TD 1 : Tableaux

Algorithmes et programmation.

2019

Exercice 1 — *Complexité de la multiplications de n matrices*

1. Ecrire un algorithme $MULT$ qui renvoie la multiplication de deux matrices. Quelle est la complexité, en nombre de multiplications, de cet algorithme ?

▶ **Correction**

Entrées: Deux matrices $A \in \mathcal{M}_{np}(\mathbb{R})$ et $B \in \mathcal{M}_{pq}(\mathbb{R})$
Sorties: Une matrice $C \in \mathcal{M}_{nq}(\mathbb{R})$ égale au produit $A \times B$.

```
1: Pour i de 1 à n Faire
2:     Pour j de 1 à q Faire
3:         C_ij ← 0
4:         Pour k de 1 à p Faire
5:             C_ij ← C_ij + A_ik · B_kj
6: Renvoyer C
```

Le nombre de multiplications de cet algorithme est égal au nombre de fois qu'on appelle la ligne 5, c'est à dire $n \cdot p \cdot q$ fois. En effet, on a 3 boucles imbriquées.
La boucle sur i exécute n itérations. La boucle sur j exécute q itérations pour chaque itération de la boucle sur i, donc nq itérations. La boucle sur k exécute p itérations pour chaque itération de la boucle sur j, donc nqp itérations.
Donc la complexité en nombre de multiplications est $O(npq)$.

2. Ecrire un algorithme $n - MULT$, qui connaissant n matrices $A_1, A_2, \ldots, A_n$, calcule le produit $((A_1 \cdot A_2) \cdot A_3) \cdots A_n)$. Quelle est la complexité, en nombre de multiplications, de cet algorithme ?

▶ **Correction**

On effectue les multiplications de gauche à droite. On va donc faire un algorithme qui fera les instructions suivantes :

$$C \leftarrow A_1$$
$$C \leftarrow C \cdot A_2$$
$$\cdots$$
$$C \leftarrow C \cdot A_n$$

Entrées: n matrices $A_i \in \mathcal{M}_{p_i q_i}(\mathbb{R})$ pour $1 \leq i \leq n$; avec $q_i = p_{i+1}$.
Sorties: Une matrice $C \in \mathcal{M}_{p_1 q_n}(\mathbb{R})$ égale au produit $((A_1 \cdot A_2) \cdot A_3) \cdots A_n)$.

```
1: C ← I_{p_1}
2: Pour i de 1 à n Faire
3:     C ← MULT(C, A_i)
4: Renvoyer C
```

Le nombre de multiplications de cet algorithme est égal à la somme des nombres de multiplications effectuées par l'algorithme MULT.

La première fois qu'on appelle MULT, quand $i = 1$, c'est avec la matrice $C = I_{p_1} \in \mathcal{M}_{p_1 p_1}(\mathbb{R})$ et la matrice $A_1 \in \mathcal{M}_{p_1 q_1}(\mathbb{R})$. D'après la question 1, on effectue donc $p_1 p_1 q_1$ multiplications. A la suite de cet appel, on a $C \in \mathcal{M}_{p_1 q_1}(\mathbb{R})$.

Quand $i = 2$, on a MULT(C, A_2) avec $C \in \mathcal{M}_{p_1 q_1}(\mathbb{R})$ et $A_2 \in \mathcal{M}_{p_2 q_2}(\mathbb{R})$. On rappelle que $p_2 = q_1$. On effectue donc $p_1 p_2 q_2$ multiplications. A la suite de cet appel, on a $C \in \mathcal{M}_{p_1 q_2}(\mathbb{R})$.

Pour $i > 1$ quelconque, on a MULT(C, A_i) avec $C \in \mathcal{M}_{p_1 q_{i-1}}(\mathbb{R})$ et $A_i \in \mathcal{M}_{p_i q_i}(\mathbb{R})$. On rappelle que $p_i = q_{i-1}$. On effectue donc $p_1 p_i q_i$ multiplications. A la suite de cet appel, on a $C \in \mathcal{M}_{p_1 q_i}(\mathbb{R})$.

