Informatique – D.S. 1 *(2 heures)*

Le sujet comporte deux problèmes indépendants, pouvant être traités dans un ordre quelconque.
Les algorithmes doivent être écrits de la manière la plus courte possible, parfaitement lisibles, avec une indentation convenable, sans aucune rature et en respectant scrupuleusement les notations introduites. Ils doivent être documentés par des explications concises et précises sur les points qui le nécessitent.

Les fonctions demandées seront écrites en langage Python. Les tableaux utilisés sont supposés fournis sous forme de liste Python (donc indexés à partir de 0 !).

Problème A : tranches maximales

Le but du problème est d'étudier diverses méthodes de calcul du maximum des sommes d'éléments consécutifs d'un tableau donné (non trié) de flottants. Plus précisément, t étant donné, contenant n valeurs indexées de 0 à $n-1$, on cherche la plus grande des sommes des valeurs des "tranches" **non vides** $t\,[g:d]$, c'est-à-dire des sommes de la forme $\sum_{k=g}^{d-1} t[k]$, où les entiers g et d vérifient $0 \leq g < d \leq n$.

Par exemple, pour $n = 6$:
- si $t = [-4, 8, -1, -3, 5, -2]$, le résultat est 9, obtenu pour $g = 1$ et $d = 5$;
- si $t = [-4, -3, -1, -2, -7, -5]$, le résultat est -1, obtenu pour $g = 2$ et $d = 3$.

Cette valeur sera notée $S(t)$; elle est bien définie, en tant que plus grand élément d'un ensemble fini totalement ordonné. On ne se préoccupera pas des problèmes de débordement : on suppose que toutes les sommes à calculer restent dans le domaine de validité du type `float`.

Question préliminaire : écrire une fonction d'en-tête `max(a,b)` renvoyant le plus grand des deux flottants `a`, `b`. Cette fonction pourra être utilisée dans la suite.

Partie I – Algorithme naïf et première amélioration

La fonction ci-dessous (que l'on ne demande pas de justifier) détermine $S(t)$ en calculant successivement toutes les sommes décrites ci-dessus :

```
def som_max1(t):
    n=len(t)
    s_max=t[0]
    for g in range(n):
        for d in range(g+1,n+1):
            s=t[g]
            for k in range(g+1,d):
                s+=t[k]
            s_max=max(s_max,s)
    return s_max
```

On peut montrer (**et l'on admettra**) que le nombre $T_1\,(n)$ d'additions de flottants effectuées au cours de l'appel de `som_max1(t)` est de l'ordre de n^3 lorsque t contient n valeurs.

On peut supprimer l'une des boucles `for` imbriquées de la fonction `som_max1`, en mettant à jour la variable `s_max` au fur et à mesure du calcul des $\sum_{k=g}^{d-1} t[k]$, pour g fixé et d variant de $g+1$ à n.

Justifier cela en écrivant une fonction d'en-tête `som_max2(t)` renvoyant $S(t)$, au prix d'un nombre $T_2\,(n)$ d'additions de flottants, qui devra être de l'ordre de n^2 lorsque t contient n valeurs. On calculera $T_2\,(n)$.

Partie II – Version récursive

On remarque (**et l'on admet**) que, pour $n \geq 2$ et pour t contenant n valeurs, $S(t)$ est le plus grand des $n+1$ nombres suivants : les $\sum_{k=0}^{d-1} t[k]$, $d \in [\![1, n]\!]$ et $S(t')$ où t' désigne t privé de sa première valeur.

En déduire une fonction **récursive** d'en-tête `som_max3(t)` renvoyant $S(t)$.

Préciser le nombre $T_3(n)$ d'additions de flottants effectuées au cours de l'appel de `som_max3(t)` lorsque t contient n valeurs.

Partie III – Un algorithme linéaire

On note dans cette partie, pour tout p de $[\![1, n]\!]$:

$$\text{où} \quad \mathcal{E}_p = \left\{ \sum_{k=g}^{p-1} t[k],\ 0 \leq g < p \right\} \quad \text{et} \quad \mathcal{F}_p = \left\{ \sum_{k=g}^{d-1} t[k],\ 0 \leq g < d \leq p \right\}$$

et l'on pose

$$\delta_p = \max \mathcal{E}_p \quad \text{et} \quad \sigma_p = \max \mathcal{F}_p$$

1) Soit $p \in [\![1, n-1]\!]$; préciser la valeur de δ_{p+1} en fonction de δ_p et de $t[p]$, puis la valeur de σ_{p+1} en fonction de σ_p et de δ_{p+1}.

