un(mid + 1, hi)
            sinon
                retourner premier-un(lo, mid)
    pos ← premier-un(1, |s|)
    si pos ≠ aucune alors retourner |s| − pos + 1
    sinon retourner 0
```

Remarquez qu'on peut également déterminer en temps constant si la séquence contient au moins une occurrence de de `1`. En effet, comme la séquence est ordonnée, il suffit de vérifier si le dernier bit est `1`. Cela mène à cette implémentation alternative (mais très similaire):

Entrées : séquence de bits s ordonnée de façon croissante
Sorties : nombre d'occurrences de `1` dans s

```
compter(s):
    premier-un(lo, hi):
        si lo = hi alors
            retourner lo
        sinon
            mid ← (lo + hi) ÷ 2
            si s[mid] = 0 alors
                retourner premier-un(mid + 1, hi)
            sinon
                retourner premier-un(lo, mid)
    si |s| > 0 et s[|s|] = 1 alors
        pos ← premier-un(1, |s|)
        retourner |s| − pos + 1
    sinon retourner 0
```

Voici une solution alternative proposée par Thomas Arcand (A22):

Entrées : séquence de bits s ordonnée de façon croissante
Sorties : nombre d'occurrences de `1` dans s

```
compter(s):
    si |s| = 0 alors retourner 0
    sinon si |s| = 1 alors retourner s[1]
    sinon
        m ← |s| ÷ 2
        si s[m] = 1 alors
            retourner compter(s[1 : m]) + (|s| − m)
        sinon
            retourner compter(s[m + 1 : |s|])
```

5.14)

a) On adapte la recherche dichotomique afin d'identifier le côté de la séquence qui contient l'index de départ i:

Entrées : séquence s de $n \in \mathbb{N}_{\geq 1}$ éléments comparables distincts triés circulairement
Sorties : $\max(s)$

```
max-circ(s):
    max-circ'(lo, hi):
        si hi − lo = 1 alors
            retourner s[lo]
        sinon
            mid ← (lo + hi) ÷ 2
            si s[lo] > s[mid] alors
                retourner max-circ'(lo, mid)
            sinon si s[mid] > s[hi] alors
                retourner max-circ'(mid, hi)
            sinon
                // Impossible car les éléments sont
                   distincts
    si s[1] > s[n] alors
        retourner max-circ'(1, n)
    sinon
        retourner s[n]
```

b) ★ La sous-routine `max-circ'(`lo, hi`)` retourne $\max(s[lo : hi])$ car elle satisfait cette pré-condition:

- $1 \leq lo < hi \leq |s|$,
- $s[lo] > s[hi]$, et
- $s[lo : hi]$ est ordonnée circulairement.

En effet, le premier appel satisfait trivialement cette pré-condition et les appels subséquents sont choisis de telle sorte à satisfaire les deux premiers. De plus, toute sous-séquence d'une séquence ordonnée circulairement est elle-même ordonnée circulairement. Remarquons que le bloc **sinon** ne peut pas être atteint, même avec des doublons, car $s[lo] > s[hi]$ par hypothèse, alors que l'atteinte du **sinon** signifierait que $s[lo] \leq s[mid] \leq s[hi]$. □

c) ★★ Afin d'obtenir une contradiction, supposons qu'il existe un algorithme $\mathcal{A}$ qui identifie $\max(s)$ en *moins* de n requêtes sur *toute* séquence s de n éléments ordonnés circulairement. Nous procédons par argument adversarial où chaque fois que $\mathcal{A}$ désire accéder à un nouvel élément $s[i]$, on lui fournit un élément de notre choix qui n'enfreint pas le fait que s soit ordonnée circulairement. Chaque fois que $\mathcal{A}$ effectue une requête, on lui fournit la valeur 0. Ainsi, lorsque $\mathcal{A}$ termine, il annonce forcément que $\max(s) = 0$. Par hypothèse, il existe au moins une position i pour laquelle $\mathcal{A}$ n'a jamais consulté $s[i]$. En effet, $\mathcal{A}$ effectue au plus $n-1$ requêtes, alors que s contient n éléments. Nous posons $s[i] \coloneqq 1$ et $s[j] \coloneqq 0$ pour toutes les autres positions j non consultées (s'il y en a d'autres). Cette séquence s est ordonnée circulairement car elle est de la forme

$$s = [\underbrace{0, \ldots, 0}_{\substack{\text{un certain nombre} \\ \text{de fois } k \in [0\,:\,n-1]}}, 1, \overbrace{0, \ldots, 0}^{n-k-1 \text{ fois}}].$$

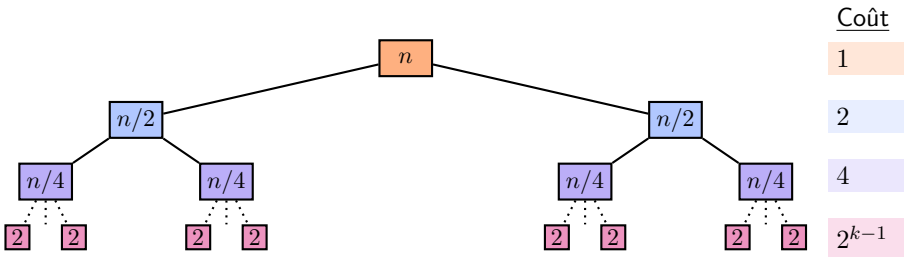
Ainsi, $\max(s) = 1$, alors que $\mathcal{A}$ a retourné 0, ce qui contredit le bon fonctionnement de $\mathcal{A}$. □

5.15) Approche diviser-pour-régner:

Entrées : séquence s de $n \in \mathbb{N}_{\geq 1}$ entiers
Sorties : somme maximale parmi toutes les sous-séquences contigües non vides

```
somme-max(s):
    si n = 0 alors
        retourner −∞
    sinon si n = 1 alors
        retourner s[1]
    sinon
        m ← n ÷ 2
        gauche ← s[1 : m]
        droite ← s[m + 1 : n]
        // Cumulatif maximal vers la droite
        cumul, c ← 0, −∞
        pour x ∈ droite
            cumul ← cumul + x
            c ← max(c, cumul)
        // Cumulatif maximal vers la gauche
        cumul, d ← 0, −∞
        pour x ∈ gauche en ordre inverse
            cumul ← cumul + x
            d ← max(d, cumul)
        // Somme maximale des deux côtés
        a ← somme-max(gauche)
        b ← somme-max(droite)
        // Somme maximale
        retourner max(a, b, c + d)
```

5.19) Considérons le cas où n est une puissance de deux. Chaque appel récursif effectue une multiplication, sauf lorsque $n \in \{0, 1\}$ où il n'y en a aucune. Ainsi, l'arbre de récursion (partiel) est comme suit, où $k := \log n$:



Posons $t(n)$ le nombre de multiplications sur entrée n. Nous avons:

$$\begin{aligned} t(n) &= \sum_{i=0}^{k-1} 2^i \\ &= 2^k - 1 \\ &\in \Theta(n). \end{aligned}$$

En général, pour une valeur $n \in \mathbb{N}$ arbitraire, nous avons:

$$t(n) = \begin{cases} 0 & \text{si } n \in \{0,1\}, \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + 1. & \text{sinon.} \end{cases}$$

Dans le jargon du théorème maître, nous avons $c = 1 + 1 = 2$, $b = 2$ et $d = 0$. Ainsi, nous obtenons $t \in \mathcal{O}(n^{\log_b c}) = \mathcal{O}(n^{\log_2 2}) = \mathcal{O}(n)$.

5.20) ★★ Nous avons $t(n) + t(n-2) = 0$ dont le polynôme caractéristique est $x^2 + 1 = (x - i)(x + i)$, où i est l'unité imaginaire, c-à-d. le nombre complexe tel que $i^2 = -1$. Par conséquent, $t(n) = c_1 \cdot i^n + c_2 \cdot (-i)^n$ pour certaines constantes c_1 et c_2. Afin d'identifier la valeur de ces constantes, nous devons résoudre ce système d'équations:

$$\begin{aligned} 1 &= c_1 \quad + c_2 \\ 0 &= c_1 \cdot i - c_2 \cdot i. \end{aligned}$$

Nous obtenons $c_1 = c_2 = 1/2$ et ainsi $t(n) = (i^n + (-i)^n)/2$.

5.21) ★★

a)

Entrées : $b \in S, n \in \mathbb{N}$
Résultat : b^n

`exp-rapide(`b, n`):`
  si $n = 0$ **alors**
    retourner e
  sinon
    $m \leftarrow$ `exp-rapide'(`$b, n \div 2$`)`
    $k \leftarrow e$
    si n *est impair* **alors** $k \leftarrow b$
    retourner $(m \oplus m) \oplus k$

b) Montrons que $b^{x+y} = b^x \oplus b^y$ pour tous $b \in S$ et $x, y \in \mathbb{N}$. Soit $b \in S$. Nous procédons par induction sur $x + y$. Si $x = 0$, alors nous avons

$$b^{x+y} = b^x = b^x \oplus e = b^x \oplus b^0 = b^x \oplus b^y.$$

Similairement, si $y = 0$, alors nous avons

$$b^{x+y} = b^y = e \oplus b^y = b^0 \oplus b^y = b^x \oplus b^y.$$

Considérons le cas général où $x > 0$ et $y > 0$. Nous avons:

$$\begin{aligned}
b^{x+y} &= b^{(x-1)+y} \oplus b && \text{(par } x > 0 \text{ et déf. d'expo.)} \\
&= (b^{x-1} \oplus b^y) \oplus b && \text{(par hypothèse d'induction)} \\
&= (b^{x-1} \oplus (b \oplus b^{y-1})) \oplus b && \text{(par } y > 0 \text{ et hyp. d'induction)} \\
&= ((b^{x-1} \oplus b) \oplus b^{y-1}) \oplus b && \text{(par associativité)} \\
&= (b^x \oplus b^{y-1}) \oplus b && \text{(par déf. d'exponentiation)} \\
&= b^x \oplus (b^{y-1} \oplus b) && \text{(par associativité)} \\
&= b^x \oplus b^y && \text{(par déf. d'exponentiation).} \quad \square
\end{aligned}$$

Montrons maintenant que l'algorithme est correct par induction sur n. Autrement dit, posons $r(b, n) \coloneqq$ `exp-rapide'`(b, n) et montrons que $r(b, n) = b^n$. Lorsque $n = 0$, l'algorithme retourne $b^0 = e$ comme attendu. Soit $n > 0$. L'algorithme retourne:

$$\begin{aligned}
r(b, n) &= (r(b, n \div 2) \oplus r(b, n \div 2)) \oplus b^{n \bmod 2} && \text{(par déf. de l'algo.)} \\
&= (b^{n \div 2} \oplus b^{n \div 2}) \oplus b^{n \bmod 2} && \text{(par hyp. d'induction)} \\
&= b^{2(n \div 2)} \oplus b^{n \bmod 2} && \text{(par la proposition)} \\
&= b^{2(n \div 2) + (n \bmod 2)} && \text{(par la proposition)} \\
&= b^n. && \square
\end{aligned}$$

5.23) a) *Solution non disponible.*

b) ★ *Solution non disponible.*

c)

►

Entrées : séquence s de n éléments partiellement ordonnés
Sorties : éléments maximaux de s

```
maximaux(s):
    // Identifier éléments dominés
    r ← [i ↦ faux : i ∈ [1..n]]
    pour i ∈ [1..n]
        pour j ∈ [1..n]
            si (i ≠ j) ∧ (s[i] ⪯ s[j]) alors
                r[i] ← vrai
    // Retirer éléments dominés
    s' ← [ ]
    pour i ∈ [1..n]
        si ¬r[i] alors
            ajouter s[i] à s'
    retourner s'
```

► ★★ Afin d'obtenir une contradiction, supposons qu'il existe un algorithme qui résout le problème avec moins de $n(n-1)/2$ comparaisons. Considérons l'entrée $s \coloneqq [\{1\}, \{2\}, \ldots, \{n\}]$. Comme il y a n éléments, il y a $(n-1)+\ldots+1+0 = n(n-1)/2$ comparaisons possibles. Par hypothèse, il existe des indices $i \neq j$ tels que $s[i]$ et $s[j]$ ne sont pas comparés. Comme tous les éléments de s sont maximaux, l'algorithme ne retire rien. Considérons la séquence s' identique à s sauf pour $s'[j] \coloneqq \{i, j\}$. Tous les éléments de s' sont maximaux, sauf le $i^{\text{ème}}$ car $s'[i] = \{i\} \subseteq \{i, j\} = s'[j]$. Comme l'algorithme ne compare jamais ces deux éléments, il retourne la même chose que sur entrée s, ce qui est contradictoire. □

Chapitre 6

6.2) Cet algorithme essaie toutes les sous-chaînes contiguës en temps $\mathcal{O}(mn \cdot \min(m,n))$ où $m = |u|$ et $n = |v|$:

Entrées : chaînes u et v
Résultat : plus longue sous-chaîne contiguë commune à u et v

```
sous-chaine-contiguë(u, v):
    préfixe(i, j):
        p ← []
        tant que i ≤ |u| ∧ j ≤ |v| ∧ u[i] = v[j]
            ajouter u[i] à p
            i ← i + 1
            j ← j + 1
        retourner p
    s ← []
    pour i ∈ [1..|u|]
        pour j ∈ [1..|v|]
            s' ← préfixe(i, j)
            si |s'| > |s| alors s ← s'
    retourner s
```

6.3) Cet algorithme essaie toutes les sous-chaînes en temps $\Omega(2^{|u|+|v|})$:

Entrées : chaînes u et v
Résultat : plus longue sous-chaîne commune à u et v

```
sous-chaine(u, v):
    aux(x, y, i, j):
        si i ≤ |u| alors
            a ← aux(x + u[i], y, i + 1, j)
            b ← aux(x,        y, i + 1, j)
            si |a| ≥ |b| alors retourner a
            sinon retourner b
        sinon si j ≤ |v| alors
            a ← aux(x, y + v[j], i, j + 1)
            b ← aux(x, y,        i, j + 1)
            si |a| ≥ |b| alors retourner a
            sinon retourner b
        sinon
            si x = y alors retourner x
            sinon retourner []
    retourner aux([], [], 1, 1)
```

6.7) À chaque tour de boucle, m décroît au moins d'une unité ou d augmente au moins d'une unité. En effet, m est ou bien découpé en $d \geq 2$, ou bien d est incrémenté. Ainsi, la distance entre m et d, c.-à-d. $m - d$, rétrécit à chaque tour de boucle. À l'entrée du corps de la boucle on a toujours:

$$m \geq d, \qquad (*)$$

Ainsi, la valeur de $m - d$ ne peut pas chuter sous 0 sans que la boucle termine. En effet, autrement on obtiendrait $m - d < 0 \equiv m < d$, ce qui contredit (*). Cela montre que le temps appartient à $\mathcal{O}(m)$. Lorsque m est premier, le seul diviseur possible est $d = m$, il y a donc $m - 1$ tours de boucle. L'algorithme fonctionne donc en temps $\Theta(m)$ dans le pire cas. Néanmoins, cette expression est de la forme $\Theta(2^{\log m})$, ce qui est exponentiel par rapport à la représentation binaire de m.

