UE d'Ouverture Algorithmique

Première partie
Diviser pour régner

Rappel du théorème maître

Théorème 1. *(Résolution de récurrences « diviser pour régner »). Soient $a \geq 1$ et $b > 1$ deux constantes, soient f une fonction à valeurs dans $\mathbb{R}^+$ et T une fonction de $\mathbb{N}^*$ dans $\mathbb{R}^+$ vérifiant, pour tout n suffisamment grand, l'encadrement suivant : $aT(\lfloor n/b \rfloor) + f(n) \leq T(n) \leq aT(\lceil n/b \rceil) + f(n)$. Alors T peut être bornée asymptotiquement comme suit :*

1. *Si $f(n) = O(n^{(\log_b a)-\varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si on a asymptotiquement pour une constante $c < 1$, $af(n/b) < cf(n)$, alors $T(n) = \Theta(f(n))$.*

1 Fonctions définies par récurrence

Exercice 1.1. Soit un algorithme dont la complexité $T(n)$ est donnée par la relation de récurrence :

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 3\,T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2. \end{aligned}$$

a. Calculer $T(n)$ en résolvant la récurrence.

b. Déterminer $T(n)$ à l'aide du théorème maître.

Remarque :
On a

$$a^{\log_x b} = b^{\log_x a} \quad \text{et} \quad \sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}.$$

Exercice 1.2. Le but de cet exercice est d'utiliser le théorème maître pour donner des ordres de grandeur asymptotique pour des fonctions définies par récurrence.

1. En utilisant le théorème maître, donner un ordre de grandeur asymptotique pour $T(n)$:

a.

$$T(1) = 1, \quad T(n) = 2\,T\left(\frac{n}{2}\right) + n^2;$$

b.

$$T(1) = 0, \quad T(n) = 2\,T\left(\frac{n}{2}\right) + n;$$

c.

$$T(1) = 1, \quad T(n) = 2\,T\left(\frac{n}{2}\right) + \sqrt{n};$$

d.

$$T(1) = 0, \quad T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

2. Essayer de donner des exemples d'algorithmes dont le coût est calculé par l'une de ces récurrences.

Exercice 1.3. Le but de cet exercice est de choisir l'algorithme de type « diviser pour régner » le plus rapide pour un même problème.

1. Pour résoudre un problème manipulant un nombre important de données, on propose deux algorithmes :

- **a.** un algorithme qui résout un problème de taille n en le divisant en 2 sous-problèmes de taille $n/2$ et qui combine les solutions en temps quadratique,
- **b.** un algorithme qui résout un problème de taille n en le divisant en 4 sous-problèmes de taille $n/2$ et qui combine les solutions en temps $O(\sqrt{n})$.

Lequel faut-il choisir ?

2. Même question avec les algorithmes suivants :

- **a.** un algorithme qui résout un problème de taille n en le réduisant à un sous-problème de taille $n/2$ et qui combine les solutions en temps $\Omega(\sqrt{n})$,
- **b.** un algorithme qui résout un problème de taille n en le divisant en 2 sous-problèmes de taille $n/2$ et qui combine les solutions en temps constant.

Exercice 1.4. Considérons maintenant un algorithme dont le nombre d'opérations sur une entrée de taille n est donné par la relation de récurrence :

$$T(1) = 0 \quad \text{et} \quad T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2\,n}{3}\right) + n.$$

1. Construire un arbre représentant les appels récursifs de l'algorithme. En déduire la complexité de $T(n)$.

2. Vérifier le résultat par récurrence (méthode par « substitution »).

3. Cette récurrence peut-elle être résolue en utilisant le théorème maître ?

2 Recherche d'éléments, de suite d'éléments

Exercice 2.1. Donnez un algorithme de recherche d'un élément dans un tableau **trié** (dans l'ordre croissant) :

- **a.** en utilisant une recherche séquentielle,
- **b.** en utilisant le principe « diviser pour régner ».

Donnez la complexité (en nombre de comparaisons) dans le pire cas de chacun de ces deux algorithmes en fonction de la taille n du tableau.

Exercice 2.2. On cherche le k-ième plus petit élément dans un tableau **non trié**.

- **a.** Proposer un algorithme utilisant le principe « diviser pour régner » en s'inspirant de celui du tri rapide.
- **b.** Faire la trace de l'algorithme sur le tableau | 2 | 4 | 1 | 6 | 3 | 5 | pour $k = 4$.
- **c.** Donnez la complexité (en nombre de comparaisons) dans le pire cas de cet algorithme en fonction de la taille n du tableau.
- **d.** Que se passe-t-il si, à chaque étape, le tableau considéré est coupé en 2 sous-tableaux de tailles équivalentes ?

