

IFT436 – Algorithmes et structures de données

NOTES DE COURS

Michael Blondin




2 décembre 2020

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, en français, et taillées sur mesure pour chaque cohorte. 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 évoluera 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.



Cette œuvre est mise à disposition selon les termes de la licence
Creative Commons Attribution 4.0 International.

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 le code source associé au passage.

Table des matières

I Fondements	1
0 Rappel de notions mathématiques	2
0.1 Notation et objets élémentaires	2
0.1.1 Ensembles	2
0.1.2 Séquences	3
0.1.3 Nombres	3
0.1.4 Combinatoire	4
0.2 Techniques de preuve	4
0.2.1 Preuves directes	4
0.2.2 Preuves par contradiction	5
0.2.3 Preuves par induction	5
0.3 Exercices	11
1 Analyse des algorithmes	13
1.1 Temps d'exécution	13
1.1.1 Temps en fonction d'un paramètre	13
1.1.2 Temps en fonction de plusieurs paramètres	14
1.2 Notation asymptotique	15
1.2.1 Notation \mathcal{O}	15
1.2.2 Notation Ω	19
1.2.3 Notation Θ	20
1.3 Étude de cas: vote à majorité absolue	21
1.4 Simplification du décompte des opérations	23
1.5 Règle de la limite	25
1.6 Notation asymptotique à plusieurs paramètres	26
1.7 Correction et terminaison	27
1.7.1 Correction	27
1.7.2 Bon fonctionnement	28
1.8 Étude de cas: vote à majorité absolue (suite)	28
1.9 Exercices	33

2	Tri	35
2.1	Approche générique	35
2.2	Tri par insertion	39
2.3	Tri par monceau	40
2.4	Tri par fusion	41
2.5	Tri rapide	42
2.6	Propriétés intéressantes	44
2.7	Tri sans comparaison	45
2.8	Exercices	47
3	Graphes	49
3.1	Graphes non dirigés	49
3.2	Graphes dirigés	50
3.3	Chemins et cycles	51
3.4	Sous-graphes et connexité	52
3.5	Représentation	52
3.5.1	Matrice d'adjacence	52
3.5.2	Liste d'adjacence	53
3.5.3	Complexité des représentations	53
3.6	Accessibilité	54
3.6.1	Parcours en profondeur	54
3.6.2	Parcours en largeur	55
3.7	Calcul de plus court chemin	56
3.8	Ordre topologique et détection de cycle	57
3.9	Arbres	58
3.10	Exercices	60
II	Paradigmes	62
4	Algorithmes gloutons	63
4.1	Arbres couvrants minimaux	63
4.1.1	Algorithme de Prim–Jarník	64
4.1.2	Algorithme de Kruskal	66
4.2	Approche générique	70
4.3	Exercices	72
5	Algorithmes récursifs et approche diviser-pour-régner	73
5.1	Tours de Hanoï	73
5.2	Réurrences linéaires	76
5.2.1	Cas homogène	76
5.2.2	Cas non homogène	79
5.3	Exponentiation rapide	79
5.4	Multiplication rapide	81
5.5	Théorème maître	83

5.6	Problème de la ligne d'horizon	85
5.7	Racines multiples et changement de domaine	88
5.8	Exercices	90
6	Force brute	96
6.1	Problème des n dames	96
6.2	Problème du sac à dos	99
6.3	Problème du retour de monnaie	102
6.4	Satisfaction de formules de logique propositionnelle	103
6.5	Programmation linéaire entière	104
6.6	Exercices	105
7	Programmation dynamique	107
7.1	Approche descendante	107
7.2	Approche ascendante	108
7.2.1	Problème du retour de monnaie	108
7.2.2	Problème du sac à dos	110
7.3	Plus courts chemins	111
7.3.1	Algorithme de Dijkstra	112
7.3.2	Algorithme de Floyd-Warshall	114
7.3.3	Algorithme de Bellman-Ford	118
7.3.4	Sommaire	120
7.4	Exercices	121
8	Algorithmes et analyse probabilistes	125
8.1	Nombres aléatoires	125
8.2	Paradigmes probabilistes	127
8.2.1	Algorithmes de Las Vegas et temps espéré	128
8.2.2	Algorithmes de Monte Carlo et probabilité d'erreur	129
8.3	Coupe minimum: algorithme de Karger	129
8.4	Amplification de probabilité	132
8.5	Temps moyen	133
8.6	Exercices	135
	Solutions des exercices	137
	Fiches récapitulatives	171
	Bibliographie	175
	Index	176

Fondements

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*, $\{aa, ab, ba, bb\}$ est l'ensemble des mots de taille deux formés des lettres a et b, $\{0, 2, 4, 6, 8, \dots\}$ 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. $|\{a, b, c\}| = 3$ 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*

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

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

$$\begin{aligned} X \cup Y &:= \{x : x \in X \vee x \in Y\}, & X \setminus Y &:= \{x : x \in X \wedge x \notin Y\}, \\ X \cap Y &:= \{x : x \in X \wedge x \in Y\}, & X \times Y &:= \{(x, y) : x \in X \wedge y \in Y\}. \end{aligned}$$

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

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

Par exemple, $\mathcal{P}(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ 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, $s = [k, a, y, a, k]$ représente la chaîne de caractères « kayak » et $f = [1, 1, 2, 3, 5, 8, \dots]$ 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. $s[1] = k$ 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:

$$s[i:j] := [s[i], s[i+1], \dots, s[j]].$$

Par convention, $s[i:j] := []$ lorsque $i > j$. Nous écrivons $s+t$ afin de dénoter la séquence obtenue en ajoutant t à la suite de s .

0.1.3 Nombres

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

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

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 $[a..b] := \{a, a+1, \dots, b\}$.

Logarithmes et exponentielles. Le *logarithme en base* $b \in \mathbb{N}_{\geq 2}$ est la fonction $\log_b : \mathbb{R}_{>0} \rightarrow \mathbb{R}$ 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:

$$\begin{aligned}\log_b(xy) &= \log_b(x) + \log_b(y), & \log_b(x/y) &= \log_b(x) - \log_b(y), \\ \log_b(x^y) &= y \cdot \log_b(x), & \log_a(x) &= \frac{\log_b(x)}{\log_b(a)}, \\ \log_b(1) &= 0, & x < y &\implies \log_b(x) < \log_b(y).\end{aligned}$$

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

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

Arithmétique modulaire. Pour tout $n \in \mathbb{N}$ 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 $n \bmod d := n - (n \div d) \cdot d$. 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 $n! := 1 \cdot 2 \cdot \dots \cdot n$ avec $0! := 1$ 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 $[a, b, c]$:

$$[a, b, c], [a, c, b], [b, a, c], [b, c, a], [c, a, b], [c, b, a].$$

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

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

Par convention, $\binom{n}{k} := 0$ 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 $\{\text{●}, \text{■}, \text{▲}, \text{◆}\}$:

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

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:

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

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. \square

0.2.2 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. \square

0.2.3 Preuves par induction

Rappelons la technique de *preuve par induction*. Soit $\varphi: \mathbb{N} \rightarrow \{\text{faux}, \text{vrai}\}$ 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 $n \geq b$.

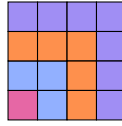
Ces deux étapes se nomment respectivement *cas de base* et *étape d'induction*.

Voyons quelques exemples de preuves par induction.

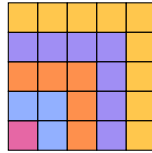
Carré d'un nombre. En inspectant quelques valeurs, la somme des n premiers nombres impairs semble égalé n^2 :

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

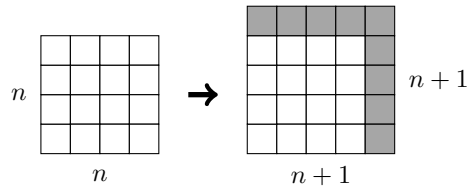
Cherchons à démontrer que cela est vrai *pour tout* $n \in \mathbb{N}_{\geq 1}$. Considérons le cas où $n = 4$. Nous pouvons visualiser graphiquement que $1 + 3 + 5 + 7 = 4^2$:



En ajoutant 9 cases au 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 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 $s_n := \sum_{i=1}^n (2i - 1)$ 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:

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

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 : $n \in \mathbb{N}_{\geq 1}$

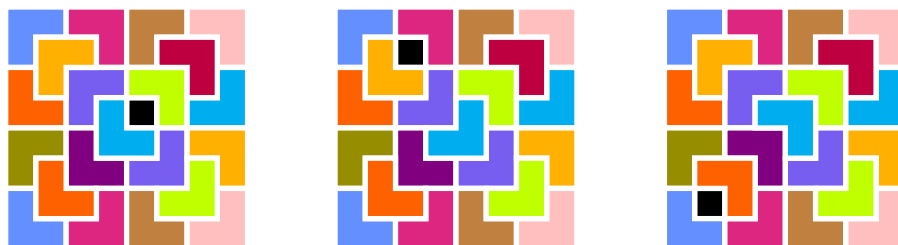
Résultat : n^2

- 1 $c \leftarrow 0; m \leftarrow 1$
 - 2 **faire** n **fois fois**
 - 3 | $c \leftarrow c + m; m \leftarrow m + 2$
 - 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:

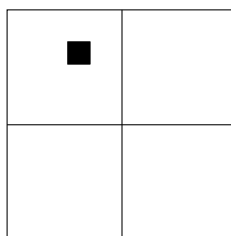
Proposition 4. Pour tout $n \in \mathbb{N}$ et pour tous $i, j \in [1..2^n]$, il est possible de paver la totalité d'une grille $2^n \times 2^n$ à l'exception de la case (i, j) .

Démonstration. Nous prouvons la proposition par induction sur n .

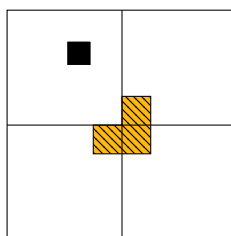
Cas de base ($n = 0$). La grille possède une seule case qui est forcément la case (i, j) . Le pavage ne nécessite donc aucune tuile.

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:



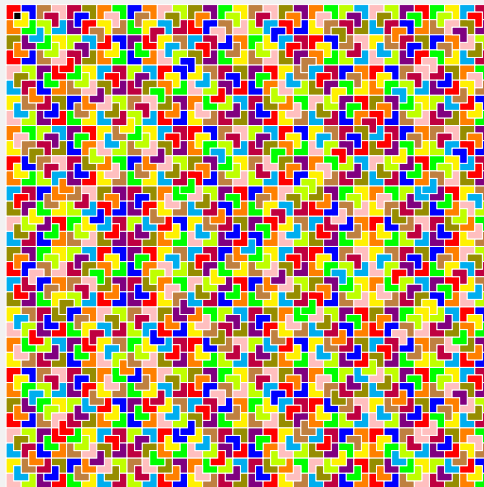
Retrançons une case à chacune des trois autres sous-grilles de façon à former un « L ». Par exemple, dans le cas précédent, nous retrançons ces trois cases hachurées en couleur:



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. \square

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 $\varphi(b), \varphi(b+1), \dots, \varphi(n)$ 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. \square

0.3 Exercices

- 0.1) Quelle est la taille de l'ensemble $\{\{a, b\}, \{\emptyset, \{1, 2\}\}, \{\emptyset, \{\emptyset, \emptyset\}\}$?
- 0.2) Soient les ensembles $X := \{1, 5, 6, a, 9, 23, c\}$ et $Y := \{-23, 3, 5, 9, a\}$.
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 $\mathcal{P}(\{a, b, c\})$.
- 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 $\binom{n}{0} + \binom{n}{1} + \dots + \binom{n}{n} = 2^n$ 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. \square

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 récurrents 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 $t_{\max} : \mathbb{N} \rightarrow \mathbb{N}$ telle que:

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

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:

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

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:

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

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:

$$t_{\max}(n_1, \dots, n_k) := \max\{f(x) : \text{entrée } x \text{ de taille } (n_1, \dots, n_k)\},$$

$$t_{\min}(n_1, \dots, n_k) := \min\{f(x) : \text{entrée } x \text{ de taille } (n_1, \dots, n_k)\}.$$

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:

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

$$= 7mn + 4m - 3.$$

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

Entrées : matrice \mathbf{A} d'entiers de taille $m \times n$, où $m, n \in \mathbb{N}_{>0}$

Sorties : valeur maximale parmi toutes les entrées de \mathbf{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:

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

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

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

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

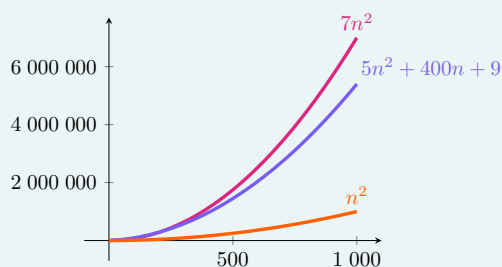
Intuitivement, $f \in \mathcal{O}(g)$ indique que f croît *aussi ou moins* rapidement que la fonction g . Reconsidérons l'algorithme 2. Nous avons déjà établi que son temps d'exécution est d'au plus $7n - 4$ pour tout $n \geq 1$. Nous avons donc:

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

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 $n \geq 2$:

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

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 $c := c' \cdot c''$ comme constante multiplicative et $n_0 := \max(n'_0, n''_0)$ comme seuil. Pour tout $n \geq n_0$, nous avons:

$$\begin{aligned} f(n) &\leq c' \cdot g(n) && \text{(car } n \geq n_0 \geq n'_0) \\ &\leq c' \cdot c'' \cdot h(n) && \text{(car } n \geq n_0 \geq n''_0) \\ &= c \cdot h(n) && \text{(par définition de } c). \end{aligned} \quad \square$$

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:

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

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

Démonstration. Soit m_i le seuil à partir duquel f_i est positive. Nous prouvons la proposition en prenant $c := \max(1/c_1, 1/c_2, \dots, 1/c_k)$ comme constante multiplicative et $n_0 := \max(m_1, m_2, \dots, m_k)$ comme seuil. Pour tout $n \geq n_0$:

$$\begin{aligned} \sum_{i=1}^n f_i(n) &= \sum_{i=1}^n \frac{1}{c_i} \cdot (c_i \cdot f_i(n)) && \text{(puisque } c_i \neq 0) \\ &\leq \sum_{i=1}^n c \cdot (c_i \cdot f_i(n)) && \text{(par déf. de } c \text{ et car } c_i \cdot f_i(n) > 0) \\ &= c \cdot \sum_{i=1}^n c_i \cdot f_i(n). \end{aligned} \quad \square$$

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

Démonstration. Nous démontrons la proposition en prenant $c := k$ comme constante multiplicative et $n_0 := 0$ comme seuil. Pour tout $n \geq n_0$, nous avons:

$$\begin{aligned} \sum_{i=1}^k f_i(n) &\leq k \cdot \max(f_1(n), f_2(n), \dots, f_k(n)) \\ &= c \cdot \max(f_1(n), f_2(n), \dots, f_k(n)). \quad \square \end{aligned}$$

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 $g(n) := 9000n^2 + 3^n + 8$. Par la proposition 7, nous avons $g \in \mathcal{O}(n^2 + 3^n + 1)$. 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 .

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 négatif¹. 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. □

1. Cela découle du fait que tout polynôme possède un nombre fini de zéros.

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:

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

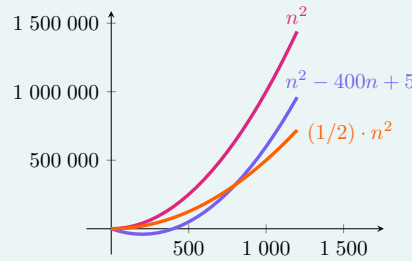
Nous appelons les valeurs c et n_0 une *constante multiplicative* et un *seuil*.

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 $f(n) := n^2 - 400n + 5$ 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 $c := 1/2$ comme constante multiplicative et $n_0 := 800$ 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

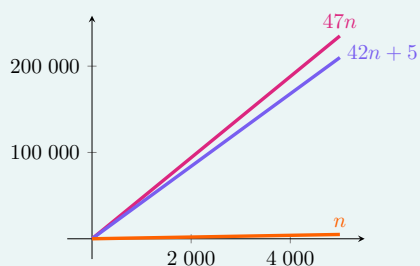
Nous introduisons une troisième et dernière notation asymptotique qui permet de borner une fonction à la fois inférieurement et supérieurement. Pour toute fonction $g \in \mathcal{F}$, nous définissons:

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

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 $f(n) := 42n + 5$ 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 $f \in \Theta(n)$.

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:

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

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 $[a, b, b, a, a, a, c, a]$ contient la valeur majoritaire a qui apparaît 5 fois parmi 8 éléments; alors que la séquence $[a, b, a, b, c]$ 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.



