Série 8

Les exercices précédés d'une astérisque sont optionnels (mais pas nécessairement difficiles). Vous pouvez utiliser le notebook `cours8_complement.ipynb` mis à votre disposition sur Moodle pour visualiser le temps de parcours de vos algorithmes en fonction de la taille de l'entrée.

---

1. Ordonnez sans justification les fonctions suivantes par ordre de croissance, c'est-à-dire pour deux fonctions données f et g, faites apparaître f avant g si $f = \mathcal{O}(g)$. Groupez ensemble les fonctions qui ont le même ordre de croissance (c'est-à-dire groupez ensemble f et g si $f = \Theta(g)$).

$$n + 50,\ n\sqrt{n},\ 2n,\ 100,\ 10n^{0.1},\ n^2,\ n^3 + n^2,\ n^3,\ n^{1.1},\ |\sin(n)| + 1,\ n^{10}.$$

Pour vous aider dans cette tâche, vous trouverez ci-dessous le code Python qui a été utilisé en cours pour illuster la vitesse de croissance de diverses fonctions (le code donné produit le graphe des fonctions $f(n) = n$ et $g(n) = n^2$ pour n allant de 1 à 100). Il utilise le module `matplotlib`, qui donne accès à un ensemble de fonctions de visualisation en Python. Le sous-module `pyplot` est une collection de fonctions qui permettent de visualiser des graphes de fonctions mathématiques dans un style similaire au langage de programmation MATLAB.

Adaptez ce code pour vérifier votre réponse en dessinant le graphes des fonctions que vous avez ordonnées, groupées par deux ou par trois, et observant leur croissance relative. N'oubliez pas de modifier la plage des valeurs de n au besoin.

```
import matplotlib.pyplot as plt

n = []
f = []
g = []
for i in range(1,101):
    n.append(i)
    f.append(i)
    g.append(i**2)

plt.plot(n, f, label = 'n')
plt.plot(n, g, label = 'n^2')
plt.legend()
plt.show()
```

Remarque: `matplotlib` est installé sur Noto. Si vous travaillez sur votre machine et n'avez pas `matplotlib` installé, vous pouvez l'installer en exécutant `pip install matplotlib` dans une ligne de commande ou directement dans un Jupyter notebook. Vous pouvez aussi utiliser un outil de plotting en ligne comme Wolfram Alfa, Geogebra, Desmos, ou une multitude d'autres.

2. Pour $n \in \mathbb{N}$, prouvez les affirmations suivantes en exhibant une constante C (ou des constantes C_1 et C_2) et un rang N appropriés:

   (a) $2n + 100 = \Theta(n)$

   (b) $an + b = \Theta(n)$ pour a, b des réels strictement positifs

   (c) $100n\sqrt{n} = \mathcal{O}(n^2)$.

3. (a) Soit $n \in \mathbb{N}$, et f, g, h des fonctions positives de n telles que $f = \mathcal{O}(g)$ et $g = \mathcal{O}(h)$. Prouvez que $f = \mathcal{O}(h)$.

   (b) Soient $T'(n)$ et $T''(n)$ les temps de parcours respectifs des algorithmes `max_somme` et `max_somme_lineaire` tels qu'ils ont été définis au cours. Déduisez du point (a) que $T''(n) = \mathcal{O}(T'(n))$.

4. (a) Donnez un algorithme `carre` qui prend un entier positif n en entrée et affiche un carré de côté n. Par exemple, l'appel `carre(5)` doit afficher:

   ```
   * * * * *
   * * * * *
   * * * * *
   * * * * *
   * * * * *
   ```

   L'espaçage entre les astérisques n'est pas important.

   (b) Combien d'astérisques est-ce que votre code affiche, pour l'entrée n?

   (c) Donnez, en notation $\Theta(\cdot)$, l'ordre de croissance du temps de parcours de votre algorithme en fonction de n.

   Remarque: si votre algorithme contient des instructions de la forme `print("*" * i)`, sachez qu'une telle instruction ne prend pas temps constant! En effet la création et le stockage en mémoire d'une chaîne de caractère de taille n prendra un temps (et un espace en mémoire) au moins linéaire en n.