Exercice 2.3. On dispose d'un tableau d'entiers *relatifs* de taille n. On cherche à déterminer la suite d'entrées consécutives du tableau dont la somme est maximale. Par exemple, pour le tableau $T = [-1, 9, -3, 12, -5, 4]$, la solution est 18 (somme des éléments de $T[2..4] = [9, -3, 12]$).

- **a.** Proposer un algorithme élémentaire pour résoudre ce problème. Donner sa complexité.
- **b.** Donner un (meilleur) algorithme de type « diviser pour régner ». Quelle est sa complexité ?

Exercice 2.4. Un élément x est majoritaire dans un ensemble E de n éléments si E contient strictement plus de $n/2$ occurrences de x. La seule opération dont nous disposons est la comparaison (égalité ou non) de deux éléments.

Le but de l'exercice est d'étudier le problème suivant : étant donné un ensemble E, représenté par un tableau, existe-t-il un élément majoritaire, et si oui quel est-il ?

1. Algorithme naïf

- **a.** Écrivez un algorithme NombreOccurrences qui, étant donné un élément x et deux indices i et j, calcule le nombre d'occurrences de x entre $E[i]$ et $E[j]$. Quelle est sa complexité (en nombre de comparaisons) ?
- **b.** Écrivez un algorithme Majoritaire qui vérifie si un tableau E contient un élément majoritaire, et si oui le retourne. Quelle est sa complexité ?

2. Algorithme de type « diviser pour régner »
Pour calculer l'élément majoritaire de l'ensemble E (s'il existe), on découpe l'ensemble E en deux sous-ensembles E_1 et E_2 de même taille, et on calcule récursivement dans chaque sous-ensemble l'élément majoritaire. Pour qu'un élément x soit majoritaire dans E, il suffit que l'une des conditions suivantes soit vérifiée :
— x est majoritaire dans E_1 et dans E_2,
— x est majoritaire dans E_1 et non dans E_2, mais suffisamment présent dans E_2,
— x est majoritaire dans E_2 et non dans E_1, mais suffisamment présent dans E_1.
Écrivez un algorithme utilisant cette approche « diviser pour régner ». Analysez sa complexité.

3. Pour améliorer l'algorithme précédent, on propose de construire un algorithme PseudoMajoritaire tel que :

— soit l'algorithme garantit que E ne possède pas d'élément majoritaire,
— soit il rend un couple (x, p) tel que $p > n/2$, x est un élément apparaissant au plus p fois dans E, et tout élément $y \neq x$ de E apparaît au plus $n - p$ fois dans E.

Donnez un algorithme récursif vérifiant ces conditions. Donnez sa complexité. Déduisez-en un algorithme de recherche d'un élément majoritaire, et donnez sa complexité.

3 Tris

Exercice 3.1 (Tri fusion). Le but de l'exercice est de trier un tableau en utilisant l'approche « diviser pour régner » : on commence par couper le tableau en deux, on trie chaque moitié avec notre algorithme, puis on fusionne les deux moitiés.

1. Écrire un algorithme Fusion(A, B) permettant de fusionner deux tableaux A et B déjà triés de tailles n_A et n_B en un seul tableau trié T. Quelle est sa complexité ?

2. À l'aide de l'algorithme décrit ci-dessus, écrire l'algorithme du tri fusion.

3. Quelle est la complexité du tri fusion ?

4 Algèbre et arithmétique

Exercice 4.1 (Multiplication de deux polynômes). On représente un polynôme $p(x) = \sum_{i=0}^{m-1} a_i x^i$ par la liste de ses coefficients $[a_0, a_1, \dots, a_{m-1}]$. Nous nous intéressons au problème de la multiplication de deux polynômes : étant donnés deux polynômes $p(x)$ et $q(x)$ de degré au plus $n - 1$, calculer $pq(x) = p(x)q(x) = \sum_{i=0}^{2n-2} c_i x^i$.

1. Préliminaires : énumérez les opérations basiques (de coût 1) qui serviront à calculer la complexité de vos algorithmes. Vérifiez que l'addition de deux polynômes de degré $n - 1$ se fait en $\Theta(n)$.

