1 Partition d'une liste d'entiers

1.1 Sous-listes et partitions

1. Rappel : il y a deux manières de parcourir une liste en Python, par les indices ou par les valeurs.

Parcours par les indices
```
def occurrences(L,x):
    n = len(L)
    c = 0
    for i in range(n):
        if L[i]==x:
            c += 1
    return c
```

Parcours par les valeurs
```
def occurrences(L,x):
    c = 0
    for y in L:
        if y==x:
            c += 1
    return c
```

2.
```
def est_sous_liste(L,L1):
    for x in L1:
        if x not in L:
            return False
    for x in L:
        if occurrences(L,x) < occurrences(L1,x):
            return False
    return True
```

3.
```
def est_partition(L,L1,L2):
    if not est_sous_liste(L,L1):
        return False
    if not est_sous_liste(L,L2):
        return False
    concat = L1+L2
    if not est_sous_liste(L,concat):
        return False
    for x in L:
        if occurrences(L,x) != occurrences(concat,x):
            return False
    return True
```

1.2 Problème de la partition

4.

Parcours par les indices
```
def tous_positifs(L):
    n = len(L)
    for i in range(n):
        if L[i] < 0:
            return False
    return True
```

Parcours par les valeurs
```
def tous_positifs(L):
    for x in L:
        if x < 0:
            return False
    return True
```

5.

Parcours par les indices
```
def somme(L):
    n = len(L)
    s = 0
    for i in range(n):
        s += L[i]
    return s
```

Parcours par les valeurs
```
def somme(L):
    s = 0
    for x in L:
        s += x
    return s
```

6. (a) Si `somme(L)` est impair, le problème de la partition de `L` n'a pas de solution.
   En effet, si `L1` et `L2` sont une partition de `L` satisfaisant le problème, alors `somme(L1)` = `somme(L2)` et `somme(L)` = `somme(L1)` + `somme(L2)` = $2 \times$ `somme(L1)`, donc `somme(L)` est forcément pair.

   (b) Si `L` une liste telle que `somme(L)` soit pair.
   - Si le problème de la partition de `L` a une solution (`L1`, `L2`), alors `somme(L1)` = `somme(L2)` et `somme(L)` = `somme(L1)` + `somme(L2)`, donc `somme(L1)` = $\frac{\texttt{somme(L)}}{2}$.
   - Réciproquement, s'il existe une sous-liste `L1` de `L` telle que `somme(L1)` = $\frac{\texttt{somme(L)}}{2}$, alors en prenant `L2` le complémentaire de `L1`, on obtient bien une partition de `L` satisfaisant le problème.

1.3 Résolution naïve

7. (a) Si `L` est de taille n, il y a 2^n sous-listes `L1` de `L` dont les éléments apparaissent dans le même ordre que dans `L`.
   En effet, pour chaque élément `x` de `L`, on a deux choix indépendants : soit on garde `x` dans `L1`, soit on ne le garde pas.

   (b) Puisqu'il y aurait un nombre exponentiel de sous-listes `L1` à tester, et que pour chacune d'entre elles, le calcul de `somme(L1)` se fera en temps linéaire, alors dans le pire des cas la réponse est `False` et il faudra toutes les tester : on aura donc une complexité temporelle exponentielle.

8. Tout d'abord, on remarque que si $0 \leq$ `n_max` $<$ `len(L)`, l'accès à `L[n_max]` (ligne 6) ne produira pas d'erreur.
   Ensuite, si on passe par le `return` de la ligne 6, on effectue au plus deux appels récursifs, avec `n_max`$'$ = `n_max` $- 1$. Donc la valeur de `n_max` décroit strictement à chaque appel récursif. De plus, cette valeur est entière, donc elle va atteindre une valeur strictement négative en un nombre fini d'étapes. Or la fonction `sous_somme` termine immédiatement si `n_max` < 0 (lignes 4 et 5).
   Ainsi, l'appel à `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. On montre cette propriété par récurrence sur `n_max` :
   - cas `n_max` < 0 : dans ce cas, `L[0:n_max+1]` est la liste vide, donc une telle sous-liste n'existe pas et il est correct de renvoyer `False` (lignes 4 et 5) ;
   - cas `n_max` ≥ 0 :
     - si `total` < 0, comme `L` est supposée ne contenir que des entiers positifs, une telle sous-liste n'existe pas, et il est correct de renvoyer `False` (lignes 4 et 5) ;
     - si `total` $= 0$, la sous-liste vide convient, et il est correct de renvoyer `True` (lignes 2 et 3) ;
     - sinon :
       - Soit une telle sous-liste `L1` existe et contient la valeur `L[n_max]` : dans ce cas, en enlevant cette valeur à `L1`, on obtient une sous-liste de `L[0:n_max]` dont la somme des éléments vaut `total-L[n_max]`. Ainsi, par hypothèse de récurrence, le premier appel récursif de la ligne 6 renvoie `True`, donc `sous_somme(L,n_max,total)` renvoie `True`, ce qui est correct.
       - Soit une telle sous-liste `L1` existe et ne contient pas la valeur `L[n_max]` : dans ce cas, `L1` est une sous-liste de `L[0:n_max]` dont la somme des éléments vaut `total`. Ainsi, par hypothèse de récurrence, le second appel récursif de la ligne 6 renvoie `True`, donc `sous_somme(L,n_max,total)` renvoie `True`, ce qui est correct.
       - Soit une telle sous-liste n'existe pas : dans ce cas, par hypothèse de récurrence, les deux appels récursifs de la ligne 6 vont renvoyer `False`, et `sous_somme(L,n_max,total)` renvoie `False`, ce qui est correct.

