

Devoir d'informatique n°2.

Partitions équilibrées d'entiers

On considère un ensemble E de n entiers naturels non nuls, codé en PYTHON par une liste.

On cherche à partitionner E en deux parties A et B de sorte à minimiser $\Delta = \left| \sum_{x \in A} x - \sum_{x \in B} x \right|$.

On note $s = \sum_{x \in E} x$ la somme des éléments de E . On peut reformuler le problème de la façon suivante :

$$\text{on cherche une partie } A \text{ de } E \text{ minimisant } \Delta = \left| 2 \sum_{x \in A} x - s \right|$$

Rappels : La valeur absolue s'obtient en PYTHON par la fonction `abs`.

$E[0 : k]$ désigne la liste des k premiers éléments de la liste E , c'est-à-dire $E[0], \dots, E[k-1]$.

1) Écrire une fonction `somme_liste(E)` qui renvoie s la somme de tous les éléments de E .

2) Méthode gloutonne

On lit les éléments de E dans l'ordre. On calcule **au fur et à mesure la somme s des éléments déjà lus** et la liste A des éléments sélectionnés. Pour chaque nouvel élément $y \in E$, on ajoute y à la liste A des éléments sélectionnés **si et seulement si ce choix diminue** la valeur de $\Delta = \left| 2 \sum_{x \in A} x - s \right|$, où s a été **mis à jour** (en prenant en compte le nouvel élément y).

Écrire la fonction `glouton(E)` qui renvoie la liste A obtenue par cet algorithme.

3) Force brute

On considère une fonction `genere(E)` qui renvoie la liste des 2^n parties de E .

Par exemple, `genere([])` renvoie `[[] ]` et `genere([3,5])` renvoie `[[] , [3] , [5] , [3,5] ]`.

On considère le code suivant :

```
01. def genere(E) :
02.     n = len(E) ; L = [[]]
03.     for x in E :
04.         M = [A.copy() for A in L]
05.         for A in M :
06.             L.append(A.append(x))
07.     return L
```

a) Expliquer pourquoi on utilise une copie de L à la ligne 04 (qui est utilisée lignes 05 et 06).

b) Écrire une fonction `equilibre(E)` qui par le calcul de la somme associée à chaque partie de E renvoie une partie A de E minimisant Δ .

c) Préciser sans justification la complexité des fonctions `genere` et `equilibre`.

4) Programmation dynamique de bas en haut

Pour tout $0 \leq k \leq n$ et $0 \leq i \leq s$, on définit

$$P(k, i) = \begin{cases} \text{True} & \text{s'il existe une partie } A \text{ de } E[0 : k] \text{ telle que } \sum_{x \in A} x = i \\ \text{False} & \text{sinon} \end{cases}$$

En particulier, $P(0, i)$ vaut **True** si $i = 0$, et vaut **False** sinon.

On rappelle que $E[0 : k]$ est la sous liste des $E[i]$ pour $0 \leq i \leq k - 1$. Elle contient k éléments.

En PYTHON, P est codé par une liste de listes. Autrement dit, $P(k, i)$ est codé par $P[k][i]$.

En particulier, la liste P est de longueur $(n + 1)$ et la liste $P[0]$ est de longueur $(s + 1)$.

a) Écrire une fonction **sommes(E)** qui renvoie P selon le principe suivant :

- On initialise P par $P = [[\text{False}] * (s+1) \text{ for } k \text{ in range}(n+1)]$; $P[0][0] = \text{True}$

- On définit $P(k, i)$ par récurrence, car $P(k, i)$ est fonction de $P(k - 1, i)$ et de $P(k - 1, i - E[k - 1])$.

Il faut faire bien attention à la valeur de $i - E[k - 1]$ pour éviter tout dépassement de tableau.

b) Donner sans justification la complexité de **sommes(E)** en fonction de n et de s .

c) En déduire une fonction **equilibre(E)** qui renvoie la valeur minimale de Δ .

d) Écrire une fonction **extraction(P, E, i)** qui étant donnés la liste de listes P , la liste E (de longueur n et de somme s) et un entier i tel que $0 \leq i \leq s$, renvoie :

- une liste A incluse dans E telle que $\sum_{x \in A} x = i$, si elle existe

- **None** sinon, c'est-à-dire si $P(n, i)$ vaut **False**.

On demande une fonction de complexité $O(n)$ indépendante de s .

Indication : On part de $(k, j) = (n, i)$ et de la liste vide A , et à l'aide de P et d'une boucle **décroissante** sur j , on détermine à chaque étape s'il convient d'ajouter ou non $E[k - 1]$ à A .

5) Programmation dynamique de haut en bas

On souhaite construire via une procédure récursive (avec mémorisation) le dictionnaire d dont les clés sont des couples (k, i) , où $0 \leq k \leq n$ et $0 \leq i \leq s$ et telle que la valeur de la clé (k, i) est une liste A incluse dans $E[0 : k]$ vérifiant $\sum_{x \in A} x = i$, si elle existe et la valeur **None** sinon. On considère le code suivant :

```
01. def sommes(E) :
02.     d = {}
03.     def traite(k,i) :
04.         if not (k,i) in d :
05.             if (k,i) == 0 : d[(0,0)] = []
06.             elif k == 0 : d[(0,i)] = None
07.             else :
08.                 ### à compléter
09.     return d
```

Compléter le code de la procédure auxiliaire récursive **traite** indiqué en ligne 08.

On utilisera autant de lignes que nécessaire.