Donc au total, on a $\sum_{i=1}^{n} p_1 p_i q_i = p_1 \left(\sum_{i=1}^{n} p_i q_i \right)$ multiplications.

3. Trouvez un exemple avec $n = 3$ où le calcul $(A_1 \cdot A_2) \cdot A_3$ nécessite plus de multiplications que $A_1 \cdot (A_2 \cdot A_3)$.

► **Correction**

Prenons $A_1 = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, $A_2 = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$, $A_3 = \begin{pmatrix} i & j \end{pmatrix}$.

Lors qu'on fait $(A_1 \cdot A_2) \cdot A_3$, on a : $2*2*2 = 8$ multiplications pour $C = A_1 \cdot A_2$, et qui donne une matrice 2×2. $2*2*1 = 4$ multiplications pour $C = C \cdot A_3$. Donc 12 multiplications en tout.

Lors qu'on fait $A_1 \cdot (A_2 \cdot A_3)$, on a : $2*2*1 = 4$ multiplications pour $C = A_2 \cdot A_3$, et qui donne une matrice 2×1. $2*2*1 = 4$ multiplications pour $C = A_1 \cdot C$. Donc 8 multiplications en tout.

Pour info, si on refait le même calcul qu'à la question 2, mais en multipliant les matrices dans l'autres sens, on a l'algo suivant :

Entrées: n matrices $A_i \in \mathcal{M}_{p_i q_i}(\mathbb{R})$ pour $1 \leq i \leq n$; avec $q_i = p_{i+1}$.

Sorties: Une matrice $C \in \mathcal{M}_{p_1 q_n}(\mathbb{R})$ égale au produit $((A_1 \cdot A_2) \cdot A_3) \cdots A_n)$.

1: $C \leftarrow I_{q_n}$
2: **Pour** i de n à 1 **Faire**
3: $\quad C \leftarrow \text{MULT}(A_i, C)$
4: **Renvoyer** C

On a alors la complexité suivante : $\sum_{i=1}^{n} p_i q_i q_n = q_n \left(\sum_{i=1}^{n} p_i q_i \right)$

Donc si $p_1 > q_n$, le second algorithme sera plus efficace que le premier.

Il existe donc un ordre dans lequel multiplier les matrices $A_1, A_2, \ldots, A_n$ nécessite le moins de multiplications. On désire connaître cet ordre.

4. On souhaite écrire un algorithme *nbMULT* qui, connaissant n matrices $A_1, A_2, \ldots, A_n$ et un ordre $\mathcal{O}$ dans lequel les multiplier, renvoie le nombre de multiplications nécessaires pour obtenir le résultat. Sous quelle forme peut-on donner cet ordre en entrée de l'algorithme ? Ecrire l'algorithme correspondant. Quelle est la complexité de cet algorithme ?

► **Correction**

Prenons un exemple, pour 4 matrices A_1, A_2, A_3, A_4, il y a 5 manières de les multiplier :

— $((A_1 \cdot A_2) \cdot A_3) \cdot A_4$
— $(A_1 \cdot (A_2 \cdot A_3)) \cdot A_4$
— $A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)$
— $A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))$
— $(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$

On peut donner l'ordre sous la forme d'une liste d'entiers $\mathcal{O} = (i_1, i_2, \ldots, i_{n-1})$.

— le premier indice i_1 indique les 2 premières matrices à multiplier. Cet indice est entre 1 et $n - 1$; on multiplie les matrice A_{i_1} et A_{i_1+1}.

— le deuxième indice i_2 indique les 2 matrices suivantes à multiplier. A la suite de la première multiplication on a une matrice de moins, renommons, sans perte de généralité toutes les matrices : $B_1, B_2, \ldots, B_{n-1}$ où une des matrices B_i est le produit $A_{i_1} \cdot A_{i_1+1}$. L'indice i_2 est entre 1 et $n-2$. On multiplie les matrices B_{i_2} et B_{i_2+1}.

