TD 03 – Programmation Dynamique (corrigé)

(Polygones) Exercice 1. *Triangulation de polygones*

On considère les polygones convexes du plan. Une triangulation d'un polygone est un ensemble de cordes qui ne se coupent pas à l'intérieur du polygone et qui le divisent en triangles.

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

☞ On procède par récurrence sur $n \geq 3$.
Pour $n = 3$, le polygône est un triangle, qui a bien $n-2=1$ triangle et $n-3=0$ corde.
Soit $n \in \mathbb{N}, n \geq 3$. On suppose le résultat démontré pour tout polygône convexe possédant un nombre de côtés strictement inférieur à n. Soit alors un polygône à n côtés et une triangulation de ce polygône. Considérons une corde qui divise ce polygône en deux polygônes, respectivement à $(i+1)$ et $(j+1)$ côtés avec $i+j=n, i \geq 2, j \geq 2$ (comme le montre la figure 1).

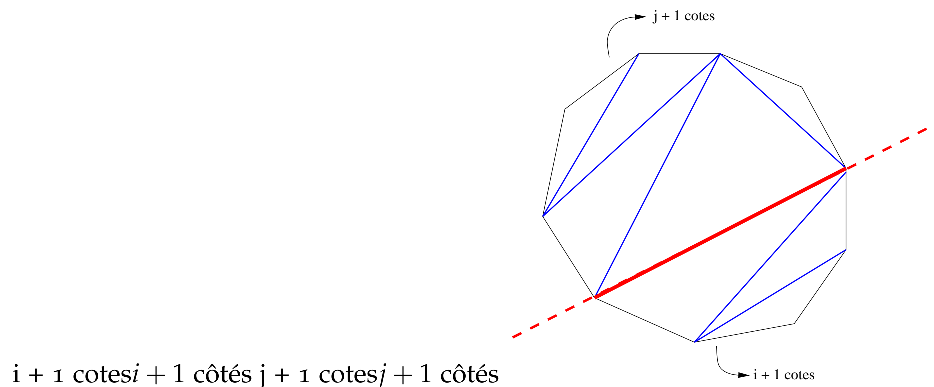

Figure 1 – Une corde (en bleu) qui divise un polygone en deux

Par hypothèse de récurrence, on a $(i-2)$ cordes dans le premier polygône et $j-2$ dans le second, ce qui fait pour le polygône entier $(i-2)+(j-2)+1 = n-3$ cordes en tout (le $+1$ représente la corde qui séparait le polygône en deux).

De même, on obtient un total de $(i-1)+(j-1) = n-2$ triangles dans le polygône en tout, ce qui achève la démonstration.

Le problème est celui de la triangulation optimale de polygones. On part d'un polygone convexe $P = \langle v_0, ..., v_n \rangle$, où $v_0, \ldots, v_n$ sont les sommets du polygone donnés dans l'ordre direct, et d'une fonction de pondération w définie sur les triangles formés par les côtés et les cordes de P (par exemple $w(i,j,k) = \|v_i v_j\| + \|v_j v_k\| + \|v_k v_i\|$ est le périmètre du triangle $v_i v_j v_k$). Le problème est de trouver une triangulation qui minimise la somme des poids des triangles de la triangulation.
On définit pour $1 \leq i < j \leq n$, $t[i,j]$ comme la pondération d'une triangulation optimale du polygone $\langle v_{i-1}, ..., v_j \rangle$, avec $t[i,i] = 0$ pour tout $1 \leq i \leq n$.

2. Définir t récursivement, en déduire un algorithme et sa complexité.

☞ Pour trouver une formule de récurrence sur t, il suffit de s'aperçevoir que le $v_{i-1}v_j$ fait partie d'un triangle dont le troisième sommet est v_k, voir Figure 2. On en déduit la formule : $t[i,j] = min_{i \leq k \leq j-1}(t[i,k] + t[k+1,j] + w(i-1,k,j))$ (*On remarque que la convention $t[i,i]$ est bien adaptée puisque la formule ci-dessus reste vraie quand $j = i+1$: on a bien $t[i,i+1] = w(i-1,i,i+1)$.*).
Nous pouvons maintenant écrire un algorithme qui calcule $t[1,n]$, et ce à partir des $t[k,j], i+1 \leq k \leq j$ et des $t[i,k], i \leq k \leq j-1$, c'est à dire, si l'on représente les valeurs à calculer dans un diagramme (i,j) comme celui donné dans la figure 3, à partir des valeurs situées "en-dessous" et "à droite" du point $(1,n)$. On s'aperçoit alors qu'il suffit de calculer les $t[i,j]$ pour i de 1 à n et pour $j \geq i$, et ce par $(j-i)$ croissant.
Ceci donne l'algorithme :

