Ce devoir est d'une durée de 2h, et et composé de deux parties indépendantes. Dans la partie 1, les fonctions sont à rédiger dans le langage Python. Dans la partie 2, les questions sont à rédiger dans le langage SQL. Le barème tient compte de la clarté et de la présentation de votre copie. On prendra garde à indiquer clairement le numéro des questions, et, autant que possible, à rédiger les questions dans l'ordre. Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

1 Partition d'une liste d'entiers

Dans cette partie, nous travaillerons avec des listes Python. L'usage de tableaux Numpy est interdit.

1.1 Sous-listes et partitions

On commence par introduire et manipuler quelques définitions utiles pour la suite du problème.

Définition 1 (sous-liste) *Soit `L` une liste. On dit que `L1` est une* ***sous-liste*** *de `L` si :*

- *tous les éléments de `L1` sont des éléments de `L` ;*
- *pour tout élément `x` de `L`, si `x` apparaît k fois dans `L`, alors `x` apparaît au plus k fois dans `L1`.*

1. Écrire une fonction `occurrences(L,x)` prenant en arguments une liste `L` et une valeur `x` et renvoyant le nombre d'occurrences de `x` dans `L`.
2. Écrire une fonction `est_sous_liste(L,L1)` prenant en arguments deux listes `L` et `L1` et renvoyant `True` si `L1` est une sous-liste de `L`, et `False` sinon.

Définition 2 (partition d'une liste) *Soit `L` une liste. On dit que deux listes `L1` et `L2` sont une* ***partition*** *de `L` si :*

- *`L1` et `L2` sont des sous-listes de `L` ;*
- *la concaténation `L1+L2` est également une sous-liste de `L` ;*
- *pour tout élément `x` de `L`, si `x` apparaît k fois dans `L`, alors `x` apparaît* ***exactement*** *k fois dans `L1+L2`.*

Autrement dit, les éléments de `L2` sont ***exactement*** *les éléments de `L` qui ne sont pas dans `L1`.*

3. Écrire une fonction `est_partition(L,L1,L2)` prenant en arguments trois listes `L`, `L1` et `L2` et renvoyant `True` si `L1` et `L2` forment une partition de `L`, et `False` sinon.

1.2 Problème de la partition

Dans cette partie, on s'intéresse au problème suivant (dit **problème de la partition**) : si `L` est une liste d'entiers positifs, est-il possible de partitionner `L` en deux listes `L1` et `L2` telles que la somme des éléments de `L1` soit égale à la somme des éléments de `L2` ?

4. Écrire une fonction `tous_positifs(L)` prenant en argument une liste `L` et renvoyant `True` si tous les éléments de `L` sont positifs ou nuls, et `False` sinon.
5. Écrire une fonction `somme(L)` prenant en argument une liste `L` et renvoyant la somme des éléments de `L`.
6. Soit `L` une liste.
   (a) Si `somme(L)` est impair, quelle est la réponse au problème de la partition de `L` ?
   (b) Si `somme(L)` est pair, montrer que le problème de la partition de `L` est équivalent au problème suivant : existe-t-il une sous-liste `L1` de `L` telle que `somme(L1)` $= \frac{\texttt{somme(L)}}{2}$?

Ainsi, on va s'intéresser au problème suivant (dit **problème de la sous-somme**) : si `L` est une liste d'entiers positifs et `total` est une valeur entière positive, existe-t-il une sous-liste `L1` de `L` telle que `somme(L1)` $=$ `total` ?

1.3 Résolution naïve

Une manière naïve de résoudre ce problème serait de tester toutes les sous-listes possibles `L1` de `L` jusqu'à en trouver une telle que `somme(L1)` = `total`. De plus, plutôt que de tester absolument toutes les sous-listes `L1` possibles, on peut se restreindre aux sous-listes `L1` de `L` dont les éléments apparaissent dans le même ordre que dans `L`.

7. (a) Si `L` est une liste de taille n, combien y a-t-il de sous-listes possibles de `L` dont les éléments apparaissent dans le même ordre que dans `L` ?
   (b) Quelle serait la complexité temporelle dans le pire des cas de cette approche naïve ?

On fournit le code de la fonction suivante :

```
def sous_somme(L,n_max,total):
    if total == 0:
        return True
    if total < 0 or n_max < 0:
        return False
    return sous_somme(L,n_max-1,total-L[n_max]) or sous_somme(L,n_max-1,total)
```

On rappelle la notation Python suivante : si `L` est une liste et i et j sont deux entiers, `L[i:j]` désigne la liste des éléments de `L` dont les indices sont dans $[\![i, j-1]\!]$.

8. Montrer que la fonction `sous_somme(L,n_max,total)` termine, pour toute liste `L` d'entiers positifs et tous entiers `n_max` < `len(L)` et `total` $\in \mathbb{N}$.
9. Montrer que pour toute liste `L` d'entiers positifs et tous entiers `n_max` < `len(L)` et `total` $\in \mathbb{N}$, `sous_somme(L,n_max,total)` renvoie `True` si et seulement s'il existe une sous-liste de `L[0:n_max+1]` dont la somme des éléments vaut `total`.
10. (a) À l'aide de la fonction `sous_somme`, écrire une fonction `partition(L)` prenant en argument une liste `L` d'entiers positifs, et renvoyant `True` s'il est possible de partitionner `L` en deux listes `L1` et `L2` telles que la somme des éléments de `L1` soit égale à la somme des éléments de `L2` (et `False` sinon).
    (b) Quelle est la complexité temporelle de votre fonction `partition` dans le pire des cas ?