— et ainsi de suite jusqu'à ce qu'il ne reste plus qu'une matrice.

L'ordre $\mathcal{O}$ est donc une liste de $n-1$ indices.

Par exemple, si $n = 4$ et $\mathcal{O} = (3, 2, 1)$ alors on a :

— D'abord la multiplication de A_3 par A_4 ; on obtient 3 matrices $A_1, A_2, A_3 \cdot A_4$.

— Puis la multiplication de A_2 par $A_3 \cdot A_4$; on obtient 2 matrices A_1 et $A_2 \cdot (A_3 \cdot A_4)$.

— Puis la multiplication de A_1 par $A_2 \cdot (A_3 \cdot A_4)$; on obtient 1 matrice $A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))$.

Pour chaque multiplication, on a les ordre suivants :

— $((A_1 \cdot A_2) \cdot A_3) \cdot A_4$ est représenté par $\mathcal{O} = (1, 1, 1)$

— $(A_1 \cdot (A_2 \cdot A_3)) \cdot A_4$ est représenté par $\mathcal{O} = (2, 1, 1)$

— $A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)$ est représenté par $\mathcal{O} = (2, 2, 1)$

— $A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))$ est représenté par $\mathcal{O} = (3, 2, 1)$

— $(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$ est représenté par $\mathcal{O} = (1, 2, 1)$ et par $\mathcal{O} = (3, 1, 1)$.

Connaissant l'ordre $\mathcal{O}$, on peut effectuer la multiplication correspondante avec l'algorithme suivant :

Entrées: n matrices $A_i \in \mathcal{M}_{p_i q_i}(\mathbb{R})$ pour $1 \leq i \leq n$; avec $q_i = p_{i+1}$ et un ordre $\mathcal{O}$ dans lequel les multiplier.

Sorties: Une matrice $C \in \mathcal{M}_{p_1 q_n}(\mathbb{R})$ égale au produit $A_1 \cdot A_2 \cdot A_3 \cdots A_n$ calculé dans l'ordre $\mathcal{O}$.

1: $L \leftarrow$ liste des n matrices $A_1, A_2, \ldots, A_n$
2: **Pour** $i \in \mathcal{O}$ **Faire**
3: $\quad M_1 \leftarrow$ la i^{e} matrice de L
4: $\quad M_2 \leftarrow$ la $i+1^{\text{e}}$ matrice de L
5: $\quad M \leftarrow \text{MULT}(M_1, M_2)$
6: $\quad$ Remplacer M_1 et M_2 par M dans L
7: **Renvoyer** C

Par exemple si on applique l'algorithme avec $n = 4$ et $\mathcal{O} = (2, 2, 1)$, on obtient :

— $L = (A_1, A_2, A_3, A_4)$ avant la première itération

— $L = (A_1, A_2 \cdot A_3, A_4)$

— $L = (A_1, (A_2 \cdot A_3) \cdot A_4)$

— $L = (A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$

On souhaite dans cette question juste connaître le nombre de multiplications nécessaires pour faire le produit des matrices ; on ne souhaite pas connaître le produit des matrices (c'est la questions suivante).

On a donc juste besoin des tailles des matrices et d'utiliser le résultat de la question 1.

Entrées: n matrices $A_i \in \mathcal{M}_{p_i q_i}(\mathbb{R})$ pour $1 \leq i \leq n$; avec $q_i = p_{i+1}$ et un ordre $\mathcal{O}$ dans lequel les multiplier.

Sorties: Le nombre de multiplications nécessaires pour calculer le produit $A_1 \cdot A_2 \cdot A_3 \cdots A_n$ dans l'ordre $\mathcal{O}$.

