Problème A : tranches maximales

Question préliminaire : sans commentaire !

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

Partie I – Algorithme naïf et première amélioration

L'idée est d'essayer d'éviter les calculs redondants dans les deux boucles internes de `som_max1` : je vais mettre à jour la variable `s_max` **à l'intérieur** de la boucle qui calcule $\sum_{k=g}^{d-1} t[k]$:

```
def som_max2(t):
    n = len(t)
    s_max = t[0]
    for g in range(n):
        s = t[g]
        s_max = max(s_max,s)
        for k in range(g+1, n):
            s += t[k]
            s_max = max(s_max, s)
    return s_max
```

La terminaison est banale puisque les boucles sont inconditionnelles.
`s_max` est initialisé avec la valeur `t[0]`, à la suite de quoi je calcule toutes les sommes à considérer en mettant à jour `s_max` à chaque nouveau calcul : à la sortie `smax` contient bien $S(t)$.
Attention ! Ne pas initialiser `s_max` par 0, pour le cas où tous les éléments du tableau seraient négatifs...
Pour g fixé dans $[\![0, n-1]\!]$, la boucle interne du programme ci-dessus effectue $n-g-1$ additions, d'où :

$$T_2(n) = \sum_{g=0}^{n-1} (n-g-1) = \sum_{k=0}^{n-1} k$$

soit

$$\boxed{T_2(n) = \frac{n(n-1)}{2} = \Theta\left(n^2\right).}$$

Partie II – Version récursive

Si le tableau contient une seule valeur, il y a une seule somme que je renvoie !

Sinon j'applique le principe de l'énoncé, en déterminant la plus grande des sommes des tranches contenant $t[0]$, puis en la comparant à $S(t')$.

À droite une variante avec un second paramètre pour éviter les recopies de tableau :

```
def som_max3(t):
    if len(t) == 1:
        return t[0]
    else:
        s_max = t[0]
        s = t[0]
        for k in range(1, len(t)):
            s += t[k]
            s_max = max(s_max, s)
        return max(s_max, som_max3(t[1:]))
```

```
def som_max3(t, d=0):
    if d == len(t) - 1:
        return t[-1]
    else:
        s_max = t[d]
        s = t[d]
        for k in range(d + 1, len(t)):
            s += t[k]
            s_max = max(s_max, s)
        return max(s_max, som_max3(t, d+1))
```

- La terminaison est justifiée par la décroissance stricte de la longueur de t à chaque appel récursif et l'arrêt lorsqu'elle vaut 1.
- La correction est justifiée par la remarque de l'énoncé, accompagnée d'une récurrence : la boucle `for` place dans `s_max` la plus grande des sommes $\sum_{k=g}^{n} t[k]$, $g \in [\![1,n]\!]$. Il n'y a plus qu'à renvoyer `max(s_max,som_max3(t[1:]))`, en effet $t\,[1:]$ correspond bien à t'.
- Le nombre d'additions de réels $T_3(n)$ vérifie ici $T_3(1) = 0 \quad$ et $\quad \forall n \geq 2 \quad T_3(n) = n-1+T_3(n-1)$, d'où — par une récurrence banale — $T_3(n) = \sum_{k=1}^{n} (k-1)$, soit

$$\boxed{T_3(n) = \frac{n(n-1)}{2}.}$$

Partie III – Un algorithme linéaire

1) Je remarque que, pour tout p de $[\![1, n-1]\!]$, avec les notations de l'énoncé,

$$\mathcal{E}_{p+1} = \{t[p]\} \cup \{x + t[p],\ x \in \mathcal{E}_p\}.$$

Il est clair que

$$\max\{x + t[p],\ x \in \mathcal{E}_p\} = \delta_p + t[p]$$

d'où : si $\delta_p \leq 0$, alors $\delta_{p+1} = t[p]$, sinon $\delta_{p+1} = \delta_p + t[p]$. Autrement dit

$$\boxed{\delta_{p+1} = \max(t[p], \delta_p + t[p]).}$$

Par ailleurs, par construction, $\mathcal{F}_{p+1} = \mathcal{F}_p \cup \mathcal{E}_{p+1}$. Il en résulte que

$$\boxed{\sigma_{p+1} = \max(\sigma_p, \delta_{p+1}).}$$

2) Avec l'initialisation triviale $\sigma_1 = \delta_1 = t[0]$, la question précédente fournit la preuve par récurrence de la correction de la fonction suivante, puisque par définition $S(t) = \sigma_n$:

```
def som_max4(t):
    sigma = t[0]
    delta = t[0]
    for p in range(1, len(t)):
        if delta > 0:
            delta += t[p]
        else:
            delta = t[p]
        sigma = max(sigma,delta)
    return sigma
```

J'économise une addition à chaque fois que δ_p est négatif ou nul...