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  $i \in [1..n]$ 
2    $c \leftarrow 0$ 
3   pour  $j \in [1..n]$  // Compter # d'occur. de  $T[i]$ 
4     si  $T[j] = T[i]$  alors  $c \leftarrow c + 1$ 
5   si  $c > n \div 2$  alors retourner  $T[i]$ 
6 retourner aucune
```

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

$$n \cdot (1 + 3n + 2) \leq t(n) \leq n \cdot (1 + 5n + 3).$$

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 $t(n) \geq 3n^2 + 3n \geq 6n^2$ 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.



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  $i \in [1..n]$ 
2   si  $T[i] \neq \perp$  alors
3      $c \leftarrow 0$ ;  $x \leftarrow T[i]$ 
4     pour  $j \in [i..n]$  // Compter # d'occur. de  $T[i]$ 
5       si  $T[j] = x$  alors
6          $c \leftarrow c + 1$ 
7          $T[j] \leftarrow \perp$  // «Retirer» en remplaçant par  $\perp$ 
8     si  $c > n \div 2$  alors retourner  $x$ 
9 retourner aucune
```

Analysons le temps d'exécution t de l'algorithme 5. En supposant que les 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 « $T[i] \neq \perp$ » 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   |  $c \leftarrow c + 1$ 
3   |  $T[j] \leftarrow \perp$ 

```

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, \dots, j_ℓ 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, \dots, j_ℓ . Par exemple, dans le code ci-dessus, le nombre d'exécutions de la ligne $j_1 := 1$ borne celle des lignes $j_2 := 2$ et $j_3 := 3$.

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  $i \in [1..n]$ 
  | si /* élémentaire */ alors
  | | pour  $j \in [i..n]$ 
  | | | /* élémentaire
  | | |
  | |
  |
retourner
  
```

Bien plus formellement, nous exploitons l'observation suivante:

Proposition 12. Soit $J = \{j_1, j_2, \dots, j_\ell\}$ 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 élémentaire 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 $t'_{\max}(n) := \max\{f'(x) : \text{entrée } x \text{ de taille } n\}$.

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:

$$\begin{aligned}
 f(x) &\leq m + \sum_{j \in J} c_j \cdot k_j \\
 &\leq m + k_1 \cdot \sum_{j \in J} c_j && \text{(car } k_1 = \max\{k_1, \dots, k_\ell\}) \\
 &\leq m + k_1 \cdot \ell \cdot \max(c_1, \dots, c_\ell) \\
 &\leq \ell \cdot \max(c_1, \dots, c_\ell) \cdot (m + k_1) \\
 &\leq \ell \cdot \max(c_1, \dots, c_\ell) \cdot (m + c_1 \cdot k_1) \\
 &= \ell \cdot \max(c_1, \dots, c_\ell) \cdot f'(x).
 \end{aligned}$$

Nous obtenons donc $f(x) \leq c \cdot f'(x)$, où $c := \ell \cdot \max(c_1, \dots, c_\ell)$.

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 $t_{\max} \in \mathcal{O}(t'_{\max})$. \square

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 13. Soient $f, g \in \mathcal{F}$ et $\ell := \lim_{n \rightarrow +\infty} f(n)/g(n)$ 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 $f(n) := \log n$ et $g(n) := \sqrt{n}$. Nous avons:

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

Par la proposition 13, 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:

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

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

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

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

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

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:

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

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 $m_0 = n_0 := 1$ comme seuils, et $c := 11$ 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 binaire, 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:

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

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 14. 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 est loin d'être 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  $x \leftarrow aucune$ ;  $c \leftarrow 0$            // Chercher une valeur majoritaire  $x$ 
2 pour  $i \in [1..n]$ 
3   | si  $c = 0$  alors  $x \leftarrow T[i]$ ;  $c \leftarrow 1$ 
4   | sinon si  $T[i] = x$  alors  $c \leftarrow c + 1$ 
5   | sinon  $c \leftarrow c - 1$ 

6  $c \leftarrow 0$                                // Vérifier que  $x$  est bien majoritaire
7 pour  $i \in [1..n]$ 
8   | si  $T[i] = x$  alors  $c \leftarrow c + 1$ 

9 si  $c > n \div 2$  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 $T = [a, b, b, a, a, a, c, a]$, la première phase de l’algorithme termine avec $x = a$ 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 $T' = [a, b, a, b, c]$. La première phase termine avec $x = c$ 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 $T[1:i] = [T[1], T[2], \dots, T[i]]$.

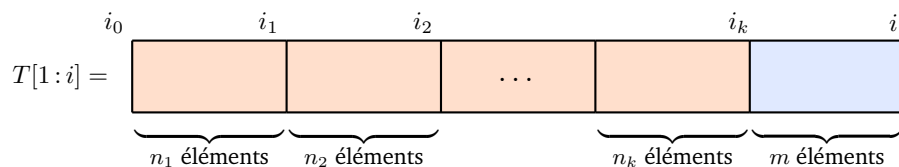
Proposition 15. *Après l'exécution du corps de la première boucle de l'algorithme 8, l'invariant suivant est satisfait:*

- (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 $i_0 < i_1 < i_2 < \dots < i_k$ les itérations auxquelles cela s'est produit, où $i_0 := 0$ (le moment avant d'entrer dans la boucle).

Posons $n_j := i_j - i_{j-1}$ pour tout $j \in [1..k]$, et $m := i - i_k$. 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 $n_j/2$ occurrences 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:

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

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

□

Démonstration formelle. 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 $T[1:i]$. \square

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 15 (c), y apparaît au plus $\frac{n-c}{2}$ fois dans $T[1:n] = T$, où $c \geq 0$ par la proposition 15 (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. \square

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: $\mathcal{O}(f) = \mathcal{O}(g) \iff f \in \mathcal{O}(g) \text{ et } g \in \mathcal{O}(f)$. 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 $R(f, g) \stackrel{\text{déf}}{\iff} f \in \Theta(g)$ est une relation d'équivalence. Est-ce aussi le cas si l'on remplace Θ par \mathcal{O} ou Ω ?
- 1.7) Comparez $f(n) := n^2$ et $g(n) := 2^n$ 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 $\log n \in \mathcal{O}(\sqrt[d]{n})$ et $\sqrt[d]{n} \notin \mathcal{O}(\log n)$ pour tout $d \in \mathbb{N}_{>0}$.
- 1.14) ★ Soient $f(n) := n$ et $g(n) := 2^{\lfloor \log n \rfloor}$. Montrez que $f \in \Theta(g)$, mais que la limite $\lim_{n \rightarrow \infty} f(n)/g(n)$ 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 14 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, A2019)

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

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

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?

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 $s[1] \leq s[2] \leq \dots \leq s[n]$. De façon générale, un *algorithme de tri* est un algorithme dont la pré-condition et la post-condition correspondent à:

$$\mathbb{D} = \{s : s \text{ est une séquence d'éléments comparables}\},$$

$$\varphi(s, t) = t \text{ est triée et } t \text{ est une permutation de } s.$$

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 tri¹. Pour toute séquence s de n éléments comparables, posons:

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

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) , le contenu de $s[i]$ et $s[j]$ sont intervertis. Cette procédure est décrite à l'algorithme 9.

1. Sauf pour quelques exceptions, tels les algorithmes loufoques comme « *bogosort* ».

Algorithme 9 : Algorithme générique de tri.

Entrées : séquence s d'éléments comparables

Résultat : s est triée

```

1 tant que  $\exists(i, j) \in \text{inv}(s)$ 
2   |  $s[i] \leftrightarrow s[j]$                                // 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 disparues, 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 16. 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:

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

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

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

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:

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

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:

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

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

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

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:

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

Fin de la preuve. Posons $P := \{(x, y) \in [1..n]^2 : |X \cap \{x, y\}| = 1\}$. 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:

$$\begin{aligned}
|\text{inv}(s)| &= \sum_{\substack{(x,y) \in [1..n]^2 \\ x < y}} f(x, y) \\
&= f(i, j) + \sum_{\substack{(x,y) \in X^2 \\ x < y}} f(x, y) + \sum_{\substack{(x,y) \in P \\ x < y}} f(x, y) \\
&> f'(i, j) + \sum_{\substack{(x,y) \in X^2 \\ x < y}} f(x, y) + \sum_{\substack{(x,y) \in P \\ x < y}} f(x, y) \quad (\text{par (2.1)}) \\
&= f'(i, j) + \sum_{\substack{(x,y) \in X^2 \\ x < y}} f'(x, y) + \sum_{\substack{(x,y) \in P \\ x < y}} f(x, y) \quad (\text{par (2.2)}) \\
&\geq f'(i, j) + \sum_{\substack{(x,y) \in X^2 \\ x < y}} f'(x, y) + \sum_{\substack{(x,y) \in P \\ x < y}} f'(x, y) \quad (\text{par (2.3)-(2.5)}) \\
&= \sum_{\substack{(x,y) \in [1..n]^2 \\ x < y}} f'(x, y) \\
&= |\text{inv}(s')|. \quad \square
\end{aligned}$$

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 16, nous avons $|\text{inv}(t_0)| > |\text{inv}(t_1)| > \dots$ 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 $(n-1) + \dots + 1 + 0 = n(n-1)/2$ inversions, l'algorithme effectue $\mathcal{O}(n^2)$ itérations. \square

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 17. *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 17 donne bien $\Theta(n)$ et $\Theta(n^2)$ pour ces deux cas.

La proposition 17 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.



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

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

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

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: $x = s[1 : m]$ et $y = s[m + 1 : n]$, 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 t et u 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.

**Algorithme 12** : Algorithme de tri par fusion.

Entrées : séquence s d'éléments comparables

Sorties : séquence s triée

```

trier( $s$ ):
    fusion( $x, y$ ): // fusionne deux seq. triées
         $i \leftarrow 1; j \leftarrow 1; z \leftarrow []$ 
        tant que  $i \leq |x| \wedge j \leq |y|$ 
            si  $x[i] \leq y[j]$  alors
                ajouter  $x[i]$  à  $z$ 
                 $i \leftarrow i + 1$ 
            sinon
                ajouter  $y[j]$  à  $z$ 
                 $j \leftarrow j + 1$ 
        retourner  $z + x[i : |x|] + y[j : |y|]$ 
    si  $|s| \leq 1$  alors retourner  $s$ 
    sinon
         $m \leftarrow |s| \div 2$ 
        retourner fusion(trier( $s[1 : m]$ ), trier( $s[m + 1 : |s|]$ ))

```

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 effectuée au plus $c \cdot (|x| + |y|)$ opérations pour une certaine constante $c \in \mathbb{R}_{>0}$, et que $|s|$ 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 18. $t(2^k) \leq 2^k(ck + 1)$ 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 :

$$\begin{aligned} t(2^{k+1}) &\leq 2 \cdot t(2^k) + c \cdot 2^{k+1} && \text{(par définition de } t) \\ &\leq 2 \cdot (2^k(ck + 1)) + c \cdot 2^{k+1} && \text{(par hypothèse d'induction)} \\ &= 2^{k+1}(ck + 1) + c \cdot 2^{k+1} \\ &= 2^{k+1}(ck + 1 + c) \\ &= 2^{k+1}(c(k + 1) + 1). \quad \square \end{aligned}$$

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 aléatoirement, 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  $pivot \in s$ 
    gauche  $\leftarrow [x \in s : x < pivot]$ 
    milieu  $\leftarrow [x \in s : x = pivot]$ 
    droite  $\leftarrow [x \in s : x > pivot]$ 
    retourner trier( $gauche$ ) + milieu + trier( $droite$ )

```

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( $lo, hi$ ):           // partitionne  $s$  autour de  $x = s[hi]$  et
     $x \leftarrow s[hi]; i \leftarrow lo$            // retourne le nouvel index  $i$  de  $x$ 
    pour  $j \in [lo..hi]$ 
      si  $s[j] < x$  alors
         $s[i] \leftrightarrow s[j]$ 
         $i \leftarrow i + 1$ 
     $s[i] \leftrightarrow s[hi]$ 
    retourner  $i$ 

  trier'( $lo, hi$ ):
    si  $lo < hi$  alors
       $pivot \leftarrow partition(lo, hi)$ 
      trier'( $lo, pivot - 1$ )           // Trier le côté gauche
      trier'( $pivot + 1, hi$ )           // Trier le côté droit

  trier'(1,  $|s|$ )
  retourner  $s$ 

```

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 :

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

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)			Sur place	Stable
	meilleur	moyen	pire		
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:

$$s = [111, 011, 101, 001, 010].$$

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:

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

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

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

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

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

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

Algorithme 15 : Algorithme de tri sans comparaison

Entrées : séquence s de n séquences de m bits

Résultat : s est triée

```

1 pour  $j \leftarrow m, \dots, 1$ 
2    $lo \leftarrow []; hi \leftarrow []$ 
3   pour  $i \leftarrow 1, \dots, n$ 
4     si  $s[i][j] = 0$  alors
5       ajouter  $s[i]$  à  $lo$ 
6     sinon
7       ajouter  $s[i]$  à  $hi$ 
8    $s \leftarrow lo + hi$ 
9 retourner  $s$ 

```

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.

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?
- 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, $[a, a, c, b, b]$ et $[c, a, a, b, b]$ sont des *regroupements* valides de $s = [b, a, a, c, b]$. 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 17.

- 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

```

      k fois
c ← [0, 0, ..., 0]
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.

Graphes

Dans ce chapitre, nous introduisons formellement les graphes comme structure abstraite, leurs propriétés, leur représentation en tant que structure de données, 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 (finis ou non) 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 supposons qu'un graphe est fini.

Nous écrivons $u \rightarrow v$ afin d'indiquer que $\{u, v\} \in E$. Observons que $u \rightarrow v \iff v \rightarrow u$. 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.

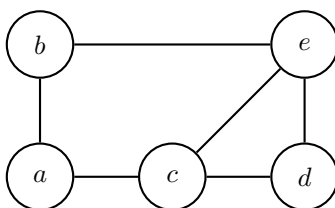


FIGURE 3.1 – Exemple de graphe (fini) non dirigé.

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 (finis ou non) 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 $u \rightarrow v$ afin d'indiquer que $(u, v) \in E$. Lorsque $u \rightarrow v$, nous disons que u est un *prédécesseur* 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 prédécesseurs, 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 $a \rightarrow b$, mais pas $b \rightarrow a$. De plus, nous avons:

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

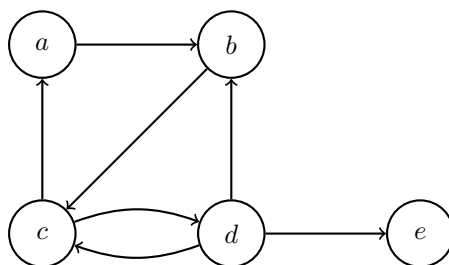


FIGURE 3.2 – Exemple de graphe fini dirigé.

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 Chemins et cycles

Soit $\mathcal{G} = (V, E)$ un graphe. Un *chemin* C , allant d'un sommet u vers un sommet v , est une séquence finie de sommets

$$C = [v_0, v_1, \dots, v_k]$$

telle que $v_0 = u$, $v_k = v$, et $v_{i-1} \rightarrow v_i$ pour tout $i \in [1..k]$. Nous disons qu'un chemin est *simple* s'il ne répète aucun sommet. La *longueur* d'un tel chemin correspond à $|C| := k$. Nous écrivons $u \xrightarrow{*} v$ afin d'indiquer qu'il existe un chemin de u vers v . En particulier, observons que $u \xrightarrow{*} u$ pour tout sommet u .

Un *cycle* est un chemin, de longueur non nulle, 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 arête¹. 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.

Exemple.

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

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

3.4 Sous-graphes et connexité

Soit $\mathcal{G} = (V, E)$ un graphe. Un *sous-graphe* de \mathcal{G} est un graphe $\mathcal{G}' = (V', E')$ tel que $V' \subseteq V$ et $E' \subseteq E$ n'utilise que des sommets de V' . Le *sous-graphe induit* par un ensemble de sommets $V' \subseteq V$ est le sous-graphe $\mathcal{G}[V'] := (V', E')$ 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 $\mathcal{G}'[\{e\}]$.

3.5 Représentation

3.5.1 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:

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

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:

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

3.5.2 Liste d'adjacence

Une *liste d'adjacence* d'un graphe $\mathcal{G} = (V, E)$ est une séquence adj telle que $adj[u] := [v \in V : u \rightarrow v]$ 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 $V = \{1, 2, \dots, n\}$, 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:

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

3.5.3 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 représentations². 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
$u \rightarrow v?$	$\Theta(1)$	$\mathcal{O}(1 + \min(\deg(u), \deg(v)))$	$\mathcal{O}(1 + \deg^+(u))$
$\{v : u \rightarrow v\}$	$\Theta(V)$	$\mathcal{O}(1 + \deg(u))$	$\mathcal{O}(1 + \deg^+(u))$
$\{u : u \rightarrow v\}$	$\Theta(V)$	$\mathcal{O}(1 + \deg(v))$	$\mathcal{O}(V + E)$
Ajouter $u \rightarrow v$	$\Theta(1)$	$\mathcal{O}(1 + \deg(u) + \deg(v))$	$\mathcal{O}(1 + \deg^+(u))$
Retirer $u \rightarrow v$	$\Theta(1)$	$\mathcal{O}(1 + \deg(u) + \deg(v))$	$\mathcal{O}(1 + \deg^+(u))$
Mémoire	$\Theta(V ^2)$	$\Theta(V + E)$	$\Theta(V + E)$

2. 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 $V \neq \{1, 2, \dots, n\}$. 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*.

