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.

Solution. Voici les fonctions données ordonnées par ordre de croissance, où on a groupé avec des accolades les fonctions qui ont le même ordre de croissance:

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

Quelques explications:

- 100 est une constante, et on peut borner $|\sin(n)|+1$ par deux constantes (par exemple, $1 \leq |\sin(n)|+1 \leq 2$). Une fonction constante ou bornée par deux constantes est $\Theta(n^0)$, c'est-à-dire $\Theta(1)$.
- $n+50$ et $2n$ sont toutes deux $\Theta(n)$.
- n^3+n^2 et n^3 sont toutes deux $\Theta(n^3)$.
- En utilisant le fait que pour l'inégalité stricte $p<q$ on a $n^p = \mathcal{O}(n^q)$ (et n^q pas $\mathcal{O}(n^p)$), on déduit de l'ordre des exposants

  $$0 < 0.1 < 1 < 1.1 < 1.5 < 2 < 3 < 10$$

  l'ordre des fonctions correspondantes.

Pour mieux visualiser ces ordres relatifs de croissance, on utilise le code Python donné pour produire les graphes donnés dans l'annexe à la question 1 (fichier AnnexeQ1.pdf).

---

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

Solution.

(a) D'une part,

$$2n+100 \leq 102n \text{ pour tout } n \geq 1,$$

il suffit donc de prendre $C=102$ et $N=1$ pour prouver que $2n+100 = \mathcal{O}(n)$.
D'autre part, comme

$$n \leq 2n+100 \text{ pour tout } n \geq 1,$$

en prenant $C=1$ et $N=1$ on obtient que $n = \mathcal{O}(2n+100)$.
On a donc bien que $2n+100 = \Theta(n)$.

(b) Puisque

$$an + b \leq (a + b)n \text{ pour tout } n \geq 1,$$

il suffit donc de prendre $C = a + b$ et $N = 1$ pour prouver que $an + b = \mathcal{O}(n)$. D'autre part, comme

$$n = \frac{1}{a} \cdot an \leq \frac{1}{a}(an + b) \text{ pour tout } n \geq 1,$$

en prenant $C = \frac{1}{a}$ et $N = 1$ on obtient que $n = \mathcal{O}(an + b)$.
On a donc bien que $an + b = \Theta(n)$.

(c) Il s'agit de trouver des réels $C, N > 0$ tels que

$$\forall n > N \quad 100n\sqrt{n} < Cn^2.$$

Or pour $n > 0$,

$$100n\sqrt{n} < Cn^2 \Leftrightarrow 100\sqrt{n} < Cn \Leftrightarrow \sqrt{n} > \frac{100}{C}.$$

On veut donc trouver des réels $C, N > 0$ tels que

$$\forall n > N \quad \sqrt{n} > \frac{100}{C}.$$

On peut par exemple choisir $C = 100, N = 1$, ou bien $C = 1, N = 10000$, ou encore $C = 10, N = 100$, ou une infinité d'autre choix.

---

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 $T''(n) = \mathcal{O}(T'(n))$.

Solution.

(a) On a par définition de $f = \mathcal{O}(g)$ qu'il existe $C_1, N_1 > 0$ tels que

$$\forall n > N_1 \;\; f(n) \leq C_1 \cdot g(n). \tag{1}$$

On a également par définition de $g = \mathcal{O}(h)$ qu'il existe $C_2, N_2 > 0$ tels que

$$\forall n > N_2 \;\; g(n) \leq C_2 \cdot h(n). \tag{2}$$

En choisissant $C = C_1 \cdot C_2$ et $N = \max(N_1, N_2)$, on a par (1) et (2) que

$$\forall n > N \;\; f(n) \leq C \cdot h(n).$$

On a donc bien que $f = \mathcal{O}(h)$.