10. (a)
```python
def partition(L):
    total = somme(L)
    n_max = len(L) - 1
    if total % 2 == 1:
        return False
    else:
        return sous_somme(L,n_max,total//2)
```

(b) Dans le pire des cas, à chaque passage dans la ligne 6 de la fonction `sous_somme`, on effectuera deux appels récursifs, ce qui donnera une complexité temporelle exponentielle.
Dans ce cas, les différents appels récursifs correspondent au test de chaque sous liste `L1` de `L` dont les éléments apparaissent dans le même ordre que dans `L`.

1.4 Résolution par programmation dynamique

11. Version 'à la main'
```python
def init_false(n,m):
    T = []
    for i in range(n):
        T_i = []
        for j in range(m):
            T_i.append(False)
        T.append(T_i)
    return T
```

Version 'en une ligne'
```python
def init_false(n,m):
    return [[False]*m for _ in range(n)]
```

12.
```python
def sous_somme_dyna(L,total):
    n = len(L)
    T = init_false(n+1,total+1)
    # première colonne
    for i in range(n+1):
        T[i][0] = True
    for i in range(1,n+1):
        for j in range(1,total+1):
            j2 = j - L[i-1]
            if j2 < 0:
                T[i][j] = T[i-1][j]
            else:
                T[i][j] = T[i-1][j] or T[i-1][j2]
    return T[n][total]
```

13.
```python
def partition_dyna(L):
    total = somme(L)
    if total % 2 == 1:
        return False
    else:
        return sous_somme_dyna(L,total//2)
```

2 Randonnée (d'après Mines-Ponts 2021)

14.
```sql
SELECT nom
FROM Participant
WHERE annee >= 1999 AND annee <= 2003
```

15.
```sql
SELECT COUNT(*)
FROM Participant
WHERE annee >= 1999 AND annee <= 2003
```

16. On peut obtenir la difficulté de la randonnée n°42 à l'aide d'une sous-requête :

```sql
SELECT nom
FROM Participant
WHERE diff_max < (SELECT diff FROM Rando WHERE id = 42)
```

17.
```sql
SELECT MAX(deniv) FROM Rando
```

18.
```sql
SELECT nom
FROM Rando
WHERE deniv = (SELECT MAX(deniv) FROM Rando)
```

19.
```sql
SELECT diff, AVG(duree)
FROM Rando
GROUP BY diff
```

20.
```sql
SELECT diff
FROM Rando
WHERE deniv <= 800 -- on ne garde que les randonnées de dénivelé <= 800m
GROUP BY diff -- on regroupe ces randonnées par difficulté
HAVING MAX(duree) <= 150 -- pour chaque difficulté, on teste la durée maximale
                         -- d'une telle randonnée
```

21.
```sql
SELECT P.nom, R.nom
FROM Participant as P JOIN Historique as H JOIN Rando as R
ON P.id = id_pers AND id_rando = R.id
```

22.
```sql
SELECT id_pers, id_rando
FROM Historique as H JOIN Participant as P JOIN Rando as R
ON id_pers = P.id AND id_rando = R.id
WHERE diff > diff_max
```

23.
```sql
SELECT P.nom, R.nom
FROM Participant as P, Rando as R -- on fait un produit cartésien
WHERE R.diff <= P.diff_max
```

24. On reprend la question 23 mais on teste aussi que `(P.id,R.id)` n'apparaît pas dans `Historique`. Pour cela, on peut écrire une sous-requête qui compte combien de fois l'entrée `(P.id,R.id)` apparaît dans `Historique`, et tester que cette sous-requête renvoie 0 :

```sql
SELECT P.nom, R.nom
FROM Participant as P, Rando as R -- on fait un produit cartésien
WHERE R.diff <= P.diff_max
AND 0 = (SELECT COUNT(*)
         FROM Historique
         WHERE id_pers = P.id AND id_rando = R.id)
```