Puisqu'il n'y a plus qu'une boucle, avec au plus une addition à chaque passage :

$$\boxed{T_4(n) \leq n-1 \quad \text{d'où} \quad T_4(n) = O(n).}$$

(ça valait le coup de réfléchir !).

"Un bon algorithme est comme un couteau tranchant — il fait exactement ce qu'on attend de lui, avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez sans doute par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire, et le résultat aura peu de chances d'être esthétiquement satisfaisant."

(Cormen, Leiserson & Rivest)

Problème B – *Produits de matrices*

Partie I : produit de deux matrices

1) Il n'y a qu'à appliquer la définition :

```
def Prod_mat(A,B):
    p, q, r = len(A), len(B), len(B[0])
    C = np.zeros((p, r))
    for i in range(p):
        for j in range(r):
            for k in range(q):
                C[i,j] += A[i,k] * B[k,j]
    return C
```

Terminaison et correction banales...

2) La boucle interne est exécutée pr fois et elle coûte q multiplications :

`Prod_mat(A,B)` coûte pqr multiplications de flottants.

Partie II : produits en chaîne

1) Pour (A_1A_2), le coût est $10\times5\times100$ et le résultat est une matrice de type $(10,100)$; pour la multiplier par A_3, le coût est $10\times100\times3$ et le produit $((A_1A_2)\,A_3)$ est une matrice de type $(10,3)$; il reste à la multiplier par A_4, pour un coût de $10\times3\times50$. Ainsi, le coût total pour le calcul de $(((A_1A_2)\,A_3)\,A_4)$ est : $10\times5\times100+10\times100\times3+10\times3\times50=9\,500$.

De même, pour $((A_1\,(A_2A_3))\,A_4)$, le coût est : $5\times100\times3+10\times5\times3+10\times3\times50=3\,150$.

Pour $((A_1A_2)\,(A_3A_4))$: $10\times5\times100+100\times3\times50+10\times100\times50=70\,000$.

Pour $(A_1\,((A_2A_3)\,A_4))$: $5\times100\times3+5\times3\times50+10\times5\times50=4\,750$.

Enfin, pour $(A_1\,(A_2\,(A_3A_4)))$: $100\times3\times50+5\times100\times50+10\times5\times50=42\,500$.

Le coût du pire des choix est plus de 22 fois supérieur au coût optimal...

2) Soit $n\geq2$; par définition, un parenthésage complet d'un produit de n matrices est le produit, placé entre parenthèses, d'un parenthésage complet d'un produit de k matrices par un parenthésage complet d'un produit de $n-k$ matrices, cela pour une valeur de k telle que $1\leq k\leq n-1$; les choix des deux sous-parenthésages étant indépendants l'un de l'autre, j'en déduis la relation de récurrence :

$$\boxed{\forall n\geq2\quad P(n)=\sum_{k=1}^{n-1}P(k)\,P(n-k)\quad\text{et}\quad P(1)=1.}$$

Je retrouve ainsi $P(2)=1$, $P(3)=1+1=2$, $P(4)=2+1+2=5$ et j'obtiens :

$$\boxed{P(5)=14\quad\text{et}\quad P(6)=42.}$$

Le lecteur curieux pourra vérifier que $P(n)$ n'est autre que le n-ième *nombre de Catalan* C_{n-1} et que :

$$\forall n\in\mathbb{N}\quad P(n+1)=C_n=\frac{1}{n+1}\binom{2n}{n}.$$

3) a) L'ensemble des coûts associés aux différents parenthésages permettant le calcul de $\Pi(i..j)$ forment une partie non vide de $\mathbb{N}$, qui admet donc un plus petit élément !