(b) On a vu au cours que $T''(n) = \Theta(n)$ et donc en particulier $T''(n) = \mathcal{O}(n)$. On a également vu que $T'(n) = \Theta(n^2)$, donc en particulier $n^2 = \mathcal{O}(T'(n))$. Puisqu'on a de plus que $n = \mathcal{O}(n^2)$, on applique deux fois la propriété de transitivité prouvée au point (a) pour obtenir 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`.

Solution.

(a) Il s'agit d'afficher n lignes de n astérisques chacune. On utilise deux boucles `for` imbriquées:

```
1 def carre(n):
2     for i in range(1, n+1):
3         for j in range(1, n+1):
4             print("*", end = " ")
5         print()
```

(b) Pour l'entier n en entrée, le nombre d'astérisques total affiché est n^2.

(c) Chaque instruction de cet algorithme s'exécute en temps constant. L'instruction `print` de la ligne 4 est exécutée n^2 fois. L'instruction `print()` de la ligne 5 est exécutée n fois (une fois pour chaque itération de la boucle extérieure). Le temps de parcours total de l'algorithme est donc $\Theta(n^2)$.

(d) Il s'agit d'afficher n lignes, telles que la $i$ème ligne contient i astérisques. Le code suivant produit le triangle demandé:

```
1 def triangle(n):
2     for i in range(1, n+1):
3         for j in range(1, i+1):
4             print("*", end = " ")
5         print()
```

– Pour l'entier n en entrée, le nombre d'astérisques total affiché est

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

– Chaque instruction de cet algorithme s'exécute en temps constant. L'instruction `print` de la ligne 4 est exécutée $n(n+1)/2$ fois. L'instruction `print` de la ligne 5 est exécutée n fois (une fois pour chaque itération de la boucle extérieure). Le temps de parcours total de l'algorithme est donc $\Theta(n^2)$.

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

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

Solution.

(a) On modifie l'algorithme `max_somme` de la manière suivante pour obtenir un algorithme qui calcule la plus grande somme de trois éléments distincts d'une liste `L`.

```python
def max_somme_3(L):
    '''
    Entree: liste L de nombres de taille n >= 3
    Sortie: Somme maximale de trois elements de L
    '''
    n = len(L)
    max_s = L[0] + L[1] + L[2]

    for i in range(n):
        for j in range(i+1, n):
            for k in range(j+1, n):
                if L[i] + L[j] + L[k] > max_s:
                    max_s = L[i] + L[j] + L[k]

    return max_s
```

On peut calculer la même valeur en modifiant l'algorithme `max_somme_lineaire` de la manière suivante (où la fonction `max_liste` est celle donnée en cours).

```python
def max_somme_lineaire_3(L):
    '''
    Entree: liste L de nombres de taille n >= 3
    Sortie: Somme maximale de trois elements de L
    '''
    n = len(L)
    max1 = max_liste(L)
    L.remove(max1)
    max2 = max_liste(L)
    L.remove(max2)
    max3 = max_liste(L)

    return max1 + max2 + max3
```

(b) Le code ci-dessous vous permettra de comparer le temps de parcours des deux algorithmes sur des listes de diverses tailles.

```python
from time import time
from random import randrange

L = []
for i in range(100):
    L.append(randrange(1000))

t0 = time()
m1 = max_somme_lineaire_3(L)
t1 = time()
print(f"l'appel a max_somme_lineaire_3(L) a pris {t1 - t0} secondes.")

t0 = time()
m2 = max_somme_3(L)
t1 = time()
print(f"l'appel a max_somme_3(L) a pris {t1 - t0} secondes.")
```

Alors que `max_somme_lineaire_3` tourne assez vite sur de grandes entrées (un échantillon de temps de parcours obtenus: environ 0.0005 secondes pour une liste de taille 10^4; environ 0.004 secondes pour une liste de taille 10^5; environ

0.03 secondes pour une liste de taille 10^6) , `max_somme_3` commence déjà à devenir inutilisable pour des listes avec quelques milliers d'entrées (environ 17.1 secondes pour une liste de taille $1,000$; environ 154 secondes pour une liste de taille 2000).