2. Méthode directe.
Donnez un algorithme calculant directement chaque coefficient c_i. Quelle est sa complexité ?

3. On découpe chaque polynôme en deux parties de tailles égales :

$$p(x) = p_1(x) + x^{n/2} p_2(x),$$
$$q(x) = q_1(x) + x^{n/2} q_2(x),$$

et on utilise l'identité

$$pq = p_1 q_1 + x^{n/2}(p_1 q_2 + p_2 q_1) + x^n p_2 q_2.$$

Donnez un algorithme récursif, utilisant le principe « diviser pour régner », qui calcule le produit de deux polynômes à l'aide de l'identité précédente. Si $T(n)$ est le coût de la multiplication de deux polynômes de taille n, quelle est la relation de récurrence vérifiée par $T(n)$? Quelle est la complexité de votre algorithme ? Comparez-la à la complexité de la méthode directe.

4. On utilise maintenant la relation

$$p_1 q_2 + p_2 q_1 = (p_1 + p_2)(q_1 + q_2) - p_1 q_1 - p_2 q_2.$$

Donnez un algorithme calculant le produit pq à l'aide de cette relation, et explicitez le nombre de multiplications et d'additions de polynômes de degré $n/2$ utilisées. Quelle est la complexité de votre algorithme ?

5. Pensez-vous que cet algorithme a une complexité optimale ?

Exercice 4.2 (Algorithme de Strassen pour la multiplication de deux matrices)**.** La multiplication de matrices est très fréquente dans les codes numériques. Le but de cet exercice est de proposer un algorithme rapide pour cette multiplication.

1. Donnez un algorithme direct qui multiplie deux matrices A et B de taille $n \times n$. Donnez sa complexité en nombre de multiplications élémentaires.

2. Notons $C = A\,B$. Dans le but d'utiliser le principe « diviser pour régner », on divise les matrices A, B et C en quatre sous-matrices $n/2 \times n/2$

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = A\,B.$$

Les sous-matrices de C peuvent être obtenues en effectuant les produits et additions de matrices $n/2 \times n/2$ suivants :

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21}, \qquad & c_{12} &= a_{11}b_{12} + a_{12}b_{22}, \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21}, \qquad & c_{22} &= a_{21}b_{12} + a_{22}b_{22}. \end{aligned}$$

Notons $T(n)$ le coût de la multiplication de deux matrices de taille n, quelle est la relation de récurrence vérifiée par $T(n)$ dans ce cas ? Donnez la solution de cette relation de récurrence, comparez-la à la complexité de la méthode directe.

3. Strassen (en 1969) a suggéré de calculer les produits intermédiaires suivants :

$$\begin{aligned} m_1 &= (a_{12} - a_{22})\,(b_{21} + b_{22}), \qquad & m_4 &= (a_{11} + a_{12})\,b_{22}, \\ m_2 &= (a_{11} + a_{22})\,(b_{11} + b_{22}), \qquad & m_5 &= a_{11}\,(b_{12} - b_{22}), \\ m_3 &= (a_{11} - a_{21})\,(b_{11} + b_{12}), \qquad & m_6 &= a_{22}\,(b_{21} - b_{11}), \\ & & m_7 &= (a_{21} + a_{22})\,b_{11}, \end{aligned}$$

puis de calculer C comme suit :

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6, \qquad & c_{12} &= m_4 + m_5, \\ c_{21} &= m_6 + m_7, \qquad & c_{22} &= m_2 - m_3 + m_5 - m_7. \end{aligned}$$

a. Montrez que C est bien le produit de A par B (attention, la multiplication de deux matrices n'est pas commutative !).

b. Donnez la relation de récurrence vérifiée par $T(n)$. Déduisez-en la complexité de l'algorithme de Strassen.

c. Pensez-vous que cet algorithme a une complexité optimale ?

Exercice 4.3 (Matrices Toeplitz)**.** Une matrice Toeplitz est une matrice de taille $n \times n$ (a_{ij}) telle que $a_{i,j} = a_{i-1,j-1}$ pour $2 \le i, j \le n$.

1. La somme de deux matrices Toeplitz est-elle une matrice Toeplitz ? Et le produit ?

2. Trouver un moyen d'additionner deux matrices Toeplitz en $O(n)$.

3. Comment calculer le produit d'une matrice Toeplitz $n \times n$ par une vecteur de longueur n ? Quelle est la complexité de l'algorithme ?

5 Géométrie algorithmique

Exercice 5.1 (Recherche des deux points les plus rapprochés)**.** Le problème consiste à trouver les deux points les plus rapprochés (au sens de la distance euclidienne classique) dans un ensemble P de n points. Deux points peuvent coïncider, auquel cas leur distance vaut 0. Ce problème a notamment des applications dans les systèmes de contrôle de trafic : un contrôleur du trafic aérien ou maritime peut avoir besoin de savoir quels sont les appareils les plus rapprochés pour détecter des collisions potentielles.