Algorithm 1: Algorithme par programmation dynamique

```
début
    pour i de 0 à n faire
        t[i,i] ← 0;
    pour d de 1 à n − 1 faire
        pour i de 1 à n − d faire
            t[i,i+d] ← min_{i≤k≤j−1}(t[i,k] + t[k+1,j] + w(i−1,k,j));
    retourner t[1,n];
```

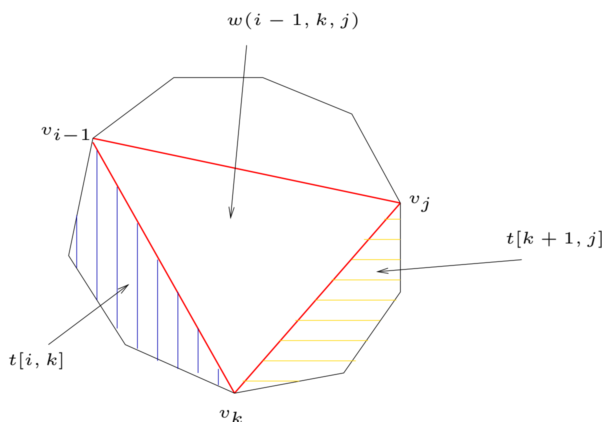

FIGURE 2 – Découpage du polygone pour la récurrence

Cet algorithme calcule $t[i,j]$ en $2(j-i)$ additions et en $j-i-1$ comparaisons pour calculer le min.
Le nombre total d'additions est donc : $A_n = \sum_{d=1}^{n-1}((n-d)\cdot(2d))$ où $(n-d)$ correspond au nombre de $t[i,j]$ à calculer sur la diagonale $j-i=d$ et $2d$ au nombre d'additions dans le calcul de chaque $t[i,j]$ où $j-i=d$, ce qui nous donne :

$$\begin{aligned} A_n &= 2n\cdot\sum_{d=1}^{n-1} d - 2\sum_{d=1}^{n-1} d^2 \\ &= 2n\cdot\frac{n\cdot(n-1)}{2} - 2\frac{(n-1)\cdot n\cdot(2n-1)}{6} \\ &= \frac{(n-1)\cdot n\cdot(n+1)}{3} \\ &\sim n^3/3 = \Theta(n^3) \end{aligned}$$

Pour calculer le nombre T_n de tests à effectuer, il suffit de remarquer qu'on fait deux fois moins de tests que d'additions dans le calcul du minimum, ce qui nous donne $T_n = A_n/2 - C_n$ où C_n représente le nombre total de $t[i,j]$ calculés (ie le nombre d'affectations) On en déduit :

$$\begin{aligned} T_n &= A_n/2 - \sum_{d=1}^{n-1}(n-d) \\ &= \frac{n^3-3n^2+2n}{6} \\ &= \Theta(n^3) \end{aligned}$$

La complexité globale de l'algorithme est donc en $\Theta(n^3)$.

3. 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.

☞ Dans le cas général, il faut autant de valeurs pour définir la fonction w qu'il n'y a de triangles possibles, soit $C_{n+1}^3 = \Theta(n^3)$. Comme il faudra bien lire chacune de ces valeurs pour construire une triangulation optimale, n'importe quel algorithme doit fonctionner en une complexité d'au moins $\Theta(n^3)$. Notre algorithme est donc optimal en complexité du point de vue de l'ordre de grandeur.

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

☞ Dans le cas où le poids d'un triangle est égal à son aire, toutes les triangulations donnent le même poids qui est l'aire du polygône. L'algorithme précédent est donc inadapté, et on peut à la place procéder par exemple comme suit :

---

début
  Prendre une triangulation quelconque (par exemple avec les cordes $(v_0 v_i)_{2\le i\le n-1}$);
  Faire la somme des poids des triangles;

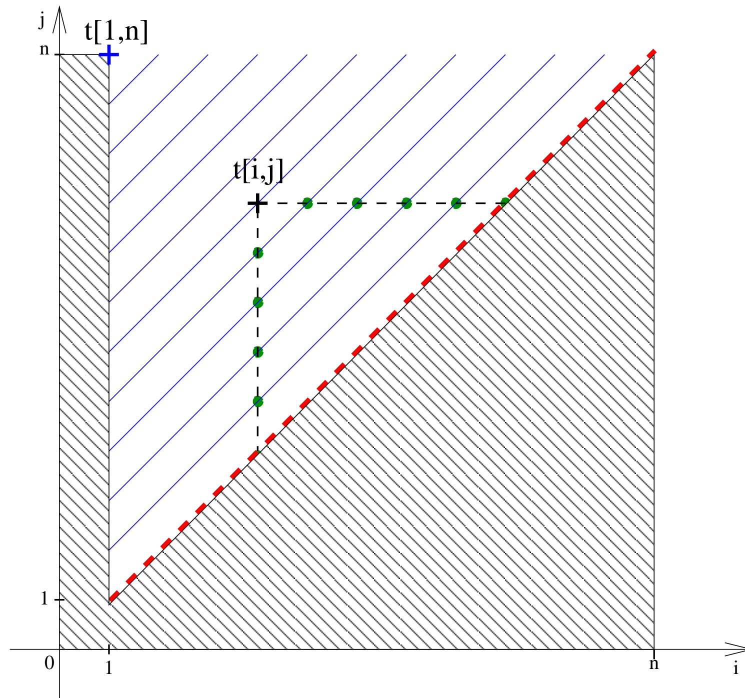

FIGURE 3 – Diagramme (i, j) des valeurs à calculer par l'algorithme de résolution

Cet algorithme n'effectue aucun test et $n - 3$ additions (bien en dessous du $\Theta(n^3)$!).

(Impression) **Exercice 2.** *Impression Equilibrée*

Le problème est l'impression équilibrée d'un paragraphe sur une imprimante. Le texte d'entrée est une séquence de n mots de longueurs $\ell_1, \ell_2, \ldots, \ell_n$ (mesurées en caractères). On souhaite imprimer ce paragraphe de manière équilibrée sur un certain nombre de lignes qui contiennent un maximum de M caractères chacune. Le critère d'*équilibre* est le suivant. Si une ligne donnée contient les mots i à j (avec $i \leq j$) et qu'on laisse exactement une espace [1] entre deux mots, le nombre de caractères d'espacement supplémentaires à la fin de la ligne est $M - j + i - \sum_{k=i}^{j} \ell_k$, qui doit être positif ou nul pour que les mots tiennent sur la ligne. L'objectif est de minimiser la somme, sur toutes les lignes *hormis la dernière*, des cubes des nombres de caractères d'espacement présents à la fin de chaque ligne.

1. Est-ce que l'algorithme glouton consistant à remplir les lignes une à une en mettant à chaque fois le maximum de mots possibles sur la ligne en cours, fournit l'optimum ?

☞ Cet algorithme ne fournit pas la solution optimale. Pour s'en convaincre prenons l'exemple suivant : on considère des lignes de 24 caractères, et on souhaite écrire la phrase "Un algorithme glouton ne peut pas être bon sempiternellement". La Figure 1 présente la solution gloutonne : la première ligne ne contient aucun espace supplémentaire, mais la deuxième en contient sept, d'où un coût de $O^3 + 7^3 = 343$. La Figure 1 présente la solution optimale : la première ligne contient trois espaces supplémentaires et la deuxième quatre, d'où un coût de $3^3 + 4^3 = 91$, ce qui montre la non optimalité de l'algorithme glouton.

U	n		a	l	g	o	r	i	t	h	m	e		g	l	o	u	t	o	n		n	e
p	e	u	t		p	a	s		ê	t	r	e		b	o	n							
s	e	m	p	i	t	e	r	n	e	l	l	e	m	e	n	t							

FIGURE 4 – Solution de l'algorithme glouton

2. Donner un algorithme de programmation dynamique résolvant le problème. Analyser sa complexité en temps et en espace.

1. En typographie *espace* est un mot féminin.

U	n		a	l	g	o	r	i	t	h	m	e		g	l	o	u	t	o	n			
n	e		p	e	u	t		p	a	s		ê	t	r	e		b	o	n				
s	e	m	p	i	t	e	r	n	e	l	l	e	m	e	n	t							

FIGURE 5 – Solution optimale

☞ Si on considère une solution optimale sur m lignes, alors nécessairement la composition sur les $m-1$ dernières lignes est optimale. En effet, supposons qu'elle ne soit pas optimale, dans ce cas il suffirait de combiner la composition de la première ligne avec une composition optimale dans les $m-1$ dernières lignes pour améliorer notre coût.
Si on note $C[i]$ le coût de la composition des mots à partir du i^e (inclus) mot jusqu'au dernier, on remarque que la première ligne se terminera à un certain mot j et que le coût de la composition sera le coût de la première ligne de i à j plus le coût des dernières lignes commençant au mot $j+1$. On a donc la relation de récurrence suivante :

$$C[i] = \begin{cases} 0 \text{ si } n-i+\sum_{k=i}^{n} \ell_k \leq M \\ \min_{1\leq i\leq j\leq n} \left\{ \left(M-j+i-\sum_{k=i}^{j} \ell_k\right)^3 + C[j+1] \middle| j-i+\sum_{k=i}^{j} \ell_k \leq M \right\} \text{ sinon} \end{cases} \quad (1)$$

Algorithm 2: Algorithme par programmation dynamique

début
- $c[n+1] \leftarrow 0$;
- **pour** i *de* n *à* 1 **faire**
  - **si** $n-i+\sum_{k=i}^{n} \ell_k \leq M$ **alors**
    - $c[i] \leftarrow 0$;
  - **sinon**
    - $c[i] \leftarrow +\infty$;
    - **pour** j *de* i *à* n **faire**
      - **si** $\left(j-i+\sum_{k=i}^{j} \ell_k \leq M\right)$ *et* $\left(\left(M-j+i-\sum_{k=i}^{j} \ell_k\right)^3 + c[j+1] < c[i]\right)$ **alors**
        - $c[i] \leftarrow \left(M-j+i-\sum_{k=i}^{j} \ell_k\right)^3 + c[j+1]$;

3. Supposons que pour la fonction de coût à minimiser, on ait simplement choisi la somme des nombres de caractères d'espacement présents à la fin de chaque ligne. Est-ce que l'on peut faire mieux en complexité que pour la question 2 ?

   ☞ Oui. Si on prend la somme des caractères d'espacement présents à la fin de chaque ligne l'algorithme glouton est suffisant.

4. (*Plus informel*) Qu'est-ce qui à votre avis peut justifier le choix de prendre les cubes plutôt que simplement les nombres de caractères d'espacement en fin de ligne ?

   ☞ Le fait de prendre des cubes va défavoriser les solutions avec de grandes différences dans le nombre d'espaces en fin de ligne : les lignes avec beaucoup d'espace vont contribuer très fortement au coût, alors qu'avec une simple somme le nombre total d'espacements en fin de ligne peut être le même que la solution optimale, mais avoir des lignes très remplies et d'autres très peu remplies (voir l'exemple avec le glouton).

(SuperSuite) Exercice 3. *Plus courte super suite*

On a vu en cours le calcul d'une plus longue sous-suite commune à deux chaînes A et B. On s'intéresse maintenant au calcul d'une plus courte super-suite de A et B, définie comme une suite S de longueur minimale dont A et B sont des sous-suites.

1. Déterminer S pour $A = ababababb$ et $B = aabbbbaab$.

   ☞

2. (Difficile) Proposer un algorithme pour déterminer une plus courte super-suite de A et B et donner sa complexité. Attention, il faut justifier soigneusement la correction.
   Indication : Se servir de la plus longue sous-suite commune à A et B.

   ☞

(JeuConstruction) Exercice 4. *Jeu de construction*

On veut construire une tour la plus haute possible à partir de différentes briques. On dispose de n types de briques et d'un nombre illimité de briques de chaque type. Chaque brique de type i est un

parallélépipède de taille (x_i, y_i, z_i) et peut être orientée dans tous les sens, deux dimensions formant la base et la troisième dimension formant la hauteur.
Dans la construction de la tour, une brique ne peut être placée au dessus d'une autre que si les deux dimensions de la base de la brique du dessus sont *strictement inférieures* aux dimensions de la base de la brique du dessous.

1. Proposer un algorithme efficace pour construire une tour de hauteur maximale.

☞ Si on prend un parallélépipède quelconque $(x_i,\ y_i,\ z_i)$, on s'aperçoit qu'on peut le poser sur la tour de 6 manières différentes.
On peut commencer par remarquer que si on peut poser une brique sur une autre de façon à ce que la longueur de la 1re brique soit parallèle à la largeur de la 2e, alors on peut aussi poser la 1re brique sur la 2e de façon à ce que les largeurs (et donc aussi les longueurs) des deux briques soient parallèles (comme le montre la figure 6). On peut donc faire l'hypothèse que les briques seront posées les unes sur les autres de telle sorte que les largeurs des briques soient toutes parallèles entre elles. (*Le nombre de configurations par parallélépipède est ainsi réduit à 3.*)

Remarquons ensuite qu'un même parallélépipède ne peut être utilisé au plus que deux fois (dans deux configurations différentes). En effet, considérons le parallélépipède (x_i, y_i, z_i) avec $x_i \leq y_i \leq z_i$: si on la pose une première fois, alors, dans le meilleur des cas, on peut encore poser sur la tour toute brique (x_j, y_j, z_j) telle que $x_j < y_i$ et $y_j < z_i$ (en supposant que $x_j \leq y_j \leq z_j$). Ainsi, si les inégalités sont strictes pour x_i, y_i, et z_i, on peut encore poser le parallélépipède. Pour le poser une troisième fois, il faudrait qu'il possède un côté de longueur strictement inférieure à x_i. Ce n'est pas le cas, on ne peut le poser que deux fois. Cela a pour conséquence que la tour de hauteur maximale sera composée d'au plus $2n$ **parallélépipèdes**.

Dorénavant, nous allons considérer les briques (L_i, l_i, h_i) avec $L_i \geq l_i$. Nous devons alors considérer $3n$ briques au lieu de n parallélépipèdes, mais il ne reste plus qu'une configuration possible par brique (*Si le parallélépipède est (x_i, y_i, z_i) avec $x_i < y_i < z_i$, les 3 briques associées sont (z_i, y_i, x_i), (z_i, x_i, y_i) et (y_i, x_i, z_i).*).

Ainsi, on peut poser la brique (L_i, l_i, h_i) sur la brique (L_j, l_j, h_j) si et seulement si $\begin{cases} L_i < L_j \\ l_i < l_j \end{cases}$

On pose $\forall i \in \{1, \ldots, 3n\}, H_i =$ hauteur maximale parmi les tours se terminant par la i^e^ brique. On a la relation de récurrence suivante :

$$\forall i \in \{1, \ldots, 3n\}, H_i = h_i + \max_{1 \leq j \leq 3n} (H_j / L_j > L_i \text{ et } l_j > l_i) \tag{2}$$

L'idée est de commencer par trier les $3n$ briques par L_i décroissants. Cela se fait en $O(n \log(n))$ comparaisons. L'équation précédente peut alors se réécrire :

$$\forall i \in \{1, \ldots, 3n\}, H_i = h_i + \underbrace{\max_{1 \leq j < i}}_{i-2\ tests} (H_j / \underbrace{l_j > l_i}_{i-1\ tests}) \tag{3}$$

On calcule ainsi chaque H_i pour i allant de 2 à $3n$ en partant de $H_1 = h_1$. Le calcul de H_i nécessite 1 addition et $2i - 3$ comparaisons dans le pire des cas. Le calcul de tous les $H_i, i \in 1, \ldots, 3n$ est donc linéaire en additions et quadratique en comparaisons.

Enfin, il faut calculer $H = Max_{1 \leq i \leq 3n}(H_i)$ qui est la solution de notre problème, ce qui coûte $3n - 1$ comparaisons. La complexité totale est par conséquent en $O(n^2)$ pour les comparaisons et en $O(n)$ pour les additions et l'algorithme est le suivant :

Algorithm 3:

début
- liste ← [];
- **pour** *i de* 1 *à* n **faire**
  - Ajouter les 3 configurations possibles du parallélépipède $(x_i,\ y_i,\ z_i)$ à *liste*;
- Trier $liste = (L_i, l_i, h_i)_{i \in \{1,\ldots,3n\}}$ par L_i décroissants;
- $H_1 \leftarrow h_1$;
- **pour** *i de* 2 *à* $3n$ **faire**
  - $H_i \leftarrow h_i + Max_{1 \leq j < i}(H_j / l_j > l_i)$;
- $H \leftarrow Max_{1 \leq i \leq 3n}(H_i)$;
- **retourner**;

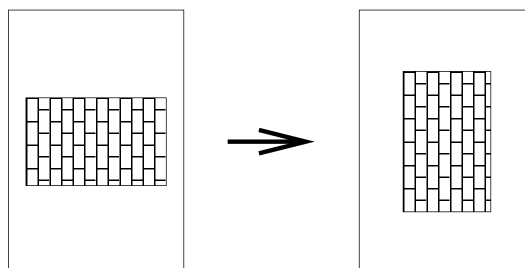

FIGURE 6 – Si une brique est posée avec sa largeur parallèle à la longueur de celle en dessous, alors elle peut être posée de telle sorte que leurs longueurs sont parallèles