Programmation dynamique

Dans le cadre du TP, toutes les formules de récurrence (question 4 de l'exercice 1, 3 de l'exercice 2 ou 7 de l'épreuve de polytechnique) pourront être admises dans un premier temps.

Exercice 1 : Sous-sommes maximales

Dans cet exercice, on considère une liste L d'entiers relatifs de longueur n. L'objectif est de déterminer la somme de valeur maximale d'éléments consécutifs de cette liste :

$$m = \max_{0\leq p\leq q\leq n-1} (L[p] + \cdots L[q])$$

A titre d'exemple, si $L = [2; -3; 4; 6; -3; 5; -8; 1; -2; 7]$, la somme de valeur maximale est la somme $4 + 6 - 3 + 5$ qui vaut donc 12.

1. Ecrire une fonction `sous_somme` qui prend en argument la liste L et deux indices p et q et calcule la somme des éléments de L compris entre les indices p et q.
2. En déduire une fonction naïve qui répond au problème en testant toutes les sous-sommes possibles. Quelle est la complexité de cette fonction ?
3. La fonction `sous_somme` est réalité une fonction nuisible car coûteuse et inutile. En utilisant uniquement deux boucles et une variable mise à jour au fur et à mesure des boucles pour calculer les sommes, écrire une fonction qui répond au problème en $O(n^2)$ opérations élémentaires.

On peut encore faire mieux que la complexité de la fonction précédente. Pour tout $i \in [\![0; n-1]\!]$, on pose

$$m_i = \max_{0\leq p\leq q\leq i} (L[p] + \cdots L[q]) \qquad \text{et} \qquad \ell_i = \max_{0\leq p\leq i} (L[p] + \cdots L[i])$$

4. Justifier que pour tout $i \in [\![0; n-2]\!]$,

$$m_{i+1} = \max(m_i, \ell_{i+1}) \qquad \text{et} \qquad \ell_{i+1} = L[i+1] + \max(\ell_i, 0)$$

5. En déduire une fonction qui calcule m en $O(n)$ opérations élémentaires.
6. Adapter la fonction pour qu'elle renvoie deux indices $p \leq q \in [\![0; n-1]\!]$ tels que $m = L[p] + \cdots + L[q]$.

Exercice 2 : Problème de la scierie

On considère une scierie qui connaît pour tout entier i le prix de vente p_i pour une planche de longueur i (avec naturellement $p_0 = 0$). Lorsqu'elle reçoit en entrée une planche de longueur n, elle peut soit en tirer directement le profit p_n, soit chercher à la découper en k morceaux pour en tirer plusieurs planches de longueur $i_1, i_2, \ldots, i_k$ avec $i_1+i_2+\cdots+i_k = n$, et obtenir comme profit la somme $p_{i_1}+\cdots+p_{i_k}$ des ventes des sous-planches. Le problème de la scierie consiste alors de déterminer le découpage lui garantissant un profit maximal. Dans tout les codes, les valeurs de $p_0, \ldots, p_n$ sont fournies en argument sous la forme d'une liste `P` de longueur $n+1$ telle que `P[k]` contient la valeur p_k.

1. On propose une première méthode de découpage gloutonne : à chaque étape, on découpe un morceau de rentabilité $r_k = p_k/k$ maximale et on recommence tant que la planche est de longueur non nulle. Ecrire une fonction qui prend en argument la liste `P` et la valeur de `n` et renvoie la découpe obtenue de cette manière.

   Note : Si possible, écrire une fonction de complexité linéaire !
2. **Facultatif :** A l'aide d'un exemple bien choisi, démontrer que cette stratégie ne fournit pas toujours une découpe de profit maximal.

Pour tout entier k, notons m_k le prix maximal que l'on peut toucher en revendant une planche de longueur k (découpée judicieusement). On note également E_k l'ensemble des listes d'entiers non nuls dont la somme vaut k et qui représente les différentes longueurs à découper pour réaliser ce gain maximal. Par convention, m_0 est nul et E_0 est un ensemble contenant la seule liste vide.

3. Justifier que pour tout $k \in \mathbb{N}$,

$$m_{k+1} = \max_{r\in[\![1;k+1]\!]} (p_r + m_{k+1-r})$$

   Expliquer comment construire E_{k+1} à partir de la formule précédente et des ensembles $E_0, \ldots, E_k$.
4. Ecrire une fonction en Python prenant en argument la liste `P` et calculant les valeurs $m_0, \ldots, m_n$.
5. Adapter la fonction de la question précédente pour qu'elle renvoie un découpage garantissant le profit maximal. Le résultat fourni sera la liste des longueurs du découpage, sous la forme d'une liste d'entiers non nuls dont la somme vaut n. Evaluer sa complexité.