1. Donnez un algorithme naïf qui calcule directement les deux points les plus rapprochés, et analysez sa complexité (on représentera P sous forme de tableau et on analysera la complexité de l'algorithme en nombre de comparaisons).

Nous allons construire un algorithme plus efficace basé sur le principe « diviser pour régner ». Le principe est le suivant :

a. Si $|P| \leq 3$, on détermine les deux points les plus rapprochés par l'algorithme naïf.

b. Si $|P| > 3$, on utilise une droite verticale Δ, d'abscisse l, séparant l'ensemble P en deux sous ensembles (Pgauche et Pdroit) tels que l'ensemble Pgauche contienne la moitié des éléments de P et l'ensemble Pdroit l'autre moitié, tous les points de Pgauche étant situés à gauche de ou sur Δ, et tous les points de Pdroit étant situés à droite de ou sur Δ.

c. On résout le problème sur chacun des deux sous-ensembles Pgauche et Pdroit, les deux points les plus rapprochés sont alors :
— soit les deux points les plus rapprochés de Pgauche,
— soit les deux points les plus rapprochés de Pdroit,
— soit un point de Pgauche et un point de Pdroit.

On supposera qu'initialement l'ensemble P est trié selon des abscisses et ordonnées croissantes ; en d'autres termes l'algorithme prend la forme : PlusProche(PX, PY), où PX est un tableau correspondant à l'ensemble P trié selon les abscisses (et selon les ordonnées pour les points d'abscisses égales) et PY à l'ensemble P trié selon les ordonnées. Les ensembles P, PX et PY contiennent les mêmes points $P[i]$ (seul l'ordre change).

2. Écrivez un algorithme DiviserPoints qui calcule à partir de P les tableaux PgaucheX (resp. PgaucheY) correspondant aux points de l'ensemble Pgauche triés selon les abscisses (resp. les ordonnées), de même pour les tableaux PdroitX et PdroitY. Montrez que cette division peut être effectuée en $O(n)$ comparaisons.

3. Notons δ_g (resp. δ_d) la plus petite distance entre deux points de Pgauche (resp. Pdroit), et $\delta = \min(\delta_g, \delta_d)$. Il faut déterminer s'il existe une paire de points dont l'un est dans Pgauche et l'autre dans Pdroit, et dont la distance est strictement inférieure à δ.

a. Si une telle paire existe, alors les deux points se trouvent dans le tableau PY', trié selon les ordonnées, et obtenu à partir de PY en ne gardant que les points d'abscisse x vérifiant $l - \delta \leq x \leq l + \delta$. Donnez un algorithme qui calcule PY' en $O(n)$ comparaisons.

b. Montrez que si 5 points sont situés à l'intérieur ou sur le bord d'un carré de coté $a > 0$, alors la distance minimale entre 2 quelconques d'entre eux est strictement inférieure à a.

c. Montrez que s'il existe un point $p_g = (x_g, y_g)$ de Pgauche et un point $p_d = (x_d, y_d)$ de Pdroit tels que $\text{dist}(p_g, p_d) < \delta$ et $y_g < y_d$, alors p_d se trouve parmi les 7 points qui suivent p_g dans le tableau PY'.

d. Donnez un algorithme, en $O(n)$ comparaisons, qui vérifie s'il existe une paire de points dans Pgauche et Pdroit dont la distance est strictement inférieurs à δ, et si oui la retourne.

4. En déduire un algorithme recherchant 2 points à distance minimale dans un ensemble de n points.

5. Donnez la relation de récurrence satisfaite par le nombre de comparaisons effectuées par cet algorithme. En déduire sa complexité.

Deuxième partie

Programmation dynamique

6 Pour débuter

Exercice 6.1. Sous-séquence de plus grande somme.
On dispose d'un tableau d'entiers *relatifs* de taille n. On cherche à déterminer la suite d'entrées consécutives du tableau dont la somme est maximale. Par exemple, pour le tableau $T = [5, 15, -30, 10, -5, 40, 10]$, la somme maximale est 55 (somme des éléments $[10, -5, 40, 10]$).

1. Dans un premier temps, on s'intéresse uniquement à la valeur de la somme de cette sous-séquence.

  a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

  b. Caractériser (par une équation) cette sous-structure optimale.

  c. En déduire la valeur de la somme maximale.

  d. Écrire un algorithme de programmation dynamique pour calculer cette somme.

  e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également les indices de début et de fin de la somme. Sa complexité est-elle modifiée ?

