TSI2. Concours Blanc 2016

Épreuve d'informatique

Durée 3 heures. Les calculatrices sont autorisées

La rigueur du raisonnement et la clarté seront prises en compte dans la notation. Les deux problèmes ci-après sont indèpendants et au niveau du barème, le premier problème a un poids de 3/ 5 et le second a un poids de 2/5 environ.
Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Instructions propre à l'informatique : les algorithmes doivent être écrits de la manière la plus optimale possible, avec une indentation convenable, sans aucune rature et en respectant scrupuleusement les notations introduites. Vous pouvez utiliser des fonctions prédéfinies à condition d'écrire les lignes d'importation des modules contenant ces fonctions sauf si l'énoncé vous impose une façon autre. Vos algorithmes doivent en outre être documentés par des explications concises et précises sur les points qui le nécessitent.
Les fonctions demandées seront écrites en langage Python.

T.S.V.P →

Problème 01 : tranches maximales

Les tableaux qui peuventêtre utilisés dans ce problème sont fournis sous forme de liste Python (donc indéxés à partir de 0). 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 nombres réels. 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 $0 \leq g < d \leq n$.

Cette valeur sera notée $S(t)$; elle est bien définie en tant que plus grand élément d'un ensemble ordonné fini.

Partie A. Les préliminaires c'est important !

1. Calculer $S(t)$ pour : a) $t = [-7, -8]$. b) $t = [-3, 5, -2]$.
   On donnera aussi les valeurs de g et d correspondantes.
2. Écrire une fonction $max(a, b)$ renvoyant le plus grand des deux réels a et b. Cette fonction pourra être utilisée dans la suite. (Attention, on sait que Python a déjà une fonction prédéfinie qui calcule le maximum mais il n'est pas question de l'utiliser ici.)