1: $L \leftarrow$ liste des tailles des n matrices, $((p_1, q_1), (p_2, q_2), \ldots, (p_n, q_n))$
2: $nbMult \leftarrow 0$
3: **Pour** $i \in \mathcal{O}$ **Faire**
4: $\quad (p, q) \leftarrow$ la i^{e} taille de L
5: $\quad (p', q') \leftarrow$ la $i+1^{\text{e}}$ taille de L, avec $q = p'$
6: $\quad nbMult \leftarrow nbMult + p \cdot q \cdot q'$
7: $\quad$ Remplacer (p, q) et (p', q') par (p, q') dans L
8: **Renvoyer** $nbMult$

La complexité de cet algorithme est le suivant : pour chaque élément de $\mathcal{O}$, on récupère 2 éléments de la liste L, on effectue un calcul et on insère un élément. On fait donc 4 instructions élémentaires par éléments de $\mathcal{O}$.

On parle d'*instruction élémentaire* pour signifier que les instructions ne sont pas des instructions complexes qui prennent du temps. Ici, on considère donc qu'ajouter ou supprimer un élément d'une liste prend autant de temps que de faire la multiplication de 2 nombres.

Puisque $\mathcal{O}$ est une liste de $n-1$ entiers, la complexité est de l'ordre de $4 \cdot (n-1) = O(n)$.

5. Ecrire un algorithme $n - MULT - \mathcal{O}$ qui connaissant n matrices $A_1, A_2, \ldots, A_n$ et un ordre $\mathcal{O}$ dans lequel les multiplier, renvoie la multiplications de ces matrices dans l'ordre $\mathcal{O}$. Quelle est la complexité, en nombre de multiplications, de ces matrices ?

► **Correction**

L'algorithme est donné dans la correction de la question précédente.

Sa complexité est le résultat de $\text{NBMULT}(A_1, A_2, \ldots, A_n, \mathcal{O})$ puisque cet algorithme calcule le nombre de multiplications que ferait N-MULT-$\mathcal{O}(A_1, A_2, \ldots, A_n, \mathcal{O})$.

6. Ecrire enfin un algorithme $n - MULT - OPT$, qui connaissant n matrices $A_1, A_2, \ldots, A_n$, calcule le produit de ces matrices en faisant le moins de multiplications possibles. Quelle est la complexité de cet algorithme ?

► **Correction**

Il faut tester tous les ordres possibles. Pour chaque ordre on va calculer le nombre de multiplications qu'on ferait avec $\text{NBMULT}(A_1, A_2, \ldots, A_n, \mathcal{O})$; on choisit ensuite le meilleur des ordre et on effectue le produit.

Il existe plusieurs manières d'écrire cet algorithme. La plus concise est :

Entrées: n matrices $A_i \in \mathcal{M}_{p_i q_i}(\mathbb{R})$ pour $1 \leq i \leq n$; avec $q_i = p_{i+1}$.

Sorties: Une matrice $C \in \mathcal{M}_{p_1 q_n}(\mathbb{R})$ égale au produit $A_1 \cdot A_2 \cdot A_3 \cdots A_n$ calculé dans l'ordre nécessitant le moins de multiplications.

1: $\mathcal{O}^* \leftarrow \arg\min\{\text{NBMULT}(A_1, A_2, \ldots, A_n, \mathcal{O}) | \mathcal{O} \text{ ordre possible de multiplication}\}$
2: **Renvoyer** N-MULT-$\mathcal{O}(A_1, A_2, \ldots, A_n, \mathcal{O})$

La ligne 2 prend un temps $\text{NBMULT}(A_1, A_2, \ldots, A_n, \mathcal{O}^*)$ d'après la question précédente.