Exercice 6.2. Plus grande sous-suite croissante.
Soit une séquence de nombre $s = x_1, ..., x_n$. Une sous-suite est un sous-ensemble de ces nombres pris dans l'ordre, c'est à dire une séquence de la forme : $x_{i_1}, x_{i_2}, ..., x_{i_k}$ telle que $1 \leq i_1 < i_2 < ... < i_k \leq n$. Une sous-suite est dite croissante si les x_i sont croissants.
On cherche à déterminer la plus longue sous-suite croissante de s. Par exemple, pour (2,1,7,5,2,4,8,6), une solution est (1,2,4,6).

1. Dans un premier temps, on s'intéresse uniquement à la longueur de cette sous-suite.

  a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

  b. Caractériser (par une équation) cette sous-structure optimale.

  c. En déduire la longueur de la sous-suite croissante maximale.

  d. Écrire un algorithme de programmation dynamique pour calculer longueur.

  e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également les indices des éléments de la suite. Sa complexité est-elle modifiée ?

Exercice 6.3. Plus grande sous-chaîne commune.
Étant données deux chaînes de caractères $x = x_1x_2...x_n$ et $y = y_1y_2...y_m$, on cherche à déterminer la plus longue sous-chaîne commune (lettres contigües). Cela revient à dire que l'on cherche des indices i, j et k tels $x_ix_{i+1}...x_{i+k-1} = y_jy_{j+1}...y_{j+k-1}$ avec k maximal.

1. Dans un premier temps, on s'intéresse uniquement à la longueur de cette sous-chaîne commune.

  a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

b. Caractériser (par une équation) cette sous-structure optimale.

c. En déduire la longueur maximale d'une sous-chaîne commune.

d. Écrire un algorithme de programmation dynamique pour calculer cette longueur maximale.

e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également i,j et k. Sa complexité est-elle modifiée ?

Exercice 6.4. Sac-à-dos, avec répétitions.
Lors du cambriolage d'une bijouterie, le voleur s'aperçoit qu'il ne peut pas tout emporter dans son sac-à-dos... Son sac peut supporter, au maximum, P kilos de marchandise (on supposera P entier). Or, dans cette bijouterie, se trouvent n objets différents, chacun en quantité illimitée. Chaque objet x_i à un poids p_i et une valeur v_i.
Quelle combinaison d'objets, transportable dans le sac, sera la plus rentable pour notre voleur ?
Exemple : pour $P = 10$ et les objets suivants :

objet	poids	valeur
x_1	6	30 €
x_2	3	14 €
x_3	4	16 €
x_4	2	9 €

le meilleur choix est de prendre un x_1 et deux x_4, ce qui fait un sac à 48 €.

1. Dans un premier temps, on s'intéresse uniquement à la valeur maximale que pourra emporter le voleur.

a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

b. Caractériser (par une équation) cette sous-structure optimale.

c. En déduire le montant maximal des objets volés.

d. Écrire un algorithme de programmation dynamique pour calculer cette valeur maximale.

e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également la combinaison d'objets volés. Sa complexité est-elle modifiée ?

Exercice 6.5. Ensembles indépendants dans des arbres.
Un sous-ensemble de noeuds dans un graphe $G = (V, E)$ est dit *indépendant* s'il n'existe pas d'arête entre eux. Trouver le plus grand ensemble indépendant dans un graphe est un problème difficile. Par contre, si le graphe est un arbre, c'est un sous-problème plus facile. On cherche donc l'ensemble indépendant de plus grande taille dans un arbre A.
On note $r(A)$, la racine de A et $fils(A)$, la liste des fils de A. On pourra également noter $A(s)$, le sous-arbre de A dont la racine est s.

1. Dans un premier temps, on s'intéresse uniquement à la taille maximale d'un sous-ensemble.

a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

b. Caractériser (par une équation) cette sous-structure optimale.

c. En déduire la taille maximale d'un ensemble indépendant de A.

d. Écrire un algorithme de programmation dynamique pour calculer cette taille maximale.

e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également la liste des noeuds. Sa complexité est-elle modifiée ?