Partie B. Un algorithme naif et première amélioration

La fonction ci-dessous 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)
```

1. Appliquer som_max1 à la liste $t = [-4, 0, -1, 2]$, en écrivant à la main toutes les étapes décrites par l'algorithme. Combien y a t-il d'étapes ?
2. On peut montrer (et **on l'admettra**) que le nombre $T_1(n)$ d'additions de réels 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 $som\_max2(t)$ renvoyant $S(t)$, au prix d'un nombre $T_2(n)$ d'additions de réels qui devra être de l'ordre de n^2 lorsque t contient n valeurs.
   (On supprimera la boucle indexée par d.)
3. Appliquer som_max2 à la liste $t = [-4, 0, -1, 2]$, en écrivant à la main toutes les étapes décrites par l'algorithme et comparer avec som_max1. Conclure.

4. Justifier que $T_2(n) = \sum_{g=0}^{n-1}(n-g-1)$. En déduire la valeur de $T_2(n)$ en fonction de n seul.

Partie C. Une 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 $\sum_{k=0}^{d-1} t[k]$ avec $d \in in[\![1, n]\!]$ et la valeur de $S(t')$, où t' désigne t privé de sa première valeur.

1. Calculer $S(t')$ pour $t = [-4, 0, -1, 2]$ et calculer les quatre valeurs $\sum_{k=0}^{d-1} t[k]$ avec $d \in in[\![1, 4]\!]$. Retrouver le résultat trouvé à la partie B.
2. Écrire une fonction Python **récursive** nommée $som\_max3(t)$ renvoyant $S(t)$.
   (On rappelle que $t[1:]$ est l'écriture Python de t'.)
3. On note $T_3(n)$ d'additions de réels effectuées au cours de l'appel de $som\_max3(t)$ lorsque t contient n valeurs.
   (a) Justifier que $T_3(1) = 0$ et que pour tout $n \geq 2$, $T_3(n) = n - 1 + T_3(n-1)$.
   (b) En déduire $T_3(n)$ en fonction de n. Que retrouve-t-on ?

Partie D. Un algorithme linéaire

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

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

et l'on pose $\delta_p = \max_{g \in [\![1,p]\!]} \sum_{k=g}^{p-1} t[k]$ et $\sigma_p = \max_{(g,d) \in [\![1,p]\!]^2,\, g<d} \sum_{k=g}^{d-1} t[k]$.

1. Déterminer ε_i, $\mathcal{F}_i$, δ_i et σ_i pour tout $i \in [\![1, 4]\!]$ dans le cas de $t = [-4, 0, -1, 2]$.
2. Soit $p \in [\![1, n-1]\!]$.
   (a) Justifier $\varepsilon_{p+1} = \{t[p]\} \cup \{x + t[p],\ x \in \varepsilon_p\}$.
   (b) Montrer que $\delta_{p+1} = \max(t[p],\ \delta_p + t[p])$.
   (c) Justifier $\mathcal{F}_{p+1} = \mathcal{F}_p \cup \varepsilon_{p+1}$.
   (d) Montrer que $\sigma_{p+1} = \max(\sigma_p,\ \delta_{p+1})$.
3. Comparer $S(t)$ et σ_n.
4. On fera l'initialisation triviale $\sigma_1 = \delta_1 = t[0]$. Écrire une fonction $som\_max4(t)$ renvoyant $S(t)$ en terminaison d'une boucle simple calculant les quantités δ_p et σ_p à chaque pas p de la boucle.
5. Calculer $T_4(n)$ le nombre d'additions effectuées. Que remarque t-on ?

Problème 02 : chemins de poids minimum

Soit M un tableau de n lignes et p colonnes de nombres réels, les lignes étant numérotées de 0 à $n-1$ de haut en bas et les colonnes de 0 à $p-1$, de gauche à droite. On note $M[i, j]$ la valeur située dans la case de coordonnées (i, j).

*Un **chemin** est une liste de couples d'indices, coordonnées de cases du tableau, telles que deux couples consécutifs de la liste vérifient la règle de déplacement suivante : « si l'on se trouve dans la case de coordonnées (i, j), on peut aller soit dans la case de droite $(i, j+1)$, soit dans la case du dessous $(i+1, j)$, à condition de ne pas sortir du tableau.*

T.S.V.P →

*Le **poids** d'un chemin est égal à la somme des nombres contenus dans les cases parcourues.*

*On se donne comme point de départ la case D de coordonnées $(0,0)$, située en haut à gauche, et comme point d'arrivée la case A de coordonnées $(n-1,p-1)$, située en bas à droite. On cherche à déterminer un **chemin optimal**, c'est-à-dire de poids miminum parmi tous les chemins qui mènent de D à A.*

Partie A. Nombre de chemins possibles

On note $\gamma_{n,p}$ le nombre de chemins possibles.

1. On suppose $n = p = 3$. Trouver $\gamma_{3,3}$.
2. Justifier que choisir un chemin de D à A revient à choisir $n+p-2$ déplacements élémentaires successifs (en ne distinguant pas un déplacement vers le bas et un déplacement vers la gauche).
3. En déduire alors $\gamma_{n,p}$ sous la forme $\binom{n+p-2}{l}$, où l est un entier à déterminer.
4. Démontrer qu'il existe au moins un chemin optimal.

Partie B. Détermination du poids d'un chemin optimal

Énumérer tous les chemins possibles s'avère très coûteux pour de grandes valeurs de n, p (on peut montrer par exemple (ce n'est pas demandé ici) que $\gamma_{n,n}$ est un $O(4^n/\sqrt{n})$. Donc on ne va pas chercher le chemin optimal en les trouvant tous. On va s'appuyer sur le principe suivant : toute politique optimale est composée de « sous-politiques » optimales, comme a dit un célèbre enarque (je sais plus qui !).

1. Un exemple. Soit le tableau suivant : $\begin{array}{|c|c|c|}\hline 1 & 4 & 6 \\ \hline 2 & 4 & 0 \\ \hline 3 & 5 & 0 \\ \hline 8 & 0 & 7 \\ \hline \end{array}$.

   Déterminer un chemin optimal et son coût.

   On reprend le cas général.
2. Pour tout couple $(i,j) \in [\![0, n-1]\!] \times [\![0, p-1]\!]$, on note $cout(i,j)$ le minimum des poids des chemins menant de la case de coordonnées (i,j) à la case A de coordonnées $(n-1,p-1)$.
   (a) Calculer $cout(n-1,p-1)$.
   (b) On fixe $j \in [\![0, p-1]\!]$. Écrire $cout(n-1,j)$ sous forme d'une somme.
   (c) On fixe $i \in [\![0, n-1]\!]$. Écrire de même $cout(i,p-1)$ sous forme d'une somme.
   (d) On suppose que $i \in [\![0, n-2]\!]$ et $j \in [\![0, p-2]\!]$. Établir la relation :
   $$cout(i,j) = M[i,j] + \min(cout(i,j+1), cout(i+1,j)).$$
   (e) En déduire en Python une fonction récursive $cout(i,j)$ qui renvoie le côut d'un chemin optimal partant de (i,j). On se basera sur les réponses ci-dessus.
3. Écrire une fonction Python nommée $poids\_min(M)$ recevant comme argument M, tableau correspondant aux données et renvoyant le poids d'un chemin optimal. On utilisera $cout(i,j)$ comme sous-procédure.
   On rappelle qu'après avoir importé *numpy* avec l'alias *np*, la commande $n, p = np.shape(M)$ donne la dimension du tableau M, c'est-à -dire son nombre de lignes n et son nombre de colonnes p.