Chapitre 3

Complexité algorithmique

3.1 Exercice 3.1.

Soit $f : \mathbb{R}_+ \to \mathbb{R}_+$ une fonction quelconque. Montrons que $\mathcal{O}(f)$ est stable par l'addition et la multiplication.

Soient $g_1, g_2 : \mathbb{R}_+ \to \mathbb{R}_+$, appartenant à $\mathcal{O}(f)$. Alors, il existe deux constantes positives K_1 et K_2, ainsi que deux entiers n_1 et n_2, tels que :

$$\forall\ n \geqslant n_1 \quad g_1(n) \leqslant K_1 f(n)$$

$$\forall\ n \geqslant n_2 \quad g_2(n) \leqslant K_2 f(n)$$

Donc, en sommant les deux inégalités, et en posant $n_0 = \max(n_1, n_2)$ et $K = \lambda K_1 + \mu K_2$:

$$\forall\ n \geqslant n_0 \quad \lambda g_1(n) + \mu g_2(n) \leqslant (\lambda K_1 + \mu K_2) f(n) = K f(n)$$

D'où : $\lambda g_1 + \mu g_2 \in \mathcal{O}(f)$, ce qui montre le résultat recherché.

3.2 Exercice 3.2.

Montrons que l'équivalence est bien une relation d'équivalence. Soient f, g et h trois fonctions de $\mathbb{R}_+ \to \mathbb{R}_+$.

- **Reflexivité** : pour tout réel $\varepsilon > 0$ et pour tout $n \geqslant n_0 = 0$, $|f(n) - f(n)| = 0 \leqslant \varepsilon f(n)$, donc $f \sim f$.

- **Symétrie** : si $f \sim g$ alors pour tout $\varepsilon > 0$, il existe un entier n_0 tel que pour tout $n \geqslant n_0$, $|f(n) - g(n)| \leqslant \varepsilon g(n)$. Autrement dit :

$$\frac{|f(n)-g(n)|}{g(n)} \underset{n\to+\infty}{\to} 0 \Rightarrow \left|\frac{f(n)}{g(n)} - 1\right| \underset{n\to+\infty}{\to} 0 \Rightarrow \frac{f(n)}{g(n)} \underset{n\to+\infty}{\to} 1$$

$$\Rightarrow \frac{g(n)}{f(n)} \underset{n\to+\infty}{\to} 1 \Rightarrow \left|\frac{g(n)}{f(n)} - 1\right| \underset{n\to+\infty}{\to} 0 \Rightarrow \frac{|g(n)-f(n)|}{f(n)} \underset{n\to+\infty}{\to} 0$$

  Et donc : $g \sim f$

- **Transitivité** : si $f \sim g$ et $g \sim h$ (que l'on peut aussi écrire $h \sim g$ d'après le point précédent), alors pour tout réel $\varepsilon > 0$ il existe un entier n_0 (maximum des deux entiers entrant en jeu dans les définitions de $f \sim g$ et $h \sim g$) tel que pour tout $n \geqslant n_0$: $|f(n) - g(n)| \leqslant \varepsilon g(n)$ et $|h(n) - g(n)| \leqslant \varepsilon g(n)$. Donc :

$$|f(n)-h(n)| = |f(n)-g(n)+g(n)-h(n)| \leqslant |f(n)-g(n)|+|g(n)-h(n)| \leqslant 2\varepsilon g(n)$$

  Or $g \sim f$, donc on peut trouver une constante K telle que $g(n) \geqslant f(n)$ à partir d'un certain rang. Donc : $|f(n) - h(n)| \leqslant 2\varepsilon g(n) \leqslant 2K\varepsilon f(n) = \varepsilon' f(n)$.

  Et donc : $f \sim h$.

L'équivalence est donc bien une relation d'équivalence.

3.3 Exercice 3.3.

Soient f, g et h trois fonctions de $\mathbb{N} \to \mathbb{R}^{+*}$. Supposons que $f \sim g$.

Si $f \sim g$ alors :

$$\forall\ \varepsilon > 0,\ \exists\ n_0 \in \mathbb{N} \text{ t.q. } n \geqslant n_0\ \Rightarrow\ \left|\frac{g(n)-f(n)}{f(n)}\right| < \varepsilon$$

Sous les même conditions, on a donc :

$$\left|\frac{g(n)}{f(n)} - 1\right| < \varepsilon\ \Rightarrow\ 1-\varepsilon < \frac{g(n)}{f(n)} < 1+\varepsilon$$

Par ailleurs, on $f = \Theta(h)$ donc par définition, h ne s'annuant pas, on peut écrire que le rapport f/h est borné (par une constante K) à partir d'un certain rang $n_1 \in \mathbb{N}$. Donc :

$$\frac{g(n)}{f(n)} = \frac{g(n)}{h(n)}\frac{h(n)}{f(n)} < 1 + \varepsilon \;\Rightarrow\; \frac{g(n)}{h(n)} < (1+\varepsilon)\frac{f(n)}{h(n)}$$

Soit, avec tous les quantificateurs, et en identifiant ρ à $(1+\varepsilon)K$:

$$\forall\ \rho > 0 \quad \exists\ n^* = \min(n_0, n_1) \quad \text{t.q.} \quad \forall\ n \geqslant n^* \quad \frac{g(n)}{h(n)} < \rho$$

Autrement dit, le rapport g/h est borné à partir d'un certain rang, et donc $g = \Theta(h)$.

3.4 Exercice 3.4.

La formule de Stirling nous dit que $n!$ est équivalent à $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ en l'infini. Soit $a \in \mathbb{R}^{+*}$ une constante. Evaluons :

$$\frac{n!}{a^n} \sim \sqrt{2\pi n}\left(\frac{n}{ae}\right)^n \underset{n\to+\infty}{\to} +\infty$$

Donc un algorithme de complexité factorielle appartient bien *stricto sensu* à la classe hyper-exponentielle (bien qu'on ait souvent tendance à le répertorier dans la classe exponentielle).

3.5 Exercice 3.5.

Le polynôme caractéristique de la suite s'écrit $P(X) = X^2 - 3X - 4$. Son discriminant vaut $\Delta = 25$, donc il admet deux racines simples : $\{-1, 4\}$.

D'après le théorème de la section 3.10.2 sur les suites récurrentes d'ordre multiple, on a $u_n = \lambda_1 \times 1^n + \lambda_2 \times 4^n = \Theta(4^n)$.

Si on retire le terme $3u_{n+1}$ l'équation de récurrence devient $u_{n+2} = 4u_n$ et le polynôme caractéristique s'écrit $P(X) = X^2 - 4$ dont la solution est $\{-2, 2\}$, et donc $u_n = \Theta(2^n)$. Notons que dans ce second cas, le résultat peut être trouvé directement en posant $w_n = u_{2n}$:

$$w_{n+1} = 4w_n \Rightarrow w_n = 4^n w_0 \Rightarrow w_n = \Theta(4^n)$$

$$u_{2n} = \Theta(4^n) \Rightarrow u_n = \Theta(4^{\frac{n}{2}}) = \Theta\big((\sqrt{4})^n\big) = \Theta(2^n)$$

3.6 Exercice 3.6.

Soit f un algorithme de complexité $f(n) = n^{\log n}$.

Soit $\alpha \in \mathbb{R}$, $\alpha > 1$. Considérons le quotient :

$$Q_\alpha(n) = \frac{n^\alpha}{n^{\log n}} = n^{\alpha - \log n} \underset{n \to +\infty}{\to} 0$$

Donc, il existe une constante $K \in \mathbb{R}^+$ telle qu'à partir d'un entier n_0 suffisamment grand, $\forall\ n \geqslant n_0\ \ Q_\alpha(n) \leqslant K$, et donc $n^\alpha = \mathcal{O}(n^{\log n}) \Rightarrow f(n) = n^{\log n} = \Omega(n^\alpha)$, ce qui montre la première partie du résultat recherché.

Pour $\beta > 1$, considérons à présent le quotient :

$$R_\beta(n) = \frac{n^{\log n}}{\beta^n}$$

On a : $\log\big(R_\beta(n)\big) = \log n \times \log n - n \log \beta \underset{n \to +\infty}{\to} -\infty$.