★ L'invariant (*) découle du fait qu'on retire les facteurs de m en ordre croissant. Voici un argument plus formel. Nous montrons qu'au début du corps de la boucle, aucun nombre de $[2..d-1]$ ne divise m. Cette propriété implique $m \geq d$ car $m > 1$ et m se divise lui-même.

Lorsque $d = 2$, la propriété est trivialement vraie car l'intervalle est vide. Considérons maintenant une itération de la boucle (étape d'induction):

- **Si**: Le nombre d divise m et la nouvelle valeur est $m' \coloneqq m/d$. Aucun nombre de $[2..d-1]$ ne divise m', car ils ne divisent pas $m = d \cdot m'$.
- **Sinon**: Le nombre d ne divise pas m et la nouvelle valeur est $d' \coloneqq d+1$. Aucun nombre de $[2..d-1] + [d] = [2..d'-1]$ ne divise m. □

Chapitre 7

7.1) On mémorise seulement la dernière ligne de T:

Entrées : montant $m \in \mathbb{N}$, séquence s de $n \in \mathbb{N}$ pièces
Résultat : nombre minimal de pièces afin de rendre m

```
monnaie-dyn(m, s):
    initialiser séquence T[0..m] avec ∞
    T[0] ← 0
    pour i ∈ [1..n]
        initialiser séquence U[0..m] avec ∞
        pour j ∈ [0..m]
            sans ← T[j]                          // sol. sans s[i]
            avec ← ∞
            si j ≥ s[i] alors avec ← U[j − s[i]] + 1   // avec s[i]
            U[j] ← min(sans, avec)
        T ← U
    retourner T[m]
```

7.3)

Entrées : chaînes u et v
Résultat : distance entre u et v

```
dist(u, v):
    initialiser T[0..|u|, 0..|v|] avec 0
    pour i ∈ [1..|u|]
        T[i, 0] ← i
    pour j ∈ [1..|v|]
        T[0, j] ← j
    pour i ∈ [1..|u|]
        pour j ∈ [1..|v|]
            a ← T[i − 1, j]                  // Insérer u[i] dans v
            b ← T[i, j − 1]                  // Supprimer u[i] de v
            si u[i] = v[j] alors
                c ← 0                        // Préserver v[j]
            sinon
                c ← 1                        // Modifier v[j] en u[i]
            T[i, j] ← min(a + 1, b + 1, T[i − 1, j − 1] + c)
    retourner T[|u|, |v|]
```

Par exemple, nous obtenons ce tableau T pour $u =$ `allo!` et $v =$ `alllo1`:

	ε	a	l	l	l	o	1
ε	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
l	2	1	0	1	2	3	4
l	3	2	1	0	1	2	3
o	4	3	2	1	1	1	2
!	5	4	3	2	2	2	2

7.4) On calcule la taille d'un plus long suffixe commun de $u[1:i]$ et $v[1:j]$ en temps $\mathcal{O}(|u| \cdot |v|)$, puis on retourne la plus grande taille identifiée:

Entrées : chaînes u et v
Résultat : plus longue sous-chaîne commune à u et v

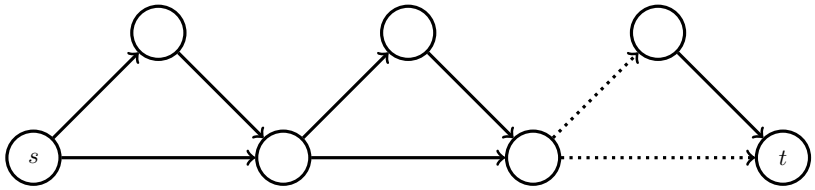
```
sous-chaine-contiguë-dyn(u, v):
    initialiser T[0..|u|, 0..|v|] avec 0
    i', j' ← 0, 0
    pour i ∈ [1..|u|]
        pour j ∈ [1..|v|]
            si u[i] = v[j] alors
                T[i, j] ← T[i − 1, j − 1] + 1
            sinon
                T[i, j] ← 0
            // Nouvelle sol. meilleure que l'ancienne?
            si T[i, j] > T[i', j'] alors
                i', j' ← i, j
    retourner T[i', j']
```

7.5) On calcule la taille d'une plus longue sous-chaîne commune de $u[1:i]$ et $v[1:j]$ en temps $\mathcal{O}(|u| \cdot |v|)$:

Entrées : chaînes u et v
Résultat : plus longue sous-chaîne commune à u et v

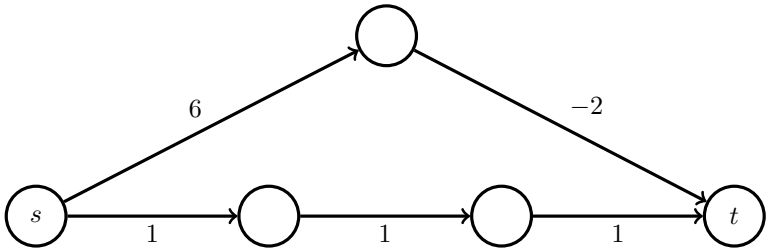
```
sous-chaine-dyn(u, v):
    initialiser T[0..|u|, 0..|v|] avec 0
    pour i ∈ [1..|u|]
        pour j ∈ [1..|v|]
            si u[i] = v[j] alors
                T[i, j] ← T[i − 1, j − 1] + 1
            sinon
                T[i, j] ← max(T[i − 1, j], T[i, j − 1])
    retourner T[m, n]
```

7.6)



7.9) On remplace l'initialisation $d[v,v] \leftarrow$ `vrai` par $d[v,v] \leftarrow$ `faux` pour ne pas tenir compte des chemins vides.

7.13) Non, contre-exemple:



7.14) Soient s et t les sommets qui correspondent aux deux villes. On cherche à identifier

$$\min\{\max(d[v], d'[v]) : v \in V\},$$

où $d[v]$ et $d'[v]$ dénotent respectivement la distance minimale de s vers v, et t vers v. Ainsi, on calcule d et d' avec l'algorithme de Dijkstra à partir de s et t respectivement. Ensuite, on identifie itérativement le sommet v qui minimise $\max(d[v], d'[v])$. L'algorithme fonctionne en temps $\mathcal{O}(|V| \log |V| + |E| + |V|) = \mathcal{O}(|V| \log |V| + |E|)$.

7.17) Si s et t ne sont pas dans la même composante connexe, alors il n'existe simplement pas de chemin de s vers t. S'ils font partie de la même composante connexe C et qu'elle possède une arête $e = \{u, v\}$ telle que $p[e] < 0$, alors nous avons $s \xrightarrow{*} u \xrightarrow{e} v \xrightarrow{e} u \xrightarrow{e} \cdots \xrightarrow{e} u \xrightarrow{*} t$. En effet, s peut atteindre u qui peut atteindre t, et, comme il n'y a pas de direction, on peut traverser e autant de fois que désiré. Ainsi, il n'y a pas de longueur minimale de s vers t. Si C ne contient pas de poids négatif, alors la distance de s vers t est bornée inférieurement par 0, donc il existe forcément un plus court chemin (simple).

7.18) Si $c \geq 0$, alors on lance un parcours en largeur sur $\mathcal{G}$. La distance entre s et un sommet v correspond au résultat obtenu multiplié par c. Si $c < 0$, nous calculons d'abord le graphe des composantes fortement connexes de $\mathcal{G}$. Si

une composante est non triviale, alors tous ses sommets ont une distance de $-\infty$. Nous pouvons ensuite retirer ces composantes, ce qui mène à un graphe dirigé acyclique. Il suffit ensuite d'ordonner les sommets en ordre topologique et de procéder par programmation dynamique.

7.19) Car en général cela va créer des cycles négatifs, auquel cas il n'y a pas de plus court chemin. Notons que si $\mathcal{G}$ est dirigé et acyclique, alors cette approche fonctionnera à coup sûr car tous les chemins sont simples.

7.20) ★ Nous procédons ainsi:

a) Enraciner l'arbre à un sommet arbitraire r;

b) Calculer le nombre de sommets n_v du sous-arbre enraciné en v, pour chaque sommet v;

c) Calculer la mesure de centralité m_r de la racine;

d) Calculer la mesure de centralité de m_v pour tout autre sommet v.

e) Retourner $[v \mapsto m_v : v \in V]$.

Nous expliquons comment réaliser chaque étape en temps linéaire:

a) On choisit une racine r quelconque, par ex. le premier sommet, puis on rend les arêtes dirigées en creusant récursivement à partir de r (et en faisant attention à ne pas revenir sur un sommet déjà exploré);

b) La taille du sous-arbre enraciné en v est la somme des tailles de ses enfants, c.-à-d. $n_v = \sum_{w:v\to w} n_w$. On calcule donc les tailles récursivement à partir de r en exploitant cette identité.

c) La mesure de centralité m'_v d'un sommet v dans le sous-arbre enraciné en v est

$$m'_v = \sum_{w:v\to w} (m'_w + p[v,w] \cdot n_w),$$

puisque chaque descendant d'un enfant w doit passer par l'arête de poids $p[v,w]$ pour atteindre v. En particulier, $m_r = m'_r$ car le sous-arbre enraciné en r est l'arbre entier. On calcule donc m'_r récursivement à l'aide de cette identité.