1.4 Résolution par programmation dynamique

On cherche maintenant à améliorer la complexité dans le pire des cas de la fonction `sous_somme` en utilisant de la programmation dynamique. Pour cela, si `L` est une liste de taille n, on va construire une matrice de booléens `T` de taille $(n+1) \times (\texttt{total}+1)$ telle que :

$$\texttt{T[i][j]} = \begin{cases} \texttt{True} & \text{s'il existe une sous-liste de } \texttt{L[0:i]} \text{ dont la somme vaut } j\,; \\ \texttt{False} & \text{sinon.} \end{cases}$$

Cette matrice `T` sera implémentée par une liste de listes de booléens.

11. Écrire une fonction `init_false(n,m)` prenant en arguments de entiers positifs n et m, et renvoyant une matrice `T` de taille $n \times m$ (sous la forme d'une liste de listes) dont toutes les cases sont initialisées à `False`.
    Attention : on prendra garde à ce que les lignes de `T` ne soient pas liées dans la mémoire.
12. Écrire une fonction `sous_somme_dyna(L,total)` prenant en entrée une liste `L` d'entiers positifs et une valeur `total` $\in \mathbb{N}$ et renvoyant `True` s'il existe une sous-liste de `L` dont la somme des éléments vaut `total`, et `False` sinon.
    Attention : votre fonction devra avoir une complexité polynomiale en `len(L)` et `total`.
13. En déduire une fonction `partition_dyna(L)` prenant en argument une liste `L` d'entiers positifs, et renvoyant `True` s'il est possible de partitionner `L` en deux listes `L1` et `L2` telles que la somme des éléments de `L1` soit égale à la somme des éléments de `L2` (et `False` sinon).
    Attention : votre fonction devra avoir une complexité polynomiale en `len(L)`.

2 Randonnée (d'après Mines-Ponts 2021)

Lors de la préparation d'une randonnée, une accompagnatrice doit prendre en compte les exigences des participants. Elle dispose d'informations rassemblées dans trois tables d'une base de données :

- la table `Rando` décrit les randonnées possibles :
  - champ `id` : **clé primaire** (numéro unique pour chaque randonnée) ;
  - champ `nom` : le nom de la randonnée ;
  - champ `diff` : le niveau de difficulté du parcours (entier entre 1 et 5) ;
  - champ `deniv` : le dénivelé de la randonnée (en mètres) ;
  - champ `duree` : la durée moyenne de la randonnée (en minutes) ;

id	nom	diff	deniv	duree
1	La belle des champs	1	20	30
2	Lac de Castellane	4	650	150
3	Le tour du mont	2	200	120
4	Les crêtes de la mort	5	1200	360
5	Yukon Ho !	3	700	210
...	...	...	...	...

- la table `Participant` décrit les randonneurs :
  - champ `id` : **clé primaire** (numéro unique pour chaque randonneur) ;
  - champ `nom` : le nom du randonneur ;
  - champ `annee` : l'année de naissance du randonneur ;
  - champ `diff_max` : le niveau de difficulté maximal des randonnées qu'il est capable de faire ;

id	nom	annee	diff_max
1	Calvin	2014	2
2	Hobbes	2015	1
3	Susie	2014	2
4	Rosalyn	2001	4
5	Tintin	1991	5
...	...	...	...

- la table `Historique` décrit les randonnées que chaque randonneur a déjà effectuées :
  - champ `id_pers` : la clé primaire d'un randonneur ;
  - champ `id_rando` : la clé primaire d'une randonnée.

id_pers	id_rando
1	3
2	1
3	1
4	2
4	5
5	2
5	4
...	...

14. Écrire une requête SQL renvoyant le nom des participants nés entre 1999 et 2003 inclus.
15. Écrire une requête SQL comptant le nombre de participants nés entre 1999 et 2003 inclus.
16. Écrire une requête SQL renvoyant le nom des participants pour lesquels la randonnée n°42 est trop difficile.
17. Écrire une requête SQL renvoyant la valeur maximale du dénivelé d'une randonnée.
18. Écrire une requête SQL renvoyant le nom des randonnées dont le dénivelé est maximal parmi les randonnées de la table.
19. Écrire une requête SQL calculant la durée moyenne des randonnées pour chaque niveau de difficulté.
20. Écrire une requête SQL renvoyant les niveaux de difficulté tels que pour toute randonnée de cette difficulté et de dénivelé inférieur ou égal à 800m, la durée d'une telle randonnée est inférieure ou égale à 150min.
21. Écrire une requête SQL renvoyant tous les couples (`nom_pers`, `nom_rando`) tels que `nom_pers` soit le nom d'une personne ayant déjà effectué la randonnée appelée `nom_rando`.
22. Écrire une requête SQL détectant les incohérences dans la base de données, c'est-à-dire renvoyant tous les couples (`id_pers`, `id_rando`) qui sont des entrées de la table `Historique` telles que la randonnée `id_rando` soit trop difficile pour la personne `id_pers`.
23. Écrire une requête SQL renvoyant tous les couples (`nom_pers`, `nom_rando`) tels que `nom_pers` soit le nom d'un participant et `nom_rando` soit le nom d'une randonnée qui n'est pas trop difficile pour `nom_pers`.
24. Écrire une requête SQL renvoyant tous les couples (`nom_pers`, `nom_rando`) où `nom_pers` est le nom d'une personne et `nom_rando` est le nom d'une randonnée telle que `nom_pers` n'a pas déjà effectué `nom_rando` et `nom_rando` n'est pas trop difficile pour `nom_pers`.