Qu'en est-t-il de la ligne 1 ? Pour rappel, l'arg min est l'argument pour lequel la valeur est minimum ; alors que min seul renverrait la valeur minimum. La ligne 1 n'est pas une instruction élémentaire. Elle prend du temps. On pourrait la développer ainsi :

1: $min \leftarrow +\infty$
2: $\mathcal{O}^* \leftarrow NULL$
3: **Pour** $\mathcal{O}$ ordre possible de multiplication **Faire**
4: $\quad m = \text{NBMULT}(A_1, A_2, \ldots, A_n, \mathcal{O})$
5: $\quad$ **Si** $m < min$ **Alors**
6: $\quad\quad min = m$
7: $\quad\quad \mathcal{O}^* = \mathcal{O}$
8: **Renvoyer** $\mathcal{O}^*$

Combien de temps est nécessaire pour effectuer cet algorithme ? Pour chaque ordre possible, on effectue un appel à NBMULT, une condition et, possiblement, 2 affectations ; soit au pire $O(n) + 3$ instructions.

Combien y a-t-il d'ordres possibles ? Un ordre est une liste d'indice $(i_1, i_2, \ldots, i_{n-1})$. On a $n-1$ choix pour l'indice i_1, $n-2$ pour l'indice i_2, $n-3$ pour l'indice i_3, ... et un seul choix pour l'indice i_{n-1}. Donc il y a $(n-1) \cdot (n-2) \cdots (1) = n!$ ordres possibles.

Donc le calcul total de l'arg min a une complexité de l'ordre de $(O(n) + 3) \cdot n! = O(n \cdot n!)$.

Donc la complexité de l'algorithme N-MULT-OPT est $O(n \cdot n! + \text{NBMULT}(A_1, A_2, \ldots, A_n, \mathcal{O}^*))$. On ne peut pas dire mieux car on ne peut pas expliciter $\mathcal{O}^*$ sans l'avoir calculé. Tout ce qu'on peut dire c'est que meilleur est l'ordre $\mathcal{O}^*$ moins de temps prendra cet algorithme.

Exercice 2 — *Tableaux de taille dynamique*

On considère l'algorithme suivant :

Ajout d'un élément dans un tableau

Entrées: Un tableau T de taille maximum N contenant n éléments, un entier i

Sorties: Un tableau T' contenant les n de T plus l'entier i.

Si $n < N$ **Alors**
  $T[n+1] \leftarrow i$
  Renvoyer T
$T' \leftarrow$ un tableau de taille maximum $2N$
Copier les n éléments de T dans les n premières cases de T'
$T'[n+1] \leftarrow i$
Renvoyer T'

---

1. En commençant avec un tableau de taille $N = 1$, comment se déroulerait l'algorithme si dessus si on l'appelait 10 fois (avec les entiers de 1 à 10 par exemple).

▶ **Correction**

Voici ce qu'il se passe à chaque itération :

- Avant la première itération, on dispose d'un tableau de taille 1.
- On ajoute 1 au tableau.
- On essaie d'ajouter 2, le tableau est plein, on crée un tableau de taille 2, on copie 1 dans le nouveau tableau, et on ajoute 2.
- On essaie d'ajouter 3, le tableau est plein, on crée un tableau de taille 4, on copie 1 et 2 dans le nouveau tableau, et on ajoute 3.
- On ajoute 4 au tableau.
- On essaie d'ajouter 5, le tableau est plein, on crée un tableau de taille 8, on copie 1, 2, 3 et 4 dans le nouveau tableau, et on ajoute 5.
- On ajoute 6 au tableau.
- On ajoute 7 au tableau.
- On ajoute 8 au tableau.
- On essaie d'ajouter 9, le tableau est plein, on crée un tableau de taille 16, on copie 1, 2, 3, 4, 5, 6, 7 et 8 dans le nouveau tableau, et on ajoute 9.
- On ajoute 10 au tableau.

On a donc fait en tout 25 ajouts dans un tableau.

2. En commençant avec un tableau de taille $N = 1$, on suppose qu'on fait n insertions, montrer que la complexité de l'algorithme est de l'ordre de $O(n)$. En déduire que la complexité en moyenne est constante.

▶ **Correction**

Pour calculer la complexité en moyenne sur chaque insertion, il faut calculer la complexité de toutes les affectations dans un tableau divisée par le nombre d'insertion. (On parle en fait de **complexité amortie** ; le terme de **complexité en moyenne** est utilisé pour autre chose).

La complexité d'une insertion est de deux type :

- si le tableau n'est pas plein, la complexité est constante, on fait une seule affectation.
- si le tableau est plein, la complexité est la taille du tableau plus 1, on recopie le tableau plus une affectation.

Le second cas n'arrive que lorsque la taille du tableau est 2^p, avec $p \geq 0$, autrement dit au moment de la $2^p + 1^e$ insertion, pour tout $p \geq 0$, on fait $2^p + 1$ affectations.

Supposons qu'on insert $n = 2^q$ éléments, alors, il existe q insertions dans le second cas, les insertions numéros $2^0 + 1$, $2^1 + 1$, …, $2^{q-1} + 1$; et $2^q - q$ insertions dans le premier cas.

Le nombre total d'affectations est donc :

$$\begin{aligned}
&\sum_{p=0}^{q-1}(2^p + 1) + 2^q - q \\
&= \frac{2^q - 1}{2 - 1} + q + 2^q - q \\
&= 2^q - 1 + 2^q \\
&= 2^{q+1} - 1 \\
&= 2n - 1
\end{aligned}$$

Le nombre moyen d'affectation par insertion est donc inférieur à 2, donc constant.

Si maintenant n est entre 2^q et 2^{q+1}, disons $2^q + k$ avec $k < 2^{q+1} - 2^q = 2^q$. Le raisonnement est le même :

Il existe $q+1$ insertions dans le second cas, les insertions numéros 2^0+1, 2^1+1, …, $2^{q-1}+1$, $2^q + 1$; et $2^q + k - q - 1$ insertions dans le premier cas.

Le nombre total d'affectations est donc :

$$\begin{aligned}
&\sum_{p=0}^{q}(2^p + 1) + 2^q + k - q - 1 \\
&= \frac{2^{q+1} - 1}{2 - 1} + q + 1 + 2^q + k - q - 1 \\
&= 2^{q+1} - 1 + 2^q + k \\
&= 3 \cdot 2^q + k - 1 \\
&= 3n - 2k - 1
\end{aligned}$$

Le nombre d'affectation par insertion est donc inférieur à 3, donc constant. Le nombre est plus grand car quand n n'est pas une puissance de 2, on fait moins d'insertion en temps constant, donc on compense un peu moins les insertions qui nécessitent de copier le tableau.

Exercice 3 — *Tri*

1. **Tri par sélection**. L'algorithme est le suivant : connaissant un tableau T de n entiers, pour tout i de 0 à $n-1$, chercher l'entier minimum parmi les éléments d'indice i à $n-1$ et l'échanger avec l'entier placé en position i. Quelle est la complexité de cet algorithme ? Prouver qu'il se termine et qu'il répond correctement.

▶ **Correction**

La complexité de cet algorithme est $O(n^2)$; car i va parcourir les entiers de 0 à $n-1$; et pour chaque entier i, on recherche un minimum parmi les éléments d'indice i à $n-1$ qui peut se faire avec une simple boucle en $n - i$ opérations.

Donc on effectue en tout $\sum_{i=0}^{n-1} n - i$ opérations, et $\sum_{i=0}^{n-1} n - i = \frac{n(n+1)}{2} = O(n^2)$.

Il se termine sinon la complexité serait infinie. Pour prouver qu'il répond correctement, il faut montrrer que à chaque itération, les éléments de la liste d'indice 0 à $i-1$ sont bien triés. (C'est ce qu'on nomme **l'invariant de boucle**). C'est à dire que si T' est le tableau T trié, alors $T_j = T'_j$ pour tout $j \leq i - 1$.