Ecole Polytechnique (Tronc commun) 2006

Deux candidats paresseux décident de mettre en commun leurs efforts pour passer les concours. Afin de s'épargner de trop nombreux déplacements, ils décident qu'à chaque épreuve, un seul d'entre eux se déplacera et rendra discrètement deux fois la même copie pour chacun d'eux. Comme ils sont de plus particulièrement fainéants, ils cherchent également à minimiser le total de leurs déplacements à tous les deux (notamment ils ne rentrent jamais chez eux et vont d'épreuve en épreuve).

Dans tout l'énoncé, la France est assimilée à l'ensemble $\mathbb{N}$. Une liste `a` recense, dans l'ordre de passage, les adresses des lieux (un entier positif) où doit se rendre l'un des étudiants pour composer : la case `a[k]` contient l'adresse du k-ième centre d'examen (avec la convention `a[0] = 0`). Initialement, les deux élèves sont à la position 0.

Une séquence de déplacements est une liste `d` de longueur n dont les éléments sont dans $\{1, 2\}$. La case d'indice i de `d` indique alors quel étudiant est désigné pour aller les représenter pour la $(i+1)$-ième épreuve. Le coût de la séquence est alors la somme totale des distances parcourues par chaque candidat. Par exemple, pour `a = [0,5,2,4]`, le coût de la séquence `d1 = [1,1,2]` vaut $5 + 3 + 4 = 12$ tandis que celui de la séquence `d2 = [1,2,1]` vaut $5 + 2 + 1 = 8$.

Préliminaires et solutions naïves

1. Ecrire une fonction prenant en argument les listes `a` et `d` et renvoyant le coût de la séquence de déplacements `d` pour les adresses `a`.
2. Quel serait, en fonction de n, la complexité d'une méthode naïve consistant à calculer le coût de toutes les séquences et de renvoyer la séquence réalisant le minimum de ces valeurs ?
3. Ecrire une fonction `cout_opt2` qui prend en argument la liste `a` des adresses et donnant le coût optimal lorsqu'il n'y a que deux épreuves à passer.
4. Ecrire une fonction `cout_naif` qui prend `a` en argument et renvoie le coût obtenu par la méthode suivante : pour chaque examen, le candidat le plus proche du centre est désigné pour passer l'épreuve.
5. En considérant l'exemple `a = [0,20,9,1|]`, justifier que cette stratégie n'est pas optimale.

Coût optimal pour n adresses

Il s'agit maintenant de mettre en place une stratégie dynamique répondant au problème. Pour tout couple d'entiers (i, j) avec $i \neq j \in [\![0; n]\!]$, on note $c_{i,j}$ le coût optimal pour passer les épreuves de sorte que la dernière épreuve passée par le candidat 1 (resp. 2) soit la i-ième (resp. la j-ième). Il est clair que $c_{i,j} = c_{j,i}$ pour tous entiers i et j, par symétrie des rôles des deux candidats. On notera bien entendu que :

- Les candidats ne passent jamais la même épreuve, ce qui explique que l'on considère $i \neq j$,
- La dernière épreuve passée par l'un des candidats est donc celle d'indice $\max(i, j)$.

On construit donc une matrice `c` de taille $(n+1) \times (n+1)$ pour calculer ces coûts. Initialement, les cases reçoivent la valeur (-1).

6. Déterminer les valeurs de $c_{0,1}$ et $c_{1,0}$.
7. Soit $i < j \in [\![0; n]\!]$. Justifier les formules de récurrences suivantes :
   - Si $i < j - 1$, alors $c_{i,j} = c_{i,j-1} + |a_j - a_{j-1}|$,
   - Si $i \geq 1$ et $j = i + 1$, alors $c_{i,i+1} = \min_{k \in [\![0;i-1]\!]} c_{i,k} + |a_{i+1} - a_k|$.

   Expliquer dans quel ordre remplir la liste `c` à partir des formules.
8. En déduire la fonction `cout_opt` qui étant donné la liste des adresses `a` calcule le coût minimal pour présenter la totalité des épreuves.
   Quelle est la complexité de cette fonction ?
9. Ecrire une fonction `deplacement_opt` qui étant donné la liste des adresses `a` fournit une séquence de déplacements réalisant le coût minimal.
   On pourra modifier la fonction précédente et utiliser un dictionnaire `kopt` (ou une vulgaire liste) pour stocker les indices k pour lesquels le minimum est atteint dans la formule de la question 7.