3.6 Accessibilité

Nous considérons deux façons de parcourir un graphe. Plus précisément, étant donné un sommet de départ u , nous présentons deux approches afin de calculer 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.

3.6.1 Parcours en profondeur

La première approche consiste à débiter 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  $s \leftarrow []$ 
2 parcours( $x$ ):
3   si  $x$  n'est pas marqué alors
4     marquer  $x$ 
5     pour  $y \in V : x \rightarrow y$  // 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 permet 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  $P \leftarrow [u]$ 
2  $s \leftarrow []$ 
3 tant que  $P$  n'est pas vide
4   | dépiler  $x$  de  $P$ 
5   | si  $x$  n'est pas marqué alors
6   |   | marquer  $x$ 
7   |   | pour  $y \in V : x \rightarrow y$            // explorer voisins/succ.
8   |   |   | empiler  $y$  dans  $P$ 
9   |   | ajouter  $x$  à  $s$            // ajouter aux sommets accessibles
10 retourner  $s$ 

```

3.6.2 Parcours en largeur

Une alternative au parcours en profondeur consiste à explorer le graphe en largeur. Autrement dit, nous débutons au sommet u , puis nous 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.



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  $F \leftarrow [u]$ 
2  $s \leftarrow []$ 
3 tant que  $F$  n'est pas vide
4   | retirer  $x$  de  $F$ 
5   | si  $x$  n'est pas marqué alors
6   |   | marquer  $x$ 
7   |   | pour  $y \in V : x \rightarrow y$            // explorer voisins/succ.
8   |   |   | ajouter  $y$  à  $F$ 
9   |   | ajouter  $x$  à  $s$            // ajouter aux sommets accessibles
10 retourner  $s$ 

```

3.7 Calcul de plus court chemin

Les algorithmes d'exploration de la section précédente permettent de déterminer si $u \xrightarrow{*} v$ étant donné deux sommets u et v . Il serait possible d'adapter légèrement ces algorithmes afin qu'ils construisent explicitement un chemin simple de u vers v . Le parcours en profondeur aurait tendance à construire des chemins longs, parfois même de longueur maximale. Cependant, le parcours en largeur construit toujours un chemin de longueur *minimale* puisqu'il explore progressivement les sommets à distance $1, 2, \dots, |V| - 1$ du sommet de départ.

Algorithme 19 : 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  $F \leftarrow [u]$ 
3 trouvé  $\leftarrow$  faux
4 pred  $\leftarrow []$ 
5 tant que  $\neg$ trouvé et  $F$  n'est pas vide
6   retirer  $x$  de  $F$ 
7   marquer  $x$ 
8   pour  $y \in V : x \rightarrow y$  // explorer voisins/succ.
9     si  $y$  n'est pas marqué alors
10      |   pred[y]  $\leftarrow$   $x$  // se souvenir d'où on arrive
11      |   si  $y = v$  alors // cible  $v$  atteinte?
12      |   |   trouvé  $\leftarrow$  vrai
13      |   sinon
14      |   |   ajouter  $y$  à  $F$ 

  // Construire le chemin de  $u$  vers  $v$ 
15 si  $\neg$ trouvé alors
16   |   retourner aucun
17 sinon
18   |    $s \leftarrow []$ 
19   |    $x \leftarrow v$ 
20   |   tant que  $x \neq u$  // rebrousser chemin de  $v$  vers  $u$ 
21   |   |   ajouter  $x$  à  $s$ 
22   |   |    $x \leftarrow$  pred[ $x$ ]
23   |   renverser  $s$ 
24   retourner  $s$ 

```

L'algorithme 19 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 $[v, pred[v], pred[pred[v]], \dots, u]$ 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 $v, pred[v], pred[pred[v]], \dots$ directement dans une file.

3.8 Ordre topologique et détection de cycle

Un *ordre topologique* d'un graphe dirigé $\mathcal{G} = (V, E)$ est une séquence de sommets v_1, v_2, \dots, v_k qui satisfait:

- $V = \{v_1, v_2, \dots, v_k\}$, 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! En effet, ce scénario indiquerait qu'il existe un cycle dans \mathcal{G} :

Proposition 19. *Un graphe possède un ordre topologique ssi il est acyclique.*

La recherche d'un ordre topologique dans un graphe acyclique peut être vue comme un problème de tri selon l'ordre défini sur les sommets par $u \preceq v \stackrel{\text{déf}}{\iff} u \xrightarrow{*} v$. Cependant, comme cet ordre est un ordre partiel, et non un ordre total, nous ne pouvons pas utiliser les algorithmes de tri du chapitre 2³. De plus, même si cela était possible, évaluer $u \preceq v$ nécessiterait un temps linéaire. Nous présentons donc, à l'algorithme 20, une procédure qui calcule un ordre topologique d'un graphe en temps linéaire, s'il en existe, et qui détecte la présence d'un cycle autrement.



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.

3. En fait, le calcul d'un ordre topologique correspond précisément à la construction d'un ordre total compatible avec \preceq , c.-à-d. une *extension linéaire* de \preceq .

Algorithme 20 : 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
 1 $d \leftarrow [v \mapsto 0 : v \in V]$
 2 **pour** $(u, v) \in E$
 3 | $d[v] \leftarrow d[v] + 1$
// Calcul des sommets de degré entrant 0
 4 **initialiser** une file $F \leftarrow []$
 5 **pour** $v \in V$
 6 | **si** $d[v] = 0$ **alors ajouter** v à F
// Tri topologique
 7 $ordre \leftarrow []$
 8 **tant que** F n'est pas vide
 9 | **retirer** u de F
 10 | **ajouter** u à $ordre$
 11 | **pour** $v \in V : u \rightarrow v$ *// explorer successeurs*
 12 | | $d[v] \leftarrow d[v] - 1$
 13 | | **si** $d[v] = 0$ **alors ajouter** v à F
 14 **si** $|ordre| = |V|$ **alors retourner** $ordre$
 15 **sinon retourner** cycle détecté

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

Analysons le temps d'exécution de l'algorithme 20. 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 $\mathcal{O}(|V| + |E|)$.

3.9 Arbres

Nous disons qu'un graphe non dirigé est *acyclique* s'il ne possède pas de cycle simple. Une *forêt* est un graphe non dirigé acyclique et un *arbre* est une forêt connexe. Les arbres peuvent être caractérisés de plusieurs façons équivalentes:

Proposition 20. 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 $|E| = |V| - 1$.

Nous disons qu'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 $\deg(r) = 1$.

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. Par exemple, la figure 3.3 identifie un arbre couvrant. Tout graphe non dirigé connexe possède au moins un arbre couvrant, et il est possible d'en construire un en adaptant, par exemple, le parcours en profondeur.

Proposition 21. Un graphe non dirigé possède au moins un arbre couvrant ssi il est connexe.

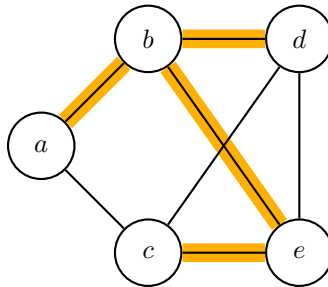
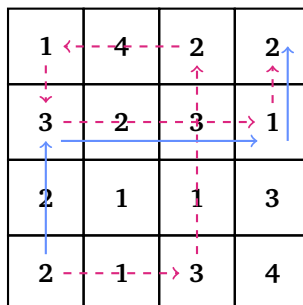


FIGURE 3.3 – Exemple d'arbre couvrant identifié par un trait gras de couleur.

3.10 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 de la figure 3.3.
- 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*.
- 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 sont de degré pair.
 - Montrez que si un graphe non dirigé \mathcal{G} possède au plus une composante connexe non triviale et ne contient que des sommets de degré pair, alors \mathcal{G} possède un cycle eulérien.
Indice: pensez à construire un cycle à partir d'un sommet arbitraire.
 - Donnez un algorithme qui détermine si un graphe non dirigé possède un cycle eulérien.
 - Adaptez l'algorithme pour qu'il identifie un cycle s'il y en a un.
- 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** x_i, j : saute à la ligne j si $x_i = 0$, et à la ligne suivante sinon;
- **cbnz** x_i, 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 $\langle x_1 = 1, x_4 = 0, x_5 = 1, \dots \rangle$ mais ne termine pas sur $\langle x_1 = 1, x_4 = 1, x_5 = 1, \dots \rangle$.

```

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

Paradigmes

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

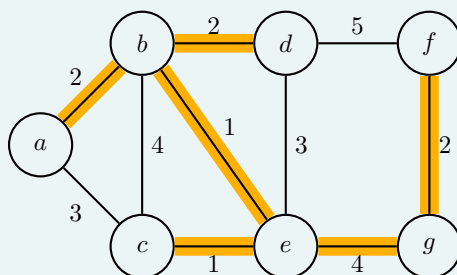
Un *graphe pondéré* est un graphe $\mathcal{G} = (V, E)$ dont chaque arête $e \in E$ est étiquetée par un poids $p[e]$, souvent un nombre entier, naturel ou rationnel. Le *poids* d'un graphe correspond à

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

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.

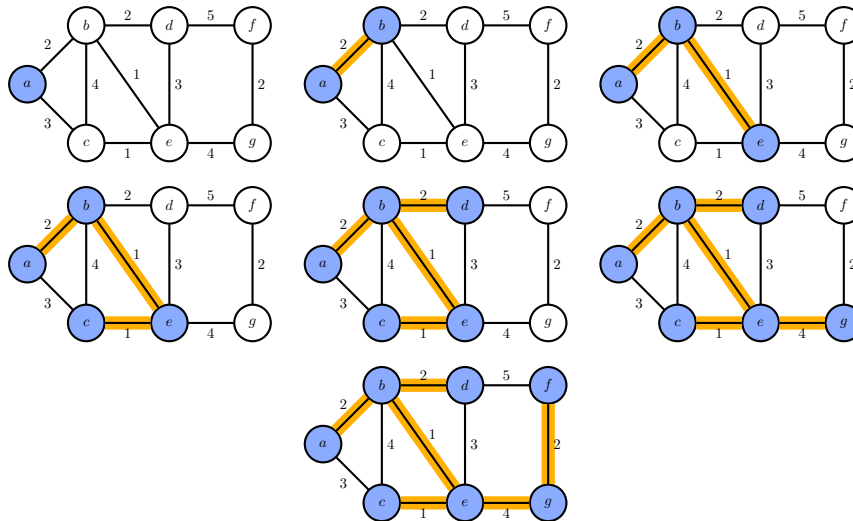


FIGURE 4.1 – Arbre couvrant minimal obtenu par l’algorithme de Prim–Jarník.

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 21.

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 21 : 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** $\text{candidats} \leftarrow [e \in E : v \in E]$ 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 $\mathcal{O}(|E| \log |V|)$.

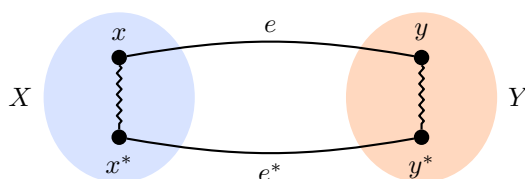
Démontrons maintenant que l'algorithme de Prim–Jarník calcule bel et bien un arbre couvrant minimal.

Théorème 3. *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 (G, p) . Il est assez simple de démontrer que E' forme un arbre couvrant $\mathcal{T} := (V, E')$. Montrons qu'il s'agit d'un arbre couvrant *minimal*.

Soit $E^* \subseteq E$ un ensemble d'arêtes tel que $\mathcal{T}^* := (V, E^*)$ 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 différents. 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 $Y := V \setminus X$ 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 $\mathcal{T}^{**} := (V, E^* \setminus \{e^*\} \cup \{e\})$. 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. \square

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 1. *L'algorithme 21 peut identifier tous les arbres couvrants minimaux.*

4.1.2 Algorithme de Kruskal

L'algorithme de Kruskal suit l'approche suivante:

- on débute avec $E' := \emptyset$ 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.

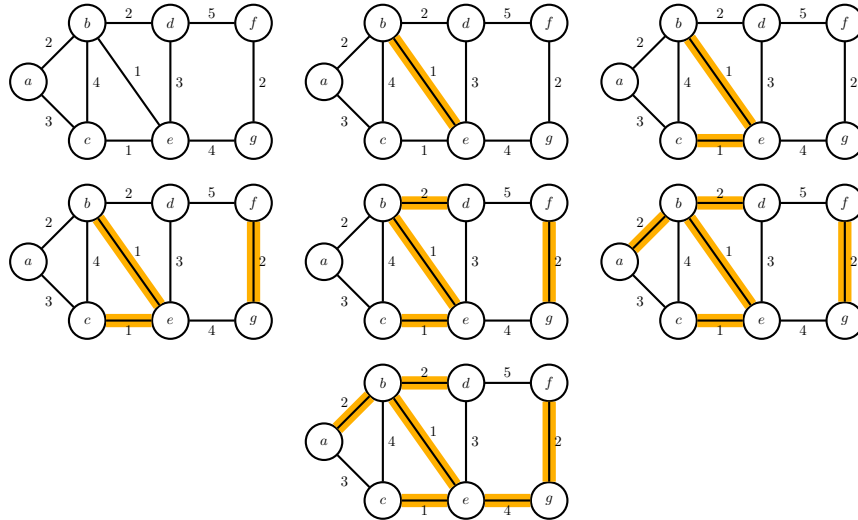


FIGURE 4.2 – Arbre couvrant minimal obtenu par l’algorithme de Kruskal.

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 22 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 4. *L’algorithme de Kruskal est correct. De plus, il peut identifier tous les arbres couvrants minimaux.*

Algorithme 22 : 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:

$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$
$\{a\}$	$\{b, e\}$	$\{c\}$	$\{d\}$	$\{f\}$	$\{g\}$	
$\{a\}$	$\{b, c, e\}$	$\{d\}$	$\{f\}$	$\{g\}$		
$\{a\}$	$\{b, c, e\}$	$\{d\}$	$\{f, g\}$			
$\{a\}$	$\{b, c, e, d\}$	$\{f, g\}$				
$\{a, b, c, e, d\}$	$\{f, g\}$					
$\{a, b, c, e, d, f, g\}$						

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 $\{a\}, \{b, c, e, d\}, \{f, g\}$.

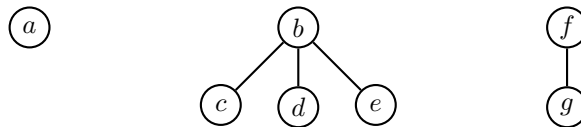


FIGURE 4.3 – Représentation de $\{a\}, \{b, c, e, d\}, \{f, g\}$ sous arborescences.

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 adjoit l'arbre de plus petite hauteur à l'autre arbre. Une telle implémentation est décrite sous forme de pseudocode à l'algorithme 23.



Algorithme 23 : 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 contiennent¹. 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 24.

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, par induction généralisée sur k , que chaque arborescence de $k \in \mathbb{N}_{\geq 1}$ sommets possède une hauteur d'au plus $\lceil \log k \rceil$.

Algorithme 24 : 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}

$E' \leftarrow \emptyset$

$D \leftarrow \text{init}(V)$

pour $\{u, v\} \in E$ en ordre croissant selon p
 | **si** $D.\text{trouver}(u) \neq D.\text{trouver}(v)$ **alors**
 | | $D.\text{union}(u, v)$
 | | **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 25.

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 25 : Recherche avec compression de chemin.

$\text{trouver}(x)$:

| $r \leftarrow x$

| // Remonter l'arborescence jusqu'à sa racine

| **tant que** $r \neq \text{parent}[r]$

| | $r \leftarrow \text{parent}[r]$

| // Compression du chemin de x vers r

| **tant que** $x \neq r$

| | $x, \text{parent}[x] \leftarrow \text{parent}[x], r$

| **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 26.

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

Composante	Prim–Jarník	Kruskal
« candidats »	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 26 : 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 Exercices

- 4.1) Expliquez comment calculer un arbre couvrant de poids *maximal*.
- 4.2) Supposons 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.
- 4.3) 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?
- 4.4) Donnez un algorithme simple qui calcule les composantes connexes d'un graphe non dirigé grâce à une structure d'ensembles disjoints.
- 4.5) 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.

