Série 2 : Solutions

1 Complexités temporelles

algorithme 1 : La complexité temporelle est $\Theta(n)$: dans le pire des cas (à savoir quand tous les nombres de la liste L sont plus petits ou égaux à M), la boucle "tant que" est exécutée n fois, car le nombre i est incrémenté de 1 à chaque passage de la boucle.

algorithme 2 : La complexité temporelle est également $\Theta(n)$: de l'ordre de $2n$ opérations sont effectuées, mais le facteur 2 ne nous importe pas ici.

algorithme 3 : La complexité temporelle est $\Theta(n^2)$: ici, il y a deux boucles imbriquées, et le nombre d'instructions effectuées, dans le pire des cas (qui correspond au cas où toutes les différences $|L(i) - L(j)|$ sont plus petites ou égales à x), est de l'ordre du nombre de paires d'éléments dans l'ensemble $\{1, \ldots, n\}$, qui vaut $\frac{n(n-1)}{2}$.

algorithme 4 : Malgré le fait qu'il y ait aussi deux boucles imbriquées dans cet algorithme, sa complexité temporelle est $\Theta(n)$: le nombre d'opérations effectuées à l'intérieur de chaque boucle "tant que" est au plus au plus égal à 5.

2 Création d'algorithmes

a)

moyenne arithmétique

entrée : *liste L de nombres réels, de taille n*
sortie : *nombre réel m_A*

$m_A \longleftarrow 0$
Pour *i allant de* 1 *à n*
  $m_A \longleftarrow m_A + L(i)$
$m_A \longleftarrow m_A/n$
Sortir : m_A

b) En utilisant l'indication de l'énoncé, on remarque que $\log_2(m_G) = \frac{1}{n}\left(\log_2(L(1)) + \cdots + \log_2(L(n))\right)$, et donc on peut écrire :

moyenne géométrique

entrée : *liste L de nombres réels positifs, de taille n*
sortie : *nombre réel positif m_G*

$M \longleftarrow$ liste de longueur n (initialisée avec des zéros, par exemple)
Pour *i allant de* 1 *à n*
  $M(i) \longleftarrow \log_2(L(i))$
$m_G \longleftarrow 2^{\textbf{moyenne arithmétique}(M,n)}$
Sortir : m_G

Note : L'initialisation de la liste M n'est pas forcément nécessaire (suivant le langage de programmation utilisé).

c) Voici une possibilité : (mais on peut penser à d'autres façons de faire)

le plus grand produit

entrée : *liste L de nombres réels positifs, de taille n*
sortie : *nombre réel positif r*

$x_1 \longleftarrow \max(L(1), L(2))$
$x_2 \longleftarrow \min(L(1), L(2))$
Pour i *allant de* 3 *à* n
  Si $L(i) > x_1$
    $x_2 \longleftarrow x_1$
    $x_1 \longleftarrow L(i)$
  Sinon
    Si $L(i) > x_2$
      $x_2 \longleftarrow L(i)$
Sortir : $x_1 \times x_2$

d) La complexité temporelle de l'algorithme ci-dessus est $\Theta(n)$.

3 Trier un tableau

a) La complexité temporelle de l'algorithme de tri par insertion est $\Theta(n^2)$ dans le pire des cas (qui correspond au tableau trié initialement à l'envers) : chacune des insertions de l'algorithme peut en effet coûter de l'ordre de n opérations, et de l'ordre de n insertions sont effectuées.

b) Une première chose à remarquer est qu'il ne suffit pas de trier chaque ligne du tableau A séparément. En effet, si on fait ceci, on obtient à partir du tableau A donné en exemple :

$$A' = \begin{pmatrix} 3 & 4 & 7 & 8 & 11 & 12 & 14 \\ 2 & 3 & 5 & 5 & 12 & 13 & 15 \end{pmatrix}$$

où on voit par exemple que la première colonne n'est pas ordonnée comme on le voudrait. Une deuxième étape est donc nécessaire, qui consiste à ordonner les colonnes une à une. En suivant les notations de l'énoncé, voici l'algorithme complet :

tri de tableau

entrée : *tableau A de dimensions $n \times 2$*
sortie : *tableau trié A''*

$A'(1) \longleftarrow$**tri par insertion**$(A(1))$
$A'(2) \longleftarrow$**tri par insertion**$(A(2))$
Pour i *allant de* 1 *à* n
  $A''(1,i) \longleftarrow \min(A'(1,i), A'(2,i))$
  $A''(2,i) \longleftarrow \max(A'(1,i), A'(2,i))$
Sortir : A''

Dans l'exemple de l'énoncé, cet algorithme produit le tableau :

$$A'' = \begin{pmatrix} 2 & 3 & 5 & 5 & 11 & 12 & 14 \\ 3 & 4 & 7 & 8 & 12 & 13 & 15 \end{pmatrix}$$

qui est bien une version triée du tableau A d'origine.