Exercice 6.6. Texte corrompu.
Soit $s[1...n]$ un chaîne de caractères sans aucun espace (ex : Ilétaitunefoisuneprincesse...). On cherche à savoir si cette chaîne correspond à un texte lisible dont on aurait effacé les espaces et le cas échéant, on veut pouvoir reconstituer ce texte. Pour cela, on dispose d'un dictionnaire un peu particulier : il s'agit d'une fonction qui, étant donné une chaîne de caractères t quelconque, renvoie VRAI si t représente un mot correct :

$$dict(t) = \begin{cases} VRAI & \text{si } t \text{ est un mot correct} \\ \text{FAUX} & \text{sinon} \end{cases}$$

1. Dans un premier temps, on cherche uniquement à savoir si s correspond à un texte valide.

a. Quelle est la sous-structure optimale dont on a besoin pour résoudre ce problème ?

b. Caractériser (par une équation) cette sous-structure optimale.

c. Comment détermine-t-on si s correspond à un texte correct ?

d. Écrire un algorithme de programmation dynamique pour le faire.

e. Quelle est la complexité de cet algorithme ?

2. Modifier l'algorithme pour qu'il renvoie également les positions des espaces. Sa complexité est-elle modifiée ?

7 Ordonnancement

Exercice 7.1. Le but de cet exercice est de trouver un ordonnancement optimal de chaînes de montage. Considèrons un atelier de production comportant deux chaînes de montage. Chaque chaîne comporte n postes, notées $S_{i,j}$ ($i = 1, 2$, $j = 1, \ldots, n$). Les postes $S_{1,j}$ et $S_{2,j}$ font le même travail, mais les temps de montage peuvent varier. Le temps de montage au poste $S_{i,j}$ est $a_{i,j}$. Une pièce brute arrive sur une chaîne, passe par les postes 1 à n où elle subit un traitement, puis sort par l'autre extrémité de la chaîne de montage. Le temps de passage d'un poste à l'autre est négligeable sur une même chaîne, mais une pièce peut être transférée du poste $S_{i,j}$ au poste $S_{3-i,j+1}$ moyennant un délai $t_{i,j}$. Une pièce met un temps e_i à arriver sur le premier poste de la chaîne i, et x_i à sortir de la chaîne i. Le problème de l'ordonnancement de chaînes de montage consiste à déterminer les postes à sélectionner sur les chaînes 1 et 2 pour minimiser le délai de transit d'une pièce à travers tout l'atelier. La complexité des algorithmes sera comptée en nombre d'additions.

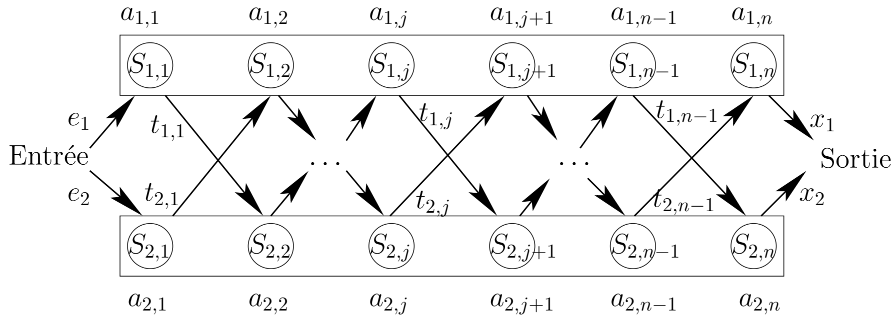


FIGURE 1 – Ordonnancement de chaînes de montage

1. Structure du chemin optimal
Soit $C_j^1 = [S_{i_1,1}, S_{i_2,2}, \dots, S_{i_{j-1},j-1}, S_{1,j}]$ un chemin optimal vers le poste $S_{1,j}$, caractérisez les sous-chemins $C_{j-1} = [S_{i_1,1}, S_{i_2,2}, \dots, S_{i_{j-1},j-1}]$ possibles. Caractérisez de même les chemins optimaux vers le poste $S_{2,j}$.

2. Solution récursive
Notons $f_{i,j}$ le délai le plus court possible avec lequel une pièce sort du poste $S_{i,j}$, et f^* le plus court délai pour qu'une pièce traverse tout l'atelier. Exprimez f^* en fonction des $f_{i,j}$. Ecrivez une relation de récurrence vérifiée par $f_{i,j}$.

3. Calcul du temps optimal
Quelle serait la complexité d'un algorithme récursif calculant f^* à partir des relations de récurrence précédentes ? Donnez un algorithme utilisant de la programmation dynamique pour calculer f^*. Quelle est sa complexité dans le pire cas ?

4. Construction du chemin optimal
Donnez un algorithme renvoyant la séquence des postes utilisés par un chemin optimal traversant l'atelier. Quelle est sa complexitè dans le pire cas ?