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

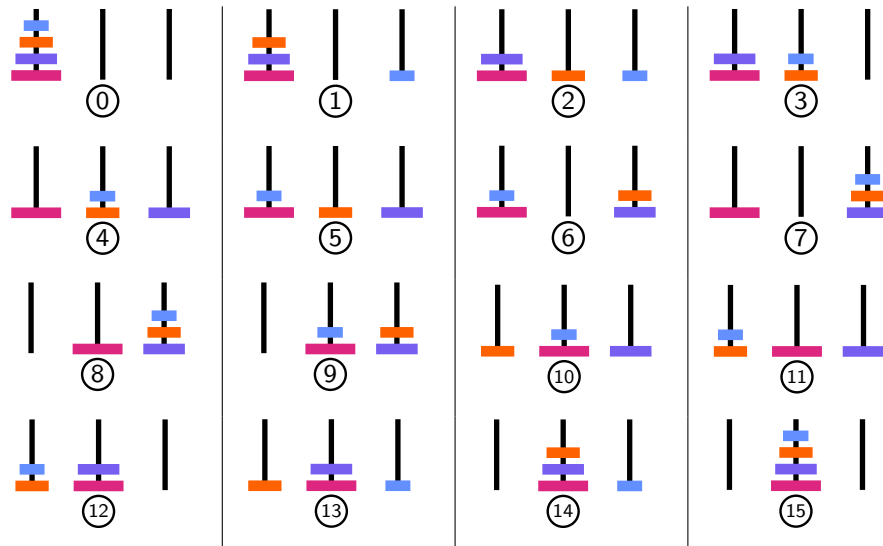
Considérons le problème classique des *tours de Hanoï*:

- n disques de diamètre $n, \dots, 2, 1$ 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'entière 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 27.



Algorithme 27 : 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( $n$ ):
  déplacements  $\leftarrow []$ 
  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 déplacements
      // 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 déplacements
  
```

Cherchons à déterminer la taille de la solution calculée par l'algorithme 27 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 substituant la récurrence à répétition:

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

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 $11 \dots 1_2 = 2^k - 1$.

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



Algorithme 28 : 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 []

si $n = 2$ **alors**

retourner [ ,  , ]

sinon

$P \leftarrow$ pavages($n - 1$)

$Q \leftarrow$ pavages($n - 2$)

retourner [ + $p : p \in P$] + [ + $q : q \in Q$] + [ + $q : q \in Q$]

5.2.1 Cas homogène

Afin d'estimer le temps d'exécution de l'algorithme 28, 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.

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

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

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 $\lambda_1, \lambda_2, \dots, \lambda_d$ du polynôme caractéristique. Si toutes les racines sont distinctes¹, la récurrence possède cette forme close:

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

où c_1, c_2, \dots, 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, \dots, 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. 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:

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

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 $\lambda_1 := (1 + \sqrt{5})/2 \approx 1,618$ et $\lambda_2 := (1 - \sqrt{5})/2 = 1 - \lambda_1 \approx -0,618$. 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:

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

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 27 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 $a_0 \cdot t(n) + a_1 \cdot t(n - 1) + \dots + a_d \cdot t(n - d) = c \cdot b^n$, 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 :


$$\begin{aligned} 0 &= c_1 + c_2, \\ 2 &= 2 \cdot c_1 + c_2. \end{aligned}$$

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 donné $b, n \in \mathbb{N}$. Une approche simple, décrite à l'algorithme 29, 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 29 : Exponentiation naïve.

Entrées : $b, n \in \mathbb{N}$ **Résultat** : b^n exp(b, n):| **si** $n = 0$ **alors**| | **retourner** 1| **sinon**| | **retourner** $b \cdot \text{exp}(b, n - 1)$

Algorithme 30 : Exponentiation rapide.

Entrées : $b, n \in \mathbb{N}$ **Résultat** : b^n exp-rapide(b, n):| **si** $n = 0$ **alors**| | **retourner** 1| **sinon**| | $m \leftarrow \text{exp-rapide}(b, n \div 2)$ | | $k \leftarrow 1$ | | **si** n est impair **alors** $k \leftarrow b$ | | **retourner** $m \cdot m \cdot k$

Cette procédure est décrite à l'algorithme 30. Sa correction découle du fait que l'ordre des multiplications n'a aucune importance; ou en termes plus techniques, par la commutativité et 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:

$$\begin{aligned}
 t(n) &= t(n \div 2) + 2 && \text{(par définition de } t) \\
 &\leq 2 \log(n \div 2) + 4 && \text{(par hypothèse d'induction)} \\
 &\leq 2 \log(n/2) + 4 && \text{(car } n \div 2 \leq n/2) \\
 &= 2 \log n - 2 \log 2 + 4 \\
 &= 2 \log n + 2. && \square
 \end{aligned}$$

L'algorithme 30 effectue donc $t \in \Theta(\log n)$ multiplications, ce qui offre un gain exponentiel par rapport à l'algorithme 29.

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 « cœur 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 31.



Algorithme 31 : 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
     $k \leftarrow \lceil n/2 \rceil$ 
     $a, b \leftarrow x \div 10^k, x \bmod 10^k$ 
     $c, d \leftarrow y \div 10^k, y \bmod 10^k$ 
     $e \leftarrow \text{mult}(k, a, c)$ 
     $f \leftarrow \text{mult}(k, b, d)$ 
     $g \leftarrow \text{mult}(k, a - b, c - d)$ 
    retourner  $10^{2k} \cdot e + 10^k \cdot (e + f - g) + f$ 

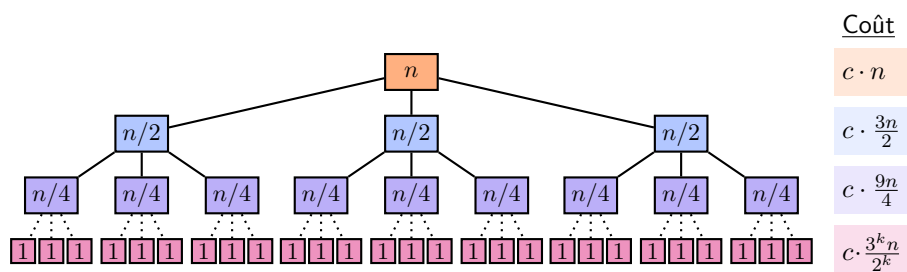
```

Analysons l'algorithme 31 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 modulus 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:

$$\begin{aligned}
 t(n) &\approx cn \cdot \sum_{i=0}^k \left(\frac{3}{2}\right)^i \\
 &= cn \cdot \frac{(3/2)^{k+1} - 1}{3/2 - 1} \quad (\text{série géométrique de raison } 3/2) \\
 &\leq cn \cdot \frac{(3/2)^{k+1}}{3/2 - 1} \\
 &= 3cn \cdot (3/2)^k \\
 &= 3cn \cdot (3/2)^{\log n} \quad (\text{par définition de } k) \\
 &= 3cn \cdot n^{\log(3/2)} \\
 &= 3c \cdot n^{1+\log(3/2)} \\
 &= 3c \cdot n^{\log 3} \quad (\text{car } 1 + \log(3/2) = \log(2) + \log(3/2) = \log(2 \cdot 3/2)).
 \end{aligned}$$

Nous nous sommes donc convaincus semi-formellement que le temps d'exécution de l'algorithme 31 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).

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 planchers² et plutôt considérer

$$t(n) = c \cdot t(n \div b) + f(n) \text{ où } c := a + a'.$$

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 5. Soient $t, f \in \mathcal{F}$, $b \in \mathbb{N}_{\geq 2}$ et $c, d \in \mathbb{R}_{>0}$ telles que $f \in \mathcal{O}(n^d)$ 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 $t \in \mathcal{O}(\log n)$.

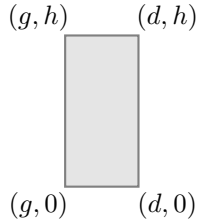
Multiplication rapide. Nous avons déjà établi que l'algorithme de Karatsuba donne lieu à la récurrence $t(n) = 3 \cdot t(n \div 2) + f(n)$ où $f \in \mathcal{O}(n)$. Nous avons $b = 2$, $c = 3$ et $d = 1$. Le troisième cas du théorème maître s'applique car $c = 3 > 2 = b^d$. Nous obtenons donc $t \in \mathcal{O}(n^{\log 3})$.

². 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.

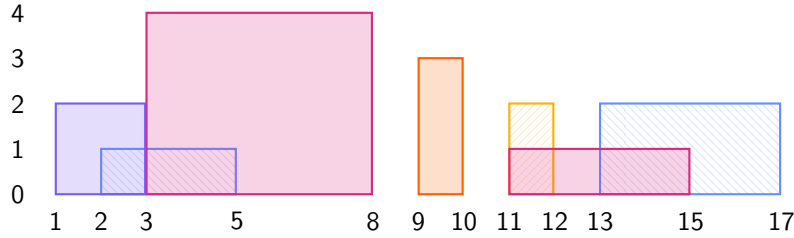
Un *bloc* est un triplet $(g, h, d) \in \mathbb{N}_{\geq 1}^3$ qui décrit ce rectangle dans le plan:



Un *paysage* est une séquence de blocs. Par exemple, le paysage

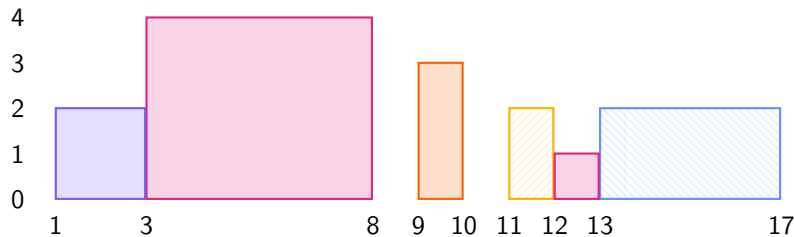
$$[(1, 2, 3), (2, 1, 5), (3, 4, 8), (9, 3, 10), (11, 2, 12), (11, 1, 15), (13, 2, 17)]$$

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:



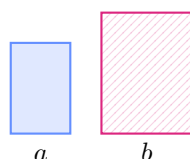
Nous présentons un algorithme qui découpe un paysage en blocs disjoints à la manière du tri par fusion:

- 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);

des blocs (comme pour le tri par fusion); on les ajoute simplement à z . L'algorithme 32 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

si $gauche(c) \neq droite(c)$ **alors empiler** c **sur** w .

Autrement dit, on se débarrasse des blocs dégénérés avec une surface de 0.

Analysons le temps d'exécution t de l'algorithme 32. 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 donc $t \in \mathcal{O}(n \log n)$.

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:

$$d_0 \cdot n^0 \cdot \lambda^n + d_1 \cdot n^1 \cdot \lambda^n + \dots + d_{k-1} \cdot n^{k-1} \cdot \lambda^n.$$

Par exemple, considérons une récurrence t dont le polynôme est t 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 $s(k) := t(2^k)$. 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:

$$\begin{aligned} s(0) &= 3 = c_1 \\ s(1) &= 8 = 2c_1 + 2c_2 \end{aligned}$$

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 $b^n \bmod m$ 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
     $k \leftarrow \lceil n/2 \rceil$ 
     $a, b \leftarrow x \div 10^k, x \bmod 10^k$ 
     $c, d \leftarrow y \div 10^k, y \bmod 10^k$ 
     $e \leftarrow \text{mult}(k, a, c)$ 
     $f \leftarrow \text{mult}(k, a, d)$ 
     $g \leftarrow \text{mult}(k, b, c)$ 
     $h \leftarrow \text{mult}(k, b, d)$ 
    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:

$$\begin{array}{c}
 \mathbf{A}[i-1, j] \\
 \wedge \\
 \mathbf{A}[i, j-1] < \mathbf{A}[i, j] > \mathbf{A}[i, j+1] \\
 \vee \\
 \mathbf{A}[i+1, j]
 \end{array}$$

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

$$s[i] \leq s[i+1] \leq \dots \leq s[n] \leq \dots \leq s[i-1].$$

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:

$$\max\{s[i] + \dots + s[j] : 1 \leq i \leq j \leq n\}.$$

Par exemple, votre algorithme doit retourner 9 sur entrée

$$s = [3, 1, -5, 4, -2, 1, 6, -3].$$

- 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.
- 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?
 - Donnez un algorithme qui identifie la position où se placer pour être élu.
 - ★ Montrez que $t(n) = 2 \cdot (n - 2^{\lceil \log n \rceil}) + 1$ par induction généralisée.
 - ★ 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

$\text{exp}'(b, n)$:

```

si  $n = 0$  alors
  | retourner 1
sinon si  $n = 1$  alors
  | retourner  $b$ 
sinon
  |  $a \leftarrow n \div 2$ 
  |  $a' \leftarrow n - a$ 
  | retourner  $\text{exp}'(b, a) \cdot \text{exp}'(b, a')$ 

```

- 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 $\oplus: S \times S \rightarrow S$, 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 $b^0 := e$ et $b^n := b^{n-1} \oplus b$ 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) $S := \mathbb{N}$, $e := 1$ et $a \oplus b := a \cdot b$;
 - (ii) $S := \mathbb{N}$, $e := 0$ et $a \oplus b := a + b$;
 - (iii) $S := \{0, 1, \dots, m-1\}$, $e := 0$ et $a \oplus b := (a + b) \bmod m$;
 - (iv) $S := \{0, 1, \dots, m-1\}$, $e := 1$ et $a \oplus b := (a \cdot b) \bmod m$;
 - (v) $S := \mathbb{N}^{k \times k}$, $e := \mathbf{I}$ et $\mathbf{A} \oplus \mathbf{B} := \mathbf{AB}$.

Une ancienne version de l'exercice supposait que \oplus était commutative. Merci à Christopher 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:

$$t(n) := \begin{cases} 0 & \text{si } n = 0, \\ c \cdot t(n \div b) + n^d & \text{sinon.} \end{cases}$$

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.

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 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 possibilités.

Ces matrices sont éparées, 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:

	1	2	3	4	5	6	7	8
1	X							
2					X			
3								X
4						X		
5			X					
6							X	
7		X						
8				X				

Peu des $\binom{n^2}{n}$ 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 33 : 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 débiter 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 33. Cette stratégie générale se nomme *retour arrière* puisqu’on revient sur nos décisions lorsqu’on n’indentifie aucune solution.



L’algorithme 34 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 → nord-est » est caractérisée par $i + j$, alors que la diagonale « nord-ouest → 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 34 : 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

Considérons maintenant le *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 $\{1, 2, \dots, c\}$. 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 à 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.

S'il était permis de « découper » les objets, on pourrait utiliser un algorithme glouton simple afin d'identifier la solution optimale. En effet, il suffirait de considérer les objets en ordre décroissant du ratio entre leur valeur et leur poids, et d'ajouter 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 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 $250 + (10/13) \cdot 845 = 900$.

Toutefois, l'approche gloutonne ne fonctionne pas en général comme on ne peut pas découper les objets; elle donne simplement des bornes sur la solution optimale. Dans l'exemple précédent, nous apprenons que la solution optimale est d'au moins 30 et qu'il est impossible de faire mieux que 80.

Nous pouvons plutôt résoudre le problème à l'aide d'un algorithme de force brute qui explore les 2^n possibilités. L'algorithme 35 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 35 : 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  $poids \leq c$  alors retourner  $valeur$ 
            sinon retourner 0
        sinon
             $valeur' \leftarrow valeur + v[i]$ 
             $poids' \leftarrow poids + p[i]$ 
            retourner max(remplir( $i + 1, valeur, poids$ ),
                          remplir( $i + 1, valeur', poids'$ ))
    retourner remplir(1, 0, 0)

```

L'algorithme 35 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 36 décrit cette modification.



Reconsidérons l'exemple précédent, c'est-à-dire l'instance:

$$\begin{aligned}
 v &= [50, 5, 65, 10, 12, 20], \\
 p &= [700, 320, 845, 70, 420, 180], \\
 c &= 900.
 \end{aligned}$$

L'algorithme 35 teste $2^6 = 64$ combinaisons, alors que l'algorithme 36 n'en teste que 18.

1. On parle de « *pruning* » en anglais.

Algorithme 36 : 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
       $valeur' \leftarrow valeur + v[i]$ 
       $poids' \leftarrow poids + p[i]$ 
       $sol \leftarrow$  remplir( $i + 1, valeur, poids$ ) // sans objet  $i$ 
      si  $poids' \leq c$  alors // avec objet  $i$ 
        |  $sol \leftarrow \max(sol, remplir(i + 1, valeur', poids'))$ 
      retourner  $sol$ 
  retourner remplir(1, 0, 0)

```

Nous pouvons améliorer l'algorithme 36 à l'aide d'un élagage plus agressif:

- on stocke la meilleure solution identifiée jusqu'ici;
- 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'excéderait pas la meilleure solution découverte, alors il est inutile de poursuivre.

Afin d'amorcer cette procédure, nous pourrions considérer $-\infty$ comme la meilleure solution connue. Cependant, nous pouvons faire mieux en débutant par une solution prometteuse. Par exemple, nous pouvons débiter avec la valeur du meilleur objet ou avec la solution obtenue par l'algorithme glouton (sans découper le dernier objet choisi). L'algorithme 37 implémente cette procédure.

Par exemple, considérons cette instance:

$$v = [1, 2, \dots, 200],$$

$$p = [200, \dots, 2, 1],$$

$$c = 150.$$

L'algorithme 36 effectue 605 701 807 appels récursifs et explore 278 031 704 assignations complètes, alors que l'algorithme 37 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.



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 38 présente cette procédure sous forme de pseudocode.



Algorithme 37 : 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  $\leftarrow$  sol-prometteuse( $v, p, c$ )
  potentiel  $\leftarrow [v[i] + \dots + v[n] : i \in [1..n]]$ 
  remplir( $i, valeur, poids$ ):
    meilleure  $\leftarrow$  max( $meilleure, valeur$ )
    si ( $i < n$ )  $\wedge$  ( $valeur + potentiel[i] > meilleure$ ) alors
      valeur'  $\leftarrow$  valeur +  $v[i]$ 
      poids'  $\leftarrow$  poids +  $p[i]$ 
      remplir( $i + 1, valeur, poids$ )           // sans objet  $i$ 
      si poids'  $\leq c$  alors                     // avec objet  $i$ 
        remplir( $i + 1, valeur', poids'$ )
  remplir(1, 0, 0)
  retourner meilleure

```

Algorithme 38 : 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  $\infty$ 
    sinon
      |  $sans \leftarrow aux(k, num, i + 1)$            // sol. sans  $s[i]$ 
      |  $avec \leftarrow \infty$ 
      | si  $k \geq s[i]$  alors
      | |  $avec \leftarrow aux(k - s[i], num + 1, i)$  // 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 brute². 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 $x = \text{vrai}$, $y = \text{vrai}$ et $z = \text{faux}$. 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:

$$\bigwedge_{1 \leq i \leq n} \bigvee_{1 \leq j \leq n} x_{i,j}$$

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 $n = 4$ 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 $x \in \mathbb{N}^n$ qui maximise ou minimise une *fonction objectif* linéaire sujet à des contraintes linéaires de cette forme:

$$\begin{aligned} &\text{optimiser} && c^T \cdot x, \\ &\text{sujet à} && A \cdot x \leq b \text{ et } x \in \mathbb{N}^n. \end{aligned}$$

Par exemple, l'instance $m = 10$ et $s = [1, 5, 7]$ du problème du retour de monnaie s'exprime par:

$$\text{minimiser } x + y + z \text{ sujet à } x + 5y + 7z = 10.$$

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:

$$\begin{aligned} &\text{maximiser} && 50 x_1 + 5 x_2 + 65 x_3 + 10 x_4 + 12 x_5 + 20 x_6 \\ &\text{sujet à} && 700 x_1 + 320 x_2 + 845 x_3 + 70 x_4 + 420 x_5 + 180 x_6 \leq 900. \end{aligned}$$

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 $\text{dist}(u, v)$, 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: $\text{dist}(ab, ac) = 1$, $\text{dist}(abc, ba) = 2$ et $\text{dist}(\varepsilon, ab) = 2$. Donnez un algorithme de force brute qui calcule la distance entre deux chaînes données. Pensez d'abord à une borne supérieure sur $\text{dist}(u, v)$.
- 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 4.5).
- 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, \dots, p_k tels que $m = p_1 \cdots p_k$

$p \leftarrow []$; $d \leftarrow 2$

tant que $m > 1$

si $m \bmod d = 0$ **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.

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 39 de calcul de la suite de Fibonacci forme un exemple classique de ce phénomène.

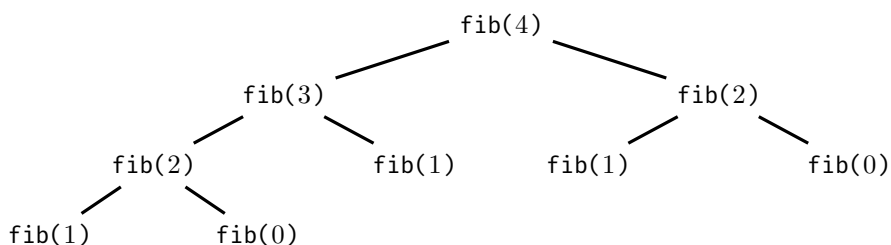
Algorithme 39 : Calcul d'un élément de la suite de Fibonacci.

Entrées : $n \in \mathbb{N}$

Résultat : $n^{\text{ème}}$ terme de la suite de Fibonacci

```
fib(n):  
    | si  $n \leq 1$  alors  
    |   | retourner  $n$   
    | sinon  
    |   | retourner  $\text{fib}(n-1) + \text{fib}(n-2)$ 
```

Par exemple, l'arbre des appels récursifs de $\text{fib}(4)$ montre que les valeurs de $\text{fib}(2)$, $\text{fib}(1)$ et $\text{fib}(0)$ 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 40 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 40 : 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 41 : 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  $\infty$ 
// Aucun pièce pour rendre 0
 $T[0, 0] \leftarrow 0$ 
// Calculer # pièces progressivement
pour  $i \in [1..n]$ 
    pour  $j \in [0..m]$ 
         $sans \leftarrow T[i - 1, j]$  // sol. sans  $s[i]$ 
         $avec \leftarrow \infty$ 
        si  $j \geq s[i]$  alors  $avec \leftarrow T[i, j - s[i]] + 1$  // sol. avec  $s[i]$ 
         $T[i, j] \leftarrow \min(sans, avec)$ 
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 41 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:

$$\begin{aligned} v &:= [5, 3, 4] \text{ (valeurs),} \\ p &:= [4, 2, 3] \text{ (poids),} \\ c &:= 8 \text{ (capacité).} \end{aligned}$$

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: $T[3, 8] = 9$.

Algorithme 42 : 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]

```

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 42 implémente cette procédure en remplissant une ligne à la fois.



7.3 Plus courts chemins

Soit \mathcal{G} un graphe pondéré par une séquence de poids entiers p . Le poids d'un chemin $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \dots \xrightarrow{e_n} v_n$ correspond à $p[e_1] + p[e_2] + \dots + p[e_n]$. Autrement dit, le poids d'un chemin allant de v_0 vers v_n , en empruntant les arêtes e_1, e_2, \dots, 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: $a \rightarrow b \rightarrow d \rightarrow e$ et $a \rightarrow b \rightarrow e$ 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*.

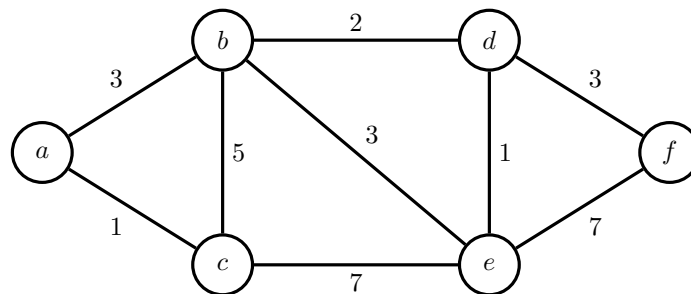


FIGURE 7.1 – Exemple de graphe (non dirigé) pondéré.

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 43 décrit cette procédure sous forme de pseudocode de haut niveau.



Algorithme 43 : 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 $v : u \rightarrow v$

$d[v] \leftarrow \min(d[v], d[u] + p[u, v])$

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 43 ne construit pas les plus courts chemins. Afin de les construire, on peut stocker le

prédécesseur 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 prédécesseur 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 prédécesseurs.

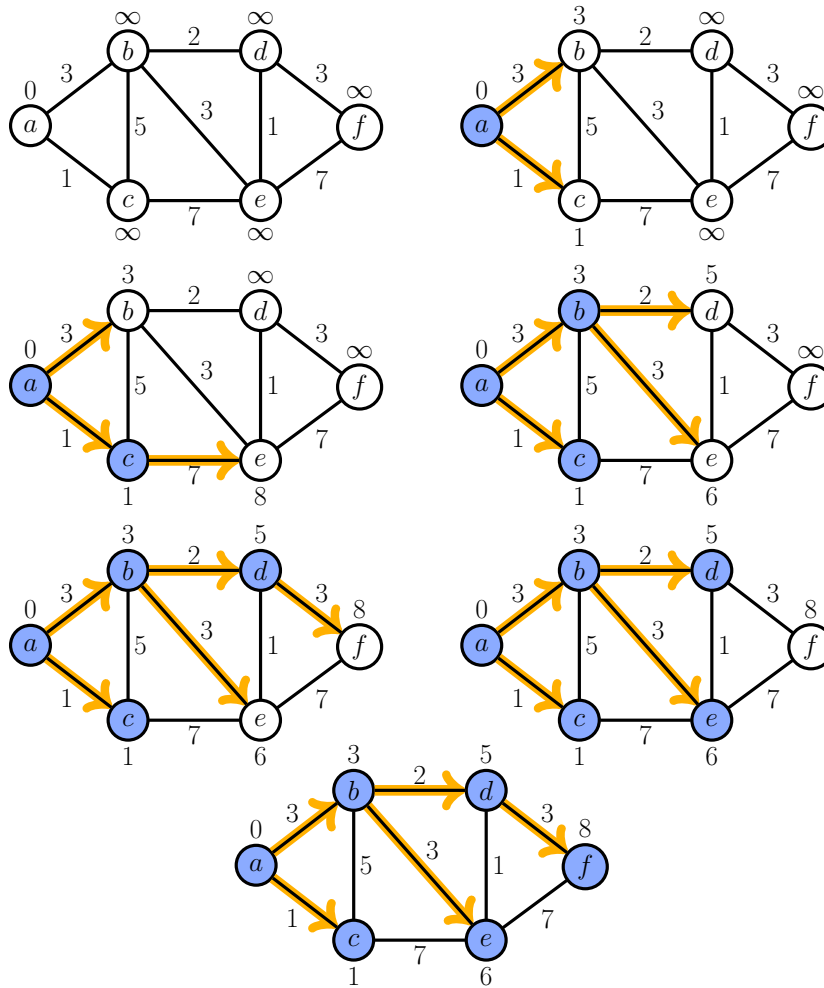


FIGURE 7.2 – Exemple d'exécution de l'algorithme de Dijkstra.

Correction. On peut montrer que l'algorithme de Dijkstra est correct en démontrant la proposition suivante par induction:

Proposition 22. 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 43 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:

$$\mathcal{O}\left(|V| \cdot |V| + \sum_{u \in V} \deg^+(u)\right) = \mathcal{O}(|V|^2 + |E|).$$

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 $3 + 2 - 2 - 2 = 1$.

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,

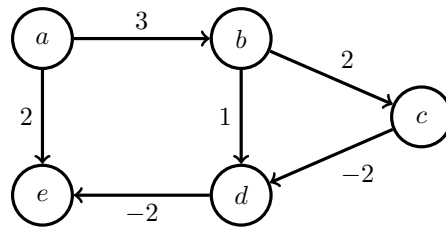


FIGURE 7.3 – Graphe avec poids négatifs sur lequel l’algo. de Dijkstra échoue.

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 44 : Algorithme de Floyd-Warshall.

Entrées : graphe $\mathcal{G} = (V, E)$ par une séquence p de poids entiers (sans cycle négatif), où $V = \{v_1, v_2, \dots, v_n\}$

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:

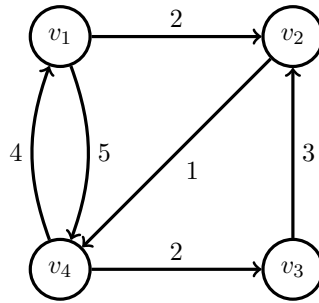


FIGURE 7.4 – Exemple de graphe (dirigé) pondéré.

—	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 44 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 $\{v_1, v_2, \dots, v_k\}$. Nous avons:

Proposition 23. *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

$$d'[v_i, v_k] = d[v_i, v_k] \quad \forall i \in [1..n], \quad (7.1)$$

$$d'[v_k, v_j] = d[v_k, v_j] \quad \forall j \in [1..n]. \quad (7.2)$$

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 $\{v_1, v_2, \dots, v_k\}$. 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:

$$\delta_k(v_i, v_j) = \min(\delta_{k-1}(v_i, v_j), \underbrace{\delta_{k-1}(v_i, v_k) + \delta_{k-1}(v_k, v_j)}_{\text{longueur du chemin qui passe par } v_k}). \quad (7.3)$$

Nous avons donc:

$$\begin{aligned} d'[v_i, v_j] &= \min(d[v_i, v_j], d[v_i, v_k] + d[v_k, v_j]) && \text{(par déf., (7.1) et (7.2))} \\ &= \min(\delta_{k-1}(v_i, v_j), \delta_{k-1}(v_i, v_k) + \delta_{k-1}(v_k, v_j)) && \text{(par hyp. d'ind.)} \\ &= \delta_k(v_i, v_j) && \text{(par (7.3)).} \quad \square \end{aligned}$$

Corollaire 2. *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 23, 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 . \square

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'accessibilité¹ $\xrightarrow{*}$ d'un graphe (non pondéré), en remplaçant:

- le poids $p[u, v]$ par vrai si $u = v$ ou $u \rightarrow v$, 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 $d[u, v] = \text{vrai}$ ssi $u \xrightarrow{*} v$. Cette modification est décrite à l'algorithme 45.

Algorithme 45 : Algorithme de calcul de relation d'accessibilité.

Entrées : graphe $\mathcal{G} = (V, E)$ où $V = \{v_1, v_2, \dots, v_n\}$

Résultat : matrice d t.q. $d[u, v]$ indique si $u \xrightarrow{*} v$

$d \leftarrow [(u, v) \mapsto \text{faux} : u, v \in V]$

pour $v \in V$ // chemins vides

 | $d[v, v] \leftarrow \text{vrai}$

pour $(u, v) \in E$ // chemins directs

 | $d[u, v] \leftarrow \text{vrai}$

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

Remarque.

L'algorithme 44 (avec poids) est attribué à *Robert W. Floyd*, alors que l'algorithme 45 (sans poids) est attribué à *Stephen Warshall*.

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*.

1. Aussi connue sous le nom de *clôture réflexive transitive* de la relation \rightarrow .

Algorithme 46 : 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 fois**

pour chaque arête $u \rightarrow v$
 $d[v] \leftarrow \min(d[v], d[u] + p[u, v])$

retourner d

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 46 décrit cette procédure.

L'identification des chemins requiert simplement l'ajout d'une séquence de prédecesseurs comme pour les autres algorithmes. La figure 7.5 donne une trace de l'exécution de l'algorithme de Bellman-Ford sur le graphe de la figure 7.3, incluant la construction des prédecesseurs.

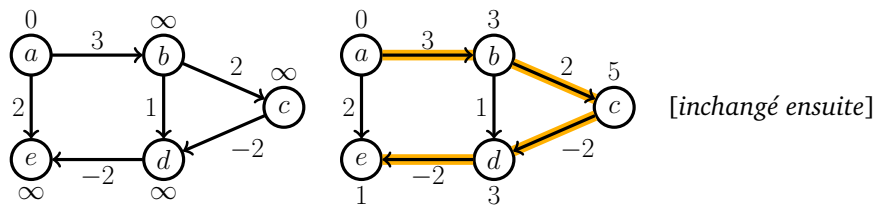


FIGURE 7.5 – Exemple d'exécution de l'algorithme de Bellman-Ford.

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 24. 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  $u \rightarrow v$ 
  | 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 41 appartient à $\mathcal{O}(m \cdot n)$. Adaptez l'algorithme afin de la réduire à $\mathcal{O}(m)$.
- 7.2) L'algorithme 41 et l'algorithme 42 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 $\overset{+}{\rightarrow}$ d'un graphe? Cette relation est définie par:

$$u \overset{+}{\rightarrow} v \stackrel{\text{déf}}{\iff} \text{il existe un chemin } \textit{non vide} \text{ de } u \text{ vers } v.$$

- 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 $u \rightarrow v$ 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 $a \rightarrow b \rightarrow e$ possède moins d'arêtes que le chemin $a \rightarrow b \rightarrow d \rightarrow e$, 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 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}

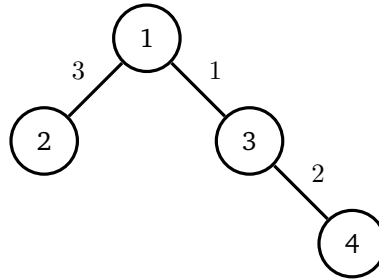
Pour le résoudre, on pourrait être tenté de multiplier chaque poids $p[e]$

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

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 47, 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 47 : 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** $4 \cdot y_2 + 2 \cdot y_1 + y_0$

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 48.

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 48 : 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):

```

   $k \leftarrow \lceil \log c \rceil$ 
  faire
     $x \leftarrow 0$ 
    pour  $i \in [0..k - 1]$ 
      choisir un bit  $y$  à pile ou face
       $x \leftarrow x + 2^i \cdot y$ 
  tant que  $x \geq c$ 
  retourner  $x$ 

```

de quitter la boucle pour une itération donnée est de $p := c/2^k$. Ainsi:

$$\begin{aligned}
 \mathbb{E}[X] &= 1/p && \text{(car } X \text{ suit une loi géométrique de paramètre } p) \\
 &= 2^k/c && \text{(par définition de } p) \\
 &= 2^{\lceil \log c \rceil} / 2^{\log c} && \text{(par définition de } k \text{ et par } c = 2^{\log c}) \\
 &= 2^{\lceil \log c \rceil - \log c} \\
 &< 2 && \text{(car } \lceil \log c \rceil < 1 + \log c).
 \end{aligned}$$

Puisque chaque itération effectuée 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 49. Celui-ci:

- choisit un élément aléatoire $s[i]$;
- retourne $\max(s[i], s[1])$ si $s[i] \neq s[1]$, et recommence sinon.



Algorithme 49 : 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  $i \in [1..n]$  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 $t_{\text{esp}}: \mathbb{N} \rightarrow \mathbb{N}$ telle que:

$$t_{\text{esp}}(n) := \max \{ \mathbb{E}[Y_x] : \text{entrée } x \text{ de taille } n \}.$$

Soit X_s la variable aléatoire qui dénote le nombre d'itérations effectuées par la boucle principale de l'algorithme 49. À une itération donnée, la probabilité de choisir $s[i] \neq s[1]$ est de $p := 1/2$, 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 49 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 50 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 50 : 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  $i \in [1..|s|]$  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 $\text{err} : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ 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 50. 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 relie X et Y . Plus formellement:

$$\text{taille}(X, Y) := |\{\{x, y\} \in E : x \in X, y \in Y\}|.$$

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 concluons 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.

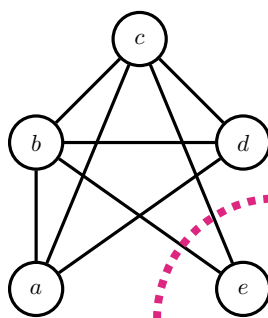


FIGURE 8.1 – Exemple de coupe minimum identifiée par un trait tireté.

L'*algorithme de Karger* est un algorithme de Monte Carlo qui identifie une coupe minimum en:

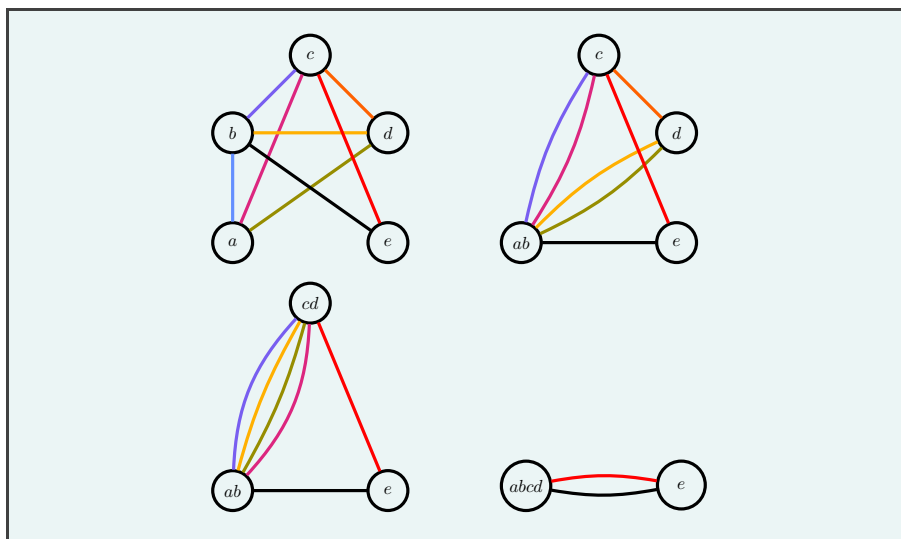
- choisissant une arête $e = \{u, v\}$ aléatoirement de façon uniforme;
- contractant e par la fusion de u et v ;
- répétant tant qu'il existe plus de deux sommets.

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 51.



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 51 : 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  $\{u, v\} \in E$  aléatoirement de façon uniforme
    retirer  $u$  et  $v$  de  $V$ 
    ajouter  $uv$  à  $V$ 
    pour  $\{x, y\} \in E$ 
        retirer  $\{x, y\}$  de  $E$ 
        si  $x \in \{u, v\}$  alors  $x \leftarrow uv$ 
        si  $y \in \{u, v\}$  alors  $y \leftarrow uv$ 
        si  $x \neq y$  alors ajouter  $\{x, y\}$  à  $E$ 
    retourner  $|E|$ 
    
```

Proposition 25. 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 :

$$\begin{aligned} |V| \cdot k &= \overbrace{k + k + \dots + k}^{|V| \text{ fois}} \\ &\leq \sum_{v \in V} \deg(v) \\ &= 2|E|. \end{aligned}$$

En divisant les deux côtés par $|V|$, nous obtenons $k \leq 2|E|/|V|$. □

Théorème 6. *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 25. 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 6 affirme que la probabilité d'erreur de l'algorithme de Karger est d'au plus $q := 1 - 1/|V|^2$. 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 :

$$\begin{aligned} q^k &\leq (1 - 1/|V|^2)^k \\ &\leq \left(2^{-1/|V|^2}\right)^k && \text{(car } 1 - x \leq 2^{-x} \text{ pour tout } x \in \mathbb{R}_{\geq 0}) \\ &= 2^{-k/|V|^2}. \end{aligned}$$

Ainsi, en prenant $k := \varepsilon \cdot |V|^2$, 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 $t_{\text{moy}}: \mathbb{N} \rightarrow \mathbb{N}$ telle que:

$$t_{\text{moy}}(n) := \sum_{\substack{\text{entrée } x \\ \text{de taille } n}} f(x) / (\text{nombre d'entrées de taille } n).$$

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, nous analysons le temps moyen du tri par insertion:

Proposition 26. *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 17, 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 substituant f à répétition, nous obtenons:

$$\begin{aligned}
 f(n) &= n \cdot f(n-1) + n! \cdot (n-1)/2 \\
 &= n \cdot [(n-1) \cdot f(n-2) + (n-1)! \cdot (n-2)/2] + n! \cdot (n-1)/2 \\
 &= n \cdot (n-1) \cdot f(n-2) + n!/2 \cdot [(n-2) + (n-1)] \\
 &\quad \vdots \\
 &= n! \cdot f(0) + n!/2 \cdot \sum_{i=0}^{n-1} i \\
 &= n!/2 \cdot \sum_{i=1}^{n-1} i \\
 &= n!/2 \cdot n(n-1)/2 \\
 &= n! \cdot n(n-1)/4.
 \end{aligned}$$

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

$$\Theta(n + n(n-1)/4) = \Theta(n^2).$$

□

8.6 Exercices

- 8.1) Montrez que la distribution des nombres générés par l'algorithme 47 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 $\mathbf{v} \in \{0, 1\}^n$ de façon uniforme
retourner $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{v}) = \mathbf{C} \cdot \mathbf{v}$

Indice: pensez à la probabilité que $\mathbf{D} \cdot \mathbf{v} = \mathbf{0}$, 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 49 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 $i \in [2..|s|]$ 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 $i^{\text{ème}}$ 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 49 et 50 par rapport à ce problème.

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 $k := a + b$. Nous concluons donc que $m + n$ est impair. \square 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:

$$\begin{aligned}
 (n + 1)! &= 1 \cdot 2 \cdots n \cdot (n + 1) \\
 &> 2^n \cdot (n + 1) && \text{(par hypothèse d'induction)} \\
 &> 2^n \cdot 2 && \text{(car } n + 1 \geq 4 + 1 > 2) \\
 &= 2^{n+1}. && \square
 \end{aligned}$$

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:

$$\begin{aligned}
 \binom{n}{k} + \binom{n}{k+1} &= \frac{n!}{k! \cdot (n-k)!} + \frac{n!}{(k+1)! \cdot (n-k-1)!} \\
 &= n! \cdot \left(\frac{1}{k! \cdot (n-k)!} + \frac{1}{(k+1)! \cdot (n-k-1)!} \right) \\
 &= n! \cdot \left(\frac{(k+1)}{(k+1)! \cdot (n-k)!} + \frac{(n-k)}{(k+1)! \cdot (n-k)!} \right) \\
 &= n! \cdot \frac{(k+1) + (n-k)}{(k+1)! \cdot (n-k)!} \\
 &= n! \cdot \frac{n+1}{(k+1)! \cdot (n-k)!} \\
 &= \frac{(n+1)!}{(k+1)! \cdot (n-k)!} \\
 &= \binom{n+1}{k+1}. && \square
 \end{aligned}$$

- 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. \square
- 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.

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, $\mathcal{O}(f) \subseteq \mathcal{O}(g)$. \square

1.4)

$$\begin{aligned} \mathcal{O}(1000000) &\subset \mathcal{O}(8(n+2) - 1 + 9n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(5n^2 - n) \\ &= \mathcal{O}(3n^2) \subset \mathcal{O}(n^3 - n^2 + 7) \subset \mathcal{O}(2^n) \subset \mathcal{O}(4^n) \subset \mathcal{O}(n!). \end{aligned}$$

1.7) Nous avons:

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

Ainsi, par la règle de la limite, $n^2 \in \mathcal{O}(2^n)$ et $2^n \notin \mathcal{O}(n^2)$.

1.10) Soit $d \in \mathbb{N}$. Posons $f(n) := \sum_{i=1}^n i^d$. Montrons d'abord que $f \in \mathcal{O}(n^{d+1})$. Nous avons:

$$\begin{aligned} f(n) &= 1^d + 2^d + \dots + n^d \\ &\leq n^d + n^d + \dots + n^d && \text{(pour tout } n \geq 1) \\ &= n \cdot n^d \\ &= n^{d+1}. \end{aligned}$$

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 $f \in \Omega(n^{d+1})$. Nous avons:

$$\begin{aligned}
 f(n) &= \sum_{i=1}^n i^d \\
 &\geq \sum_{i=\lceil (n+1)/2 \rceil}^n i^d && \text{(on garde les termes } \geq \text{ à la médiane)} \\
 &\geq \sum_{i=\lceil (n+1)/2 \rceil}^n \lceil (n+1)/2 \rceil^d && \text{(car chaque } i \geq \lceil (n+1)/2 \rceil) \\
 &= \lceil n/2 \rceil \cdot \lceil (n+1)/2 \rceil^d \\
 &\geq (n/2) \cdot (n/2)^d \\
 &= (n/2)^{d+1} \\
 &= \frac{1}{2^{d+1}} \cdot n^{d+1}.
 \end{aligned}$$

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 $f \in \Theta(n^{d+1})$. \square

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 $n^d \in \mathcal{O}(n!)$. \square

1.12) Soit $d \in \mathbb{N}_{>0}$. Montrons d'abord que $\log n \in \mathcal{O}(\sqrt[d]{n})$. Pour tout $n \in \mathbb{N}_{\geq 1}$:

$$\begin{aligned}\log n &= \log((\sqrt[d]{n})^d) \\ &= d \cdot \log \sqrt[d]{n} \\ &\leq d \cdot \sqrt[d]{n} \quad (\text{car } \log x \leq x \text{ pour tout } x \in \mathbb{R}_{>0}).\end{aligned}$$

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 $n \in \mathbb{N}_{\geq 1}$:

$$\begin{aligned}(\log n)^d &\leq (d \cdot \sqrt[d]{n})^d \quad (\text{par l'observation ci-dessus}) \\ &= d^d \cdot n.\end{aligned}$$

Ainsi en prenant d^d comme constante multiplicative et 1 comme seuil, nous concluons que $(\log n)^d \in \mathcal{O}(n)$. \square

1.15) Nous avons:

$$\begin{aligned}f(m, n) &= \frac{mn}{2} + 3m \log(n \cdot 2^n) + 7n \\ &\leq mn + 3m \log(n \cdot 2^n) + 7n \\ &= mn + 3m \log(n) + 3m \log(2^n) + 7n \\ &= mn + 3m \log(n) + 3mn + 7n \\ &\leq mn + 3mn + 3mn + 7n \quad \text{pour tout } n \geq 1 \\ &\leq mn + 3mn + 3mn + mn \quad \text{pour tout } m \geq 7 \\ &= 8mn.\end{aligned}$$

Ainsi, nous concluons que $f \in \mathcal{O}(mn)$ en prenant $c := 8$ comme constante multiplicative, et $m_0 := 7$ et $n_0 := 1$ comme seuils. \square

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,

$$x_i = 3x_{i-1}/2 = 3^2x_{i-2}/2^2 = \dots = 3^i x_0 / 2^i = 3^i n / 2^i.$$

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 contredit la maximalité de k . \square

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 $s_n := [0, 0, \dots, 0]$ 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 $u_n := [-1, -1, \dots, -1]$ 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:

$$\begin{aligned}
 g(n) &= \sum_{i=1}^{n-2} \left(3 + \sum_{j=i+1}^{n-1} 9 \right) \\
 &\geq \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
 &= \sum_{i=1}^{n-2} ((n-1) - (i+1) + 1) \\
 &= \sum_{i=1}^{n-2} (n-i-1) \\
 &= \sum_{i=1}^{n-2} i \\
 &= (n-2)(n-1)/2 \\
 &= (n^2 - 3n + 2)/2 \\
 &\geq (n^2 - 3n)/2 \\
 &\geq (n^2 - n^2/2)/2 && \text{(pour tout } n \geq 6) \\
 &= (1/4) \cdot n^2.
 \end{aligned}$$

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 $t_{\max} \in \Theta(n^2)$.

Chapitre 2

2.2) Algorithme sur place *non stable*:



```

Entrées : séquence binaire  $s$ 
Sorties : séquence  $s$  triée
 $i \leftarrow 1$ ;  $j \leftarrow |s|$ 
tant que  $i < j$       // Invar.:  $s[1:i-1]$  ne contient que des 0
    si  $s[i] = 0$  alors // et  $s[j+1:n]$  ne contient que des 1
         $i \leftarrow i + 1$ 
    sinon
         $s[i] \leftrightarrow s[j]$ 
         $j \leftarrow j - 1$ 
retourner  $s$ 

```

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. Notons qu'il est **possible d'obtenir une meilleure constante multiplicative**.



2.12) On peut identifier le centre de la liste en avançant en alternance un pointeur d'une position et un autre de deux positions. Le reste de l'approche demeure la même:



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 : liste chaînée s d'éléments comparables

Sorties : liste chaînée correspondant à s triée

fusion(x, y):

```

si  $x = \perp$  alors retourner  $y$ 
sinon si  $y = \perp$  alors retourner  $x$ 
sinon
  // Identifier la première valeur
  si  $x.valeur \leq y.valeur$  alors
     $z \leftarrow x$ 
     $x \leftarrow x.succ$ 
  sinon
     $z \leftarrow y$ 
     $y \leftarrow y.succ$ 
  // Construire le reste de la liste
   $debut \leftarrow z$ 
  tant que  $x \neq \perp$  ou  $y \neq \perp$ 
    si  $(y = \perp)$  ou  $(x \neq \perp$  et  $x.valeur \leq y.valeur)$  alors
       $z.succ \leftarrow x$ 
       $x \leftarrow x.succ$ 
    sinon
       $z.succ \leftarrow y$ 
       $y \leftarrow y.succ$ 
     $z \leftarrow z.succ$ 
  retourner  $debut$ 

```

trier(s):

```

si  $s = \perp$  ou  $s.succ = \perp$  alors
  retourner  $s$ 
sinon
  // Trouver le centre
   $x \leftarrow s$ 
   $y \leftarrow s.succ$ 
  tant que  $(y \neq \perp)$  et  $(y.succ \neq \perp)$ 
     $x \leftarrow x.succ$ 
     $y \leftarrow y.succ.succ$ 
  // Scinder en deux
   $y \leftarrow x.succ$ 
   $x.succ \leftarrow \perp$ 
   $x \leftarrow s$ 
  // Trier récursivement et fusionner
  retourner fusion(trier( $x$ ), trier( $y$ ))

```

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

Sorties : séquence s triée

$c \leftarrow \overbrace{[0, 0, \dots, 0]}^{k \text{ fois}}$

pour $x \in s$
| $c[x] \leftarrow c[x] + 1$

$total \leftarrow 0$

pour $i \in [1..k]$
| $c[i], total \leftarrow total, total + c[i]$

$t \leftarrow \overbrace{[\perp, \perp, \dots, \perp]}^{n \text{ fois}}$

pour $x \in s$
| $t[c[x]] \leftarrow x$
| $c[x] \leftarrow c[x] + 1$

retourner t

Chapitre 3

3.1) Voir `est_acyclique`.



3.2)

```

Entrées : matrice  $A \in \{0, 1\}^{n \times n}$ 
Résultat : indice de l'intrus

i ← 1
j ← 1
tant que j ≤ n
  | si i = j alors
  | | j ← j + 1
  | sinon si  $A[i, j] = 0$  alors // i n'est pas l'intrus
  | | i ← i + 1
  | sinon // j n'est pas l'intrus
  | | j ← j + 1
retourner i

```

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 ← vrai
couleur ← [v ↦ aucune : v ∈ V]

colorier(u, v):
  | si couleur[u] = aucune alors
  | | couleur[u] ← c
  | | pour v : u → v
  | | | colorier(v, couleur inverse de c)
  | sinon si couleur[u] ≠ c alors
  | | biparti ← faux

pour v ∈ 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 $C = [v_0, v_1, \dots, v_k]$ 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 $C' := [v_0, v_1, \dots, v_i, v_{j+1}, \dots, v_k]$. Remarquons que C' est un cycle plus court que C . Ainsi, en répétant ce processus suffisamment de fois, nous devons obtenir un cycle simple. \square

- 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 $C = [v_0, v_1, \dots, v_k]$. Écrivons $x \prec y$ pour dénoter « x apparaît avant y dans l'ordre topologique T ». Par définition, nous devons avoir $v_0 \prec v_1 \prec \dots \prec v_k$, et ainsi $v_0 \prec v_k$ par transitivité. Nous obtenons une contradiction puisque $v_0 = v_k$. \square

Chapitre 4

- 4.1) 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.

Si l'on ne croit 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 $p'[e] := p[e] + |c| + 1$. 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) Minimiser $p[1] \cdot p[2] \cdot \dots \cdot p[n]$ est équivalent à minimiser son logarithme: $\log(p[1] \cdot p[2] \cdot \dots \cdot p[n]) = \log p[1] + \log p[2] + \dots + \log p[n]$. Il suffit donc de remplacer chaque poids par son logarithme et de rechercher un arbre couvrant minimal sous la définition standard.
- 4.4) 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  $v : u \rightarrow v$ 
|   |   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

```

- 4.5) La procédure gloutonne remplit trois bacs plutôt que deux pour $c := 20$ et $p := [10, 6, 6, 6, 5, 5]$.

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:

$$\begin{aligned} 1 &= c_1 + c_2 \\ 2 &= 3c_1 - 2c_2 \end{aligned}$$

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:

$$\begin{aligned} 0 &= c_1 + c_2 + c_3 \\ 1 &= 2c_1 - c_2 + c_3 \\ 4 &= 4c_1 + c_2 + c_3 \end{aligned}$$

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:

$$\begin{aligned} 1 &= c_1 + c_2 \\ 0 &= c_1 - c_2 \end{aligned}$$

Donc, $c_1 = c_2 = 1/2$, et ainsi

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


5.7) 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( $x, y$ ):
     $i \leftarrow 1; j \leftarrow 1; z \leftarrow []$ 
     $c \leftarrow 0$  // nombre d'inversions entre  $x$  et  $y$ 
    tant que  $i \leq |x| \wedge j \leq |y|$ 
      si  $x[i] \leq y[j]$  alors
        ajouter  $x[i]$  à  $z$ 
         $i \leftarrow i + 1$ 
      sinon
        ajouter  $y[j]$  à  $z$ 
         $j \leftarrow j + 1$ 
         $c \leftarrow c + (|x| - i + 1)$ 
    retourner ( $z + x[i : |x|] + y[j : |y|], c$ )
  si  $|s| \leq 1$  alors retourner ( $s, 0$ )
  sinon
     $m \leftarrow |s| \div 2$ 
     $x, a \leftarrow$  trier( $s[1 : m]$ )
     $y, b \leftarrow$  trier( $s[m + 1 : |s|]$ )
     $z, c \leftarrow$  fusion( $x, y$ )
    retourner ( $z, a + b + c$ )
```


- 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
```

- 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 $s[lo] \leq s[mid] \leq s[hi]$. \square

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 $s[i] := 1$ et $s[j] := 0$ 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

$$s = [\underbrace{0, \dots, 0}_{\text{un certain nombre}}, 1, \overbrace{0, \dots, 0}^{n-k-1 \text{ fois}}].$$

de fois $k \in [0 : n - 1]$

Ainsi, $\max(s) = 1$, alors que \mathcal{A} a retourné 0, ce qui contredit le bon fonctionnement de \mathcal{A} . \square

5.15) Approche diviser-pour-régner:



Entrées : séquence s de $n \in \mathbb{N}_{\geq 1}$ entiers

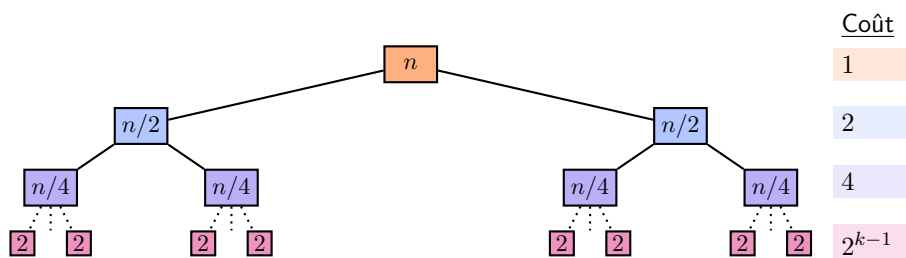
Sorties : somme maximale parmi toutes les sous-séquences contigües non vides

```

somme-max( $s$ ):
  si  $n = 0$  alors
    retourner  $-\infty$ 
  sinon si  $n = 1$  alors
    retourner  $s[1]$ 
  sinon
     $m \leftarrow n \div 2$ 
     $gauche \leftarrow s[1:m]$ 
     $droite \leftarrow s[m+1:n]$ 
    // Cumulatif maximal vers la droite
     $cumul, c \leftarrow 0, -\infty$ 
    pour  $x \in droite$ 
       $cumul \leftarrow cumul + x$ 
       $c \leftarrow \max(c, cumul)$ 
    // Cumulatif maximal vers la gauche
     $cumul, d \leftarrow 0, -\infty$ 
    pour  $x \in gauche$  en ordre inverse
       $cumul \leftarrow cumul + x$ 
       $d \leftarrow \max(d, cumul)$ 
    // Somme maximale des deux côtés
     $a \leftarrow somme-max(gauche)$ 
     $b \leftarrow somme-max(droite)$ 
    // 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:

$$\begin{aligned} 1 &= c_1 + c_2 \\ 0 &= c_1 \cdot i - c_2 \cdot i. \end{aligned}$$

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'($b, n \div 2$)	
$k \leftarrow 1$	
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 $r(b, n) := \text{exp-rapide}'(b, n)$ 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:

$$\begin{aligned} r(b, n) &= (r(b, n \div 2) \oplus r(b, n \div 2)) \oplus b^{n \bmod 2} && \text{(par déf. de l'algo.)} \\ &= (b^{n \div 2} \oplus b^{n \div 2}) \oplus b^{n \bmod 2} && \text{(par hyp. d'induction)} \\ &= b^{2(n \div 2)} \oplus b^{n \bmod 2} && \text{(par la proposition)} \\ &= b^{2(n \div 2) + (n \bmod 2)} && \text{(par la proposition)} \\ &= b^n. && \square \end{aligned}$$

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-chaîne-contiguë(u, v):

```

préfixe( $i, j$ ):
   $p \leftarrow []$ 
  tant que  $i \leq |u| \wedge j \leq |v| \wedge u[i] = v[j]$ 
    ajouter  $u[i]$  à  $p$ 
     $i \leftarrow i + 1$ 
     $j \leftarrow j + 1$ 
  retourner  $p$ 
 $s \leftarrow []$ 
pour  $i \in [1..|u|]$ 
  pour  $j \in [1..|v|]$ 
     $s' \leftarrow$  préfixe( $i, j$ )
    si  $|s'| > |s|$  alors  $s \leftarrow s'$ 
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-chaîne(u, v):

```

aux( $x, y, i, j$ ):
  si  $i \leq |u|$  alors
     $a \leftarrow$  aux( $x + u[i], y, i + 1, j$ )
     $b \leftarrow$  aux( $x, y, i + 1, j$ )
    si  $|a| \geq |b|$  alors retourner  $a$ 
    sinon retourner  $b$ 
  sinon si  $j \leq |v|$  alors
     $a \leftarrow$  aux( $x, y + v[j], i, j + 1$ )
     $b \leftarrow$  aux( $x, y, i, j + 1$ )
    si  $|a| \geq |b|$  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:

$$m \geq d, \quad (*)$$

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 en 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 $m' := m/d$. 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 $d' := d + 1$. Aucun nombre de $[2..d - 1] + [d] = [2..d' - 1]$ ne divise m . \square

Chapitre 7

7.1) On mémorise seulement la dernière ligne de T :



Algorithme 52 : Variante de l'algorithme 41 économe en mémoire.

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  $\infty$ 
 $T[0] \leftarrow 0$ 
pour  $i \in [1..n]$ 
  initialiser séquence  $U[0..m]$  avec  $\infty$ 
  pour  $j \in [0..m]$ 
     $sans \leftarrow T[j]$  // sol. sans  $s[i]$ 
     $avec \leftarrow \infty$ 
    si  $j \geq s[i]$  alors  $avec \leftarrow U[j - s[i]] + 1$  // sol. avec  $s[i]$ 
     $U[j] \leftarrow \min(sans, avec)$ 
   $T \leftarrow U$ 
retourner  $T[m]$ 

```

7.3) Voir levenshtein.py.



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
 $i', j' \leftarrow 0, 0$ 
pour  $i \in [1..|u|]$ 
  pour  $j \in [1..|v|]$ 
    si  $u[i] = v[j]$  alors
       $T[i, j] \leftarrow T[i - 1, j - 1] + 1$ 
    sinon
       $T[i, j] \leftarrow 0$ 
    // Nouvelle solution meilleure que l'ancienne?
    si  $T[i, j] > T[i', j']$  alors
       $i', j' \leftarrow i, j$ 
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 $\mathcal{O}(|u| \cdot |v|)$:

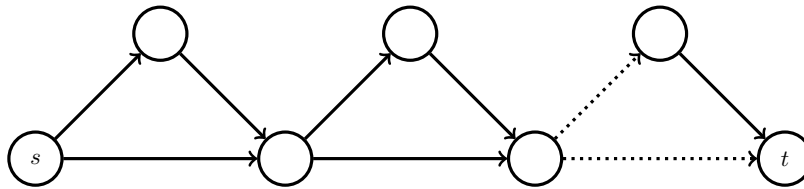


Entrées : chaînes u et v
Résultat : plus longue sous-chaîne commune à u et v

```

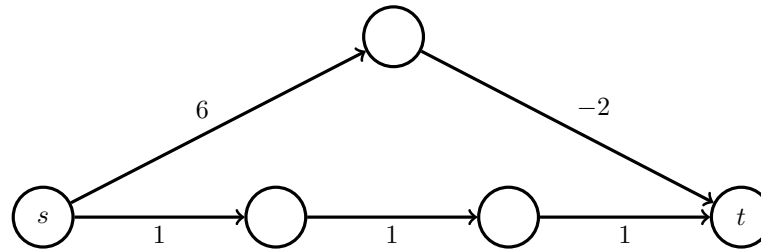
sous-chaîne-dyn( $u, v$ ):
  initialiser  $T[0..|u|, 0..|v|]$  avec 0
  pour  $i \in [1..|u|]$ 
    pour  $j \in [1..|v|]$ 
      si  $u[i] = v[j]$  alors
         $T[i, j] \leftarrow T[i - 1, j - 1] + 1$ 
      sinon
         $T[i, j] \leftarrow \max(T[i - 1, j], T[i, j - 1])$ 
  retourner  $T[m, n]$ 
    
```

7.6)



7.9) On remplace l'initialisation $d[v, v] \leftarrow \text{vrai}$ par $d[v, v] \leftarrow \text{faux}$ 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 $\mathcal{O}(|V| \log |V| + |E| + |V|) = \mathcal{O}(|V| \log |V| + |E|)$.

Chapitre 8

8.1) Considérons la dernière itération de l'algorithme. Soit A l'événement « $y_2 = y_1 = y_0$ ». Nous avons:

$$\begin{aligned}
 \Pr[A] &= \Pr[y_2 = y_1 = y_0 = 0 \vee y_2 = y_1 = y_0 = 1] \\
 &= \Pr[y_2 = y_1 = y_0 = 0] + \Pr[y_2 = y_1 = y_0 = 1] \quad (\text{évén. disjoints}) \\
 &= \prod_{i=0}^2 \Pr[y_i = 0] + \prod_{i=0}^2 \Pr[y_i = 1] \quad (\text{par indép.}) \\
 &= \prod_{i=0}^2 (1/2) + \prod_{i=0}^2 (1/2) \\
 &= 1/8 + 1/8 \\
 &= 1/4.
 \end{aligned}$$

Ainsi,

$$\Pr[\bar{A}] = 1 - \Pr[A] = 1 - (1/4) = 3/4.$$

Soit X la variable aléatoire qui dénote la valeur retournée et soit $k \in [1, 6]$. Nous avons:

$$\begin{aligned}
 \Pr[X = k] &= \Pr[y_2 y_1 y_0 = \text{bin}(k) \mid \bar{A}] \\
 &= \Pr[y_2 y_1 y_0 = \text{bin}(k) \wedge \bar{A}] / \Pr[\bar{A}] \quad (\text{prob. cond.}) \\
 &= \Pr[y_2 y_1 y_0 = \text{bin}(k)] / \Pr[\bar{A}] \\
 &= (1/8) / (3/4) \\
 &= 4/(3 \cdot 8) \\
 &= 1/6.
 \end{aligned}$$

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:

$$\begin{aligned}
 \sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i \cdot \frac{1}{8} &= \frac{1}{8} \cdot \sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i \\
 &= \frac{1}{8} \cdot \frac{1}{1 - 1/4} \quad (\text{série géométrique}) \\
 &= \frac{1}{8} \cdot \frac{4}{3} \\
 &= \frac{1}{6}.
 \end{aligned}$$

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 $a := 1$ et $b := 6$, 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 probabilité d'obtenir $x = 0$ à une itération quelconque est donc de:

$$\begin{aligned}
\sum_{i=0}^{\infty} (p^2 + q^2)^i \cdot pq &= pq \cdot \sum_{i=0}^{\infty} (p^2 + q^2)^i \\
&= pq \cdot \frac{1}{1 - (p^2 + q^2)} && \text{(série géométrique)} \\
&= p(1 - p) \cdot \frac{1}{1 - p^2 - (1 - p)^2} && \text{(car } q = 1 - p) \\
&= p(1 - p) \cdot \frac{1}{1 - p^2 - 1 + 2p - p^2} \\
&= p(1 - p) \cdot \frac{1}{2p(1 - p)} \\
&= \frac{1}{2}.
\end{aligned}$$

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 $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{v}) = \mathbf{C} \cdot \mathbf{v}$, par ex. avec $\mathbf{v} = \mathbf{0}$. Analysons donc la probabilité p que $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{v}) = \mathbf{C} \cdot \mathbf{v}$ lorsque $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$.

Posons $\mathbf{D} := \mathbf{A} \cdot \mathbf{B} - \mathbf{C}$ et $\mathbf{x} := \mathbf{D} \cdot \mathbf{v}$. Remarquons que p correspond à la probabilité que $\mathbf{x} = \mathbf{0}$. Puisque $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, nous avons $\mathbf{D} \neq \mathbf{0}$. Ainsi, il existe $i, j \in [n]$ tels que $\mathbf{D}[i, j] \neq 0$. Nous avons

$$\begin{aligned} \mathbf{x}(i) &= \sum_{k=1}^n \mathbf{D}[i, k] \cdot \mathbf{v}(k) \\ &= \mathbf{D}[i, j] \cdot \mathbf{v}(j) + \sum_{\substack{k=1 \\ k \neq j}}^n \mathbf{D}[i, k] \cdot \mathbf{v}(k). \end{aligned}$$

Posons $a := \mathbf{D}[i, j]$ et $b := \sum_{\substack{k=1 \\ k \neq j}}^n \mathbf{D}[i, k] \cdot \mathbf{v}(k)$. Nous avons

$$\begin{aligned} \Pr[\mathbf{x} = \mathbf{0}] &\leq \Pr[\mathbf{x}(i) = 0] \\ &= \Pr[\mathbf{x}(i) = 0 \mid b = 0] \cdot \Pr[b = 0] + \\ &\quad \Pr[\mathbf{x}(i) = 0 \mid b \neq 0] \cdot \Pr[b \neq 0] \\ &= \Pr[\mathbf{v}(j) = 0] \cdot \Pr[b = 0] + \tag{1} \\ &\quad \Pr[\mathbf{v}(j) = 1 \wedge b = -a] \cdot \Pr[b \neq 0] \\ &\leq \Pr[\mathbf{v}(j) = 0] \cdot \Pr[b = 0] + \\ &\quad \Pr[\mathbf{v}(j) = 1] \cdot \Pr[b \neq 0] \\ &= (1/2) \cdot \Pr[b = 0] + \\ &\quad (1/2) \cdot \Pr[b \neq 0] \\ &= (1/2) \cdot (\Pr[b = 0] + \Pr[b \neq 0]) \\ &= 1/2, \end{aligned}$$

où (1) découle du fait que

$$\mathbf{x}(i) = 0 \iff [\mathbf{v}(j) = b = 0 \vee (\mathbf{v}(j) = 1 \wedge b = -a)].$$

8.6) Soit X la variable aléatoire qui dénote le nombre d'itérations effectuées par l'algorithme. Posons $p_i := (n/2)/(n-i)$ et $q_i := 1 - p_i$. 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

$$i \cdot p_i \leq i \iff \frac{i \cdot (n/2)}{n-i} \leq i \iff (n/2) \leq n-i \iff i \leq n/2. \quad (2)$$

De plus, pour tout $0 \leq i < n$, nous avons:

$$q_i = 1 - \frac{(n/2)}{n-i} \leq 1 - \frac{(n/2)}{n} = 1/2. \quad (3)$$

Ainsi, nous obtenons:

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot \prod_{j=1}^{i-1} q_j \right) \\ &\leq \sum_{i=1}^{n/2} (i \cdot p_i \cdot (1/2)^{i-1}) \quad (\text{par (3)}) \\ &\leq \sum_{i=1}^{n/2} (i \cdot (1/2)^{i-1}) \quad (\text{par (2)}) \\ &\leq \sum_{i=1}^{\infty} (i \cdot (1/2)^{i-1}) \quad (\text{car chaque terme est non négatif}) \\ &= \frac{1}{(1-1/2)^2} \quad (\text{série géométrique dérivée}) \\ &= 4. \quad \square \end{aligned}$$

★★ 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 49. Observons d'abord que:

$$\begin{aligned} \prod_{j=1}^{i-1} q_j &= \frac{(n/2-1)(n/2-2)\cdots(n/2-i+1)}{(n-1)(n-2)\cdots(n-i+1)} \\ &= \frac{(n/2-1)!}{(n/2-i)!} \cdot \frac{(n-i)!}{(n-1)!} \\ &= \frac{(n-i)!}{(n/2)! \cdot (n/2-i)!} / \frac{(n-1)!}{(n/2)! \cdot (n/2-1)!} \\ &= \binom{n-i}{n/2} / \binom{n-1}{n/2}. \quad (4) \end{aligned}$$

De plus, remarquons les propriétés suivantes du coefficient binomial:

$$\binom{m}{k} = \frac{m}{k} \cdot \binom{m-1}{k-1} \quad \binom{m+1}{k+1} = \sum_{j=k}^m \binom{j}{k}. \quad (5)$$

Ainsi, nous avons:

$$\mathbb{E}[X] = \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot \prod_{j=1}^{i-1} q_j \right)$$

$$= \sum_{i=1}^{n/2} \left(i \cdot p_i \cdot \binom{n-i}{n/2} / \binom{n-1}{n/2} \right) \quad (\text{par (4)})$$

$$= \sum_{i=1}^{n/2} \left(i \cdot \frac{n/2}{n-i} \cdot \binom{n-i}{n/2} / \binom{n-1}{n/2} \right)$$

$$= \sum_{j=n/2}^{n-1} \left(n/2 \cdot \frac{n-j}{j} \cdot \binom{j}{n/2} / \binom{n-1}{n/2} \right)$$

$$= \frac{n/2}{\binom{n-1}{n/2}} \cdot \sum_{j=n/2}^{n-1} \left(\frac{n-j}{j} \cdot \binom{j}{n/2} \right)$$

$$= \frac{n/2}{\binom{n-1}{n/2}} \cdot \left[\sum_{j=n/2}^{n-1} \frac{n}{j} \cdot \binom{j}{n/2} - \sum_{j=n/2}^{n-1} \binom{j}{n/2} \right]$$

$$= \frac{n/2}{\binom{n-1}{n/2}} \cdot \left[\frac{n}{n/2} \cdot \sum_{j=n/2}^{n-1} \binom{j-1}{n/2-1} - \sum_{j=n/2}^{n-1} \binom{j}{n/2} \right] \quad (\text{par (5)})$$

$$= \frac{n/2}{\binom{n-1}{n/2}} \cdot \left[\frac{n}{n/2} \cdot \binom{n-1}{n/2} - \binom{n}{n/2+1} \right] \quad (\text{par (5)})$$

$$= n - (n/2) \cdot \frac{\binom{n}{n/2+1}}{\binom{n-1}{n/2}}$$

$$= n - (n/2) \cdot \frac{n}{n/2+1} \cdot \frac{\binom{n-1}{n/2}}{\binom{n-1}{n/2}} \quad (\text{par (5)})$$

$$= n - (n/2) \cdot \frac{n}{n/2+1}$$

$$= n - \frac{n^2}{n+2}$$

$$= \frac{n^2+2n}{n+2} - \frac{n^2}{n+2}$$

$$= \frac{2n}{n+2}$$

$$= \frac{2(n+2)-4}{n+2}$$

$$= 2 - \frac{4}{n+2}$$

$$< 2.$$

□

8.7) Analysons l'algorithme 49. 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 50. 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 $\text{err}(n) = \max(0, (2/3)^{275}, (1/3)^{275}) = (2/3)^{275} \leq 1/2^{260}$.

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 $n \geq n_0 \rightarrow f(n) \leq c \cdot g(n)$ pour certains c, n_0
- ▶ *Signifie*: f croît moins ou aussi rapid. que g pour $n \rightarrow \infty$
- ▶ *Transitivité*: $f \in \mathcal{O}(g)$ et $g \in \mathcal{O}(h) \rightarrow f \in \mathcal{O}(h)$
- ▶ *Règle des coeff.*: $f_1 + \dots + f_k \in \mathcal{O}(c_1 \cdot f_1 + \dots + c_k \cdot f_k)$
- ▶ *Règle du max.*: $f_1 + \dots + f_k \in \mathcal{O}(\max(f_1, \dots, f_k))$
- ▶ *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 $d \rightarrow f \in \Theta(n^d)$

Notation asymptotique (suite)

- ▶ *Simplification*: lignes élem. comptées comme une seule opér.
- ▶ *Règle de la limite*:

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f \in \mathcal{O}(g) \text{ et } g \notin \mathcal{O}(f) \\ +\infty & f \notin \mathcal{O}(g) \text{ et } g \in \mathcal{O}(f) \\ \text{const.} & \Theta(f) = \Theta(g) \end{cases}$$
- ▶ *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)			Sur place	Stable
	meilleur	moyen	pire		
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. $u_0 \rightarrow \dots \rightarrow u_k$ (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

		Mat.	Liste (non dirigé)	Liste (dir.)
	$u \rightarrow v?$	$\Theta(1)$	$\mathcal{O}(\min(\deg(u), \deg(v)))$	$\mathcal{O}(\deg^+(u))$
$a \begin{pmatrix} a & b & c \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$	$[a \mapsto [b, c],$	$\Theta(V)$	$\mathcal{O}(\deg(u))$	$\mathcal{O}(\deg^+(u))$
	$b \mapsto [a],$	$\Theta(V)$	$\mathcal{O}(\deg(v))$	$\mathcal{O}(V + E)$
$c \mapsto [b]$	Modif. $u \rightarrow v$	$\Theta(1)$	$\mathcal{O}(\deg(u) + \deg(v))$	$\mathcal{O}(\deg^+(u))$
	Mémoire	$\Theta(V ^2)$	$\Theta(V + E)$	

Propriétés et algorithmes

- ▶ *Plus court chemin*: parcours en largeur + stocker préd.
- ▶ *Ordre topologique*: $u_1 \preceq \dots \preceq u_n$ 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 qui contient tous les sommets d'un graphe

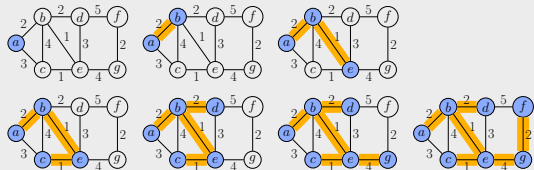
4. Algorithmes gloutons

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

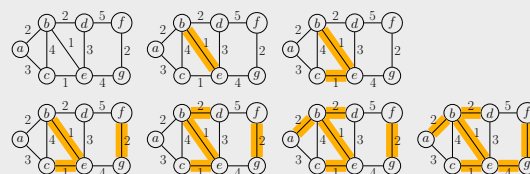
Algorithme de Prim-Jarník

- ▶ *Approche*: faire grandir un arbre en prenant l'arête min.
- ▶ *Complexité*: $\mathcal{O}(|E| \log |V|)$ avec monceau



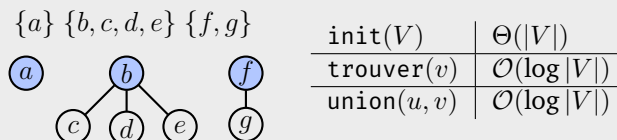
Algorithme de Kruskal

- ▶ *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

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 $t(n), t(n-1), t(n-2), \dots$ 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*: $\text{src}[1:n-1] \rightarrow \text{tmp}, \text{src}[n] \rightarrow \text{dst}, \text{tmp}[1:n-1] \rightarrow \text{dst}$ $\mathcal{O}(2^n)$
- ▶ *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*: débiter 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 + mémorisation (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 $s = [1, 5, 7]$

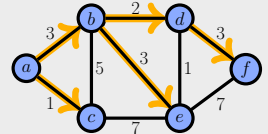
	0	1	2	3	4	5	6	7	8	9	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	1	2	3	4	5	6	7	8	9	10
2	0	1	2	3	4	1	2	3	4	5	2
3	0	1	2	3	4	1	2	1	2	3	2

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 $\geq 1, 2, \dots, |V| - 1$ arêtes
- ▶ *Sommaire*:



	Dijkstra	Bellman-Ford	Floyd-Warshall
Types de chemins	d'un sommet vers les autres		paires de sommets
Poids négatifs?	✗	✓	✓
Temps d'exécution	$\mathcal{O}(V \log V + E)$	$\Theta(V \cdot E)$	$\Theta(V ^3)$
Temps ($ E \in \Theta(1)$)	$\mathcal{O}(V \log V)$	$\Theta(V)$	$\Theta(V ^3)$
Temps ($ E \in \Theta(V)$)	$\mathcal{O}(V \log V)$	$\Theta(V ^2)$	$\Theta(V ^3)$
Temps ($ E \in \Theta(V ^2)$)	$\mathcal{O}(V ^2)$	$\Theta(V ^3)$	$\Theta(V ^3)$

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 $\Pr(Y_x \neq \text{bonne sortie sur } x)$

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*: $\sum(\text{temps instances de taille } n) / \# \text{ instances}$
- ▶ *Attention*: pas la même chose que le temps espéré
- ▶ *Hypothèse*: entrées distribuées uniformément (\pm réaliste)
- ▶ *Exemple*: $\Theta(n^2)$ pour le tri par insertion

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.
- [Eri19] Jeff Erickson. *Algorithms*. 2019.

Index

- \mathbb{N} , 3
- \mathcal{O} , 15
- Ω , 19
- \mathcal{P} , 2
- \mathbb{Q} , 3
- \mathbb{R} , 3
- Θ , 20
- \mathbb{Z} , 3
- 3SUM, 34

- accessibilité, 54, 118
- Ackermann, 70
- acyclique, 51, 57
- adjacence, 49
 - liste, 53
 - matrice, 52
- algorithme
 - de Bellman-Ford, 118
 - de Dijkstra, 112
 - de Floyd-Warshall, 114
 - de Freivalds, 135
 - de Karatsuba, 81
 - de Karger, 129
 - de Kruskal, 66
 - de Las Legas, 128
 - de Monte Carlo, 129
 - de Prim–Jarník, 64
 - déterministe, 125
 - glouton, 63
 - probabiliste, 125
 - récuratif, 73

 - vorace, 63
- amplification, 132
- analyse, 13
- approche
 - ascendante, 108
 - descendant, 107
- arborescence, 58
- arbre, 58
 - couvrant, 58
 - couvrant minimal, 63
 - de récursion, 82
- arithmétique, 79, 81
- arête, 49

- Bellman-Ford, 118
- binaire, 45
- branch-and-bound*, 99

- carré, 5
- chemin, 51
 - plus court, 111
 - simple, 51, 111
- coefficient binomial, 4
- combinatoire, 4
- comparable, 35
- complément, 2
- composante
 - connexe, 52
 - fortement connexe, 52
- connexité, 52
- constante multiplicative, 15, 19, 20

- correction, 27
- coupe minimum, 129
- cycle, 51, 57
 - négatif, 111, 118, 120
 - simple, 51
- degré, 49
 - entrant, 49
 - sortant, 49
- différence, 2
- Dijkstra, 112
- diviser-pour-régner, 73
- division entière, 4
- détection de cycle, 57
- ensemble, 2
 - des parties, 2
 - des sous-ensembles, 2
 - vide, 2
- ensembles disjoints, 67
- espérance, 125
- explosion combinatoire, 96
- exponentiation, 79
- exponentielle, 3
- factorielle, 4
- feuille, 58
- Floyd-Warshall, 114
- fonction d'Ackermann, 70
- force brute, 96
- forêt, 58
- graphe, 49
 - acyclique, 51
 - pondéré, 63
 - représentation, 52
- Hanoï, 73
- homogène, 76, 79
- induction, 5
 - généralisée, 9
- intersection, 2
- intervalle, 2
- invariant, 27
- inversion, 35
- jeu de Nim, 9
- Karatsuba, 81
- knapsack problem*, 99
- Las Vegas, 128
- ligne d'horizon, 85
- limite, 25
- liste d'adjacence, 53
- logarithme, 3
- logique propositionnelle, 103
- loi géométrique, 125
- longueur, 51
- majorité, 21, 28
- mathématiques discrètes, 2
- matrice d'adjacence, 52
- meilleur cas, 13
- modulo, 4
- monceau, 40, 64
 - de Fibonacci, 114
- Monte Carlo, 129
- multiplication, 81
- médiane, 42
- mémorisation, 107
- nombre
 - aléatoire, 125
 - pseudo-aléatoire, 125
- nombres, 3
 - entiers, 3
 - naturels, 3
 - rationnels, 3
 - réels, 3
- notation asymptotique, 15
- NP-complétude, 96
- opération élémentaire, 13
- ordonnancements, 4
- ordre
 - partiel, 57
 - topologique, 57
 - total, 35
- paramètres, 14, 26
- parcours, 54
 - en largeur, 55

- en profondeur, 54
- pavage, 7, 76
- paysage, 85
- permutations, 4
- pire cas, 13
- pivot, 42
- plus court chemin, 56
- plus courts chemins, 111
- poids
 - négatif, 114
- polynômes, 15, 19, 20
- post-condition, 27
- preuve, 4
 - directe, 4
 - par contradiction, 5
 - par induction, 5
 - par l'absurde, 5
 - par récurrence, 5
- principe d'optimalité, 107
- probabilité, 125
 - d'erreur, 129
 - de succès, 129
- problème
 - des n dames, 96
 - du retour de monnaie, 102, 108
 - du sac à dos, 99, 110
- produit cartésien, 2
- programmation
 - dynamique, 107
 - linéaire entière, 104
- pruning*, 99
- pré-condition, 27
- prédécesseur, 50
- racine, 58
- radix, 45
- retour arrière, 96
- règle de la limite, 25
- récurrence
 - homogène, 76
 - linéaire, 76
 - non homogène, 79
- récurtivité, 73
- sac à dos, 99
- SAT, 103
- satisfaction, 103
- seuil, 15, 19, 20
- sommet, 49
 - interne, 58
- sous-graphe, 52
 - induit par, 52
- sous-séquence, 3
- stabilité, 44
- stable, 44
- successeur, 50
- suite de Fibonacci, 3, 77
- sur place, 44
- séquence, 3
 - de Collatz, 28
 - de Fibonacci, 3, 77
- tableaux, 108
- taille, 14
- tas, 40, 64
- temps d'exécution, 13
- temps espéré, 128
- temps moyen, 133
- terminaison, 28
- théorème maître, 83
- tours de Hanoï, 73
- tri, 35
 - de crêpes, 47
 - fusion, 41
 - heapsort*, 40
 - merge sort*, 41
 - par fusion, 41
 - par insertion, 39, 133
 - par monceau, 40
 - par tas, 40
 - quicksort*, 42
 - radix, 45
 - rapide, 42
 - sans comparaison, 45
 - topologique, 57
- union, 2
- union-find*, 67
- variable aléatoire, 125
- voisin, 49
- élagage, 99