Donc : $R_\beta(n) \underset{n \to +\infty}{\to} 0$, et il existe une constante $K' \in \mathbb{R}^+$ telle qu'à partir d'un entier n'_0 suffisamment grand, $\forall\ n \geqslant n'_0\ \ R_\beta(n) \leqslant K'$ d'où : $n^{\log n} = \mathcal{O}(\beta^n)$.

Il s'agit donc d'un algorithme de complexité quasi-polynomiale.

3.7 Exercice 3.7.

Un entier $n \in \mathbb{N}^*$, $n \geqslant 2$, est premier si, et seulement si, pour tout $k \in \{2, n-1\}$, k ne divise pas n. Algorithmiquement :

PRIME$(N \downarrow)$: $\Big[\!\Big[$ PRIME $\leftarrow$ ⇃; $\Big\{_{k:2}^{N-1}$**MOD**$(N,k) = 0$? PRIME $\leftarrow$ ↾; ! | ¿ $\Big\}_k$ $\Big]\!\Big]$

Le squelette de l'algorithme est :

$$\left\{ ? \quad | \ ¿ \right\} \qquad\qquad d_E = 1$$

L'algorithme contient une boucle simple effectuant n itérations dans le pire des cas, d'où une complexité $C(n) = \Theta(n)$.

On peut optimiser la procédure en notant $k \in \{2, n-1\}$ le plus petit entier qui divise n. Alors, on peut écrire $n = kp$ et, par définition p divise n donc $p \geqslant k$. Donc par suite : $n \geqslant k^2 \Rightarrow k \leqslant \sqrt{n}$. Il suffit donc de tester tous les entiers $k \in \{2, \lceil \sqrt{n} \rceil + 1\}$.

PRIME$(N \downarrow)$: $[\![$ PRIME $\leftarrow$ ⇃; $B \leftarrow$ **FLOOR**(**SQRT**(N))+1; $\left\{ {}^{B}_{k:2} \right.$**MOD**$(N, k) = 0$? PRIME $\leftarrow$ ↱; ! | ¿ $\left. \right\}_k$ $]\!]$

La complexité de l'algorithme devient alors : $C(n) = \Theta(\sqrt{n})$.

Comme mentionné en remarque 7 de la section 3.7 du cours, il est d'usage en théorie de la complexité de considérer que la quantité d'information nécessaire pour encoder un nombre n est égale à la longueur de sa décomposition binaire. Ainsi, la fonction taille est définie par $|.| : n \to \log_2 n$. Une donnée de taille n permet alors d'encoder tous les nombres jusqu'à concurrence de 2^n d'où l'expression de la complexité avec cette convention : $C(n) = \Theta(\sqrt{2^n}) = \Theta(2^{\frac{n}{2}})$, complexité exponentielle. Cette seconde unité de taille paraît sous certains aspects plus naturelle : même en étant réduit à un recherche des $\sqrt{n}$ premiers entiers, ce test de primalité agit néanmoins peu ou prou comme une recherche exhaustive dans l'espace des solutions.

Il existe des tests de primalité plus rapides, par exemple le test *cyclotomique*, de complexité (toujours avec la convention consistant à encoder la taille du nombre à traiter en fonction de la longueur de sa décomposition binaire) $\Theta(n^{a \log(\log n)})$, avec $a \in \mathbb{R}^+$ une constante. On peut montrer avec des techniques similaires à celles employées dans l'exercice précédent, que le test cyclotomique est de complexité quasi-polynomiale. Le problème de primalité est donc l'un des problèmes de décision qui étaient un temps supposés appartenir à la classe NP mais qui ont finalement trouvé une résolution en temps polynomial.

Similairement, le test de *Agrawal-Kayal-Saxena* (AKS), paru en 2002, est asymptotiquement le plus rapide à ce jour, avec une complexité polynomiale en $\Theta(n^{12})$. Notons qu'un algorithme de complexité encore plus faible ($\Theta(n^4)$) est déjà prêt dans les tiroirs, à la condition que l'hypothèse de Riemann généralisée puisse être démontrée.

3.8 Exercice 3.8.

$$a_0 = a_2 = 1 \qquad a_1 = 2 \qquad \forall\ n \in \mathbb{N} \quad a_{n+3} = 2a_{n+2} + 5a_{n+1} - 6a_n$$

On se plonge dans l'espace vectoriel $\mathbb{R}^3$ et on utilise la suite $(\mathbf{X_n})_{n\geqslant 2}$:

$$\mathbf{X_n} = \begin{bmatrix} a_{n-2} \\ a_{n-1} \\ a_n \end{bmatrix}$$

Sous ce nouveau formalisme, l'équation récurrente s'exprime de la manière suivante :

$$\mathbf{X_2} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & 5 & 2 \end{bmatrix} \qquad \text{et :} \qquad \mathbf{X_{n+1}} = \mathbf{A}\mathbf{X_n} \quad \forall\ n \geqslant 2$$

Et donc : $\mathbf{X_n} = \mathbf{A^{n-2}}\mathbf{X_2}$

La complexité de l'algorithme de calcul est toujours linéaire $C(n) = \Theta(n)$. Mais on peut à présent diagonaliser la matrice $\mathbf{A}$. On calcule son polynôme caractéristique :

$$P(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I_3}) = \begin{vmatrix} -\lambda & 1 & 0 \\ 0 & -\lambda & 1 \\ -6 & 5 & 2-\lambda \end{vmatrix} = -\lambda^3 + 2\lambda^2 + 5\lambda - 6$$

$\lambda = 1$ est une solution triviale, on factorise donc P sous la forme :

$$P(\lambda) = (\lambda - 1)(-\lambda^2 + \lambda + 6)$$

Par caclul du discriminant, on trouve facilement que le trinôme $-\lambda^2 + \lambda + 6$ possède deux racines simples $\{-2, 3\}$. On en déduit donc le spectre de la matrice $\mathbf{A}$:

$$\mathrm{Sp}(\mathbf{A}) = \{-2, 1, 3\}$$

La matrice est donc diagonalisable : $\mathbf{A} = \mathbf{PDP^{-1}}$ avec :

$$\mathbf{D} = \begin{bmatrix} -2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

Le calcul de la matrice de passage (et de son inverse) vient immédiatement après résolution du système $\mathbf{AX} = \lambda\mathbf{X}$ pour $\lambda \in \mathrm{Sp}(\mathbf{A})$, qui nous donne trois vecteurs propres :

$$\left\{ \begin{pmatrix} 1 \\ -2 \\ 4 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -3 \\ 9 \end{pmatrix} \right\}$$

Après normalisation, on obtient la matrice de passage :

$$\mathbf{P} = \begin{bmatrix} 1\sqrt{21} & 1\sqrt{3} & 1\sqrt{91} \\ -2\sqrt{21} & 1\sqrt{3} & 3\sqrt{91} \\ 4\sqrt{21} & 1\sqrt{3} & 9\sqrt{91} \end{bmatrix}$$

Le calcul de la suite numérique $(a_n)_{n\in\mathbb{N}}$ se réduit donc à celui de la suite vectorielle $(\mathbf{X}_n)_{n\geqslant 2}$ via son expression explicite :

$$\mathbf{X}_{n+2} = \mathbf{A}^n\mathbf{X}_2 = \left(\mathbf{PDP^{-1}}\right)^n\mathbf{X}_2 = \mathbf{PD^nP^{-1}X}_2$$

où la dernière égalité se démontre facilement par récurrence sur n (exercice classique de L2).

Le calcul de a_n nécessite donc seulement de connaître les puissances n-ieme des 3 valeurs propres {-2,1,3}. L'algorithme d'exponentiation rapide (introduit dans la section 1.5.7 du cours) permet d'obtenir ces quantités en un temps logarithmique. On en déduit alors la complexité de l'algorithme :

$$C(n) = \Theta(\log n)$$

3.9 Exercice 3.9.

On considère l'équation de partition : $T(n) = 9T(n/3) + n^2 + 2n + 1$.

On utilise la Master Theorem avec $a = 9$, $b = 3$ et $f(n) = n^2 + 2n + 1 = \Theta(n^2)$.