Il existe un parenthésage optimal.

b) Supposons un parenthésage optimal $\mathcal{P}(i..j)$ pour le calcul de $\Pi(i..j)$ et k dans $[\![i,j-1]\!]$ tel que le dernier calcul soit le produit de $\Pi(i..k)$ par $\Pi(k+1..j)$; si, par exemple, le sous-parenthésage $\mathcal{P}(i..k)$ du calcul de $\Pi(i..k)$ n'était pas optimal, je disposerais d'un autre parenthésage $\mathcal{P}^*(i..k)$ de coût strictement inférieur à celui de $\mathcal{P}(i..k)$, pour le calcul de $\Pi(i..k)$.

Alors, en remplaçant $\mathcal{P}(i..k)$ par $\mathcal{P}^*(i..k)$, j'obtiendrais un parenthésage pour le calcul de $\Pi(i..j)$, de coût strictement inférieur à celui de $\mathcal{P}(i..j)$, ce qui est contradictoire avec son caractère optimal. De même pour le parenthésage du calcul de $\Pi(k+1..j)$.

Les parenthésages pour les calculs de $\Pi(i..k)$ et de $\Pi(k+1..j)$ sont optimaux.

c) $m(i,i)$ est bien sûr nul puisqu'il n'y a aucune multiplication à effectuer pour fournir A_i ! Supposons maintenant $i<j$. $m(i,j)$ est égal à l'une des valeurs :

$$m(i,k)+m(k+1,j)+d_{i-1}d_kd_j\ ,\ i\leq k<j\ ;$$

en effet, le calcul de $\Pi(i..j)$ s'achèvera par le calcul d'un produit de la forme

$$(\Pi(i..k)\times\Pi(k+1..j))\ ,\ i\leq k<j\ ;$$

le coût du calcul de $\Pi(i..k)$ (resp. $\Pi(k+1..j)$) étant $m(i,j)$ (resp. $m(k+1,j)$) d'après la question précédente ; enfin, $\Pi(i..k)$ (resp. $\Pi(k+1..j)$) étant une matrice de type (d_{i-1},d_k) (resp. (d_k,d_j)) — récurrence immédiate — le coût du dernier produit est bien $d_{i-1}d_kd_j$. Ainsi, $m(i,j)$ est nécessairement la plus petite de ces valeurs, pour k décrivant $[\![i,j-1]\!]$.

En conclusion :

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

4) a) La fonction `Cout_min` est triviale (dernière ligne) ! Tout est dans la fonction auxiliaire récursive `m` :

```
def Cout_min(d):
    def m(i, j):
        if i == j:
            return 0
        else:
            dd = d[i-1]*d[j]
            min = m(i+1,j) + dd*d[i]
            for k in range(i+1, j):
                essai = m(i, k) + m(k+1, j) + dd*d[k]
                if essai < min:
                    min = essai
            return min

    return m(1, len(d)-1)
```

∗ La fonction `m` se termine bien, car la valeur entière $j-i$ décroît strictement à chaque appel récursif, et le test d'arrêt renvoie un résultat lorsque $i=j$.

∗ La correction de la fonction `m` se prouve par récurrence forte sur $h=j-i$: pour $j-i=0$, $m(i,j)$ renvoie bien le résultat attendu, 0 ; si je suppose $h\geq 1$ tel que `m`(u,v) renvoie le résultat attendu pour tout couple (u,v) d'entiers tels que : $1\leq u\leq v\leq n$ et $v-u<h$ et si je considère (i,j) tel que $j-i=h$, alors l'appel $m(i,j)$ renvoie bien le résultat attendu d'après **3)c)**, cela en utilisant l'hypothèse de récurrence pour chaque valeur de k testée (la boucle déterminant le minimum est standard, ne pas oublier toutefois l'initialisation de la variable `min` !).