2) En déduire une fonction d'en-tête `som_max4(t)`renvoyant $S(t)$, au prix d'un nombre $T_4(n)$ d'additions de flottants, qui devra être un $O(n)$.

Problème B – *Produits de matrices*

Dans tout le problème, on appellera *matrice de type* (p, q) tout tableau rectangulaire de nombres réels, comportant p lignes et q colonnes, p et q étant deux entiers naturels non nuls. Une telle matrice A sera encore notée $(a_{i,j})_{\substack{0 \leqslant i < p \\ 0 \leqslant j < q}}$ où $a_{i,j}$ désigne, pour tout couple (i, j) de $[\![0, p-1]\!] \times [\![0, q-1]\!]$, l'élément situé à l'intersection de la ligne i et de la colonne j.

Pour les programmes en Python, on stockera les matrices dans des tableaux `numpy` de flottants.
On supposera `numpy` chargé avec l'alias `np` par : `import numpy as np`.
On rappelle qu'alors `np.zeros((p,q))` renvoie un tableau de type (p, q) rempli de zéros.
On rappelle enfin que, si `M` est un tel tableau `numpy`, `len(M)` donne le nombre de lignes de `M` et `len(M[0])` le nombre de colonnes (à savoir le nombre d'éléments de la première ligne !).

On ne se préoccupera pas de la place nécessitée en mémoire par les tableaux manipulés.

Partie I : produit de deux matrices

Soient $A = (a_{i,j})_{\substack{0 \leqslant i < p \\ 0 \leqslant j < q}}$ une matrice de type (p, q) et $B = (b_{i,j})_{\substack{0 \leqslant i < q \\ 0 \leqslant j < r}}$ une matrice de type (q, r), *le nombre de colonnes de A étant égal au nombre de lignes de B.*

La *matrice produit de A par B* est la matrice $C = AB$, de type (p, r), définie par

$$C = (c_{i,j})_{\substack{0 \leqslant i < p \\ 0 \leqslant j < r}} \quad \text{où} \quad \forall (i, j) \in [\![0, p-1]\!] \times [\![0, r-1]\!] \quad c_{i,j} = \sum_{k=0}^{q-1} a_{i,k} b_{k,j}.$$

1) En appliquant cette définition, écrire une fonction Python d'en-tête `Prod_mat(A,B)` recevant comme paramètres les tableaux `A`, `B` représentant A, B et renvoyant un tableau représentant le produit AB.

2) Calculer, en fonction de p, q, r, le nombre de multiplications de flottants effectuées lors de l'appel `Prod_mat(A,B)`.

Partie II : produits en chaîne

On considère dorénavant $n \in \mathbb{N}^*$, un $(n+1)$-uplet $(d_0, d_1, \dots, d_n)$ d'entiers naturels non nuls et une famille $(A_1, \dots, A_n)$ de matrices telles que, pour tout i de $[\![1, n]\!]$, A_i soit de type (d_{i-1}, d_i). On peut ainsi itérer la multiplication matricielle pour calculer le produit $(\dots((A_1A_2)\,A_3)\,A_4\dots)\,A_n$, de type (d_0, d_n). On peut montrer — *ce que l'on admettra ici* — que le choix du parenthésage précisant l'ordre dans lequel on effectue les multiplications (en respectant les positions respectives de $A_1, \dots, A_n$, de gauche à droite !) n'a pas d'influence sur le résultat *(pseudo-associativité de la multiplication matricielle).*

Puisqu'il n'y a pas d'ambiguïté, on peut noter le produit $A_1 \dots A_n$ sans parenthèses.
Mais, pour effectuer le calcul, il faut bien décider de l'ordre des opérations !

On définit récursivement un *produit complètement parenthésé* de matrices comme étant, soit une matrice unique, soit le produit — placé entre parenthèses — de deux produits complètement parenthésés.

Par exemple, pour $n = 4$, on a 5 produits complètement parenthésés permettant le calcul de $A_1A_2A_3A_4$:

$$(((A_1A_2)\,A_3)\,A_4) = ((A_1\,(A_2A_3))\,A_4) = ((A_1A_2)\,(A_3A_4)) = (A_1\,((A_2A_3)\,A_4)) = (A_1\,(A_2\,(A_3A_4)))\,.$$

Dans la suite, on appellera *coût* d'un calcul de produit matriciel le nombre de multiplications de flottants nécessaires aux calculs successifs. On admettra que le coût du produit d'une matrice de type (p, q) par une matrice de type (q, r) est pqr.