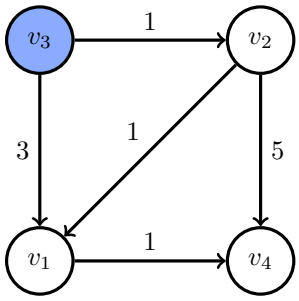
d) Soit u un sommet et v un enfant de u. Nous avons

$$m_v = m_u + p[u,v] \cdot \underbrace{(|V| - n_v)}_{\text{nombre de descendants de } u \text{ mais pas de } v} - p[u,v] \cdot \overbrace{n_v}^{\text{nombre de descendants de } v}.$$

En effet, la mesure de u et v diffère selon l'arête (u,v). Les descendants de v sont à $p[u,v]$ unités plus près de v, alors que les autres sommets sont à $p[u,v]$ unités plus loin de v. On peut ainsi calculer m_v en temps constant à partir de l'information de son parent u. On applique donc cette identité récursivement à partir de la racine.

e) Il n'y a rien à faire ici, on retourne les valeurs calculées.

7.21) L'algorithme conclut que la distance de v_3 vers v_4 est 4 plutôt que 3:



7.22) a) On identifie les distances en une itération de la boucle principale (meilleur cas possible).

b) On identifie les distances en $n - 1$ itérations de la boucle principale (pire cas possible).

c) On identifie les distances en $n \div 2$ itérations de la boucle principale.

Observation.

L'ordre dans lequel les arêtes sont stockées en mémoire influence donc la vitesse à laquelle l'algorithme de Bellman-Ford identifie les distances.

7.23)

Entrées : séquence s de $n \in \mathbb{N}_{\geq 1}$ entiers
Résultat : plus grande somme contigüe de s

initialiser séquence $T[0..n]$ avec $-\infty$

pour $i \in [1..n]$
  $T[i] \leftarrow \max(T[i-1] + s[i], s[i])$

$m \leftarrow -\infty$

pour $i \in [1..n]$
  $m \leftarrow \max(m, T[i])$

retourner m

On peut améliorer l'algorithme en utilisant moins de mémoire:

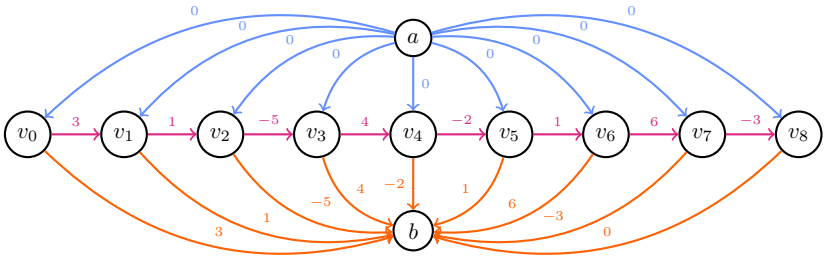
Entrées : séquence s de $n \in \mathbb{N}_{\geq 1}$ entiers
Résultat : plus grande somme contigüe de s

$t \leftarrow -\infty$
$m \leftarrow -\infty$
pour $i \in [1..n]$
  $t \leftarrow \max(t + s[i], s[i])$
  $m \leftarrow \max(m, t)$
retourner m

Remarque.

Cette version optimisée est l'algorithme de Kadane.

7.24) Nous expliquons l'approche à l'aide d'un exemple. Sur entrée $s = [3, 1, -5, 4, -2, 1, 6, -3]$, nous construisons ce graphe $\mathcal{G}_s$:



La distance maximale de a vers b correspond précisément à une somme maximale de s. Comme le graphe est acyclique, nous pouvons renverser les poids (par ex. 3 devient -3, et -5 devient 5), puis identifier la distance minimale. Comme il y a possiblement des poids négatifs, nous utilisons l'algorithme de Bellman-Ford. Celui-ci se termine en une itération de la boucle principale si les arêtes sont considérées selon cet ordre des sommets: $[a, v_0, \ldots, v_n, b]$.

Chapitre 8

8.1) Considérons la dernière itération de l'algorithme. Soit A l'événement « $y_2 = y_1 = y_0$ ». Nous avons: ?

$$\begin{aligned}
\Pr[A] &= \Pr[y_2 = y_1 = y_0 = 0 \vee y_2 = y_1 = y_0 = 1] \\
&= \Pr[y_2 = y_1 = y_0 = 0] + \Pr[y_2 = y_1 = y_0 = 1] && \text{(évén. disjoints)} \\
&= \prod_{i=0}^{2} \Pr[y_i = 0] + \prod_{i=0}^{2} \Pr[y_i = 1] && \text{(par indép.)} \\
&= \prod_{i=0}^{2} (1/2) + \prod_{i=0}^{2} (1/2) \\
&= 1/8 + 1/8 \\
&= 1/4.
\end{aligned}$$

Ainsi,

$$\Pr[\overline{A}] = 1 - \Pr[A] = 1 - (1/4) = 3/4.$$

Soit X la variable aléatoire qui dénote la valeur retournée et soit $k \in [1, 6]$. Nous avons:

$$\begin{aligned}
\Pr[X = k] &= \Pr[y_2 y_1 y_0 = \mathrm{bin}(k) \mid \overline{A}] \\
&= \Pr[y_2 y_1 y_0 = \mathrm{bin}(k) \wedge \overline{A}] / \Pr[\overline{A}] && \text{(prob. cond.)} \\
&= \Pr[y_2 y_1 y_0 = \mathrm{bin}(k)] \;/\; \Pr[\overline{A}] \\
&= (1/8) \;/\; (3/4) \\
&= 4/(3 \cdot 8) \\
&= 1/6.
\end{aligned}$$

Alternativement et plus succinctement, la probabilité de choisir une valeur $k \in [1, 6]$ à l'itération i est de $(1/4)^i \cdot (1/8)$. La probabilité de choisir k à une itération quelconque est donc de:

$$\begin{aligned}
\sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i \cdot \frac{1}{8} &= \frac{1}{8} \cdot \sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i \\
&= \frac{1}{8} \cdot \frac{1}{1 - 1/4} && \text{(série géométrique)} \\
&= \frac{1}{8} \cdot \frac{4}{3} \\
&= \frac{1}{6}.
\end{aligned}$$

8.2) On choisit le nombre en adaptant la recherche dichotomique:

Entrées : $k \in \mathbb{N}_{\geq 1}$
Résultat : nombre $x \in [0, 2^k - 1]$ choisi de façon aléatoire et uniforme

`uniforme-puissance`(i, j)**:**
  si $i = j$ **alors**
    retourner i
  sinon
    choisir un bit b à pile ou face
    $m \leftarrow (i + j) \div 2$
    si $b = 0$ **alors**
      retourner `uniforme-puissance`(i, m)
    sinon
      retourner `uniforme-puissance`$(m + 1, j)$
retourner `uniforme-puissance`$(0, 2^k - 1)$

8.3) Avec $a \coloneqq 1$ et $b \coloneqq 6$, la probabilité de générer 2, par exemple, est de $1/4$ plutôt que $1/6$. On peut s'en convaincre en dessinant l'arbre de récursion.

8.4) On lance deux pièces jusqu'à ce qu'elles donnent un résultat différent:

Entrées : —
Résultat : pile ou face

répéter
  choisir un bit x à pile ou face avec la pièce biaisée
  choisir un bit y à pile ou face avec la pièce biaisée
jusqu'à $x \neq y$
si $x = 0$ **alors** **retourner** *pile*
sinon **retourner** *face*

Soient X et Y les variables aléatoires qui dénotent respectivement les valeurs de x et y à la dernière itération. Nous simulons bien une pièce non biaisée puisque:

$$
\begin{aligned}
&\Pr[\text{retourner pile}] \\
&= \Pr[X = 0 \mid X \neq Y] \\
&= \Pr[X = 0 \wedge X \neq Y] \;/\; \Pr[X \neq Y] \\
&= \Pr[X = 0 \wedge Y = 1] \;/\; \Pr[X \neq Y] \\
&= (\Pr[X = 0] \cdot \Pr[Y = 1]) \;/\; \Pr[X \neq Y] && \text{(par indép.)} \\
&= pq / \Pr[X \neq Y] \\
&= pq / \Pr[(X = 0 \wedge Y = 1) \vee (X = 1 \wedge Y = 0)] \\
&= pq / (\Pr[X = 0 \wedge Y = 1] + \Pr[X = 1 \wedge Y = 0]) && \text{(évén. disjoints)} \\
&= pq / (\Pr[X = 0] \cdot \Pr[Y = 1] + \Pr[X = 1] \cdot \Pr[Y = 0]) && \text{(par indép.)} \\
&= pq / [pq + qp] \\
&= pq / 2pq \\
&= 1/2.
\end{aligned}
$$

Le nombre d'itérations espéré est de $1/(2pq) = 1/2(p - p^2)$ puisqu'il s'agit d'une loi géométrique de paramètre $2pq$.

Alternativement et plus succinctement, la probabilité de terminer à l'itération i avec $x = 0$ est de $(p^2 + q^2)^i \cdot pq$. La probabiltié d'obtenir $x = 0$ à une itération quelconque est donc de:

$$
\begin{aligned}
\sum_{i=0}^{\infty} \left(p^2 + q^2\right)^i \cdot pq &= pq \cdot \sum_{i=0}^{\infty} \left(p^2 + q^2\right)^i \\
&= pq \cdot \frac{1}{1 - (p^2 + q^2)} && \text{(série géométrique)} \\
&= p(1-p) \cdot \frac{1}{1 - p^2 - (1-p)^2} && \text{(car } q = 1 - p\text{)} \\
&= p(1-p) \cdot \frac{1}{1 - p^2 - 1 + 2p - p^2} \\
&= p(1-p) \cdot \frac{1}{2p(1-p)} \\
&= \frac{1}{2}.
\end{aligned}
$$

Alternativement, il est possible d'analyser l'algorithme à l'aide d'un graphe de probabilités.

8.5) Il s'agit d'un algorithme de Monte Carlo puisque l'algorithme fonctionne toujours en temps $\mathcal{O}(n^2)$, mais n'est pas toujours correct. En effet, lorsque $\mathbf{A}\cdot\mathbf{B} \neq \mathbf{C}$, il est possible que $\mathbf{A}\cdot(\mathbf{B}\cdot\boldsymbol{v}) = \mathbf{C}\cdot\boldsymbol{v}$, par ex. avec $\boldsymbol{v} = \mathbf{0}$. Analysons donc la probabilité p que $\mathbf{A}\cdot(\mathbf{B}\cdot\boldsymbol{v}) = \mathbf{C}\cdot\boldsymbol{v}$ lorsque $\mathbf{A}\cdot\mathbf{B} \neq \mathbf{C}$.