Calculons : $n^{\log_b a} = n^{\log_3 9} = n^2$, donc $f(n) = \Theta(n^{\log_b a})$, et le coût de décomposition/recomposition est asymptotiquement du même ordre que l'indice de récursion. Nous sommes donc dans la cas 2 du Master Theorem, et $T(n) = \Theta(f(n) \log n)$, d'où :

$$T(n) = \Theta(n^2 \log n)$$

Si on parvient à supprimer l'un des sous-problèmes (par exemple en imaginant que l'un des 9 cas est trivial et ne nécessite aucun travail), alors $a = 8$ et $n^{\log_b a} = n^{\log_3 8} \approx n^{1.893} = \mathcal{O}(f(n))$, le coût de décomposition/recomposition devient alors prépondérant dans l'analyse, ce qui donne une complexité $T(n) = \Theta(f(n)) = \Theta(n^2)$.

3.10 Exercice 3.10.

Par application directe du Master Theorem :

1. $C(n) = \Theta(n)$
2. Pas de solution (a dépend de n).
3. $C(n) = \Theta(n \log n)$
4. $C(n) = \Theta(n^2)$
5. $C(n) = \Theta(n \log n)$
6. $C(n) = \Theta(n^2)$
7. $C(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.808})$
8. $C(n) = \Theta(n^{\log_3 5}) \approx \Theta(n^{1.467})$
9. Pas de solution ($a \notin \mathbb{N}^*$).
10. $C(n) = \Theta(n^3 \log n)$
11. $C(n) = \Theta(n^2 \log n)$
12. $C(n) = \Theta(n^3)$
13. $C(n) = \Theta(n!)$
14. $C(n) = \Theta(n^{0.51})$
15. Pas de solution ($a \notin \mathbb{N}^*$).
16. Pas de solution (a dépend de n).

Pour les cas 9 et 10 où le Master Theorem n'est pas applicable puisque $a \notin \mathbb{N}^*$, on peut néanmoins proposer des majorations. Par exemple dans le cas 9 :

$C(n) = 0.5C(n/2) + n \log n \leqslant C(n/2) + n \log n$

On étudie donc l'équation de partition annexe : $T(n) = T(n/2) + n \log n$ dont on peut montrer à l'aide de l'inéquation ci-dessus qu'elle domine C. Le Master Theorem nous indique que $T(n) = \Theta(n \log n)$ et donc : $C(n) = \mathcal{O}(n \log n)$. Avec une approche similaire, on obtient :

15. $C(n) = \mathcal{O}(n)$

3.11 Exercice 3.11.

Le problème a déjà été étudié dans l'exemple 5 de la section 2.2.3 ainsi que dans la section 3.10.1. Nous avions montré que la procédure globale au rang n pouvait se décomposer en deux appels de $H(n-1)$ plus un appel trivial de $H(1)$. L'équation de récurrence sur la complexité de la procédure s'écrit donc de la même manière :

$$T(1) = 1 \qquad \text{et :} \qquad T(n) = 2T(n-1) + 1$$

$$\frac{T(n)}{2^n} = \frac{T(n-1)}{2^{n-1}} + 2^{-n}$$

$$\frac{T(n)}{2^n} = T(1) + \sum_{k=1}^{n} 2^{-k} = 2(1 - \frac{1}{2^n}) = 2 - \frac{1}{2^{n-1}}$$

On obtient alors l'expression explicite recherchée pour T :

$$T(n) = 2^{n+1} - 2$$

Si la tour est composée de 64 disques, à raison d'un disque par seconde, l'espérance de vie du Monde décrit par Lucas doit être de $2^{65} - 2$ secondes, soit un peu plus de mille milliards d'années. Ce problème rappelle également celui des grains de blés et de l'échiquier.

3.12 Exercice 3.12.

Q1. On suppose qu'un mot est défini par une suite de lettres précédée et succédée par un caractère blanc (en pratique, il faudrait également prendre en compte le cas où l'entité testée débute par une apostrophe). On écrit alors un programme de recherche, utilisant la fonction **SEARCH** de l'exemple 3 de la section 1.6, retournant la position de la première occurence du mot M dans la chaîne de caractères S :

SEARCH$(S\ \downarrow, M\ \downarrow)$: $[\![\ TS \leftarrow$ **LENGTH**(S); $TM \leftarrow$ **LENGTH**(M); SEARCH $\leftarrow -1$; $\left\{{}^{M}_{i:1}\right. TM - i < TS\ ?\ \overset{!}{*}\ |\, ¿\ \ j \leftarrow 1$; $\left\{{}^{/S_j = M_j}\right. j \leftarrow j+1$; $j > TM\ ?$ SEARCH $\leftarrow i$; $\overset{!}{*}\ |\, ¿\left.\right\}\left.\right\}_i\ ]\!]$

On peut alors écrire la fonction demandée, à l'aide de la récursivité en utilisant l'opérateur & de concaténation de chaînes de caractères :

$$\textbf{COUNT}(S\downarrow, M\downarrow) : [\![\ \text{COUNT} \leftarrow 0;\ M \leftarrow \text{" "} \ \&\ M\ \&\ \text{" "};\ i \leftarrow \text{SEARCH}(S, M);\ i < 0\,?\ \text{!}_{*}\ |\ \text{COUNT} \leftarrow \text{COUNT} + \textbf{COUNT}(\textbf{SUBSTR}(S, i, \textbf{LENGTH}(S) - i), M)\ ¿\ ]\!]$$

En supposant la taille du mot M fixée et indépendante de la longueur n de la chaîne de caractères, la complexité de l'algorithme obéit à l'équation de récurrence : $C(n+1) = C(n) + \Theta(1)$ (chaque appel de la fonction **SEARCH** étant effectué en temps constant), soit une complexité totale $C(n) = \Theta(n)$.

Q2. Toujours avec l'aide de la fonction **SEARCH**, on décompose la chaîne C en deux sous-chaînes de longueurs approximativement égales, puis on dénombre les occurrences du mot M dans chacune d'elles. On doit également tester s'il y a une occurrence sur la jonction.

$$\textbf{DPR}(S\downarrow, M\downarrow) : [\![\ N \leftarrow \textbf{LENGTH}(S);\ L \leftarrow \textbf{LENGTH}(M);\ \text{DPR} \leftarrow 0;\ N = 0\,?\ \text{!}_{*}\ |\ SG \leftarrow \textbf{SUBSTR}(S, 1, N/2);\ SD \leftarrow \textbf{SUBSTR}(S, N/2+1, N/2);\ \text{DPR} \leftarrow \textbf{DPR}(SG, M) + \textbf{DPR}(SD, M) + \textbf{DPR}(\textbf{SUBSTR}(\textbf{REV}(SG), 1, L-1)\ \&\ \textbf{SUBSTR}(SD, 1, L-1), M)\ ¿\ ]\!]$$

Chaque passage dans la fonction nécessite trois appels récursifs, dont 2 appels sur une donnée de taille $n/2$:

$$C(n) = 2C\Big(\frac{n}{2}\Big) + \Theta(1)$$

D'après le Master Theorem : $C(n) = \Theta(n)$ et l'application du principe diviser pour régner ne diminue pas la complexité de l'algorithme.

3.13 Exercice 3.13.

Le programme s'écrit directement à partir de l'algorithme d'Euclide :

$$\textbf{PGCD}(A\downarrow, B\downarrow) : [\![\ \text{PGCD} \leftarrow A;\ B = 0\,?\ \text{!}_{*}\ |\ \text{PGCD} \leftarrow \textbf{PGCD}(A, \textbf{MOD}(A, B))\ ¿\ ]\!]$$

Par défintion : $a = bq + r = bq + (a \bmod b)$ avec $q \in \mathbb{N}$. Donc : $a \bmod b = a - bq$. Considéreons deux cas :

- $b \leqslant \frac{a}{2}$: alors $a \bmod b < b \leqslant \frac{a}{2}$, ce qui montre l'inégalité recherchée.

- $b > \frac{a}{2}$: on a alors $q = \lfloor a/b \rfloor < \lfloor 2a/a \rfloor = 2$ et donc, comme $q \in \mathbb{N}$ on a nécessairement $q \in \{0, 1\}$. Par ailleurs, comme $b \geqslant a$, on a $q \geqslant 1$ et donc on en déduit que $q = 1$. Et donc par suite : $a \bmod b = a - bq = a - b < a - \frac{a}{2} < \frac{a}{2}$.

Donc : $a \bmod b \leqslant \frac{a}{2}$. On en déduit donc une inégalité sur la complexité (puisque C est une fonction croissante) en posant $n = a$ la taille du problème à résoudre et en comptabilisant le nombre d'appels à la fonction **MOD** :

$$C(n) = C(n \bmod b) + 1 \leqslant C\Big(\frac{n}{2}\Big) + 1$$

Plus rigoureusement, cela indique qu'il existe une fonction croissante $T : \mathbb{N} \to \mathbb{N}$, telle que $C(n) = \mathcal{O}(T(n))$ et obéissant à l'équation de récurrence :

$$T(n) = T\Big(\frac{n}{2}\Big) + 1$$

On obtient immédiatement (par le Master Theorem, voire par calcul direct avec la substitution $n \to 2^p$) que $T(n) = \Theta(\log n)$ et donc :

$$C(n) = \mathcal{O}(\log n)$$

L'algorithme d'Euclide a une complexité logarithmique.

Notons que suivant l'usage en théorie de la complexité, on pourrait comptabiliser la taille du problème à résoudre comme étant le nombre de chiffres dans la décomposition binaire de a, soit $n = \log_2 a$. Avec cette convention, l'équation sur la complexité T devient :

$$T(\log_2 a) = T\Big(\log_2\Big(\frac{a}{2}\Big)\Big) + 1 \Rightarrow T(n) = T(n-1) + 1$$

ce qui résulte en un comportement linéaire pour C : $C(n) = \mathcal{O}(n)$.

De même, si on décide de comptabiliser les opérations élémentaires effectuées sur chaque bit, on montre facilement qu'une division est aussi coûteuse qu'une multiplication, et la complexité d'une division euclidienne est donc intuitivement un $\Theta(n^2)$ du nombre n de bits considérés. Avec cette convention sur l'unité de coût, le nombre d'opérations devient un $\mathcal{O}(n) \times \Theta(n^2) = \mathcal{O}(n^3)$, soit une complexité cubique.

À nouveau donc, attention à bien savoir de quelles unités on parle lorsqu'on considère la complexité d'un algorithme.

3.14 Exercice 3.14.

Il y a n^3 passage dans la boucle externe, chacun nécessitant i^3 opérations, où i est l'indice de la boucle interne. Le nombre d'additions du programme s'évalue donc par :

$$C(n) = \sum_{i=1}^{n^3} i^3$$

L'annexe B nous montre que la somme des n premières puissances k-eme est équivalent à $\frac{n^{k+1}}{k+1}$, ce qui se montre facilement en évaluant l'intégrale de x^k entre 0 et n, puis en passant du continu au discret en encadrant le résultat par deux séries de Riemann. Donc :

$$C(n) \sim \frac{(n^3)^{3+1}}{3+1} = \Theta(n^7)$$

3.15 Exercice 3.15.

Q1. Soient $a, b \in \mathbb{R}_+$ et $L \in \mathbb{R}_+^n$ une liste triée. L'algorithme s'écrit itérativement :

SEARCH$(L\downarrow, a\downarrow, b\downarrow) : [\![$
$N \leftarrow$ **SIZE**(L); SEARCH $\leftarrow 0$; $\left\{ {}^{N}_{i:1} L_i \geqslant a\,?\ \ L_i \leqslant b\,?\ \ \text{SEARCH} \leftarrow i;\ \text{!} \mid \text{¿} \mid \text{¿} \right\}_i [\![$

L'algorithme balaye l'intégralité de la liste dans le pire des cas. Sa complexité (en fonction de la taille n de la liste, et en comptabilisant le nombre de tests booléens) est linéaire : $C(n) = \Theta(n)$.

Q2. on exploite cette fois-ci pleinement le fait que la liste est triée en procédant par dichotomie : on récupère l'élément en position médiane. Si sa valeur est comprise dans l'intervalle spécifié, on s'arrête. Sinon, si elle est inférieure à a (resp. supérieure à b), on applique récursivement la procédure de recherche sur la première (resp. seconde) moitié de la liste.

DPR$(L\downarrow, a\downarrow, b\downarrow) : [\![\ N \leftarrow$ **SIZE**(L); DPR $\leftarrow 0$; $M \leftarrow$ **FLOOR**$(N/2)$ $(L_M \geqslant a) \cap (L_M \leqslant a)\,?$ DPR $\leftarrow M$; $\mid L_M \leqslant a\,? \left\{ {}^{M}_{i:1} G_i \leftarrow L_i \right\}_i$ DPR $\leftarrow$ **DPR**(G, a, b) $\mid \left\{ {}^{N}_{i:M+1} D_{i-M} \leftarrow L_i \right\}_i$ DPR $\leftarrow$ **DPR**(D, a, b) ¿$]\!]$

où G et D représentent respectivement la première et seconde moitié de la liste L.

Avec les mêmes unités qu'à la question Q1, la complexité de la procédure obéit à l'équation de partition suivante :

$$C(n) = C\Big(\frac{n}{2}\Big) + \Theta(1)$$

On obtient alors immédiatement (Master Theorem ou calcul direct) que $C(n) = \Theta(\log n)$ et l'approche *diviser pour régner* est donc asymptotiquement plus efficace.

3.16 Exercice 3.16.

On considère un tableau T de n nombres entiers relatifs.

1) Commençons par écrire un algorithme naïf permettant de déterminer la somme maximale qui peut être obtenue avec des éléments consécutifs du tableau.

SUM1$(T\downarrow)$: $[\![\ N \leftarrow$ **SIZE**(T); SMAX $\leftarrow T_1$; imax $\leftarrow 1$; jmax $\leftarrow 1$; $\Big\{_{i:1}^{N}\Big\{_{j:i}^{N} S \leftarrow 0\ \Big\{_{k:i}^{j} S \leftarrow S + T_k;\Big\}_k$ $S \geqslant$ SMAX ? SMAX $\leftarrow S$; imax $\leftarrow i$; jmax $\leftarrow j$; | ¿$\Big\}_j\Big\}_i$
SUM1$_1 \leftarrow$ SMAX; SUM1$_2 \leftarrow$ imax; SUM1$_3 \leftarrow$ jmax; $]\!]$

On définit la taille du problème à partir du nombre n d'éléments dans le tableau à traiter et on considère l'addition comme opération élémentaire. Cet algorithme comporte 3 boucles imbriquées sans débranchement. On évalue donc la complexité par sommation directe des indices de boucle :

$$C(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \sum_{i=1}^{n}\Big[\frac{(n+i)(n-i+1)}{2} - i(n-i)\Big]$$

$$= \sum_{i=1}^{n}\frac{n^2+n-2in+i+2i^2}{2} \sim \frac{1}{2}\Big(n^3+n^2-2\frac{n^3}{2}+\frac{n^2}{2}+2\frac{n^3}{3}\Big) \sim \frac{n^3}{3} = \Theta(n^3)$$

où on ne considère que les monômes de plus grands degrés et on utilise l'équivalent $\sum_{i=1}^{n} i^k \sim (k+1)^{-1}n^{k+1}$ (cf annexe B4).