(c) **Temps de parcours de `max_somme_3`:**

Chaque instruction s'exécute en temps constant. On peut intuitivement voir que l'instruction `if` de la boucle intérieure est exécutée $\Theta(n^3)$ fois à cause des trois boucles imbriquées. Pour une preuve plus formelle, on peut compter le nombre d'exécutions de la boucle intérieure pour chaque valeur de i.

- $i = 0$: pour $j = 1$, l'instruction `if` est exécutée $n - 2$ fois. Pour $j = 2$, elle est exécutée $n - 3$ fois, etc. Donc pour $i = 0$, l'instruction `if` est exécutée

$$(n-2) + (n-3) + \cdots + 1 = \frac{(n-1)(n-2)}{2} \text{ fois.}$$

- $i = 1$: l'instruction `if` est exécutée

$$(n-3) + (n-4) + \cdots + 1 = \frac{(n-2)(n-3)}{2} \text{ fois.}$$

- $i = 2$: l'instruction `if` est exécutée $\frac{(n-3)(n-4)}{2}$ fois, et ainsi de suite.

L'instruction `if` est donc exécutée au total

$$\sum_{i=0}^{n-2} \frac{(n-i-1)(n-i-2)}{2} \text{ fois.}$$

Avec un changement de variable $k = n - i$, cette somme est égale à

$$\sum_{k=2}^{n} \frac{(k-1)(k-2)}{2}.$$

En utilisant les identités données en indication, on obtient bien un total de $\Theta(n^3)$ nombre d'exécutions de l'instruction `if`, et donc un temps de parcours $\Theta(n^3)$.

Temps de parcours de `max_somme_lineaire_3`:

Chacun des appels à `max_liste` et à la méthode `L.remove()` prend un temps linéaire en n. Le temps de parcours de l'algorithme est donc $\Theta(n)$.

* 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.

Solution.

(a) Il s'agit de prouver que les deux affirmations suivantes sont équivalentes:

1. $g = \mathcal{O}(f)$
2. Il existe $C, N > 0$ tels que $\forall n > N \;\; f(n) \geq C \cdot g(n)$.

Si l'affirmation 1 est vraie, on a par définition de $g = \mathcal{O}(f)$ qu'il existe $C', N' > 0$ qui satisfont

$$\forall n > N' \;\; g(n) \leq C' \cdot f(n).$$

En prenant de tels C' et N', et en choisissant $C = 1/C'$ et $N = N'$, C et N (tous deux strictement positifs) satisfont $\forall n > N \;\; f(n) \geq C \cdot g(n)$. Donc l'affirmation 1 implique l'affirmation 2.

D'autre part, si l'affirmation 2 est vraie, c'est-à-dire s'il existe $C, N > 0$ tels que $\forall n > N \;\; f(n) \geq C \cdot g(n)$, en choisissant $C' = 1/C$ et $N' = N$, on a que C' et N' sont strictement positifs et satisfont $\forall n > N' \;\; g(n) \leq C' \cdot f(n)$ et donc on a bien que $g = \mathcal{O}(f)$. Donc l'affirmation 2 implique l'affirmation 1.

(b) Il s'agit de prouver que les deux affirmations suivantes sont équivalentes:

1. $f = \mathcal{O}(g)$ et $g = \mathcal{O}(f)$
2. Il existe $C_1, C_2, N > 0$ tels que $\forall n > N \;\; C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$.

Supposons d'abord que l'affirmation 1 est vraie. Alors

- Puisque $f = \mathcal{O}(g)$, il existe $C_2, N_2 > 0$ tels que $\forall n > N_2 \;\; f(n) \leq C_2 \cdot g(n)$.
- Puisque $g = \mathcal{O}(f)$, alors $f = \Omega(g)$ et donc il existe $C_1, N_1 > 0$ tels que $\forall n > N_1 \;\; f(n) \geq C_1 \cdot g(n)$.