Posons $\mathbf{D} \coloneqq \mathbf{A}\cdot\mathbf{B} - \mathbf{C}$ et $\boldsymbol{x} \coloneqq \mathbf{D}\cdot\boldsymbol{v}$. Remarquons que p correspond à la probabilité que $\boldsymbol{x} = \mathbf{0}$. Puisque $\mathbf{A}\cdot\mathbf{B} \neq \mathbf{C}$, nous avons $\mathbf{D} \neq \mathbf{0}$. Ainsi, il existe $i, j \in [1..n]$ tels que $\mathbf{D}[i,j] \neq 0$. Nous avons

$$\boldsymbol{x}(i) = \sum_{k=1}^{n} \mathbf{D}[i,k]\cdot\boldsymbol{v}(k) = \mathbf{D}[i,j]\cdot\boldsymbol{v}(j) + \sum_{\substack{k=1\\k\neq j}}^{n} \mathbf{D}[i,k]\cdot\boldsymbol{v}(k).$$

Posons $a \coloneqq \mathbf{D}[i,j]$ et $b \coloneqq \sum_{\substack{k=1\\k\neq j}}^{n} \mathbf{D}[i,k]\cdot\boldsymbol{v}(k)$. Nous avons

$$\begin{aligned}
\Pr[\boldsymbol{x} = \mathbf{0}] &\leq \Pr[\boldsymbol{x}(i) = 0] \\
&= \Pr[\boldsymbol{x}(i) = 0 \mid b = 0]\cdot\Pr[b = 0] + \\
&\quad \Pr[\boldsymbol{x}(i) = 0 \mid b \neq 0]\cdot\Pr[b \neq 0] \\
&= \Pr[\boldsymbol{v}(j) = 0]\cdot\Pr[b = 0] + \qquad (2) \\
&\quad \Pr[\boldsymbol{v}(j) = 1 \wedge b = -a]\cdot\Pr[b \neq 0] \\
&\leq \Pr[\boldsymbol{v}(j) = 0]\cdot\Pr[b = 0] + \\
&\quad \Pr[\boldsymbol{v}(j) = 1]\cdot\Pr[b \neq 0] \\
&= (1/2)\cdot\Pr[b = 0] + \\
&\quad (1/2)\cdot\Pr[b \neq 0] \\
&= (1/2)\cdot(\Pr[b = 0] + \Pr[b \neq 0]) \\
&= 1/2,
\end{aligned}$$

où (2) découle du fait que

$$\boldsymbol{x}(i) = 0 \iff [\boldsymbol{v}(j) = b = 0 \vee (\boldsymbol{v}(j) = 1 \wedge b = -a)].$$

8.6) Soit X la variable aléatoire qui dénote le nombre d'itérations effectuées par l'algorithme. Posons $p_i \coloneqq (n/2)/(n-i)$ et $q_i \coloneqq 1 - p_i$. Nous avons:

$$\Pr[X = i] = \underbrace{q_1\cdot q_2\cdots q_{i-1}}_{\text{échecs}}\cdot\underbrace{p_i}_{\text{succès}}.$$

De plus, l'algorithme effectue entre 1 et $n/2$ itérations. Ainsi,

$$\mathbb{E}[X] = \sum_{i=1}^{n/2}\left(i\cdot p_i\cdot\prod_{j=1}^{i-1} q_j\right).$$

★ Afin de borner cette expression, remarquons d'abord que

$$i \cdot p_i \leq i \iff \frac{i \cdot (n/2)}{n-i} \leq i \iff (n/2) \leq n - i \iff i \leq n/2. \tag{3}$$

De plus, pour tout $0 \leq i < n$, nous avons:

$$q_i = 1 - \frac{(n/2)}{n-i} \leq 1 - \frac{(n/2)}{n} = 1/2. \tag{4}$$

Ainsi, nous obtenons:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot \prod_{j=1}^{i-1} q_j \right) \\
&\leq \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot (1/2)^{i-1} \right) && \text{(par (4))} \\
&\leq \sum_{i=1}^{n/2} \left(i \cdot (1/2)^{i-1} \right) && \text{(par (3))} \\
&\leq \sum_{i=1}^{\infty} \left(i \cdot (1/2)^{i-1} \right) && \text{(car chaque terme est non négatif)} \\
&= \frac{1}{(1-1/2)^2} && \text{(série géométrique dérivée)} \\
&= 4. && \square
\end{aligned}$$

★★ Voici une preuve de *Gabriel McCarthy (A2019)* qui montre que $\mathbb{E}[X] < 2$, et par conséquent que le nombre espéré d'itérations n'excède pas celui de l'algorithme 52. Observons d'abord que:

$$\begin{aligned}
\prod_{j=1}^{i-1} q_j &= \frac{(n/2-1)(n/2-2)\cdots(n/2-i+1)}{(n-1)(n-2)\cdots(n-i+1)} \\
&= \frac{(n/2-1)!}{(n/2-i)!} \cdot \frac{(n-i)!}{(n-1)!} \\
&= \frac{(n-i)!}{(n/2)! \cdot (n/2-i)!} \Big/ \frac{(n-1)!}{(n/2)! \cdot (n/2-1)!} \\
&= \binom{n-i}{n/2} \Big/ \binom{n-1}{n/2}.
\end{aligned} \tag{5}$$

De plus, remarquons les propriétés suivantes du coefficient binomial:

$$\binom{m}{k} = \frac{m}{k} \cdot \binom{m-1}{k-1} \qquad \binom{m+1}{k+1} = \sum_{j=k}^{m} \binom{j}{k}. \tag{6}$$

Ainsi, nous avons:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot \prod_{j=1}^{i-1} q_j \right) \\
&= \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot \binom{n-i}{n/2} \Big/ \binom{n-1}{n/2} \right) && \text{(par (5))} \\
&= \sum_{i=1}^{n/2} \left(i \cdot \frac{n/2}{n-i} \cdot \binom{n-i}{n/2} \Big/ \binom{n-1}{n/2} \right) \\
&= \sum_{j=n/2}^{n-1} \left(n/2 \cdot \frac{n-j}{j} \cdot \binom{j}{n/2} \Big/ \binom{n-1}{n/2} \right) \\
&= \frac{n/2}{\binom{n-1}{n/2}} \cdot \sum_{j=n/2}^{n-1} \left(\frac{n-j}{j} \cdot \binom{j}{n/2} \right) \\
&= \frac{n/2}{\binom{n-1}{n/2}} \cdot \left[\sum_{j=n/2}^{n-1} \frac{n}{j} \cdot \binom{j}{n/2} - \sum_{j=n/2}^{n-1} \binom{j}{n/2} \right] \\
&= \frac{n/2}{\binom{n-1}{n/2}} \cdot \left[\frac{n}{n/2} \cdot \sum_{j=n/2}^{n-1} \binom{j-1}{n/2-1} - \sum_{j=n/2}^{n-1} \binom{j}{n/2} \right] && \text{(par (6))} \\
&= \frac{n/2}{\binom{n-1}{n/2}} \cdot \left[\frac{n}{n/2} \cdot \binom{n-1}{n/2} - \binom{n}{n/2+1} \right] && \text{(par (6))} \\
&= n - (n/2) \cdot \frac{\binom{n}{n/2+1}}{\binom{n-1}{n/2}} \\
&= n - (n/2) \cdot \frac{n}{n/2+1} \cdot \frac{\binom{n-1}{n/2}}{\binom{n-1}{n/2}} && \text{(par (6))} \\
&= n - (n/2) \cdot \frac{n}{n/2+1} \\
&= n - \frac{n^2}{n+2} \\
&= \frac{n^2+2n}{n+2} - \frac{n^2}{n+2} \\
&= \frac{2n}{n+2} \\
&= \frac{2(n+2)-4}{n+2} \\
&= 2 - \frac{4}{n+2} \\
&< 2. && \square
\end{aligned}$$

8.7) Analysons l'algorithme 52. Remarquons que sa sortie (s'il y en a une) est toujours correcte. Analysons son temps espéré. Si $s[1] = a$, alors la probabilité de succès d'une itération est $1/3$. Dans ce cas, le nombre espéré d'itérations est 3. Similairement, si $s[1] = b$, alors la probabilité de succès d'une itération est $2/3$ et le nombre espéré d'itérations est $3/2$. Une itération exécute au plus 8 opérations élémentaires. Ainsi, le temps espéré est d'au plus $\max(3, 3/2) \cdot 8 = 24 \in \mathcal{O}(1)$.

Analysons l'algorithme 53. Remarquons que son temps d'exécution appartient à $\mathcal{O}(1)$. Analysons sa probabilité d'erreur. Si $s[1] = \max(s)$, alors la probabilité de retourner la mauvaise sortie est de 0. Si $s[1] \neq \max(s)$ et $s[1] = a$, alors elle est de $(2/3)^{275}$. Si $s[1] \neq \max(s)$ et $s[1] = b$, alors elle est de $(1/3)^{275}$. La probabilité d'erreur est donc $\text{err}(n) = \max(0, (2/3)^{275}, (1/3)^{275}) = (2/3)^{275} \leq 1/2^{260}$.

Démonstrations

Proposition 7. *Nous avons $f_1 + \ldots + f_k \in \mathcal{O}(c_1 \cdot f_1 + \ldots + c_k \cdot f_k)$ pour toutes fonctions $f_1, \ldots, f_k \in \mathcal{F}$ et tous coefficients $c_1, \ldots, c_k \in \mathbb{R}_{>0}$.*

Démonstration. Soit m_i le seuil à partir duquel f_i est positive. Nous prouvons la proposition en prenant $c \coloneqq \max(1/c_1, 1/c_2, \ldots, 1/c_k)$ comme constante multiplicative et $n_0 \coloneqq \max(m_1, m_2, \ldots, m_k)$ comme seuil. Pour tout $n \geq n_0$:

$$\begin{aligned}
\sum_{i=1}^{n} f_i(n) &= \sum_{i=1}^{n} \frac{1}{c_i} \cdot (c_i \cdot f_i(n)) && \text{(puisque } c_i \neq 0\text{)} \\
&\leq \sum_{i=1}^{n} c \cdot (c_i \cdot f_i(n)) && \text{(par déf. de } c \text{ et car } c_i \cdot f_i(n) > 0\text{)} \\
&= c \cdot \sum_{i=1}^{n} c_i \cdot f_i(n). && \square
\end{aligned}$$

Proposition 8. *$f_1 + \ldots + f_k \in \mathcal{O}(\max(f_1, \ldots, f_k))$ pour toutes $f_1, \ldots, f_k \in \mathcal{F}$.*

Démonstration. Nous démontrons la proposition en prenant $c \coloneqq k$ comme constante multiplicative et $n_0 \coloneqq 0$ comme seuil. Pour tout $n \geq n_0$, nous avons:

$$\begin{aligned}
\sum_{i=1}^{k} f_i(n) &\leq k \cdot \max(f_1(n), f_2(n), \ldots, f_k(n)) \\
&= c \cdot \max(f_1(n), f_2(n), \ldots, f_k(n)). && \square
\end{aligned}$$

Proposition 9. *$f \in \mathcal{O}(n^d)$ pour tout polynôme $f \in \mathcal{F}$ de degré d.*

Démonstration. Nous procédons par induction sur d. Si $d = 0$, alors $f(n) = c$ pour un certain $c \in \mathbb{R}_{>0}$. Ainsi, $f \in \mathcal{O}(1)$, ce qui démontre le cas de base.

Supposons que $d > 0$ et que la proposition soit satisfaite pour tout degré inférieur à d. Remarquons que tout polynôme est ou bien éventuellement positif,

ou bien éventuellement négatif[1]. Il existe donc un coefficient $c \geq 0$ et un polynôme $g \in \mathcal{F}$ de degré $d' < d$ tel que $f(n) = c \cdot n^d + g(n)$ ou $f(n) = c \cdot n^d - g(n)$. Considérons ces deux cas.

Cas $f(n) = c \cdot n^d + g(n)$. Par hypothèse d'induction, nous avons $g \in \mathcal{O}(n^{d'})$. Par la proposition 7, nous avons donc $f \in \mathcal{O}(n^d + n^{d'})$. Puisque $\max(n^d, n^{d'}) = n^d$, nous avons $f \in \mathcal{O}(n^d)$ par la proposition 8 et la proposition 6.

Cas $f(n) = c \cdot n^d - g(n)$. Soit n_0 un seuil à partir duquel g est positive. Nous avons $f(n) \leq c \cdot n^d$ pour tout $n \geq n_0$. Ainsi, $f \in \mathcal{O}(n^d)$ en prenant c comme constante multiplicative et n_0 comme seuil. □

Proposition 30. *Soit $J = \{j_1, j_2, \ldots, j_\ell\}$ un ensemble de lignes élémentaires d'un algorithme $\mathcal{A}$. Soit f' la fonction où $f'(x)$ dénote la somme du nombre d'exécutions de la ligne j_1 et du nombre d'opérations élémentaires exécutées par les lignes n'appartenant pas à J, sur entrée x. Si sur toute entrée, chaque ligne de J est atteinte au plus le nombre de fois que j_1 est atteinte, alors $\Theta(t'_{max}) = \Theta(t_{max})$, où $t'_{max}(n) \coloneqq \max\{f'(x) : \text{entrée } x \text{ de taille } n\}$.*

Démonstration. Observons d'abord que $f'(x) \leq f(x)$ pour toute entrée x, puisque f' compte moins d'opérations que f. Nous avons donc immédiatement $t'_{\max} \in \mathcal{O}(t_{\max})$.

Ainsi, il suffit de montrer que $t_{\max} \in \mathcal{O}(t'_{\max})$. Soit x une entrée de $\mathcal{A}$. Soit m le nombre d'opérations élémentaires exécutées, sur entrée x, par les lignes de $\mathcal{A}$ qui n'appartiennent pas à J; soit c_j le nombre d'opérations élémentaires contenues sur la ligne $j \in J$; et soit k_j le nombre de fois où la ligne $j \in J$ est atteinte sur entrée x. Nous avons:

$$\begin{aligned}
f(x) &\leq m + \sum_{j \in J} c_j \cdot k_j \\
&\leq m + k_1 \cdot \sum_{j \in J} c_j && (\text{car } k_1 = \max\{k_1, \ldots, k_\ell\}) \\
&\leq m + k_1 \cdot \ell \cdot \max(c_1, \ldots, c_\ell) \\
&\leq \ell \cdot \max(c_1, \ldots, c_\ell) \cdot (m + k_1) \\
&\leq \ell \cdot \max(c_1, \ldots, c_\ell) \cdot (m + c_1 \cdot k_1) \\
&= \ell \cdot \max(c_1, \ldots, c_\ell) \cdot f'(x).
\end{aligned}$$

Nous obtenons donc $f(x) \leq c \cdot f'(x)$, où $c \coloneqq \ell \cdot \max(c_1, \ldots, c_\ell)$.

Observons que c est indépendant de la taille de l'entrée puisque ℓ est une constante et puisque chaque c_j est indépendant par hypothèse. Ainsi $t_{\max}(n) \leq c \cdot t'_{\max}(n)$ pour tout $n \in \mathbb{N}$, et par conséquent $t_{\max} \in \mathcal{O}(t'_{\max})$. □

Proposition 14. *Après l'exécution du corps de la première boucle de l'algorithme 8, l'invariant suivant est satisfait:*

1. Cela découle du fait que tout polynôme possède un nombre fini de zéros.

(a) $c \geq 0$*;*

(b) x *apparaît au plus* $\frac{i+c}{2}$ *fois dans* $T[1:i]$*;*

(c) y *apparaît au plus* $\frac{i-c}{2}$ *fois dans* $T[1:i]$*, pour tout* $y \neq x$*.*

Démonstration. Nous procédons par induction sur $i \geq 1$.

Cas de base ($i = 1$). Puisque $x = T[1]$ et $c = 1$, x apparaît au plus $\frac{i+c}{2} = 1$ fois dans $T[1:1]$, et tout $y \neq x$ apparaît au plus $\frac{i-c}{2} = 0$ fois dans $T[1:1]$.

Étape d'induction. Soit $i > 1$. Supposons que l'invariant soit satisfait après l'exécution pour $i-1$. Soient c' et x' la valeur des variables c et x au début du corps de la boucle (et ainsi à la fin du corps de l'itération précédente). Par hypothèse d'induction, nous avons:

(a') $c' \geq 0$;

(b') x' apparaît au plus $\frac{i-1+c'}{2}$ fois dans $T[1:i-1]$;

(c') y apparaît au plus $\frac{i-1-c'}{2}$ fois dans $T[1:i-1]$, pour tout $y \neq x'$.

Observons qu'*exactement une* des lignes 3, 4 et 5 est exécutée lors de l'exécution du corps de la boucle. Nous considérons ces trois cas séparément.

Ligne 3. Nous avons $c' = 0$, $c = 1$ et $x = T[i]$. Clairement, $c \geq 0$. Puisque $c' = 0$, (b') et (c') affirment que chaque valeur apparaît au plus $\frac{i-1}{2}$ fois dans $T[1:i-1]$. Puisque $T[i] = x$, la valeur x apparaît donc au plus $\frac{i-1}{2} + 1 = \frac{i+1}{2} = \frac{i+c}{2}$ fois dans $T[1:i]$. Soit $y \neq x$. Puisque $T[i] \neq y$, le nombre d'occurrences de y n'a pas changé. Ainsi, y apparaît au plus $\frac{i-1}{2} = \frac{i-c}{2}$ fois dans $T[1:i]$.

Ligne 4. Nous avons $c = c' + 1$ et $T[i] = x' = x$. Par (a'), nous avons $c > c' \geq 0$. Par (b'), x apparaît au plus $\frac{i-1+c'}{2} + 1 = \frac{i+c'+1}{2} = \frac{i+c}{2}$ fois dans $T[1:i]$. Soit $y \neq x$. Le nombre d'occurrences de y n'a pas changé. Par (c'), y apparaît donc au plus $\frac{i-1-c'}{2} = \frac{i-c}{2}$ fois dans $T[1:i]$.

Ligne 5. Nous avons $c = c' - 1$ et $T[i] \neq x' = x$. Par (a'), nous avons $c' \geq 0$. Puisque la ligne 5 est exécutée, $c' \neq 0$ et donc $c' > 0$. Ainsi, $c \geq 0$. Par (b') et puisque $T[i] \neq x$, la valeur x apparaît au plus $\frac{i-1+c'}{2} = \frac{i+c}{2}$ fois dans $T[1:i]$. Soit $y \neq x$. Le nombre d'occurrences de y a augmenté d'au plus 1. Ainsi, par (c'), y apparaît au plus $\frac{i-1-c'}{2} + 1 = \frac{i-c'+1}{2} = \frac{i-c}{2}$ fois dans $T[1:i]$. □

Fin de la preuve de la proposition 15. Posons

$$P \coloneqq \{(x, y) \in [1..n]^2 : |X \cap \{x, y\}| = 1\}.$$

Autrement dit, P contient les paires de positions où précisément une position appartient à X et l'autre à $\{i, j\}$. En combinant nos observations, nous concluons

que:

$$\begin{aligned}
|\text{inv}(s)| &= \sum_{\substack{(x,y)\in[1..n]^2\\ x<y}} f(x,y)\\
&= f(i,j) + \sum_{\substack{(x,y)\in X^2\\ x<y}} f(x,y) + \sum_{\substack{(x,y)\in P\\ x<y}} f(x,y)\\
&> f'(i,j) + \sum_{\substack{(x,y)\in X^2\\ x<y}} f(x,y) + \sum_{\substack{(x,y)\in P\\ x<y}} f(x,y) && \text{(par (2.1))}\\
&= f'(i,j) + \sum_{\substack{(x,y)\in X^2\\ x<y}} f'(x,y) + \sum_{\substack{(x,y)\in P\\ x<y}} f(x,y) && \text{(par (2.2))}\\
&\geq f'(i,j) + \sum_{\substack{(x,y)\in X^2\\ x<y}} f'(x,y) + \sum_{\substack{(x,y)\in P\\ x<y}} f'(x,y) && \text{(par (2.3)–(2.5))}\\
&= \sum_{\substack{(x,y)\in[1..n]^2\\ x<y}} f'(x,y)\\
&= |\text{inv}(s')|. \qquad \square
\end{aligned}$$

Proposition 17. *$t(2^k) \leq 2^k(ck+1)$ pour tout $k \in \mathbb{N}_{>0}$.*

Démonstration. Procédons par induction sur k. Si $k = 1$, alors $t(2^1) \leq 2 + 2c = 2(c+1)$ par définition de t. Soit $k \in \mathbb{N}_{>0}$. Supposons que l'énoncé soit vrai pour k. Nous devons montrer que $t(2^{k+1}) \leq 2^{k+1}(c(k+1)+1)$. Nous avons:

$$\begin{aligned}
t(2^{k+1}) &\leq 2 \cdot t(2^k) + c \cdot 2^{k+1} && \text{(par définition de } t\text{)}\\
&\leq 2 \cdot (2^k(ck+1)) + c \cdot 2^{k+1} && \text{(par hypothèse d'induction)}\\
&= 2^{k+1}(ck+1) + c \cdot 2^{k+1}\\
&= 2^{k+1}(ck+1+c)\\
&= 2^{k+1}(c(k+1)+1). \qquad \square
\end{aligned}$$

Proposition 20. *À chaque temps d'un parcours en profondeur d'un graphe:*

- *les sommets gris forment un chemin simple;*
- *les sommets colorés forment une forêt, avec une arboresence par source.*

Démonstration. Nous procédons par induction sur le temps t. Au temps 0, tous les sommets sont blancs, donc les sommets gris et noirs forment trivialement un chemin et une forêt vide.

Soit un temps $t > 0$. Supposons les propriétés vraies au temps $t-1$. Soient C et $\mathcal{F}$ respectivement le chemin simple et la forêt au temps $t-1$ (obtenus par hypothèse d'induction). En passant au temps t, il y a trois cas possibles: (1) un

sommet v est découvert via aucun sommet; (2) un sommet v est découvert via un sommet u; ou (3) le traitement d'un sommet v est finalisé.

Premier cas. Par définition, v est une source. Nous ajoutons donc à $\mathcal{F}$ une arborescence triviale enracinée en v. Comme v est une source, il est le seul sommet gris. Il suffit donc de prendre le chemin de longueur 0 qui débute en v.

Deuxième cas. Le sommet v est découvert via un certain $u \to v$. Remarquons que u a forcément été découvert. De plus, son traitement n'est pas complété puisque celui de v ne l'est pas. Par conséquent, u est gris. Ainsi, u appartient à C et $\mathcal{F}$. Nous pouvons étendre $\mathcal{F}$ en ajoutant v comme enfant de u. Si u est le dernier sommet de C, alors nous avons terminé puisqu'on peut aussi ajouter v à C. Il suffit donc de montrer que u est bien le dernier sommet de C. Afin d'obtenir une contradiction, supposons qu'il existe un sommet w de C tel que $\text{pred}[w] = u$. Cela signifie que w a été découvert via $u \to w$ avant le temps t. Comme v est découvert via u au temps t, le traitement de w a forcément été finalisé avant le temps t. Ainsi, $f[w] < d[v] = t$. Cela signifie que w est noir au temps $t-1$, ce qui contredit le fait qu'il appartient à C.

Troisième cas. Le sommet v était gris au temps $t-1$, et devient noir au temps t. Il appartient donc à C et $\mathcal{F}$. Nous devons le retirer de C. Si v est le dernier sommet de C, alors nous avons terminé comme nous pouvons simplement l'enlever. Afin d'obtenir une contradiction, supposons qu'il existe un sommet w de C tel que $\text{pred}[w] = v$. Cela signifie que $v \to w$ et que w est gris au temps $t-1$. Ainsi, $t-1 < f[w] < f[v] = t$, ce qui est une contradiction. □

Théorème 6. *Soient $t, f \in \mathcal{F}$, $b \in \mathbb{N}_{\geq 2}$, $c \in \mathbb{N}_{\geq 1}$ et $d \in \mathbb{R}_{>0}$ où $f \in \mathcal{O}(n^d)$ et*

$$t(n) = c \cdot t(n \div b) + f(n) \text{ pour tout } n \text{ suffisamment grand.}$$

La fonction t appartient à:

- $\mathcal{O}(n^d)$ *si* $c < b^d$,
- $\mathcal{O}(n^d \cdot \log n)$ *si* $c = b^d$,
- $\mathcal{O}(n^{\log_b c})$ *si* $c > b^d$.

Démonstration. Soit $n_0 \in \mathbb{N}$ le seuil tel que $t(n) = c \cdot t(n \div b) + f(n)$ pour tout $n \geq n_0$. Soient $c' \in \mathbb{R}_{>0}$ et $n_0' \in \mathbb{N}$ la constante multiplicative et le seuil qui montrent que $f \in \mathcal{O}(n^d)$. Posons $n_0'' \coloneqq \max(n_0, n_0', b)$ et $c'' \coloneqq \max\{t(n) : n < n_0''\}$. Nous montrons d'abord que

$$t(n) \leq \begin{cases} c'' & \text{si } n < n_0'', \\ c'' \cdot c^{\log_b n} + c' \cdot n^d \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor} \left(\frac{c}{b^d}\right)^i & \text{si } n \geq n_0''. \end{cases} \tag{7}$$

Soit $n \in \mathbb{N}$. Si $n < n_0''$ alors $t(n) \leq c''$ découle directement de la définition de c''. Si $n \geq n_0''$ et $n \div b < n_0''$, alors

$$\begin{aligned}
t(n) &= c \cdot t(n \div b) + f(n) \\
&\leq c \cdot c'' + c' \cdot n^d \\
&= c'' \cdot c + c' \cdot n^d \\
&\leq c'' \cdot c^{\log_b n} + c' \cdot n^d \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor} \left(\frac{c}{b^d}\right)^i \quad \text{(car } \log_b n \geq 1 \text{ et } \textstyle\sum_{i=0}^{\lfloor \log_b n \rfloor} \left(\frac{c}{b^d}\right)^i \geq 1\text{)}.
\end{aligned}$$

Si $n \geq n_0''$ et $n \div b \geq n_0''$, alors par induction on obtient

$$\begin{aligned}
t(n) &= c \cdot t(n \div b) + f(n) \\
&\leq c \cdot t(n \div b) + c' \cdot n^d \\
&\leq c \cdot \left(c'' \cdot c^{\log_b (n \div b)} + c' \cdot (n \div b)^d \cdot \sum_{i=0}^{\lfloor \log_b (n \div b) \rfloor} \left(\frac{c}{b^d}\right)^i \right) + c' \cdot n^d \\
&\leq c \cdot \left(c'' \cdot c^{\log_b (n/b)} + c' \cdot (n/b)^d \cdot \sum_{i=0}^{\lfloor \log_b (n/b) \rfloor} \left(\frac{c}{b^d}\right)^i \right) + c' \cdot n^d \\
&= c'' \cdot c^{\log_b (n/b)+1} + c \cdot c' \cdot (n/b)^d \cdot \sum_{i=0}^{\lfloor \log_b (n/b) \rfloor} \left(\frac{c}{b^d}\right)^i + c' \cdot n^d \\
&= c'' \cdot c^{\log_b n - \log_b b + 1} + c \cdot c' \cdot (n/b)^d \cdot \sum_{i=0}^{\lfloor \log_b n - \log_b b \rfloor} \left(\frac{c}{b^d}\right)^i + c' \cdot n^d \\
&= c'' \cdot c^{\log_b n} + c \cdot c' \cdot (n/b)^d \cdot \sum_{i=0}^{\lfloor \log_b n - 1 \rfloor} \left(\frac{c}{b^d}\right)^i + c' \cdot n^d \\
&= c'' \cdot c^{\log_b n} + c \cdot c' \cdot (n/b)^d \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{c}{b^d}\right)^i + c' \cdot n^d \\
&= c'' \cdot c^{\log_b n} + c' \cdot n^d \cdot (c/b^d) \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{c}{b^d}\right)^i + c' \cdot n^d \\
&= c'' \cdot c^{\log_b n} + c' \cdot n^d \cdot \sum_{i=1}^{\lfloor \log_b n \rfloor} \left(\frac{c}{b^d}\right)^i + c' \cdot n^d \\
&= c'' \cdot c^{\log_b n} + c' \cdot n^d \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor} \left(\frac{c}{b^d}\right)^i .
\end{aligned}$$

Nous avons fini de démontrer (7). Remarquons que $c^{\log_b n} = n^{\log_b c}$. Ainsi,

par (7), nous avons

$$t \in \mathcal{O}\left(n^{\log_b c} + n^d \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor} \left(\frac{c}{b^d}\right)^i\right).$$

Afin de simplifier cette expression, comparons c et b^d et considérons les trois cas.

Cas 1: $c < b^d$. Nous avons $0 < c/b^d < 1$. Une série géométrique de raison $0 < r < 1$ converge à la constante $1/(1-r)$. Nous avons donc

$$\begin{aligned}
t &\in \mathcal{O}(n^{\log_b c} + n^d \cdot (1/(1-(c/b^d)))) \\
&= \mathcal{O}(n^{\log_b c} + n^d) \\
&= \mathcal{O}(n^{\log_b c} + n^{\log_b b^d}) \\
&= \mathcal{O}(n^{\log_b b^d}) && \text{(par } c < b^d \text{ et la règle du max.)} \\
&= \mathcal{O}(n^d).
\end{aligned}$$

Cas 2: $c = b^d$. Nous avons

$$\begin{aligned}
t &\in \mathcal{O}\left(n^{\log_b c} + n^d \cdot \sum_{i=0}^{\lfloor \log_b n \rfloor} 1^i\right) && \text{(car } c/b^d = 1) \\
&= \mathcal{O}(n^{\log_b c} + n^d \cdot (1 + \lfloor \log_b n \rfloor)) \\
&= \mathcal{O}(n^{\log_b c} + n^d + n^d \cdot \lfloor \log_b n \rfloor) \\
&= \mathcal{O}(n^{\log_b b^d} + n^d + n^d \cdot \lfloor \log_b n \rfloor) && \text{(par } c = b^d) \\
&= \mathcal{O}(n^d + n^d + n^d \cdot \lfloor \log_b n \rfloor) \\
&= \mathcal{O}(n^d \cdot \log_b n) && \text{(par la règle du max. et } \lfloor x \rfloor \leq x) \\
&= \mathcal{O}(n^d \cdot \log(n)/\log(b)) && \text{(changement de base)} \\
&= \mathcal{O}(n^d \cdot \log(n)).
\end{aligned}$$

Cas 3: $c > b^d$. Posons $k \coloneqq \lfloor \log_b n \rfloor$. Nous avons

$$\begin{aligned}
t &\in \mathcal{O}\left(n^{\log_b c} + n^d \cdot \sum_{i=0}^{k} \left(\frac{c}{b^d}\right)^i\right) \\
&= \mathcal{O}\left(n^{\log_b c} + n^d \cdot \frac{(c/b^d)^{k+1}}{(c/b^d) - 1}\right) && \text{(série géo. de raison } c/b^d\text{)} \\
&= \mathcal{O}(n^{\log_b c} + n^d \cdot (c/b^d)^k) \\
&= \mathcal{O}(n^{\log_b c} + n^d \cdot (c/b^d)^{\log_b n}) && \text{(par déf. de } k \text{ et } \lfloor x \rfloor \leq x\text{)} \\
&= \mathcal{O}(n^{\log_b c} + n^d \cdot n^{\log_b (c/b^d)}) \\
&= \mathcal{O}(n^{\log_b c} + n^d \cdot n^{\log_b (c) - d}) \\
&= \mathcal{O}(n^{\log_b c} + n^{\log_b c}) \\
&= \mathcal{O}(n^{\log_b c}).
\end{aligned}$$

□

Proposition 25. *La boucle principale de l'algorithme de Floyd-Warshall satisfait cet invariant:* $d[u, v] = \delta_{k-1}(u, v)$ *pour tous* $u, v \in V$.

Démonstration. Remarquons que $\delta_0(v, v) = 0$ et $\delta_0(u, v) = p[u, v]$ pour tous sommets $u \neq v$, car aucun sommet intermédiaire n'est permis. Le cas de base $k = 1$ est donc satisfait.

Soit $k \geq 1$. Écrivons d' afin de dénoter la matrice obtenue à partir de d après l'exécution du corps de la boucle principale. Remarquons que

$$d'[v_i, v_k] = d[v_i, v_k] \qquad \forall i \in [1..n], \tag{8}$$
$$d'[v_k, v_j] = d[v_k, v_j] \qquad \forall j \in [1..n]. \tag{9}$$

En effet, imaginons par ex. qu'une entrée $d[v_i, v_k]$ soit modifiée, alors cela signifierait que $d[v_i, v_k] > d[v_i, v_k] + d[v_k, v_k]$ et ainsi que $d[v_k, v_k] < 0$. Cela est impossible car $\mathcal{G}$ ne possède aucun cycle négatif.

Soient $i, j \in [1..n] \setminus \{k\}$ et soit C un plus court chemin de v_i vers v_j dont les sommets intermédiaires appartiennent à $\{v_1, v_2, \ldots, v_k\}$. Si C n'utilise pas le sommet v_k, alors $\delta_k(v_i, v_j) = \delta_{k-1}(v_i, v_j)$. Sinon, nous avons $\delta_k(v_i, v_j) = \delta_{k-1}(v_i, v_k) + \delta_{k-1}(v_k, v_j)$. Par conséquent:

$$\delta_k(v_i, v_j) = \min(\delta_{k-1}(v_i, v_j), \underbrace{\delta_{k-1}(v_i, v_k) + \delta_{k-1}(v_k, v_j)}_{\text{longueur du chemin qui passe par } v_k}). \tag{10}$$

Nous avons donc:

$$\begin{aligned}
& d'[v_i, v_j] \\
=\ & \min(d[v_i, v_j], d[v_i, v_k] + d[v_k, v_j]) && \text{(par déf., (8) et (9))} \\
=\ & \min(\delta_{k-1}(v_i, v_j), \delta_{k-1}(v_i, v_k) + \delta_{k-1}(v_k, v_j)) && \text{(par hyp. d'ind.)} \\
=\ & \delta_k(v_i, v_j) && \text{(par (10)).}
\end{aligned}$$

□

Proposition 28. $q^k \leq 2^{-k/|V|^2}$.

Démonstration. Nous avons:

$$\begin{aligned} q^k &\leq (1 - 1/|V|^2)^k \\ &\leq \left(2^{-1/|V|^2}\right)^k \qquad \text{(car } 1 - x \leq 2^{-x} \text{ pour tout } x \in \mathbb{R}_{\geq 0}\text{)} \\ &= 2^{-k/|V|^2}. \end{aligned}$$

□

Proposition 29. *Le temps moyen du tri par insertion appartient à* $\Theta(n^2)$.

Démonstration. Nous nous limitons au cas où l'entrée s est une permutation de n éléments distincts. Rappelons que par la proposition 16, le tri par insertion fonctionne en temps $\Theta(n + k)$ où k est le nombre d'inversions de s. Soit $f(n)$ le nombre total d'inversions parmi *toutes* les séquences constituées de n éléments distincts. Si le $i^{\text{ème}}$ plus grand élément apparaît au début d'une séquence, alors $(i - 1)$ inversions sont engendrées par cet élément. De plus, il existe $(n - 1)!$ séquences qui débutent par cet élément. Ainsi:

$$\begin{aligned} f(n) &= \sum_{i=1}^{n} [(n-1)! \cdot (i-1) + f(n-1)] \\ &= n \cdot f(n-1) + (n-1)! \cdot \sum_{i=1}^{n} (i-1) \\ &= n \cdot f(n-1) + (n-1)! \cdot n(n-1)/2 \\ &= n \cdot f(n-1) + n! \cdot (n-1)/2. \end{aligned}$$

En substitutant f à répétition, nous obtenons:

$$\begin{aligned} f(n) &= n \cdot f(n-1) + n! \cdot (n-1)/2 \\ &= n \cdot [(n-1) \cdot f(n-2) + (n-1)! \cdot (n-2)/2] + n! \cdot (n-1)/2 \\ &= n \cdot (n-1) \cdot f(n-2) + n!/2 \cdot [(n-2) + (n-1)] \\ &\vdots \\ &= n! \cdot f(0) + n!/2 \cdot \sum_{i=0}^{n-1} i \\ &= n!/2 \cdot \sum_{i=1}^{n-1} i \\ &= n!/2 \cdot n(n-1)/2 \\ &= n! \cdot n(n-1)/4. \end{aligned}$$

Puisqu'il y a $n!$ permutations, le nombre moyen d'inversions est égal à $f(n)/n! = n(n-1)/4$. Ainsi, le temps moyen du tri par insertion appartient à

$$\Theta(n + n(n-1)/4) = \Theta(n^2).$$

□

Récurrences linéaires

Au chapitre 5, nous avons vu une méthode afin d'identifier la forme close d'une récurrence linéaire. Cette annexe montre pourquoi elle fonctionne, à l'aide d'algèbre linéaire. Nous nous limitons aux nombres réels par souci de simplicité, bien que la théorie se généralise aux nombres complexes.

Cas homogène

Rappelons qu'une récurrence linéaire homogène est une fonction t de la forme:

$$t(n) \coloneqq \begin{cases} b_{d-n} & \text{si } n < d, \\ a_1 \cdot t(n-1) + \ldots + a_d \cdot t(n-d) & \text{si } n \geq d. \end{cases}$$

Nous pouvons représenter t par la matrice $\mathbf{A}_t$ et le vecteur $\boldsymbol{b}_t$ définis par

$$\mathbf{A}_t \coloneqq \begin{pmatrix} a_1 & a_2 & \cdots & a_{d-1} & a_d \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \text{ et } \boldsymbol{b}_t \coloneqq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{d-1} \\ b_d \end{pmatrix}.$$

Afin d'alléger la notation, nous allons souvent omettre l'indice « t ».

Exemple.

La suite de Fibonacci est représentée par:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

En multipliant par $\mathbf{A}$ à répétition, nous obtenons ces vecteurs:

$$\mathbf{A}^0\boldsymbol{b} = \begin{pmatrix}1\\0\end{pmatrix}, \mathbf{A}^1\boldsymbol{b} = \begin{pmatrix}1\\1\end{pmatrix}, \mathbf{A}^2\boldsymbol{b} = \begin{pmatrix}2\\1\end{pmatrix}, \mathbf{A}^3\boldsymbol{b} = \begin{pmatrix}3\\2\end{pmatrix}, \mathbf{A}^4\boldsymbol{b} = \begin{pmatrix}5\\3\end{pmatrix}, \ldots$$

En ne conservant que le deuxième élément de chaque vecteur, nous obtenons la suite de Fibonacci: $0, 1, 1, 2, 3, 5, \ldots$.

Comme l'exemple précédent le suggère, en général, le dernier élément de $\mathbf{A}^n\boldsymbol{b}$ correspond à $t(n)$. Cela découle de la définition de $\mathbf{A}$ et $\boldsymbol{b}$ et se démontre par induction sur n. La forme close de la récurrence peut être extraite de la matrice $\mathbf{A}$ à l'aide de ce théorème:

Théorème 8. *Si $\mathbf{A}$ possède d valeurs propres $\lambda_1, \ldots, \lambda_d \in \mathbb{R}$ distinctes, alors il existe des constantes $c_1, \ldots, c_d \in \mathbb{R}$ telles que $t(n) = c_1 \cdot \lambda_1^n + \cdots + c_d \cdot \lambda_d^n$.*

Avant de démontrer le théorème 8, nous prouvons ce lemme:

Lemme 1. *Soit $\mathbf{A}$ une matrice de dimension $d \times d$. Soient $\lambda_1, \ldots, \lambda_k$ les valeurs propres de $\mathbf{A}$, et $V = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ un ensemble de vecteurs propres associés à ces valeurs propres. Les affirmations suivantes sont vraies:*

(a) $\mathbf{A}^n\boldsymbol{v}_j = \lambda_j^n\boldsymbol{v}_j$ pour tous $j \in [1..k]$ et $n \in \mathbb{N}$,

(b) Si $|\{\lambda_1, \ldots, \lambda_k\}| = d$, alors V est une base de $\mathbb{R}^d$.

Démonstration. Rappelons qu'un *vecteur propre* $\boldsymbol{u}$ de $\mathbf{A}$ est un vecteur $\boldsymbol{u} \neq \mathbf{0}$ tel que $\mathbf{A}\boldsymbol{u} = \gamma\boldsymbol{u}$ pour une certaine constante γ, appelée *valeur propre*.

(a) Procédons par induction sur $n \in \mathbb{N}$. Il est clair que $\mathbf{A}^0\boldsymbol{v}_j = \lambda_j^0\boldsymbol{v}_j$. Supposons que l'hypothèse soit vraie pour un certain $n \in \mathbb{N}$. Nous avons:

$$\begin{aligned}
\mathbf{A}^{n+1}\boldsymbol{v}_j &= \mathbf{A} \cdot (\mathbf{A}^n\boldsymbol{v}_j) \\
&= \mathbf{A} \cdot (\lambda_j^n\boldsymbol{v}_j) && \text{(par hypothèse d'induction)} \\
&= \lambda_j^n \cdot (\mathbf{A}\boldsymbol{v}_j) \\
&= \lambda_j^n \cdot \lambda_j\boldsymbol{v}_j && \text{(car } \boldsymbol{v}_j \text{ est un vecteur propre associé à } \lambda_j\text{)} \\
&= \lambda_j^{n+1}\boldsymbol{v}_j.
\end{aligned}$$

(b) Comme $|\{\lambda_1, \ldots, \lambda_k\}| = d$, nous avons d valeurs propres distinctes et ainsi $|V| = d$. Il suffit donc de montrer que V est linéairement indépendant. Afin d'obtenir une contradiction, supposons que ce ne soit pas le cas. Nous pouvons réordonner les vecteurs de telle sorte que $\boldsymbol{v}_i$ soit un combinaison linéaire de $W \coloneqq \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i-1}\}$ et que l'ensemble W soit linéairement indépendant. Il existe donc des coefficients $\alpha_1, \ldots, \alpha_{i-1} \in \mathbb{R}$ et un indice $\ell \in [1..i-1]$ tels que $\alpha_\ell \neq 0$ et

$$\boldsymbol{v}_i = \sum_{j=1}^{i-1} \alpha_j\boldsymbol{v}_j. \tag{11}$$

En multipliant les deux côtés de (11) d'une part par $\mathbf{A}$, et d'autre part par λ_i, nous obtenons:

$$\lambda_i \boldsymbol{v}_i = \sum_{j=1}^{i-1} \alpha_j \lambda_j \boldsymbol{v}_j \;\text{ et }\; \lambda_i \boldsymbol{v}_i = \sum_{j=1}^{i-1} \alpha_j \lambda_i \boldsymbol{v}_j.$$

Ainsi, nous avons $\sum_{j=1}^{i-1} \alpha_j(\lambda_j - \lambda_i)\boldsymbol{v}_j = \mathbf{0}$. Rappelons que les valeurs propres sont distinctes. En particulier, cela signifie que $\alpha_\ell(\lambda_\ell - \lambda_i) \neq 0$ et ainsi que W est linéairement dépendant, ce qui est une contradiction. □

Nous sommes maintenant en mesure de prouver le théorème 8:

Démonstration du théorème 8. Supposons que $\mathbf{A}$ possède d valeurs propres réelles distinctes. Soit $\boldsymbol{v}_i$ un vecteur propre de $\mathbf{A}$ associé à λ_i, c.-à-d. un vecteur non nul tel que $\mathbf{A}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i$. Posons $V \coloneqq \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d\}$.

Par le lemme 1(b), l'ensemble V engendre $\mathbb{R}^d$. Par conséquent, $\boldsymbol{b}$ est une combinaison linéaire de V. Autrement dit, il existe des coefficients $\alpha_1, \ldots, \alpha_d \in \mathbb{R}$ tels que $\boldsymbol{b} = \alpha_1 \boldsymbol{v}_1 + \ldots + \alpha_d \boldsymbol{v}_d$. Nous avons donc:

$$\begin{aligned}
\mathbf{A}^n \boldsymbol{b} &= \mathbf{A}^n(\alpha_1 \boldsymbol{v}_1 + \ldots + \alpha_d \boldsymbol{v}_d) \\
&= \alpha_1 \mathbf{A}^n \boldsymbol{v}_1 + \ldots + \alpha_d \mathbf{A}^n \boldsymbol{v}_d \\
&= \alpha_1 \lambda_1^n \boldsymbol{v}_1 + \ldots + \alpha_d \lambda_d^n \boldsymbol{v}_d \qquad \text{(par le lemme 1(a))}.
\end{aligned}$$

Rappelons que $t(n) = (\mathbf{A}^n \boldsymbol{b})(d)$. Nous avons donc:

$$t(n) = \alpha_1 \lambda_1^n \boldsymbol{v}_1(d) + \ldots + \alpha_d \lambda_d^n \boldsymbol{v}_d(d).$$

En posant $c_i \coloneqq \alpha_i \boldsymbol{v}_i(d)$, nous obtenons $t(n) = c_1 \lambda_1^n + \ldots + c_d \lambda_d^n$. □

Exemple.

Considérons cette récurrence linéaire homogène qui émane du problème de pavage présenté à la section 5.2.1:

$$t(n) \coloneqq \begin{cases} 1 & \text{si } n \leq 1, \\ t(n-1) + 2 \cdot t(n-2) & \text{sinon.} \end{cases}$$

Nous avons

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix} \text{ et } \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Posons $\boldsymbol{v}_1 \coloneqq (1, 1/2)$ et $\boldsymbol{v}_2 \coloneqq (-1, 1)$. Ces deux vecteurs sont des vecteurs propres associés aux valeurs propres $\lambda_1 \coloneqq 2$ et $\lambda_2 = -1$, car $\mathbf{A}\boldsymbol{v}_1 = (2, 1)$ et $\mathbf{A}\boldsymbol{v}_2 = (1, -1)$. De plus, nous avons $\boldsymbol{b} = (4/3) \cdot \boldsymbol{v}_1 + (1/3) \cdot \boldsymbol{v}_2$. Nous obtenons donc:

$$t(n) = \frac{4}{3} \cdot \frac{1}{2} \cdot 2^n + \frac{1}{3} \cdot 1 \cdot (-1)^n = \frac{2}{3} \cdot 2^n + \frac{1}{3} \cdot (-1)^n.$$

Nous venons donc de retrouver la forme close obtenue à la section 5.2.1.

Rappelons que le *polynôme caractéristique* d'une matrice $\mathbf{D}$ est la fonction $p_{\mathbf{D}}(x) \coloneqq \det(x\mathbf{I}-\mathbf{D})$ et que ses racines réelles correspondent aux valeurs propres [2] de $\mathbf{D}$. Ainsi, les valeurs propres de la matrice $\mathbf{A}$ d'une récurrence linéaire homogène t peuvent être obtenues en identifiant son polynôme caractéristique, puis en trouvant ses racines. Cela permet ensuite de construire la forme close de t à l'aide du théorème 8. La forme du polynôme caractéristique correspond précisément à celle introduite à la section 5.2.1 comme en témoigne cette proposition:

Proposition 31. *Le polynôme caractéristique de* $\mathbf{A}$ *est* $x^d - a_1 \cdot x^{d-1} - \ldots - a_d \cdot x^0$.

Démonstration. Rappelons que $\mathbf{A}$ est de dimension $d \times d$. Nous procédons par induction sur d. Si $d = 1$, alors $\mathbf{A} = (a_1)$, et ainsi son polynôme caractéristique est $x - a_1$ comme attendu.

Supposons que $d > 1$. Soit $\mathbf{B}$ la matrice de dimension $(d-1) \times (d-1)$ obtenue en retranchant la première ligne et la première colonne de $x\mathbf{I} - \mathbf{A}$. Soit $\mathbf{C}$ la matrice de dimension $(d-1) \times (d-1)$ obtenue en retranchant la deuxième ligne et la première colonne de $x\mathbf{I} - \mathbf{A}$. Ces matrices sont de la forme:

$$\mathbf{B} = \begin{pmatrix} x & 0 & \cdots & 0 & 0 \\ -1 & x & \cdots & 0 & 0 \\ 0 & -1 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & x \end{pmatrix} \text{ et } \mathbf{C} = \begin{pmatrix} -a_2 & -a_3 & \cdots & -a_{d-1} & -a_d \\ -1 & x & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & x & 0 \\ 0 & 0 & \cdots & -1 & x \end{pmatrix}.$$

Soit $\mathbf{C}'$ la matrice d'une récurrence linéaire homogène dont les coefficients sont $x + a_2, a_3, a_4, \ldots, a_d$. Remarquons que $\mathbf{B}$ est une matrice diagonale. De plus, $\det(x\mathbf{I} - \mathbf{C}') = \det(\mathbf{C})$ car $x - (x + a_2) = -a_2$. Rappelons que le déterminant d'une matrice diagonale est le produit des éléments de sa diagonale principale. En calculant le déterminant de $x\mathbf{I} - \mathbf{A}$ via sa première colonne, nous obtenons:

$$\begin{aligned} p_{\mathbf{A}}(x) &= \det(x\mathbf{I} - \mathbf{A}) \\ &= (x - a_1) \det \mathbf{B} + \det \mathbf{C} \\ &= (x - a_1)x^{d-1} + p_{\mathbf{C}'}(x) \\ &= (x - a_1)x^{d-1} + (x^{d-1} - (x + a_2)x^{d-2} - \ldots - a_d x^0) \quad \text{(par ind.)} \\ &= x^d - a_1 x^{d-1} + x^{d-1} - x^{d-1} - a_2 x^{d-2} - \ldots - a_d x^0 \\ &= x^d - a_1 x^{d-1} - a_2 x^{d-2} - \ldots - a_d x^0. \end{aligned}$$

□

2. Avec des multiplicités possiblement différentes.

Exemple.

Reconsidérons la récurrence linéaire homogène de l'exemple précédent:

$$t(n) = \begin{cases} 1 & \text{si } n \leq 1, \\ t(n-1) + 2 \cdot t(n-2) & \text{sinon.} \end{cases}$$

Par la proposition 31, le polynôme caractéristique de la matrice $\mathbf{A}$ associée à t est $p_{\mathbf{A}}(x) = x^2 - x - 2$. Puisque $p_{\mathbf{A}} = (x-2)(x+1)$, les valeurs propres de $\mathbf{A}$ sont 2 et -1. Comme il y en a deux et qu'elles sont distinctes, par le théorème 8, nous avons $t(n) = c_1 \cdot 2^n + c_2 \cdot (-1)^n$. Afin d'identifier les valeurs des constantes, nous pouvons instancier $t(0)$ et $t(1)$, c.-à-d. résoudre ce système d'équations:

$$\begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \boldsymbol{b}$$

Le système possède une unique solution:

$$\left(\begin{array}{cc|c} 1 & 1 & 1 \\ 2 & -1 & 1 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 1 & 1 \\ 0 & -3 & -1 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 1 & 1 \\ 0 & 1 & 1/3 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 0 & 2/3 \\ 0 & 1 & 1/3 \end{array}\right).$$

Nous obtenons donc bien $t(n) = (2/3) \cdot 2^n + (1/3) \cdot (-1)^n$.

Cas non homogène

Nous généralisons maintenant les énoncés et preuves de la section précédente au cas *non homogène*. Rappelons qu'une telle récurrence linéaire est de la forme:

$$t(n) = \begin{cases} b_{d-n} & \text{si } n < d, \\ a_1 \cdot t(n-1) + \ldots + a_d \cdot t(n-d) + \alpha \cdot \beta^n & \text{si } n \geq d. \end{cases}$$

Nous pouvons représenter t par la matrice $\mathbf{A}_t$ et le vecteur $\boldsymbol{b}_t$ définis par

$$\mathbf{A}_t \coloneqq \begin{pmatrix} a_1 & a_2 & \cdots & a_{d-1} & a_d & \alpha \\ 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & \beta \end{pmatrix} \text{ et } \boldsymbol{b}_t \coloneqq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{d-1} \\ b_d \\ \beta^d \end{pmatrix}.$$

Comme précédemment, nous allons souvent omettre l'indice « t ».

Exemple.

Considérons cette récurrence linéaire non homogène qui émane du problème des tours de Hanoï présenté à la section 5.2.2:

$$t(n) = \begin{cases} 0 & \text{si } n = 0, \\ 2 \cdot t(n-1) + 2 & \text{sinon.} \end{cases}$$

Puisque $\alpha = 2$ et $\beta = 1$, sa matrice et son vecteur sont:

$$\mathbf{A} = \begin{pmatrix} 2 & 2 \\ 0 & 1 \end{pmatrix} \text{ et } \boldsymbol{b} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

En multipliant par $\mathbf{A}$ à répétition, nous obtenons ces vecteurs:

$$\mathbf{A}^0\boldsymbol{b} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \mathbf{A}^1\boldsymbol{b} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \mathbf{A}^2\boldsymbol{b} = \begin{pmatrix} 6 \\ 1 \end{pmatrix}, \mathbf{A}^3\boldsymbol{b} = \begin{pmatrix} 14 \\ 1 \end{pmatrix}, \mathbf{A}^4\boldsymbol{b} = \begin{pmatrix} 30 \\ 1 \end{pmatrix}, \ldots$$

En ne conservant que le premier élément de chaque vecteur, nous obtenons les valeurs de t: $0, 2, 6, 14, 30, \ldots$.

Comme l'exemple précédent le suggère, en général, l'avant-dernier élément de $\mathbf{A}^n\boldsymbol{b}$ correspond à $t(n)$, et le dernier élément contient β^{n+d}. Cela découle de la définition de $\mathbf{A}$ et $\boldsymbol{b}$ et se démontre par induction sur n.

Soit $\mathbf{A}_0$ la matrice obtenue en retirant la dernière ligne et la dernière colonne de $\mathbf{A}$. Soit $\boldsymbol{b}_0$ le vecteur obtenu en retirant le dernier élément de $\boldsymbol{b}$. Remarquons que $\mathbf{A}_0$ et $\mathbf{b}_0$ correspondent à la récurrence linéaire *homogène* t_0 obtenue en retirant le terme $\alpha \cdot \beta^n$ du cas général de t. La forme close de t peut être extraite à l'aide de ce théorème:

Théorème 9. *Si* $\mathbf{A}$ *possède* $d+1$ *valeurs propres* $\lambda_1, \ldots, \lambda_{d+1} \in \mathbb{R}$ *distinctes, alors il existe* $c_1, \ldots, c_{d+1} \in \mathbb{R}$ *telles que* $t(n) = c_1 \cdot \lambda_1^n + \cdots + c_d \cdot \lambda_d^n + c_{d+1} \cdot \beta^n$.

Démonstration. Supposons que $\mathbf{A}$ possède $d+1$ valeurs propres distinctes. Soient $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ les valeurs propres de $\mathbf{A}_0$. Soit $\boldsymbol{v}_i$ un vecteur propre de $\mathbf{A}_0$ associé à λ_i. Pour chaque $i \in [1..d]$, posons $\boldsymbol{w}_i \coloneqq (\boldsymbol{v}_i, 0)$. Chaque $\boldsymbol{w}_i$ est un vecteur propre de $\mathbf{A}$ dont la valeur propre est λ_i. Comme toutes les valeurs propres de $\mathbf{A}$ sont distinctes, nous avons $k = d$ et il ne reste donc qu'une valeur propre λ_{d+1} à identifier.

Soit $\boldsymbol{w}_{d+1}$ un vecteur propre associé à λ_{d+1}. Posons $W \coloneqq \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{d+1}\}$. Par le lemme 1(b), l'ensemble W engendre $\mathbb{R}^{d+1}$. Comme $\boldsymbol{w}_i(d+1) = 0$ pour tout $i \in [1..d]$, nous avons forcément $\boldsymbol{w}_{d+1}(d+1) \neq 0$. Par définition de $\mathbf{A}$, nous avons $(\mathbf{A}\boldsymbol{w}_{d+1})(d+1) = \beta\boldsymbol{w}_{d+1}(d+1)$. De plus, $\mathbf{A}\boldsymbol{w}_{d+1} = \lambda_{d+1}\boldsymbol{w}_{d+1}$. Ainsi, $\lambda_{d+1}\boldsymbol{w}_{d+1}(d+1) = \beta\boldsymbol{w}_{d+1}(d+1)$. Comme $\boldsymbol{w}_{d+1}(d+1) \neq 0$, nous pouvons diviser des deux côtés, ce qui implique $\lambda_{d+1} = \beta$.

Par le lemme 1(b), le vecteur $\boldsymbol{b}$ est une combinaison linéaire de W. Il existe donc des coefficients $\alpha_1, \ldots, \alpha_{d+1} \in \mathbb{R}$ tels que $\boldsymbol{b} = \alpha_1\boldsymbol{w}_1 + \ldots + \alpha_{d+1}\boldsymbol{w}_{d+1}$.

Ainsi:

$$\begin{aligned}\mathbf{A}^n\boldsymbol{b} &= \mathbf{A}^n(\alpha_1\boldsymbol{w}_1 + \ldots + \alpha_{d+1}\boldsymbol{w}_{d+1}) \\ &= \alpha_1\mathbf{A}^n\boldsymbol{w}_1 + \ldots + \alpha_{d+1}\mathbf{A}^n\boldsymbol{w}_{d+1} \\ &= \alpha_1\lambda_1^n\boldsymbol{w}_1 + \ldots + \alpha_{d+1}\lambda_{d+1}^n\boldsymbol{w}_{d+1} \qquad \text{(par lemme 1(a))}.\end{aligned}$$

Rappelons que $t(n) = (\mathbf{A}^n\boldsymbol{b})(d)$. Nous avons donc:

$$t(n) = \alpha_1\lambda_1^n\boldsymbol{w}_1(d) + \ldots + \alpha_{d+1}\lambda_{d+1}^n\boldsymbol{w}_{d+1}(d).$$

En posant $c_i \coloneqq \alpha_i\boldsymbol{w}_i(d)$, nous obtenons $t(n) = c_1\lambda_1^n + \ldots + c_{d+1}\lambda_{d+1}^n$. Comme $\lambda_{d+1} = \beta$, nous avons bien la forme attendue. □

La proposition suivante démontre que le polynôme caractéristique de **A** s'exprime en fonction de celui de $\mathbf{A}_0$:

Proposition 32. *Le polynôme caractéristique de* **A** *est* $p_{\mathbf{A}_0}(x) \cdot (x - \beta)$.

Démonstration. Rappelons que **A** est de dimension $(d+1) \times (d+1)$. Nous procédons par induction sur d. Si $d = 1$, alors $p_{\mathbf{A}}$ est comme attendu puisque:

$$\det(x\mathbf{I} - \mathbf{A}) = \begin{vmatrix} x - a_1 & -\alpha \\ 0 & x - \beta \end{vmatrix} = (x - a_1)(x - \beta) + 0 \cdot \alpha = (x - a_1)(x - \beta).$$

Supposons que $d > 1$. Soit **B** la matrice de dimension $d \times d$ obtenue en retranchant la première ligne et la première colonne de $x\mathbf{I} - \mathbf{A}$. Soit **C** la matrice de dimension $d \times d$ obtenue en retranchant la deuxième ligne et la première colonne de $x\mathbf{I} - \mathbf{A}$. Ces matrices sont de la forme:

$$\mathbf{B} = \begin{pmatrix} x & 0 & \cdots & 0 & 0 \\ -1 & x & \cdots & 0 & 0 \\ 0 & -1 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -1 & x - \beta \end{pmatrix} \text{ et } \mathbf{C} = \begin{pmatrix} -a_2 & -a_3 & \cdots & -a_d & -\alpha \\ -1 & x & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ddots & x & 0 \\ 0 & 0 & \cdots & -1 & x - \beta \end{pmatrix}.$$

Soit $\mathbf{C}'$ la matrice d'une récurrence linéaire non homogène dont les coefficients sont $x + a_2, a_3, a_4, \ldots, a_d$. Remarquons que **B** est une matrice diagonale, et que $\det(x\mathbf{I} - \mathbf{C}') = \det(\mathbf{C})$. En calculant le déterminant de $x\mathbf{I} - \mathbf{A}$ via sa première

colonne, et en utilisant la proposition 31, nous obtenons:

$$\begin{aligned}
p_{\mathbf{A}}(x) &= \det(x\mathbf{I} - \mathbf{A}) \\
&= (x - a_1) \det \mathbf{B} + \det \mathbf{C} \\
&= (x - a_1)x^{d-1} + p_{\mathbf{C}'}(x) \\
&= (x - a_1)x^{d-1}(x - \beta) + p_{\mathbf{C}'_0}(x) \cdot (x - \beta) \\
&= (x - a_1)x^{d-1}(x - \beta) + (x^{d-1} - (x + a_2)x^{d-2} - \ldots - a_d x^0)(x - \beta) \\
&= [(x - a_1)x^{d-1} + (x^{d-1} - (x + a_2)x^{d-2} - \ldots - a_d x^0)](x - \beta) \\
&= [x^d - a_1 x^{d-1} + x^{d-1} - x^{d-1} - a_2 x^{d-2} - \ldots - a_d x^0](x - \beta) \\
&= [x^d - a_1 x^{d-1} - a_2 x^{d-2} - \ldots - a_d x^0](x - \beta) \\
&= p_{\mathbf{A}_0}(x) \cdot (x - \beta).
\end{aligned}$$

□

Exemple.

Reconsidérons la récurrence linéaire de l'exemple précédent:

$$t(n) = \begin{cases} 0 & \text{si } n = 0, \\ 2 \cdot t(n-1) + 2 & \text{sinon.} \end{cases}$$

Par la proposition 31, le polynôme caractéristique de la matrice $\mathbf{A}_0$ de t_0 est $p_{\mathbf{A}_0}(x) = (x-2)$. Par la proposition 32, le polynôme caractéristique de la matrice $\mathbf{A}$ associée à t est $p_{\mathbf{A}}(x) = (x-2)(x-1)$. Celui-ci possède deux racines distinctes. Par le théorème 9, nous avons $t(n) = c_1 \cdot 2^n + c_2 \cdot 1^n$. Afin d'identifier les valeurs des constantes, nous pouvons instancier $t(0)$ et $t(1)$, c.-à-d. résoudre ce système d'équations:

$$\begin{pmatrix} 1 & 1 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Le système possède une unique solution:

$$\left(\begin{array}{cc|c} 1 & 1 & 0 \\ 2 & 1 & 2 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & -1 & 2 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1 & -2 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 0 & 2 \\ 0 & 1 & -2 \end{array}\right).$$

Nous obtenons donc $t(n) = 2 \cdot 2^n - 2 = 2^{n+1} - 2$ comme à la section 5.2.2.

Bibliographie

[BB96] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., 1996.

[BM91] Robert S. Boyer and J. Strother Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[Cha20] Timothy M. Chan. More logarithmic-factor speedups for 3SUM, (median, +)-convolution, and some geometric 3SUM-hard problems. *ACM Transactions on Algorithms*, 16(1):7:1–7:23, 2020.

[CJ86] I-Ping Chu and Richard Johnsonbaugh. Tiling deficient boards with trominoes. *Mathematics Magazine*, 59(1):34–40, 1986.

[Eri19] Jeff Erickson. *Algorithms*. 2019.

[Gol54] S. W. Golomb. Checker boards and polyominoes. *The American Mathematical Monthly*, 61(10):675–682, 1954.

[KLP75] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

Index