   Un algorithme contenant deux boucles `for` imbriquées prendra le même temps de parcours mais ce temps de parcours sera plus facile à analyser.

   (d) Donnez un algorithme `triangle` qui prend un entier positif n en entrée et affiche un triangle de côté n de la forme ci-dessous (dans ce cas, ce triangle a été affiché par `triangle(5)`):

   ```
   *
   * *
   * * *
   * * * *
   * * * * *
   ```

   Puis répondez aux mêmes questions (b) et (c) pour l'algorithme `triangle`.

5. Soit `L` une liste de $n \geq 3$ nombres. On s'intéresse à la plus grande somme de trois éléments distincts de la liste, c'est-à-dire au maximum de la valeur de

   `L[i] + L[j] + L[k]`, pour `i, j, k` des indices de `L` distincts deux à deux.

(a) Modifiez chacun des algorithmes `max_somme` et `max_somme_lineaire` vus en cours pour calculer le maximum demandé.

(b) Utilisez le module `time` comme vu en cours pour mesurer le temps de parcours de chacun de vos algorithmes sur une liste de nombre aléatoires dont vous ferez varier la taille. Est-ce que c'est soutenable de donner une liste de taille 1000 à ces deux algorithmes? Une liste de taille $10,000$?

(c) Donnez, en notation $\Theta(\cdot)$, l'ordre de croissance du temps de parcours de chacun des deux algorithmes en fonction de la taille de l'entrée.

Indication: vous pouvez utiliser les identités

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \text{ et } \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}.$$

* 6. (a) Prouvez l'équivalence des deux définitions de $f = \Omega(g)$ données au transparent 31 du cours.

(b) Prouvez l'équivalence des deux définitions de $f = \Theta(g)$ données au transparent 32 du cours.

7. **Exercice de révision**[1]

On considère le problème de la multiplication de deux matrices numériques de dimensions $n \times n$, pour $n \geq 2$.

Pour rappel, soient A et B des matrices de dimensions $n \times n$, et $C = AB$ la matrice produit de A et B. On dénote par a_{ij} l'élément de la $i$ème ligne et $j$ème colonne de A (et de même pour B et C). Alors

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

C'est-à-dire que c_{ij} est obtenu en calculant le produit scalaire de la $i$ème ligne de A et la $j$ème colonne de B. Pour calculer l'entrée c_{ij} de la matrice produit C, il faut donc calculer une somme où chaque terme est le produit d'une entrée de A et d'une entrée de B.

Par exemple, si

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & -1 & 2 \\ 5 & -2 & 1 \end{pmatrix} \text{ et } B = \begin{pmatrix} 2 & -1 & -2 \\ 0 & 3 & 2 \\ 1 & 4 & 3 \end{pmatrix},$$

alors

$$c_{11} = 1\cdot 2 + 2\cdot 0 + 3\cdot 1 = 5,\ c_{12} = 1\cdot -1 + 2\cdot 3 + 3\cdot 4 = 17,\ \text{et } C = \begin{pmatrix} 5 & 17 & 11 \\ 10 & 1 & -4 \\ 11 & -7 & -11 \end{pmatrix}.$$

[1] Cet exercice constitue un bon entraînement pour comprendre les notions de ce cours mais n'est pas nécessaire à la compréhension des prochains cours. Vous pouvez donc le garder pour vos révisions si vous trouvez la série trop longue.

(a) Ecrivez un algorithme qui prend en entrée deux matrices numériques A et B de dimensions $n \times n$ et retourne la matrice $C = A \cdot B$. On représentera une matrice de dimensions $n \times n$ en Python par une liste de taille n dont chaque élément est une liste de taille n. Par exemple, la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

sera représentée par `A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.

N'oubliez pas de tester votre code sur plusieurs instances de petite taille.

(b) Quelle est la taille de l'entrée en fonction de n? Quel est le temps de parcours de votre algorithme en fonction de n?

(c) Donnez une borne inférieure (triviale) sur le temps de parcours de n'importe quel algorithme qui résoud le problème de la multiplication de deux matrices de dimensions $n \times n$.