On peut le faire par récurrence :

Si $i = 0$ alors il n'y a pas d'éléments d'indice 0 à $i - 1$, la propriété est donc vraie.

Si la propriété est vrai pour tout $j \leq i$, montrons qu'elle est vrai pour $i+1$. Pendant l'itération $i+1$, on cherche $\min\{T_j, j \geq i+1\}$ et on l'échange avec T_{i+1}. Montrons donc que $T'_{i+1} = \min\{T_j, j \geq i+1\}$.

Par hypothèse $T_j = T'_j$ pour tout $j \leq i$. Donc les éléments de T d'indice $i+1$ à $n-1$ sont aussi les éléments de T' d'indice $i+1$ à $n-1$ (mais ils ne sont pas dans le bon ordre) : donc $\{T_j, j \geq i+1\} = \{T'_j, j \geq i+1\}$. Par définition, T' est trié, donc $T'_{i+1} \leq T'_j$ pour $j \geq i+1$, donc $T'_{i+1} = \min\{T'_j, j \geq i+1\} = \min\{T_j, j \geq i+1\}$.

2. **Tri par dénombrement**. L'algorithme est le suivant : connaissant un tableau L de n entiers, créer un tableau S d'entiers dont la taille est $\max(L)+1$ dont la valeur initiale est 0. Pour chaque élément e de L, ajouter 1 à $S[e]$. Déduire de S la liste L triée. Quelle est la complexité de cet algorithme ? Prouver qu'il se termine et qu'il répond correctement.

▶ **Correction**

L'algorithme est le suivant :

Ajout d'un élément dans un tableau

Entrées: Une liste L

Sorties: Une liste L' égale à L triée

1: S une liste de $\max(L)+1$ entiers, tous égaux à 0
2: **Pour** $e \in L$ **Faire**
3: $\quad S[e] \leftarrow S[e]+1$
4: $L' \leftarrow \emptyset$
5: **Pour** e de 0 à $\max(L)$ **Faire**
6: $\quad$ Insérer $S[e]$ fois e dans L'
7: **Renvoyer** L'

Prenons un exemple, si $L = (1,4,0,3,2,3,4,2)$, on crée un tableau S avec $\max(L)+1 = 4+1 = 5$ éléments. A l'issue de l'algorithme, on a $S = (1,1,2,2,2)$. Le tableau S compte le nombre de fois que chaque élément apparaît dans le tableau. On peut ensuite simplement trier le tableau L avec S en créant une liste L' où, pour chaque indice i, on ajoute $S[i]$ fois l'entier i dans L' :

$L' = (0,1,2,2,3,3,4,4)$.

La complexité est de l'ordre de $|S|+|L|$ car on parcours une fois L pour remplir S, puis une fois S pour remplir L' ; plus exactement lorsqu'on parcours S, on lit chaque case (une opération) et on effectue une boucle de taille $S[i]$ ($S[i]$ opérations). Donc on effectue $1+S[i]$ opérations. Donc on effectue $|S| + \sum S[i]$ opérations lors du remplissage de L'. Et $\sum S[i] = |L|$.

Donc on effectue $|L| + |S| + |L|$ opérations en tout, la complexité est donc $|S| + 2|L| = O(|S| + |L|))O(\max(L) + |L|)$.

L'algorithme se termine (sinon sa complexité serait infinie).

Montrons qu'il répond correctement, en effet :

- on peut montrer assez facilement que la liste L' est triée, puisque la boucle ligne 6-7 tente d'insérer e dans L' avant de tenter d'insérer $e+1$.
- il reste donc à vérifier que cette boucle insère les bons éléments. Il faut donc montrer que L' est une permutation de L. Donc que pour tout élément e de L', L et L' contiennent autant de fois e.

  Supposons qu'à la ligne 6, on insère k fois un élément f dans L' ; montrons que L contient f k fois également. On sait donc que $S[f] = k$. Donc la ligne 3 a été effectuée k fois pour $e = f$, donc qu'il y a k itérations de la boucle ligne 2 où $e = f$; donc que f est présent k fois dans L.

3. — Combien existe-il de permutations possibles pour une liste L de n entiers ? Dans le pire cas, combien de permutations correspondent à la liste triée ?

▶ **Correction**

Il existe $n!$ permutations de L, au pire une seule correspond à la liste triée. Par exemple si $n = 4$, on a 24 permutations possibles.

— Si on effectue une comparaison de deux entiers de L, et aucune autre comparaison, combien existe-t-il au mieux de permutations dont on peut dire qu'elle ne correspondent pas à la liste triée ?

▶ **Correction**

Une fois qu'on a comparé 2 éléments de la liste, on peut supprimer au mieux la moitié des permutations. Par exemple, supposons qu'on sache que $L_1 < L_2$, on sait que, dans la liste triée, L_1 est avant L_2. Donc il nous reste les permutations suivantes :
$(L_1, L_2, L_3, L_4), (L_1, L_3, L_2, L_4), (L_1, L_3, L_4, L_2), (L_3, L_1, L_2, L_4), (L_3, L_1, L_4, L_2), (L_3, L_4, L_1, L_2)$
$(L_1, L_2, L_4, L_3), (L_1, L_4, L_2, L_3), (L_1, L_4, L_3, L_2), (L_4, L_1, L_2, L_3), (L_4, L_1, L_3, L_2), (L_4, L_3, L_1, L_2)$
Soit 12 permutations sur les 24.

Maintenant, si on compare L_2 et L_3, et qu'on trouve que $L_3 < L_2$, il nous reste les permutations suivantes :
(L_1, L_3, L_2, L_4), (L_1, L_3, L_4, L_2), (L_3, L_1, L_2, L_4), (L_3, L_1, L_4, L_2), (L_3, L_4, L_1, L_2)
(L_1, L_4, L_3, L_2), (L_4, L_1, L_3, L_2), (L_4, L_3, L_1, L_2)
Soit 8 permutations. On en a enlevé moins que la moitié.

Si au lieu de comparer L_2 et L_3 on avait comparé L_3 et L_4, et qu'on avait remarqué que $L_4 < L_3$, on aurait eu les 6 permutations suivantes :
$(L_1, L_2, L_4, L_3), (L_1, L_4, L_2, L_3), (L_1, L_4, L_3, L_2), (L_4, L_1, L_2, L_3), (L_4, L_1, L_3, L_2), (L_4, L_3, L_1, L_2)$.

Il existe donc des comparaisons plus efficaces que d'autres, mais, dans le pire cas, on ne peut pas supprimer plus de la moitié des permutations.

— En déduire le nombre minimum de comparaison qu'il faut effectuer pour obtenir une liste triée ; et donc la complexité minimum d'un algorithme de tri.

▶ **Correction**

Il faut donc effectuer au moins $\log_2(n!)$ comparaisons pour toutes les supprimer sauf une dans le pire cas, c'est à dire le nombre de fois qu'il faut diviser $n!$ par 2 pour tomber à 1.

Pour n'importe quel algorithme de tri, dans le pire cas, il devra donc faire $\log_2(n!) = n\log_2(n)$ comparaisons. On dit que sa complexité des $\Omega(n\log_2(n))$.

La notation O est utilisée pour désigner une borne supérieure de la complexité et la notation Ω est utilisée pour désigner une borne inférieure.

— Pourquoi le tri par dénombrement parvient à passer en dessous de cette barrière ?

▶ **Correction**

La complexité $\Omega(n\log n)$ est le nombre minimum de comparaisons que doit faire un algorithme de tri.

Le tri par dénombrement ne fonctionne pas avec des comparaisons, donc sa complexité n'est pas soumise aux mêmes contraintes.