∗ L'utilisation de la variable `dd` permet de ne pas recalculer inutilement `d[i-1]*d[j]`.

b) L'examen de la fonction `m` ci-dessus montre que :

$$M(1)=0\quad\text{et}\quad\forall n\geq 2\quad M(n)=1+M(n-1)+1+\sum_{p=1}^{n-2}\left[M(p)+M(n-p)+1\right]$$

les valeurs de n (resp. p) correspondant à celles de $j-i+1$ (resp. $k-i$) ; soit, après réindexation, $M(n)=n+2\sum\limits_{p=2}^{n-1}M(p)$ d'où, en minorant grossièrement :

$$M(2)=2\quad et\quad\forall n\geq 3\quad M(n)\geq 2M(n-1)\,.$$

Il en résulte, par une récurrence évidente :

$\boxed{\forall n\geq 2\quad M(n)\geq 2^{n-1}.\text{ Cette fonction est inutilisable pour de grandes valeurs de } n.}$

5) a) Nous reprenons ici le résultat du **3)c)**, en remarquant que nous pouvons remplir le tableau `m` (ou plus précisément sa partie utile, à savoir le triangle supérieur formé des $\mathtt{m}[i,j]$, pour $i \leq j$) en remplissant l'une après l'autre les "diagonales" définies par $j-i=h$, pour $h=0,\ldots,n-1$: comme $k-i<j-i$ et $j-(k+1)<j-i$, cela pour tout k tel que $i \leq k < j$, la détermination de chaque nouvelle valeur se fera en n'utilisant que des valeurs déjà calculées (après évidemment l'initialisation de la première diagonale, compte tenu de $\mathtt{m}[i,i]=0$). D'où la fonction :

```
def Tableau_m(d):
    n = len(d)-1
    m = np.zeros((n+1, n+1), dtype = int)
    # kop = np.zeros((n+1, n+1),dtype = int)
    for h in range(1, n):
        for i in range(1, n-h+1):
            j = i + h
            dd = d[i-1]*d[j]
            m[i, j] = m[i+1, j] + dd*d[i]
            # kop[i, j] = i
            for k in range(i+1, j):
                essai = m[i, k] + m[k+1, j] + dd*d[k]
                if essai < m[i, j]:
                    m[i, j] = essai
                    # kop[i, j] = k
    return m #, kop
```

b) $m(1,n)$ n'est obtenu qu'à la fin du remplissage (pour $h=n-1, i=1, j=n$). Pour en arriver là, il aura fallu effectuer $N(n)$ multiplications d'entiers, avec, en suivant pas à pas la procédure

$$\begin{aligned} N(n) &= \sum_{h=1}^{n-1}\left(\sum_{i=1}^{n-h}\left[1+1+(i+h-1)-(i+1)+1\right]\right) = \sum_{h=1}^{n-1}(n-h)(1+h) \\ &= \sum_{h=1}^{n-1}\left[n+(n-1)h-h^2\right] = n(n-1)+(n-1)\frac{(n-1)n}{2}-\frac{(n-1)n(2n-1)}{6} \end{aligned}$$

cela compte tenu des formules classiques :

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

Soit, tous calculs faits :

$$N(n) = \frac{n(n-1)(n+4)}{6} \underset{n\to\infty}{\sim} \frac{n^3}{6} \text{ ce qui devient presque raisonnable.}$$

c) La modification demandée s'obtient simplement en supprimant les dièses de la fonction du **a)** ci-dessus : le tableau `kop` est alors rempli en même temps que `m`. `kop[i,j]` est mis à jour à chaque fois que c'est nécessaire, avec la valeur optimale (ou l'une des valeurs optimales...) de k pour le calcul de $\Pi(i..j)$ sous la forme $(\Pi(i..k)\times\Pi(k+1..j))$.

d) Cette fonction s'écrit naturellement, à l'aide d'une fonction auxiliaire récursive, une fois de plus.

```
def Parenthesage(kop):
    def Aux(i, j):
        if i == j:
            return 'A[' + str(i) + ']'
        else:
            return '(' + Aux(i, kop[i, j]) + Aux(kop[i, j] + 1, j) + ')'

    return Aux(1, len(kop)-1)
```

La fonction `Parenthesage` elle-même est triviale (dernière ligne !). Quant à la fonction `Aux`, elle reprend "à la lettre" la définition récursive d'un parenthésage complet, avec le "bon choix" de k fourni par le tableau `kop`, par construction !