Pour ces mêmes valeurs de C_1 et C_2, et en prenant $N = \max(N_1, N_2)$, on a que

$$\forall n > N \;\; C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n).$$

L'affirmation 1 implique donc bien l'affirmation 2.

D'autre part, si l'affirmation 2 est vraie, c'est-à-dire s'il existe $C_1, C_2, N > 0$ tels que $\forall n > N \;\; C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$, alors en particulier il existe $C_1, N > 0$ tels que

$$\forall n > N \;\; f(n) \geq C_1 \cdot g(n),$$

et donc on a $f = \Omega(g)$ et donc $g = \mathcal{O}(f)$.
On a aussi en particulier qu'il existe $C_2, N > 0$ tels que

$$\forall n > N \;\; f(n) \leq C_2 \cdot g(n),$$

et donc $f = \mathcal{O}(g)$. L'affirmation 2 implique donc que $g = \mathcal{O}(f)$ et que $f = \mathcal{O}(g)$, elle implique donc l'affirmation 1.

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

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

(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 $n \times n$.

[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.

Solution.

(a) On donne ci-dessous un algorithme simple qui utilise la formule de multiplication de matrices pour calculer le produit de deux matrices carrées. On inclut un test de l'algorithme avec deux matrices de taille 3×3.

```
def mult_matrices(A, B):
    '''
    Entree: matrices A, B de taille n x n, n>=2
    Sortie: matrice produit A*B
    '''
    n = len(A[0])
    C = []
    for i in range(n):
        C.append([])
        for j in range(n):
            s = 0
            for k in range(n):
                s += A[i][k] * B[k][j]
            C[i].append(s)

    return C

A = [[1, 2, 3], [4, 0, 2], [0, -1, 1]]
B = [[2, 5, 3], [1, -1, 3], [2, 0, 2]]
print(mult_matrices(A,B))
```

On appelle parfois ce type d'algorithme (qui se base sur la formule évidente sans essayer d'optimiser les calculs) un algorithme naïf. Souvent, l'algorithme naïf pour résoudre un problème donné ne sera pas le plus efficace.

(b) Chaque matrice $n \times n$ a n^2 éléments, la taille de l'entrée est donc $2n^2$ ou $\Theta(n^2)$. Toutes les instructions du code s'exécutent en temps constant. Les trois boucles imbriquées résultent en un temps de parcours de $\Theta(n^3)$.

(c) Une borne inférieure triviale sur le temps de parcours de n'importe quel algorithme de multiplication de matrices $n \times n$ est $\Omega(n^2)$, puisqu'il faut au moins lire tous les éléments des deux matrices avant de pouvoir calculer leur produit, quelle que soit la manière dont on calcule ce produit.

L'algorithme cubique trouvé à la question (a) n'est pas le plus efficace.

L'algorithme de Strassen[2] (trouvé en 1969) se base sur une résolution récursive judicieuse du problème de multiplication de matrices en un temps de parcours $\mathcal{O}(n^{2.81})$. Le meilleur algorithme connu à ce jour (novembre 2024) a un temps de parcours $\mathcal{O}(n^{2.371552})$[3]. C'est un algorithme d'intérêt purement théorique (alors que l'algorithme de Strassen est utilisé en pratique) car la constante cachée dans la notation $\mathcal{O}(\cdot)$ est gigantesque: cet algorithme n'a donc d'avantage sur l'agorithme de Strassen que pour des valeurs de n tellement grandes qu'on ne peut pas les traiter avec des machines existantes. Le problème de l'existence d'un algorithme $\Theta(n^2)$ pour la multiplication de matrices $n \times n$ est un grand problème ouvert dans le domaine de la théorie de complexité algébrique.

[2] https://en.wikipedia.org/wiki/Strassen_algorithm
[3] https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication