TD12: Stratégies algorithmiques

MP2I Lycée Pierre de Fermat

Exercice 1.

Multiplication de matrice : algorithme de Strassen

Soit $n \in \mathbb{N}^*$. On considère deux matrices $A, B \in \mathcal{M}_n(\mathbb{R})$. On s'intéresse au calcul de AB.

Q1. Préliminaire : justifier que la **somme** $A + B$ peut se calculer en temps $\mathcal{O}(n^2)$.

Q2. Proposer un algorithme simple pour calculer AB, et donner sa complexité.

On suppose dans la suite que n est une puissance de 2, quitte à ajouter des coefficients nuls. Notons $C = AB$. On remarque que si l'on décompose A, B et C en 4 matrices de tailles moitié :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

alors on a :

- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

On peut donc calculer C en effectuant 8 produits de matrices de taille moitié.

Q3. En utilisant cette observation, proposer un algorithme récursif de multiplication de matrice. En notant $C(n)$ le coût de multiplication de deux matrices de taille $n \times n$ par cet algorithme, donner la relation de récurrence vérifiée par $C(n)$ et en déduire la complexité de l'algorithme.

Cette première méthode récursive ne donne donc pas lieu à une solution plus efficace que l'algorithme naïf. L'algorithme de **Strassen** consiste à effectuer seulement 7 multiplications de matrices de taille moitié. On introduit les matrices suivantes :

- $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
- $M_2 = (A_{21} + A_{22})B_{11}$
- $M_3 = A_{11}(B_{12} - B_{22})$
- $M_4 = A_{22}(B_{21} - B_{11})$
- $M_5 = (A_{11} + A_{12})B_{22}$
- $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
- $M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Q4. Vérifier que $C_{11} = M_1 + M_4 - M_5 + M_7$

Q5. Exprimer C_{12}, C_{21}, C_{22} en fonction de $M_1, \ldots, M_6$. Vous n'avez pas besoin d'utiliser M_7, qui n'est utilisée que pour le calcul de C_{11}.

Q6. Donner l'algorithme de Strassen, et calculer sa complexité asymptotique.

La constante de complexité se cachant derrière le $\mathcal{O}$ de l'algorithme de Strassen fait qu'en pratique, il devient plus efficace que l'algorithme naïf à partir de $n = 100$ environ. On ne connaît pas d'algorithme de multiplication optimal, des articles sont régulièrement publiés. L'algorithme de Strassen, publié en 1969, est en $\mathcal{O}(n^{2.8074})$. En 2022, trois chercheurs (Duan, Wu et Zhou) ont décrit un algorithme en $\mathcal{O}(n^{2.371866})$, et en 2024, ils ont décrit avec la chercheuse Williams un nouvel algorithme en $\mathcal{O}(n^{2.371552})$! Bien qu'intéressants d'un point théorique, ces algorithmes ne sont pas utilisables en pratique, car la constante de complexité est bien trop grande : on parle d'algorithmes galactiques.

Exercice 2. *Pièce défectueuse*

On considère un ensemble de n pièces indiscernables à l'oeil nu. On sait qu'une des pièces est une fausse, et qu'elle ne pèse pas le même poids que les autres. Cependant, on ne sait pas si elle pèse **plus** ou **moins** lourd. On dispose comme unique outil d'une balance à plateaux basique. On peut utiliser la balance pour **comparer deux tas de pièces**, ce qui donne **trois** résultats possibles : la balance peut pencher d'un côté, de l'autre, ou ne pas pencher du tout si les deux tas sont de même poids. On souhaite résoudre le problème en utilisant le moins de fois possible la balance. La complexité des algorithmes s'exprimera donc en nombre d'utilisations de la balance.

Q1. Proposer un algorithme en $\mathcal{O}(n^2)$

Q2. En regroupant les pièces 3 à 3, donner un algorithme en $\mathcal{O}(n)$.

Q3. En suivant une stratégie diviser pour régner, donner un algorithme en $\mathcal{O}(\log n)$.

Exercice 3. *Bin-packing*

On considère n fichiers de taille $t_1, \ldots, t_n$, que l'on veut stocker sur des disques durs externes, chacun de capacité maximale C, en utilisant le moins de disques durs possible.
On note $S = t_1 + \cdots + t_n$ la taille totale des fichiers.

Q1. Formaliser le problème d'optimisation sous-jacent.

Q2. Donner une minoration simple de la valeur optimale du problème en fonction de S et C

De plus, on considère que les fichiers arrivent dans un ordre $t_1, \ldots t_n$ fixé, et que l'on doit les traiter dans cet ordre. On ne peut donc pas trier les fichiers au préalable. On dit que l'on considère une version **en ligne** du problème car on doit prendre les décisions en direct, à chaque nouvelle donnée reçue.

Algorithme Next-Fit On prépare un disque dur vide initial. On stocke tous les fichiers qui arrivent sur ce disque dur, jusqu'à ce qu'un fichier ne tienne pas sur le disque. Alors, on range le disque dur, et on en prépare un nouveau, vide, dans lequel on met le fichier, et on reprend. On continue jusqu'à avoir traité tous les fichiers. En particulier, une fois que l'on a rangé un disque dur, on ne stocke plus aucun fichier dessus.

Q3. Appliquer cet algorithme avec $C = 10$, et la suite de tailles de fichiers suivante :

$$3, 2, 7, 4, 5, 1, 6, 5, 4, 5$$

obtient-on une solution optimale ?

On considère une instance $(t_1, \ldots, t_n, C)$. Notons K^* le nombre minimal de disques à utiliser, et K_g le nombre de disques utilisé par l'algorithme Next-Fit. Nous allons montrer que $K_g \leq 2K^*$. Autrement dit, l'algorithme glouton utilise au plus deux fois plus de disques qu'une solution optimale. On dit alors que l'algorithme Next-Fit est une **2-approximation**.

Q4. Montrer que dans la solution générée par Next-Fit, deux disques successifs contiennent nécessairement des fichiers dont la taille fait strictement plus que C.

Q5. Montrer que $K_g \leq 2K^*$.

Il existe de nombreuses variantes de l'algorithme glouton Next-fit : on pourrait garder tous les disques ouverts, et mettre chaque nouveau fichier dans un disque quelconque, ou bien dans le disque ayant le moins d'espace libre, etc... Cependant, aucun algorithme polynomial n'est connu à ce jour. Le problème est même NP-complet, autrement dit aussi dur que SAT !

Exercice 4.

Voyageur de commerce

Le problème du voyageur de commerce est un problème algorithmique classique. On considère n villes, de coordonnées $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$. Un voyageur de commerce veut faire un circuit passant exactement une fois par chaque ville, pour y vendre ses marchandises. Pour être le plus efficace possible, il souhaite minimiser son temps de trajet total. On suppose qu'il peut aller de ville en ville en ligne droite, à vitesse constante. On cherche donc à trouver une permutation des villes qui minimise la distance totale entre une ville et la suivante. Formellement, le problème **TSP** (Travelling Salesperson Problem) est :

$$\textbf{TSP} : \min_{\sigma \in S_n} \Sigma_{i=0}^{n-1} d(\sigma(i), \sigma(i+1 \mod n))$$

où $d(i, j)$ est la distance euclidienne entre les villes i et j.

Plutôt que de prendre en entrée les coordonnées des villes, on suppose qu'une instance est directement donnée par la matrice des distances entre chaque ville. Par exemple, voici une instance du TSP avec 7 villes :

	ARGENTEUIL	GRENOBLE	LILLE	AIX	LYON	BREST	LE MANS
ARGENTEUIL	0	495	196	650	404	497	184
GRENOBLE	495	0	639	183	94	854	526
LILLE	196	639	0	809	557	598	358
AIX	650	183	809	0	251	937	641
LYON	404	94	557	251	0	764	431
BREST	497	854	598	937	764	0	349
LE MANS	184	526	358	641	431	349	0

Q1. Donner la distance totale de l'itinéraire Argenteuil - Grenoble - Le Mans - Brest - Aix - Lille - Lyon - Argenteuil.

Q2. Chercher un itinéraire le plus cours possible.

Q3. Décrire l'algorithme exhaustif naïf pour résoudre ce problème, et donner sa complexité.

Q4. Proposer un algorithme glouton, donner sa complexité et montrer qu'il n'est pas optimal en donnant un contre exemple.

Q5. Pour $I \subseteq [\![1, n]\!]$ et $k \in [\![1, n]\!]$, on pose $C(I, k)$ la longueur du chemin le plus court qui part de la ville 1, visite toutes les villes de I, puis arrive en k. Donner une relation de récurrence pour C.

Q6. En déduire un algorithme de programmation dynamique. Quelle est sa complexité ?

Exercice 5.

Prog. Dyn. : $\binom{n}{k}$

On s'intéresse au calcul de $\binom{n}{k}$ pour $0 \leq k \leq n$. L'algorithme consistant à utiliser la formule $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ a comme inconvénient que le calcul de $n!$ peut ne pas tenir sur un entier. On cherche un algorithme qui ne manipule jamais d'entier plus grand que le résultat attendu.

Q1. On définit la complexité spatiale d'un algorithme comme l'espace total qu'il doit réserver pour s'exécuter. Expliciter l'algorithme décrit ci-dessus, et donner ses complexités temporelles et spatiales.

Q2. Rappeler la formule classique du triangle de Pascal, et en déduire un algorithme récursif. Quelle est sa complexité temporelle ? et spatiale ?

Q3. Améliorer l'algorithme précédent avec de la programmation dynamique du bas vers le haut. Donner les complexités temporelles et spatiales.

Q4. Modifier l'algorithme précédent pour avoir une complexité spatiale linéaire.

Exercice 6.

La matmult, elle assure

On s'intéresse à un autre aspect de la multiplication de matrices. On suppose que l'on utilise un algorithme naïf de multiplication, de telle sorte que la multiplication d'une matrice A de taille $m \times n$ et d'une matrice B de taille $n \times p$ coûte exactement nmp multiplications de scalaires.

Considérons trois matrices A, B, C, avec A de dimensions $m \times n$, B de dimensions $n \times p$ et C de dimensions $p \times q$. Ainsi, le produit ABC est une matrice de dimensions $m \times q$. Pour la calculer, deux possibilités :

- Calculer AB puis $(AB)C$
- Calculer BC puis $A(BC)$

Q1. Donner le coût de chaque méthode, en fonction de m, n, p, q.

Q2. Pour tout $N \in \mathbb{N}$, donner m_N, n_N, p_N, q_N, de telle sorte que le coût de la première méthode est négligeable devant le coût de la deuxième méthode lorsque $N \to +\infty$.

De manière générale, si l'on considère des matrices $A_1, \ldots, A_n$, avec chaque A_i de coordonnées $p_{i-1} \times p_i$, l'ordre dans lequel on calcule le produit $A_1 A_2 \ldots A_n$ peut avoir un impact très fort sur le temps de calcul. Notons qu'un ordre de calcul revient en fait à parenthéser le produit $A_1 \ldots A_n$ de manière non-ambigüe, ou bien à construire un arbre syntaxique.

On introduit le problème d'optimisation **MATMULT** : étant donné $p_0, p_1, \ldots p_n$ des dimensions de matrices $A_1, \ldots A_n$, quel est le coût minimal pour calculer le produit $A_1 \ldots A_n$?

Q3. Proposer un algorithme glouton, et donner un contre exemple montrant qu'il n'est pas optimal.

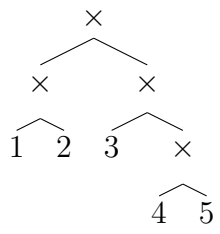
Nous allons résoudre ce problème par une approche de programmation dynamique. Pour $1 \leq i \leq j \leq n$, on note $C(i, j)$ le coût minimal nécessaire pour calculer le produit $A_i \ldots A_j$.

Q4. Donner une relation de récurrence sur $C(i, j)$, et donner ses cas de base.

Q5. En déduire un algorithme de programmation dynamique pour le problème **MATMULT**, et donner sa complexité.

On s'intéresse maintenant à la construction d'une solution optimale à partir du tableau calculé dans l'algorithme de prog. dyn. Explicitement, on cherche à construire un arbre de syntaxe, dont :

- Les feuilles sont des entiers $k \in [\![1, n]\!]$
- Les noeuds internes marquent les multiplications



de telle sorte que les feuilles sont ordonnées dans l'ordre croissant et contiennent $1, 2, \ldots, n$. Par exemple, l'arbre ci-contre correspond à l'ordre de multiplication $(A_1 A_2)(A_3(A_4 A_5))$.

Q6. Décrire un algorithme permettant de construire un arbre syntaxique à partir du tableau de prog. dyn. :

Algorithme 1 : Reconstruction d'une solution

Entrée(s) : $p_0, \ldots, p_n$ dimensions des matrices à multiplier, T tableau de taille $n \times n$ des coûts de multiplication calculé par programmation dynamique

Sortie(s) : A arbre syntaxique de coût optimal pour le calcul de $A_1, \ldots, A_n$

Indication : on pourra écrire un algorithme récursif qui prend également en entrée $0 \leq i \leq j \leq n$ et calcule l'arbre syntaxique optimal pour le calcul du produit partiel $A_i \ldots A_j$.

Q7. Quelle est la complexité de l'algorithme précédent ? Quelles informations pourrait-on stocker lors du calcul du tableau afin d'accélérer cette phase de construction ?