Ceci dit, il importe maintenant de *vérifier* que cet algorithme aboutit à une version triée du tableau pour *tout* tableau de départ A, et pas seulement pour l'exemple ci-dessus : clairement, il est vrai que $A''(1,i) \leq A''(2,i)$ pour tout i du fait de la dernière étape de l'algorithme. Reste à vérifier que $A''(1,i) \leq A''(1,j)$ et $A''(2,i) \leq A''(2,j)$ pour tout $i < j$ (la dernière étape de l'algorithme ne garantit pas en effet que l'ordre dans chaque ligne est conservé). Pour cela, raisonnons par l'absurde et supposons que l'une de ces deux inégalités ne soit pas vérifiée, par exemple, que $A''(1,i) > A''(1,j)$ pour une paire $i < j$ donnée. Ceci veut dire que

$$\min(A'(1,i), A'(2,i)) > \min(A'(1,j), A'(2,j))$$

mais comme les lignes de A' sont triées, on sait que $A'(1,i) \leq A'(1,j)$ et $A'(2,i) \leq A'(2,j)$; il n'y donc aucune possibilité que l'inégalité ci-dessus soit satisfaite, ce qui prouve que notre hypothèse ne peut être vraie et donc que $A''(1,i) \leq A''(1,j)$. Le même raisonnement vaut pour le max et l'inégalité $A''(2,i) \leq A''(2,j)$.

Note : Il est aussi possible de :

- trier d'abord les colonnes puis les lignes du tableau (on arrive alors à la version triée de l'énoncé).

- réunir les deux lignes du tableau $A(1)$ et $A(2)$ en une seule ligne L de taille $2n$, puis trier cette liste L avec le tri par insertion, et finalement réarranger cette liste en un tableau de taille $n \times 2$. Pour cela, on a deux possibilités (en fait, même beaucoup plus que ça) :

$$\begin{pmatrix} L(1) & L(2) & \dots & L(n-1) & L(n) \\ L(n+1) & L(n+2) & \dots & L(2n-1) & L(2n) \end{pmatrix}$$

ou

$$\begin{pmatrix} L(1) & L(3) & \dots & L(2n-3) & L(2n-1) \\ L(2) & L(4) & \dots & L(2n-2) & L(2n) \end{pmatrix}$$

c) La complexité temporelle du premier algorithme ci-dessus est $\Theta(n^2)$: la partie qui demande en effet le plus de temps de calcul est celle constituée des deux tris par insertion (effectués l'un après l'autre) ; la dernière boucle (en $\Theta(n)$) prend un temps négligeable en comparaison. Notez que la complexité du dernier algorithme proposé ci-dessus est également $\Theta(n^2)$, mais prend plus de temps en pratique, car trier une liste de taille $2n$ avec le tri par insertion prend deux fois plus de temps que trier deux sous-listes de taille n (exercice pour vous !).

4 Analyse d'un algorithme de tri

On considère l'algorithme suivant :

tri par insertion
entrée : *liste L de taille n* sortie : *liste L triée*

Pour *i allant de 2 à n*
  Si $L(i) < L(i-1)$
  $L \longleftarrow$ *insérer*(L, i)
Sortir : L

où le sous-algorithme `insérer` est défini par :

insérer
entrée : *liste L, indice i* sortie : *liste L*

Tant que $i > 1$ **et** $L(i) < L(i-1)$
  $L \longleftarrow$ *permuter*$(L, i, i-1)$
  $i \longleftarrow i - 1$
Sortir : L

On exécute cet algorithme sur une liste de 100 éléments. Après 50 itérations de la boucle principale (c'est-à-dire lorsque $i = 51$), on observe que `permuter` a été appelé exactement 1275 fois au total. Que peut-on déduire ?

- [ ] La liste était déjà triée
- [ ] La liste était triée en ordre décroissant
- [x] Les 51 premiers éléments étaient exactement en ordre inverse
- [ ] On ne peut rien conclure sans plus d'informations

Idée. Quand la liste est en ordre inverse, à chaque étape i, l'élément courant doit reculer d'un certain nombre de positions. Les nombres de permutations par étape forment une suite arithmétique.

Exemple simple. Pour 5 éléments entièrement inversés, les permutations effectuées à chaque insertion sont :

$$a_1 = 1, \quad a_2 = 2, \quad a_3 = 3, \quad a_4 = 4.$$

Le total après les 4 premières insertions est

$$S_4 = a_1 + a_2 + a_3 + a_4.$$

Pour une suite arithmétique de premier terme a_1, de dernier terme a_n et de longueur n, on a la formule :

$$S_n = a_1 + a_2 + \cdots + a_n = \frac{(a_1 + a_n)\, n}{2}.$$

Ici $a_1 = 1$, $a_n = 4$, $n = 4$, donc :

$$S_4 = \frac{(1+4) \times 4}{2} = 10.$$

Cas général. Après $i = 51$, les permutations cumulées valent :

$$S_{50} = 1 + 2 + 3 + \cdots + 50,$$

soit une suite arithmétique de $n = 50$ termes, avec $a_1 = 1$ et $a_{50} = 50$. D'après la formule précédente :

$$S_{50} = \frac{(1+50) \times 50}{2} = 1275.$$

Conclusion. Comme on observe exactement 1275 permutations à $i = 51$, cela correspond au cas où les 51 premiers éléments étaient en ordre strictement décroissant. On ne peut pas en déduire l'ordre des 49 derniers éléments.

5 Complexité asymptotique

Considérons trois algorithmes :
- Algorithme A : effectue $5n^2 + 100n\log(n) + 1000$ opérations
- Algorithme B : effectue $0.001n^3$ opérations

À partir de quelle valeur approximative de n l'algorithme B devient-il plus lent que l'algorithme A ?

- ☐ $n > 100$
- ☐ $n > 1000$
- ☒ $n > 5000$
- ☐ B est toujours plus rapide que A

Objectif. Comparer A et B en résolvant

$$0.001\, n^3 \overset{?}{>} 5n^2 + 100n \log n + 1000.$$

Étape 1. Divisons par n^2 (pour $n \geq 1$) :

$$0.001\, n \overset{?}{>} 5 + 100\, \frac{\log n}{n} + \frac{1000}{n^2}.$$

Étape 2. Terme dominant. Pour les grandes valeurs de n, les termes $100\, \dfrac{\log n}{n}$ et $\dfrac{1000}{n^2}$ deviennent petits. On obtient une condition de premier ordre :

$$0.001\, n > 5 \quad \Longrightarrow \quad n > 5000.$$

Remarque. Si on tient compte du terme $100n \log n$, la croissance de A est légèrement plus rapide que celle donnée par $5n^2$ seul. Il faut donc une valeur un peu plus grande pour que le terme en n^3 de B domine. Le croisement réel se produit autour de $n \approx 5200$.

Conclusion. Parmi les choix proposés, la bonne réponse est $\boxed{n > 5000}$.

6 Réutilisation d'un algorithme existant

On considère l'algorithme suivant qui cherche deux éléments d'une liste ordonnée dont la somme vaut une valeur cible T :

deux font la paire

entrée : *liste ordonnée L de n entiers, valeur cible T*
sortie : *booléen*

```
i ⟵ 1
j ⟵ n
Tant que i < j
   Si L(i) + L(j) = T
      Sortir : Vrai
   Si L(i) + L(j) < T
      i ⟵ i + 1
   Sinon
      j ⟵ j − 1
Sortir : Faux
```

On souhaite maintenant résoudre un problème similaire : trouver trois indices $i < j < k$ tels que $L(i) + L(j) + L(k) = 0$ dans une liste ordonnée. En utilisant une approche similaire (une boucle externe sur i, puis l'algorithme ci-dessus sur le reste), quelle est la complexité obtenue ?

- ☐ $\Theta(n)$
- ☒ $\Theta(n^2)$
- ☐ $\Theta(n^3)$
- ☐ $\Theta(n \log n)$

Écrivez le sous-algorithme correspondant à la boucle externe qui appelle `deux font la paire` pour chaque valeur de i. L'algorithme doit retourner `Vrai` s'il existe trois éléments dont la somme est nulle, et `Faux` sinon.

Idée. Pour chaque i, on cherche deux indices $j < k$ tels que

$$L(j) + L(k) = -L(i).$$

On peut directement réutiliser l'algorithme `deux font la paire` avec la valeur cible $T = -L(i)$.

trois font le trio

entrée : *liste ordonnée L de n entiers*
sortie : *booléen*

```
Pour i de 1 à n − 2
   Si deux_font_la_paire(L[i + 1..n], −L(i))
      Sortir : Vrai
Sortir : Faux
```

Complexité.

- Pour un i donné, l'algorithme *deux font la paire* parcourt la sous-liste $L[i+1..n]$ une seule fois, soit un coût $\Theta(n)$.
- On répète cette recherche pour environ n valeurs de i dans la boucle externe.

Ainsi, on obtient :

$$\Theta(n) \times \Theta(n) = \Theta(n^2).$$

Conclusion. L'approche basée sur deux pointeurs appliquée dans une boucle externe a donc une complexité en temps $\boxed{\Theta(n^2)}$, bien plus efficace que la recherche exhaustive en $\Theta(n^3)$.