1) *Intérêt d'un bon choix*

On considère, **dans cette question seulement**, le cas particulier suivant ;

$$n = 4 \quad ; \quad (d_0, d_1, d_2, d_3, d_4) = (10, 5, 100, 3, 50)\,.$$

Déterminer le coût du calcul de $A_1A_2A_3A_4$ pour chacun des 5 parenthésages ci-dessus. Commenter.

2) *Dénombrement des parenthésages possibles*

Trouver une relation de récurrence vérifiée par les $P(n)$, $n \in \mathbb{N}^*$, où $P(n)$ désigne le nombre de parenthésages complets possibles pour le calcul de $A_1 \dots A_n$ *(bien entendu, $P(1) = P(2) = 1$!).* Calculer $P(5)$ et $P(6)$.

On peut montrer, ce qui n'est pas demandé ici, que

$$\forall n \in \mathbb{N}^* \quad P(n) \underset{n\to\infty}{\sim} \frac{1}{\sqrt{\pi}} \cdot \frac{4^{n-1}}{n^{3/2}}.$$

Il n'est donc pas question d'envisager un à un tous les parenthésages possibles…

3) *Structure d'un choix optimal*

Soient i, j dans $[\![1, n]\!]$, tels que $i \leqslant j$; on note $\Pi(i..j)$ le produit $A_i \dots A_j$ *(avec bien sûr $\Pi(i..i) = A_i$).*

a) Justifier l'existence d'un parenthésage *optimal* (de coût minimum) pour le calcul de $\Pi(i..j)$.

b) Montrer que, si un parenthésage optimal pour le calcul de $\Pi(i..j)$ conduit en fin de calcul à effectuer le produit de $\Pi(i..k)$ par $\Pi(k+1..j)$ (où $i \leqslant k < j$), alors les parenthésages pour les calculs de $\Pi(i..k)$ et de $\Pi(k+1..j)$ sont également optimaux.

c) On note $m(i, j)$ le coût minimum pour le calcul de $\Pi(i..j)$. Établir :

$$m(i,j) = \begin{cases} 0 & \text{si } i = j \\ \min\limits_{i\leqslant k<j} \left(m(i,k) + m(k+1,j) + d_{i-1}d_kd_j\right) & \text{si } i < j \end{cases}.$$

4) *Programmation récursive*

a) Écrire une fonction Python d'en-tête `Cout_min(d)` recevant comme paramètre une liste Python `d` contenant $(d_0, \dots, d_n)$ et renvoyant la valeur de $m(1, n)$ *(on pourra utiliser une fonction auxiliaire* ***récursive*** *d'en-tête* `m(i,j)` *basée sur le résultat de la question précédente).*

b) Montrer que le nombre $M(n)$ de multiplications d'entiers effectuées lors de l'appel `Cout_min(d)` est supérieur ou égal à 2^{n-1}, pour tout $n \geq 2$. Commenter.

5) *Programmation dynamique*

L'inefficacité de la fonction précédente tient au fait qu'elle calcule plusieurs fois les mêmes valeurs. D'où l'idée de mémoriser dans un tableau auxiliaire les valeurs de $m(i,j)$, pour $1 \leqslant i \leqslant j \leqslant n$ au fur et à mesure de leur obtention.

a) Écrire une fonction itérative, d'en-tête `Tableau_m(d)`, renvoyant le tableau `m` ainsi associé à la liste `d` contenant $(d_0, \ldots, d_n)$ *(on initialisera* `m` *à zéro, puis on reprendra le résultat du* **3)c)** *et l'on pourra raisonner sur la valeur de* $j - i$ *pour remplir la partie utile de* `m` *; on n'utilisera évidemment pas la fonction du* **4)** ***!****).*

b) Calculer le nombre de multiplications d'entiers nécessaires à l'obtention de $m(1,n)$ par cette méthode. Commenter.

c) Modifier la fonction du **a)** pour renvoyer — en même temps que `m` — un second tableau `kop`, de sorte que `kop[i,j]` contienne une valeur optimale k telle que celle considérée au **3)b)**.

d) Écrire enfin une fonction d'en-tête `Parenthesage(kop)` recevant comme paramètre le tableau `kop` (préalablement rempli comme au **c)** !) et renvoyant un parenthésage complet optimal sous la forme d'une chaîne de caractères, telle que :

```
((A[1](A[2]A[3]))A[4])
```

(on rappelle que l'opérateur + réalise en Python la concaténation des chaînes de caractères et que le transtypage `str(x)` *convertit la valeur numérique* `x` *en chaîne de caractères).*

Principe de Hofstadter

Il faut toujours plus de temps que prévu, même en tenant compte du principe de Hofstadter.