Remarque : la sommation d'équivalents peut parfois poser des problèmes. Ici, toutes les quantités sont positives et ne s'annulent pas en l'infini. On peut donc sommer les équivalents sans embûches. On peut aussi s'en sortir avec la notation o :

$$C(n) = \frac{1}{2}\Big(n^3 + n^2 - 2\frac{n^3}{2} + o(n^3) + \frac{n^2}{2} + o(n^2) + 2\frac{n^3}{3} + o(n^3)\Big) = \frac{n^3}{3} + o(n^3) = \Theta(n^3)$$

Alternativement, pour s'épargner le calcul des sommations, on peut résoudre le problème en supposant que n est une variable continue :

$$C(n) = \int_0^n \int_x^n \int_x^y 1 \times dxdydz = \int_0^n \int_x^n (y-x)dxdy = \int_0^n \Big(\frac{n^2-x^2}{2} - x(n-x)\Big)dx$$
$$= \frac{n^3}{2} - \frac{n^3}{6} - \frac{n^3}{2} + \frac{n^3}{3} = \frac{3n^3 - n^3 - 3n^3 + 2n^3}{6} = \frac{n^3}{6} = \Theta(n^3)$$

2) On écrit la fonction d'intégration :

INTEGR$(T\downarrow)$: $\Big[\!\Big[\ N \leftarrow \textbf{SIZE}(T); I_1 \leftarrow 0; \Big\{_{t:1}^{N} I_{t+1} \leftarrow I_t + T_t\Big\}_t \text{ INTEGR} \leftarrow I\ \Big]\!\Big]$

Cette fonction possède une seule boucle sans débranchement, ni modification du compteur, du pas ou de la borne finale. On trouve alors immédiatement une complexité en $\Theta(n)$.

Notons que le tableau résultat contient $N+1$ éléments. Par exemple, le tableau intégral de $T = [1,2,3]$ est $I = [0,1,3,6]$. On observe alors une propriété intéressante (analogue discrète du théorème fondamental de l'analyse) : la somme des éléments de T de i à j vaut la différence $I_{j+1} - I_i$. Par exemple : $T_2 + T_3 = 2 + 3 = 5 = 6 - 1 = T_4 - T_2$.

On modifie alors la fonction **SUM1** pour utiliser le tableau intégral :

SUM2$(T\downarrow)$: $\Big[\!\Big[\ N \leftarrow \textbf{SIZE}(T); I \leftarrow \textbf{INTEGR}(T);\ \text{SMAX} \leftarrow T_1; \text{imax} \leftarrow 1; \text{jmax} \leftarrow 1; \Big\{_{i:1}^{N} \Big\{_{j:i}^{N} S \leftarrow I_{j+1} - I_i; S \geqslant \text{SMAX}\,?\,\text{SMAX} \leftarrow S;\ \text{imax} \leftarrow i;\ \text{jmax} \leftarrow j;\ |\ ¿\Big\}_j\Big\}_i \text{SUM2}_1 \leftarrow \text{SMAX};\ \text{SUM2}_2 \leftarrow \text{imax};\ \text{SUM2}_3 \leftarrow \text{jmax};\ \Big]\!\Big]$

A l'aide du tableau intégral, le calcul de chaque somme d'entiers se fait en un temps $\Theta(1)$ au lieu d'un $\Theta(n)$ initialement. La complexité de l'algorithme devient alors :

$$C(n) = \Theta(n) + \sum_{i=1}^{n}\sum_{j=i}^{n} \Theta(1) = \Theta(n^2)$$

où le premier terme correspond au calcul du tableau intégral, et le second à son utilisation effective dans la recherche de la séquence maximale.

3) Dans cette question, on ne s'intéresse plus qu'à la valeur maximale qui peut être obtenue (et plus aux indices permettant effectivement d'obtenir cette valeur). Notons que l'on peut facilement modifier le code suivant pour prendre en compte la gestion des indices.

Suivant le paradigme de la stratégie *diviser pour régner*, on suppose connaître les sommes maximales S_1 et S_2 que l'on peut obtenir sur deux sous tableaux T_1 et T_2 de tailles identiques (à la partité de n près). La question consiste à déterminer la somme maximale S du tableau complet en cherchant à utiliser les deux informations S_1 et S_2. A ce stade, on note qu'il y a trois cas de figure, dont les deux premiers sont triviaux :

- La somme maximale du tableau complet ne contient que des éléments du sous tableau T_1. La somme maximale vaut donc immédiatement S_1.

- De la même manière, si la somme maximale du tableau complet ne contient que des éléments du sous tableau T_2, alors la somme maximale vaut S_2.

- Enfin, si la somme maximale est à cheval entre les deux tableaux T_1 et T_2, on peut utiliser une information précieuse : le dernier élément de T_1 et le premier élément de T_2 appartiennent à la séquence optimale. On peut alors chercher à construire la séquence maximale en partant de ces deux éléments et en se déplaçant itérativement vers les bords du tableau. Cette étape de recombinaison se fait en un temps linéaire (puisque chaque parcours de sous-tableau est indépendant), ce qui va garantir l'efficacité de cette approche.

Le principe global de la recherche consiste donc à tester l'hypothèse 3, puis à la comparer aux sommes partielles obtenues sur chacun des deux sous-tableaux. La solution globale sur le tableau complet est alors la meilleure des 3 solutions individuelles.

Notons que l'usage du tableau intégral de la question 2 n'est pas absolument nécessaire ici, mais nous l'utilisons pour simplifier le code, sans effet néfaste sur la complexité algorithmique.

Ecrivons le pseudo-code correspondant.

On suppose avoir à disposition une fonction **REV** qui renvoit l'image miroir d'une liste. Par exemple, pour $T = [1, 2, 3]$, REV $= [3, 2, 1]$. Nous laissons le soin au lecteur d'écrire cette fonction. On se contentera ici de remarquer qu'elle possède une complexité $\Theta(1)$ (si on considère comme unité de coût le nombre d'additions), voire $\Theta(n)$, si on préfère utiliser le nombre de lectures de cellules.

On commence par écrire une fonction permettant de trouver la somme maximale d'un tableau T, avec la contrainte que cette sommation doit partir du premier élément. Ayant calculé le tableau intégral I de T, on remarque que cette fonction est réduite à une recherche du maximum de I.

$$\textbf{SUM\_MAX}(T\downarrow) : [\![\ N \leftarrow \textbf{SIZE}(T);\ I \leftarrow \textbf{INTEGR}(T);\ M \leftarrow I_1;\ \left\{{}^{N+1}_{t:2}\right. I_t > M\,?\ M = I_t\ \mid\ ¿\Big\}_t \text{ SUM\_MAX} \leftarrow M\ ]\!]$$

Une fois le tableau complet divisé en deux sous-tableaux T_1 et T_2, le calcul d'une somme

à cheval sur les deux sous-tableaux est alors réduit à l'instruction :

$$\textbf{SUM\_MAX}(\textbf{REV}(T1)) + \textbf{SUM\_MAX}(T2)$$

où l'appel de la fonction **REV** dans le premier terme permet de prendre en compte le fait que la recherche dans $T1$ se fait en sens inverse à partir de son dernier élément.

On peut alors écrire le code complet :

SUM($T\downarrow$) : $[\![$ $N \leftarrow$ **SIZE**(T); ⓒ Gestion des cas de base ⓒ $N = 0$? SUM_MAX $\leftarrow$ 0; ❗$_*$ | ¿ $N = 1$? SUM_MAX $\leftarrow T_1$; ❗$_*$ | ¿

ⓒ Recherche sur chaque sous-tableau ⓒ $M \leftarrow N \div 2; \left\{ \begin{smallmatrix} M \\ \\ i:1 \end{smallmatrix} T1_i \leftarrow T_i \right\}_i \left\{ \begin{smallmatrix} N \\ \\ i:M+1 \end{smallmatrix} T2_{i-M} \leftarrow T_i \right\}_i$ $S1 \leftarrow$ **SUM**($T1$); $S2 \leftarrow$ **SUM**($T2$); $SM \leftarrow$ **MAX**($S1, S2$);

ⓒ Recherche d'une solution à cheval sur les 2 sous-tableaux ⓒ

$S12 \leftarrow$ **SUM_MAX**(**REV**($T1$)) + **SUM_MAX**($T2$); SUM_MAX $\leftarrow$ **MAX**($S12, SM$) $]\!]$

Evaluons la complexité de la fonction récursive **SUM**. On commence par former l'équation de partition : chaque étape nécessite deux appels récursifs sur des problèmes de taille $n/2$. Par ailleurs, le coût de décomposition/recomposition vaut $n/2 + n/2 = n$ additions pour former les deux tableaux intégraux. On obtient alors :

$$C(n) = 2C\Big(\frac{n}{2}\Big) + \Theta(n)$$

Une application directe du Master Theorem avec $a = b = 2$ et $f(n) = \Theta(n)$ nous permet d'obtenir la complexité : $n^{\log_b a} = n = \Theta(f(n))$, donc la complexité de la fonction vaut :

$$C(n) = n \log n$$

Notons qu'à chaque itération, le coût de décomposition/recomposition vaut identiquement n (dans tous les cas, on doit calculer chaque tableau intégral en entier). La complexité trouvée ci-dessus est donc valable en moyenne, dans le meilleur et dans le pire des cas.

Remarque. La notion de tableau intégral est couramment utilisée en traitement d'images pour accélérer des opérations de sommation par zones. On parle alors d'*images intégrales*.

3.17 Exercice 3.17.

Q1. On considère un ensemble de n canfidats : c_1, c_2, ... c_n. Supposons que deux candidats c_i et c_j aient été éliminés à l'issue de la phase de vote. Alors, c_i et c_j ont remporté chacun au moins $\lfloor n/2 \rfloor + 1$ votes. Le nombre de votes cumulés remportés par c_i et c_j est donc au moins égal à :

$$N = \lfloor n/2 \rfloor + 1 + \lfloor n/2 \rfloor + 1 = 2 \lfloor n/2 \rfloor + 2$$

On considère alors deux cas, suivant que n est pair ou impair :

- $n = 2k$
$$N = 2 \left\lfloor \frac{2k}{2} \right\rfloor + 2 = 2 \lfloor k \rfloor + 2 = 2k + 2 = n + 2 > n$$

- $n = 2k + 1$
$$N = 2 \left\lfloor \frac{2k+1}{2} \right\rfloor + 2 = 2 \left\lfloor k + \frac{1}{2} \right\rfloor + 2 = 2k + 2 = n + 1 > n$$

Dans les deux cas on aboutit à la contradiction $N > n$, et donc on a au maximum un candidat éliminé à l'issue de chaque phase.

Q2. On encode par $c_i = j$ la fait que le candidat c_i ait voté pour éliminer le candidat c_j.

IS_ELIMINATED$(C\downarrow, x\downarrow)$: $[\![$ $N \leftarrow$ **SIZE**(C); IS_ELIMINATED $\leftarrow$ ⸘; NB $\leftarrow 0$; $\left\{ \begin{smallmatrix} N \\ j:1 \end{smallmatrix} \right.$
$C_j = i$? NB $\leftarrow$ NB $+ 1$; | ¿$\Big\}_j$ NB $>$ **FLOOR**$(N/2) + 1$? IS_ELIMINATED $\leftarrow$ ⸮; $]\!]$

On peut optimiser le calcul en stoppant la boucle dès que le nombre minimal de votes est atteint, ce qui revient à migrer le test dans la boucle :

IS_ELIMINATED$(C\downarrow, x\downarrow)$: $[\![$ $N \leftarrow$ **SIZE**(C); IS_ELIMINATED $\leftarrow$ ⸘; NB $\leftarrow 0$; $\left\{ \begin{smallmatrix} N \\ j:1 \end{smallmatrix} \right.$
$C_j = i$? NB $\leftarrow$ NB $+ 1$; NB $>$ **FLOOR**$(N/2) + 1$? IS_ELIMINATED $\leftarrow$ ⸮; ❗ | ¿$\Big\}_j$ $]\!]$

Q3. Il suffit de lancer l'algorithme de la question $Q2$ pour tous les candidats x. Etant donné qu'un seul candidat au plus est éliminé, on stoppe la procédure dès qu'un candidat est éliminé.

ELIM$(C\downarrow)$: $[\![$
$N \leftarrow$ **SIZE**(C); ELIM $\leftarrow 0$ $\left\{ \begin{smallmatrix} N \\ i:1 \end{smallmatrix} \right.$**IS_ELIMINATED**$(C\downarrow, i\downarrow)$? ELIM $\leftarrow i$; ❗ | ¿$\Big\}_i$ $]\!]$

L'algorithme écrit à la question Q2 est de complexité $\Theta(n)$. Celui de la question Q3 est donc dans le pire des cas de complexité : $C(n) = n \times \Theta(n) = \Theta(n^2)$, soit une complexité quadratique.

Q4. On exploite le principe *diviser pour régner* : on divise le tableau C en deux sous-parties de tailles à peu près égales. On lance alors la procédure de test d'élimination sur chacune des deux sous-parties. On montre facilement qu'un candidat peut être éliminé à l'issue de la phase seulement s'il est éliminé dans au moins un des deux sous-tableaux. En effet, supposons qu'un candidat c_i ne soit éliminé dans aucun des deux sous-tableaux. Alors le nombre de votes récoltés au total est nécessairement inférieur à 2 fois la moitié de chaque sous-tableau. Ce n'est donc pas suffisant pour éliminer c_i. Il reste donc trois cas de figure à examiner :

- 1 candidat c_i est éliminé dans le premier sous-tableau
- 1 candidat c_i est éliminé dans le second sous-tableau
- 2 candidats c_i et c_j sont éliminés respectivement dans chacun des sous-tableaux

On peut facilement se convaincre que les deux premiers cas sont indentiques par symétrie du problème : il n'y a qu'un candidat potentiel qui puisse être éliminé. On le confirme alors en appelant la procédure **IS_ELIMINATED** sur le tableau complet pour le candidat c_i. De même dans le troisième cas, on appelle cette même procédure pour c_i et c_j. Notons que si $i = j$ alors, il n'y a pas besoin de tester, on sait immédiatement que c_i est éliminé.

L'écriture du programme ne pose aucune difficulté avec la brique **IS_ELIMINATED** :

DPR$(C\downarrow)$: $[\![$ $N \leftarrow$ **SIZE**(C); DPR $\leftarrow 0$; $N = 0\,?$ ❗ | ¿ $M \leftarrow$ **FLOOR**$(N/2)$;
$\left\{ G_i \leftarrow C_i \right\}_{i:1}^{M}$ $\left\{ D_{i-M} \leftarrow C_i \right\}_{i:M+1}^{N}$ $g \leftarrow$ **DPR**(G); $d \leftarrow$ **DPR**(D);
$(g = 0) \cap (d = 0)\,?$ ❗ | ¿

$g = d\,?$ DPR $\leftarrow g$; ❗ | ¿

$g = 0\,?$ **IS_ELIMINATED**$(C, d)\,?$ DPR $\leftarrow g$; ❗ | ¿

$d = 0\,?$ **IS_ELIMINATED**$(C, g)\,?$ DPR $\leftarrow d$; ❗ | ¿ $]\!]$

Chaque appel récursif nécessite deux appels de **DPR** sur une liste de taille deux fois moindre, ainsi que, dans le pire des cas, un appel à **IS_ELIMINATED**. L'équation de partition s'écrit donc :

$$C(n) = 2C\Big(\frac{n}{2}\Big) + \Theta(n)$$

dont on obtient la solution immédiatement à l'aide du Master Theorem :

$$C(n) = \Theta(n \log n)$$

Le paradigme *diviser pour régner* permet de passer d'une complexité quadratique à une complexité quasi-linéaire.

3.18 Exercice 3.18.

L'ensemble des entrées de l'algorithme est $\mathcal{D} = \mathbb{N}$, qui est un ensemble bien fondé lorsqu'il est munit de la relation d'ordre classique. On utilise donc la fonction identité $\varphi : n \to n$.

Par ailleurs, l'ensemble des cas de base de l'algorithme récursif est $\mathcal{B} = \{0, 1\}$. Pour tout $n \in \mathcal{B}$, l'algorithme termine et renvoit la valeur 1.

Soit $n \in \mathbb{N}$ ($n \notin \mathcal{B}$) un argument du programme. Si n est impair, alors la récursion exécute un nombre fini d'instruction (un test et quelques opérations arithmétiques) et appelle 3 fois le programme, respectivement avec les arguments de tailles $\frac{n}{2} + 1$, $\frac{n}{2}$ et $\frac{n}{2} - 1$. La décroissance stricte des tailles des arguments est assurée si et seulement si :

$$n > \frac{n}{2} + 1 \Leftrightarrow n > 2$$

On a supposé que $n \notin \mathcal{B}$ et par ailleurs que n est impair. Donc nécessairement $n > 2$ et l'appel récursif est donc bien effectué sur une donnée de tailel strictement inférieure.

Dans le cas où n est pair, la récursion appelle 2 fois le programme, respectivement avec les arguments de tailles $\frac{n}{2}$, et $\frac{n}{2} - 1$. On a bien : $n > \frac{n}{2}$ pour tout $n \notin \mathcal{B}$ donc la décroissance stricte est assurée aussi dans le cas impair.

Au bilan, nouas avons montré que chaque appel du programme exécute un nombre fini d'instructions, ainsi qu'un nombre fini (au maximum 3) d'appels récursifs sur des arguments de tailles strictement inférieures. Les techniques vues dans la section 2.2.3 nous indiquent donc que le programme se termine.

Pour montrer la correction, supposons la formule vérifiée jusqu'au rang $2n$ (en notant f_n la valeur de la suite de Fibonacci au rang n). En particulier :

$$f_{2n} = f_n^2 + f_{n-1}^2$$

$$f_{2n-1} = f_{n-1}(f_n + f_{n-2})$$

On doit vérifier que f_{2n+1} et f_{2n+2} vérifient également la relation.

$$f_{2n+1} = f_{2n} + f_{2n-1} = f_n^2 + f_{n-1}^2 + f_{n-1}(f_n + f_{n-2}) = f_n^2 + f_{n-1}^2 + f_{n-1}(2f_n + f_{n-1})$$

$$= f_n^2 + 2f_n f_{n-1} = f_n\Big(f_n + 2f_{n-1}\Big) = f_n\Big(f_{n+1} + f_{n-1}\Big)$$

En posant $N = 2n + 1$ on retrouve la relation du cas impair de l'algorithme :

$$f_N = f_{\lfloor N/2 \rfloor}\Big(f_{\lfloor N/2 \rfloor + 1} + f_{\lfloor N/2 \rfloor - 1}\Big)$$

Etudions à présent le cas pair :

$$f_{2n+2} = f_{2n+1} + f_{2n} = f_{2n+1} + f_n^2 + f_{n-1}^2 = f_n\Big(f_{n+1} + f_{n-1}\Big) + f_n^2 + f_{n-1}^2$$

$$= f_n\Big(2f_{n+1} - f_n\Big) + f_n^2 + f_{n-1}^2 = 2f_n f_{n+1} + f_{n-1}^2 = 2f_n f_{n+1} + (f_{n+1} - f_n)^2 = f_{n+1}^2 + f_n^2$$

À nouveau, en posant $N = 2n + 2$ on retrouve :

$$f_N = f_{\frac{N}{2}}^2 + f_{\frac{N}{2}-1}^2$$

Donc, f_{2n+1} et f_{2n+2} vérifient également la relation et la récurrence forte est vérifiée (on laisse le soin au lecteur de vérifier l'initialisation).

Dans le pire des cas, la relation de récurrence sur la complexité vérifie (cas impair) :

$$C(n) = 3C\Big(\frac{n}{2}\Big) + \Theta(1)$$

On a $n^{\log_2 3} = \Omega(1)$, donc la complexité de l'algorithme est un $\Theta(n^{1.58})$. Cet algorithme est plus rapide que la récursion linéaire naïve ($\Theta(\Phi^n)$), mais moins efficace que les méthodes par produit matriciel $\Theta(n)$ et par exponentiation matricielle rapide ($\Theta(\log n)$)).

3.19 Exercice 3.19.

Q1. On commence par écrire un module **INSERT** prenant en entrée un indice $i \in \mathbb{N}^*$ et une liste L que l'on supposera triée dans l'ordre croissant de l'élément 1 à l'élément i, et retournant la liste L triée jusqu'à l'élément $i+1$. Notons que le passage de l'argument L est fait par référence (pour éviter une copie potentiellement coûteuse de la liste).

$$\textbf{INSERT}(L\updownarrow, i\downarrow) : \left[\!\left[N \leftarrow \textbf{SIZE}(L);\ e \leftarrow L_i;\ \left\{ \begin{matrix} 1,-1 \\ k:i-1 \end{matrix} e < L_k\,?\, L_{k+1} \leftarrow L_k;\ L_k \leftarrow e;\ |\,!\,¿ \right\}_k \right]\!\right]$$

Dans le code ci-dessus on suppose implicitement que $i \leqslant N$. En toute rigueur, il faudrait ajouter un test pour s'en assurer et retourner un code d'erreur si ce n'est pas le cas. En notant n la taille de la liste L et en comptabilisant le nombre d'affectations, ce module **INSERT** possède une boucle effectuant au maximum n itérations, et donc $\Theta(n)$ comparaisons. On écrit alors dans un second temps la fonction de tri par insertion :

$$\textbf{SORT\_INSERT}(L\downarrow, i\downarrow) : \left[\!\left[N \leftarrow \textbf{SIZE}(L);\ \left\{ \begin{matrix} N \\ i:1 \end{matrix} \textbf{INSERT}(L, i) \right\}_i \right]\!\right]$$

Q2. L'algorithme de tri par insertion comporte une boucle effectuant exactement n itérations (dans tous les cas), chacune d'elle en un temps $\Theta(i)$, d'où la complexité de l'algorithme :

$$C(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

Cet algorithme de tri a donc une complexité (dans tous les cas, i.e. min ou max) quadratique. D'après le théorème d'encadrement, sa complexité moyenne est donc nécessairement quadratique.

Q3. Le tri par insertion est donc du même ordre que le tri par sélection ($\Theta(n^2)$) mais moins efficace que le tri fusion ($\Theta(n \log n)$).

Q4. Reprenons l'exercice avec l'affectation comme unité de coût : dans ce cas, la complexité dans le pire des cas est un $\Theta(n^2)$, mais dans le cas le plus favorable, aucune permutation n'est nécessaire (la liste est déjà triée) et donc le traitement se fait en temps constant ($\Theta(1)$). On ne peut pas conclure quand à la complxité moyenne avec le théorème d'encadrement. Il faut donc se munir d'une distribution de probabilités : supposons que toutes les permutations de liste soient équiprobables et notons X la variable aléatoire (discrète) dénotant le nombre de permutations à effectuer pour trier la liste. On peut décomposer X en une somme des nombres de permutation à effectuer à chaque passage dans la boucle principal de l'algorithme :

$$X = \sum_{i=1}^{n} X_i$$

où X_i est la variable aléatoire désignant le nombre de permutations à effectuer lors du i-eme passage dans la boucle. On cherche à évaluer :

$$\mathbb{E}[X] = \mathbb{E}\Big[\sum_{i=1}^{n} X_i\Big] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

Notons bien que la décomposition ci-dessus ne nécessite aucune hypothèse particulière [1] sur les variables aléatoires individuelles X_i.

Par ailleurs, l'appel de **INSERT** sur l'élément i requiert en moyenne $\frac{i}{2}$ permutations, donc $\mathbb{E}[X_i] = \Theta(i)$ d'où :

$$\mathbb{E}[X] = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

En moyenne, le tri par insertion nécessite un nombre quadratique d'affectations.

3.20 Exercice 3.20.

Q1. BUBBLE$(L \updownarrow)$: $[\![\ N \leftarrow$ **SIZE**(L);
$\Big\{_{i:1}^{N} \Big\{_{j:1}^{i-1} T_j > T_{j+1}\,?\ \ G \leftarrow T_j;\ T_j \leftarrow T_{j+1};\ T_{j+1} \leftarrow G;\ \ |\ ¿\Big\}_j\Big\}_i\ ]\!]$

Q2. Ce code contient deux boucles imbriquées sans débranchement, ni modificcations du compteur de boucle et/ou des bornes. Il termine donc.

Pour démontrer la correction, on utilise l'invariant de boucle suivant : *à l'issue de la i-eme itération de la boucle externe, les i plus grands éléments de la liste sont triés dans l'ordre croissant en fin de liste.* Notons P_i cette propriété.

Si on démontre cette propriété, son application en fin de boucle P_n démontre la correction de l'algorithme.