8 Géométrie algorithmique

Exercice 8.1. Le but de cet exercice est de trouver une triangulation de polygones optimale pour une fonction de pondération donnée Etant donnés $n \geq 3$ points du plan $v_1, \dots, v_n$, le *polygone* $v_1 \cdots v_n$ est la figure constituée des n segments $[v_1 v_2]$, $[v_2 v_3]$, ..., $[v_{n-1} v_n]$, $[v_n v_1]$ (voir les exemples de la figure 8.1). Les v_i sont les *sommets* du polygone, et les segments $[v_i v_{i+1}]$ sont les *côtés* du polygone. Les segments $[v_i v_k]$ qui ne sont pas des côtés sont appelés des *cordes* du polygone. Un polygone est *simple* si ses côtés ne se coupent pas, et croisé sinon.

Un polygone simple découpe le plan en deux parties : l'intérieur et l'extérieur du polygone. Un polygone simple est *convexe* si toutes ses cordes sont à l'intérieur du polygone.

Dans cet exercice nous ne considérons que des polygones convexes. Une triangulation d'un polygone (convexe) est un ensemble T de cordes qui divisent le polygone en triangles disjoints et ne se coupent pas. La figure suivante montre deux triangulations possibles d'un polygone à 7 côtés.

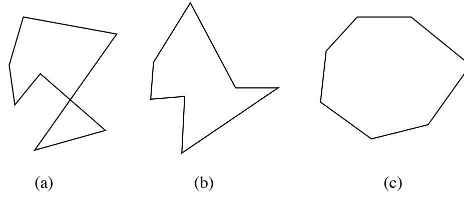
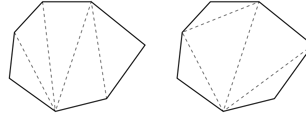


FIGURE 2 – Polygones à 7 sommets. (a) croisé (b) simple (c) convexe



1. Montrer qu'une triangulation d'un polygone à n côtés a $(n-3)$ cordes et $(n-2)$ triangles.

2. Structure du chemin optimal, Solution récursive

Soit $P = v_1 \cdots v_n$ un polygone convexe, et w une fonction de pondération définie sur les triangles formés par les côtés et les cordes de P. Un exemple naturel de fonction de pondération est le périmètre du triangle :

$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

où $|v_i v_j|$ est la distance euclidienne entre les sommets v_i et v_j.

À toute triangulation T de P on peut donc associer une pondération totale, qui est égale à la somme des pondérations des triangles de T. Le problème de la triangulation *optimale* du polygone P pour la pondération w est de trouver une triangulation de pondération totale minimale.

Notons $t[i, j]$ la pondération totale d'une triangulation optimale du polygone $v_i \cdots v_j$, pour $1 \leq i < j \leq n$, et posons $t[i, i+1] = 0$ pour tout $1 \leq i \leq n$. La pondération totale d'une triangulation optimale de P vaut $t[1, n]$.

Soit T une triangulation optimale de P, et soit v_k, $2 \leq k \leq n-1$ le troisième sommet du triangle contenant v_1 et v_n. Donnez une formule reliant la pondération totale de T à la pondération $w(v_1, v_k, v_n)$ et aux pondérations des deux sous-polygones $v_1 \cdots v_k$ et $v_k \cdots v_n$ pour la triangulation T. Déduisez-en une relation de récurrence vérifiée par la fonction t.

3. Calcul du temps optimal, Construction d'une triangulation optimale
Donnez un algorithme calculant la pondération totale d'une triangulation optimale d'un polygone P pour une fonction de pondération w. Quelle est sa complexité dans le pire cas en nombre d'appels à la fonction w ? Donnez un algorithme retournant effectivement une triangulation optimale.

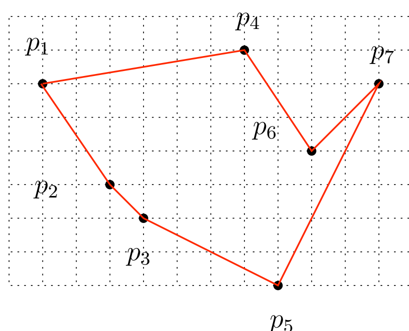
4. Si la fonction de poids est quelconque, combien faut-il de valeurs pour la définir sur tout triangle du polygone ? Comparez avec la complexité obtenue.

5. Si le poids d'un triangle est égal à son aire, que pensez-vous de l'algorithme que vous avez proposé ?

9 Problème NP-complet

Exercice 9.1. Problème euclidien du voyageur de commerce
Étant donnés n points du plan, le problème du voyageur de commerce consiste à trouver une *tournée* (i.e. un chemin reliant tous les points, et ne passant qu'une seule fois par chaque point) qui minimise la distance totale parcourue. Ce problème est un problème difficile en général (il est NP-complet).

1. J. L. Bentley (1962) à suggéré de se restreindre aux tournées *bitoniques* : ces tournées partent du point le plus à gauche, continuent strictement de gauche à droite vers le point le plus à droite, puis retournent vers le point de départ en se déplaçant strictement de droite à gauche (on suppose que deux points n'ont pas la même abscisse). Remarquons qu'une tournée bitonique optimale n'est pas nécessairement une tournée optimale.



Décrire un algorithme utilisant de la programmation dynamique qui détermine une tournée bitonique optimale (on balayera le plan de gauche à droite). On pourra utiliser la primitive DISTANCE(p_1, p_2 : **point**) : **réel** qui calcule la distance entre deux points. Quelle est sa complexité dans le pire cas en nombre d'appels à la fonction DISTANCE ?

2. On considère maintenant le problème général du voyageur de commerce. Quel est le nombre de tournées possibles ?
3. Structure du chemin optimal, solution récursive Soit $S \subset \{1, \ldots, n\}$ et $k \in S$. On note $C(S, k)$ la distance minimale pour aller de p_1 à p_k en passant une fois et une seule par tous les points de S. Exprimez $C(S, k)$ en fonction des $C(S - \{k\}, l)$, pour $l \in S - \{k\}$. Si C^* est le coût d'un chemin optimal, exprimez C^* en fonction des $C(S, k)$, pour $S \subset \{1, \ldots, n\}$ et $k \in S$.
4. Décrivez un algorithme de programmation dynamique utilisant cette relation de récurrence pour résoudre le problème du voyageur de commerce. Quel est le nombre d'étapes de calcul par cette méthode ? Comparez avec la question 3 Peut-on espérer faire mieux ?
5. Quelle est la place mémoire utilisée par l'algorithme précédent ? Peut-on faire mieux ?

10 Bio-informatique

Exercice 10.1. Alignement de séquences génétiques
Un brin d'ADN (acide désoxyribonucléique) est caractérisé par la suite de molécules, appelées *bases*, qui la composent. Il existe quatre bases différentes : deux purines (l'adénine

A, la guanine G), ainsi que deux pyrimidines (la thymine T et la cytosine C). Une séquence d'ADN peut donc être modélisée par un mot sur un alphabet à quatre lettres ($AGTC$).

Pouvoir comparer l'ADN de deux organismes est un problème fondamental en biologie. L'*alignement* entre deux séquences ADN similaires permet d'observer leur degré d'apparentée et d'estimer si elles semblent ou non homologues : c'est-à-dire si elles descendent ou non d'une même séquence ancestrale commune ayant divergé au cours de l'évolution. Un alignement met en évidence les :

— identités entre les 2 séquences,

— insertions ou délétions survenues dans l'une ou l'autre des séquences.

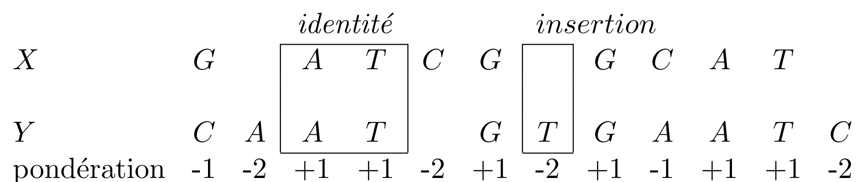


FIGURE 3 – Alignement de deux séquences ADN.

Notons X et Y deux séquences ADN. Pour aligner ces deux séquences, on insère des espaces entre deux bases de chaque séquence, à des emplacements arbitraires (extrémités comprises), de sorte que les deux séquences résultantes (notées X' et Y') aient la même longueur (mais à un emplacement donné, il ne peut pas y avoir un espace pour chacune des deux séquences). Nous considérons le modèle d'évolution simplifié suivant : à chaque emplacement j on attribue une *pondération*

— $+1$ si $X'[j] = Y'[j]$ (alors aucun des deux n'est un espace),

— -1 si $X'[j] \neq Y'[j]$ et aucun des deux n'est un espace,

— -2 si $X'[j]$ ou $Y'[j]$ est un espace.

Le poids de l'alignement est la somme des poids des emplacements.

Donnez un algorithme de programmation dynamique déterminant l'alignement optimal (i.e. de poids maximal) de deux séquences ADN. Déterminez sa complexité dans le pire cas.