Montrons que P_i est vraie par récurrence (finie). P_1 est évidemment vérifiée : à chaque itération de la boucle interne, l'élément maximal de la sous-liste gauche est décalé vers la droite. A l'issue de l'itération, cet élément est placé à la fin de la liste et il est nécessairement trié : en partant de la définition qu'une liste est triée dans l'ordre croissant si et

1. Si ce n'est bien sûr l'existence des espérances individuelles des X_i qui ne pose aucun problème puisque les v.a. sont à support borné.

seulement si il n'existe par deux indices $i < j$ tels que $L_i > L_j$, alors, la liste cible étudiée par P_1 ne comporte qu'un élément et est donc nécessairement triée.

Supposons que pour un certain rang $i \in \{2, n-1\}$, P_i soit vérifiée (notons que si $i = n$ on peut stopper la récurrence puisqu'on a atteint la propriété à démontrer. Alors, par hypothèse, les i plus grands éléments de la liste sont triés dans l'ordre croissant en fin de liste. On peut donc considérer uniquement la partie gauche de la liste. Par application du tri à bulle, le plus grand élément de cette sous-liste est placé en position $i-1$. Les $i+1$ plus grands éléments de la liste totale sont donc bien en fin de liste. Par ailleurs, le dernier élément placé en position $i-1$ est plus petit que tous les éléments de i à n, qui eux-mêmes sont triés. Donc les $i+1$ derniers éléments sont triés, ce qui achève de montrer P_{i+1}.

Q3. L'algorithme de tri à bulle comporte une boucle effectuant exactement n itérations (dans tous les cas), chacune d'elle en un temps $\Theta(i)$, d'où la complexité de l'algorithme :

$$C(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

Cet algorithme de tri a donc une complexité (dans tous les cas, i.e. min ou max) quadratique. D'après le théorème d'encadrement, sa complexité moyenne est donc nécessairement quadratique.

Le tri à bulles est donc du même ordre que le tri par sélection et le tri par insertion ($\Theta(n^2)$) mais moins efficace que le tri fusion ($\Theta(n \log n)$).

3.21 Exercice 3.21.

Pour vérifier l'identité, il suffit simplement d'effectuer les opérations matricielles :

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix} = \begin{bmatrix} I_{\frac{n}{2}} & 0 \\ 0 & I_{\frac{n}{2}} \end{bmatrix}$$

$$\begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} I_{\frac{n}{2}} & 0 \\ 0 & I_{\frac{n}{2}} \end{bmatrix}$$

Q1. La multiplication classique de deux matrices $M = AB$ s'écrit :

$$M_{ij} = A_i^T B_j = \sum_{k=1}^{n} A_{ik} B_{kj}$$

ce qui nécessite $\Theta(n)$ multiplications flottantes. Ce calcul doit être effectué pour chacun des n^2 éléments de la matrice M, d'où la complexité totale : $C(n) = \Theta(n^3)$.

Q2. À nouveau il suffit de calculer les produits matriciels :

$$\begin{bmatrix} I_n & A & \\ & I_n & B \\ & & I_n \end{bmatrix} \begin{bmatrix} I_n & -A & AB \\ & I_n & -B \\ & & I_n \end{bmatrix} = \begin{bmatrix} I_n & & \\ & I_n & \\ & & I_n \end{bmatrix}$$

$$\begin{bmatrix} I_n & -A & AB \\ & I_n & -B \\ & & I_n \end{bmatrix} \begin{bmatrix} I_n & A & \\ & I_n & B \\ & & I_n \end{bmatrix} = \begin{bmatrix} I_n & & \\ & I_n & \\ & & I_n \end{bmatrix}$$

Q3. Supposons qu'il existe un algorithme de complexité $C(n) = \Theta(f(n))$ pour effectuer l'inversion matricielle et considérons deux matrices A et B de n éléments. On peut former la matrice de taille $3n$ ci-dessous :

$$M = \begin{bmatrix} I_n & A & \\ & I_n & B \\ & & I_n \end{bmatrix}$$

On peut alors calculer l'inverse M^{-1} de M en un temps $C(3n) = \Theta(f(3n))$. Or, on peut montrer que si f est de croissance au plus polynomiale (ce qui est le cas pour l'inversion matricielle), alors $f(an) = \Theta(f(n))$. En effet, en notant $f(n) = n^\alpha$ on a :

$$f(an) = (an)^\alpha = a^\alpha n^\alpha = \Theta(n^\alpha) = \Theta(f(n))$$

On dit que la classe polynomiale est stable par changement de variable $n \mapsto an$.

Par ailleurs, la réponse à la question précédente montre que la matrice M^{-1} contient le produit matriciel AB, ce qui prouve que ce produit peut être calculé en un temps $\Theta(f(n)) = \Theta(C(n))$.

L'inversion est donc au moins aussi difficile que la multiplication matricielle.

Q4. En exploitant le principe *diviser pour régner*, et en considérant une matrice de taille $n = 2p$ on peut procéder de la manière suivante :

- On divise la matrice à inverser en quatre matrices blocs de taille p : A, B, C et D.

- On calcule chacun des termes contenus dans la matrice suivante :

$$\begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

- Chaque terme de la matrice ci-dessus invoque (par récursivité) des inversions matricielles de taille $p = n/2$.

Au total, on dénombre deux inversions matricielles (celles de A et du complément de Schur de la matrice initiale $(D - CA^{-1}B)$), ce à quoi on doit ajouter un nombre constant de multiplications de complexité $\Theta(M(n))$. L'équation de partition sur la complexité s'écrit :

$$C(n) = 2C\Big(\frac{n}{2}\Big) + \Theta(M(n))$$

On a : $n^{\log_2 2} = n$. Par ailleurs, on sait que la multiplication matricielle est de complexité au moins quadratique (il faut a minima considérer les n^2 éléments de la matrice), donc $n^{\log_2 2} = \mathcal{O}(M(n))$ et donc le Master Theorem nous donne la solution de l'équation de partition précédente :

$$C(n) = \Theta(M(n)$$

Ce développement théorique nous montre qu'il est possible de concevoir un algorithme de calcul de l'inversion matricielle ayant même complexité asymptotique que la multiplication.

Q5. Nous avons vu que l'algorithme de Strassen permet d'effectuer une multiplication matricielle en un temps $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.808})$. L'inversion matricielle peut donc être théoriquement calculée en un temps $\Theta(n^{2.808})$, ce qui est meilleur que le pivot de Gauss avec sa complexité $\Theta(n^3)$. L'algorithme de Coppersmith-Winograd permet de multiplier deux matrices en un temps $\Theta(n^{2.376})$ et il en va donc de même pour l'inversion.

Un message pratique à retenir de cet exercice : inversion et multiplication sont donc en théorie des problèmes algorithmiques appartenant à la même classe de complexité. En pratique, les constantes multiplicatives sont telles que la multiplication est plus rapide que l'inversion, mais son temps de calcul n'est pas négligeable pour autant. Pour résoudre le système d'équations linéaire $AX = B$, on a tendance à suivre à la lettre la notation mathématique $X = A^{-1}B$ et donc à procéder de la manière suivante : on calcule l'inverse A^{-1} de A ($\Theta(n^3)$ opérations), puis on le multiplie par B (à nouveau $\Theta(n^3)$ opértations). Or, le calcul de l'inverse A^{-1} est justement déterminé par la résolution du système d'équations $AX = I$, lui même de complexité $\Theta(n^3)$. Donc, en général, mis à part dans quelques situations précises où on doit précacluler l'inverse de A pour le multiplier par un grand nombre de vecteurs B, il est plus efficace de résoudre directement le système $AX = B$ sans passer par l'inverse de A (ce qui s'écrit en général `solve(A,B)` informatiquement).

3.22 Exercice 3.22.

Q1. On laisse le soin au lecteur d'écrire une fonction **AIRE** prenant en entrée la liste des coordonnées des sommets du polygone et retournant son aire. On pourra utiliser la formule de géométrie analytique :

$$\mathcal{A} = \frac{1}{2}\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

Pour pouvoir calculer la surface de plancher constructible, on doit pouvoir associer chaque parcelle du terrain à sa hauteur, ce qui signifie qu'on doit d'une manière ou d'une autre, être en mesure de calculer toutes les partitions possibles entre les bâtiments. Dans le cadre le plus général, pour un ensemble n de bâtiments, il y a 2^n combinaisons possibles des intersections, ce qui présage donc un algorithme de complexité exponentielle.