```python
# Algorithme et complexité
# Annexe des fonctions créées
# Déric Augusto FRANÇA DE SALES
# 30/12/22

import numpy as np

def balance2Plateaux(W):

  # Attribution d'éléments aux plats
  leftPlate = W[:]
  rightPlate = []

  oddLoop = True
  while 1:
    # Calcul du poids total des assiettes
    weightLeft = sum(leftPlate)
    weightRight = sum(rightPlate)

    # 1ère condition d'arrêt : sortie d'une boucle infinie
    if oddLoop:
      weightLeftPrev = weightLeft
      oddLoop = False
    else:
      if weightLeftPrev == weightLeft:
        break
      else:
        oddLoop = True

    # Calcul de la différence de poids entre les deux plaques
    diffWSides = abs(weightLeft - weightRight)

    # Passer l'élément qui minimise la différence avec l'autre plaque
    diffWList = []
    if weightLeft > weightRight:
      for i in range(len(leftPlate)):
        diffWList.append(diffWSides - leftPlate[i])
      # 2ème condition d'arrêt : si la différence ne peut être minimisée
      if all(n <= 0 for n in diffWList):
        break
      else:
        minPositive = min([x for x in diffWList if x >= 0])
        rightPlate.append(leftPlate.pop(diffWList.index(minPositive)))
    else:
      for i in range(len(rightPlate)):
        diffWList.append(diffWSides - rightPlate[i])
      if all(n <= 0 for n in diffWList):
        break
      else:
        minPositive = min([x for x in diffWList if x >= 0])
        leftPlate.append(rightPlate.pop(diffWList.index(minPositive)))

  return leftPlate, rightPlate


def backtracking(W, n = 2, X = np.array([]), solutions = {}):
  '''
    Équilibre les poids W sur n plateaux, en utilisant un algorithme
    de backtracking (retour en arrière). Renvoie la matrice de solution avec
```

```
    la meilleure solution trouvée.
  '''
  # Initialisant la fonction
  if X.size == 0 :
    lenW = len(W)
    X = np.zeros((n,lenW), dtype=int)
    X[0,:] = np.ones((lenW), dtype=int)

  # calcule la liste des poids équilibrés
  weightList = totalWeight(W, X)
  # enregistre la solution actuelle dans le dictionnaire des solutions
  solutions[balanceValue(weightList)] = np.copy(X)

  # cas trivial (les poids distribués sur chaque plateau sont exactement égals)
  elementsAreEqual = np.all(weightList == weightList[0])
  elementsAreInScales = np.all(np.any(X == 1, axis=0))
  if elementsAreEqual and elementsAreInScales:
    return X

  # cas récursif
  else:
    try:
      X = moveWeight(X, -1)
      return backtracking(W, n, X, solutions)
      # s'il a essayé toutes les possibilités,
      # renvoie la meilleure solution trouvée
    except IndexError:
      return solutions[min(solutions.keys())]


def backtrackingSol(W, n = 2, solution = False, X = np.array([]), solutions = {}):
  '''
    Effectue l'équilibrage des poids W sur n plaques, par un algorithme
    de backtracking. Renvoie la matrice de solution avec la meilleure solution
    trouvée. Si la variable solution est définie comme vraie, renvoie le tuple
    dont le premier élément est la matrice avec la meilleure solution X et le
    second, un dictionnaire de solutions, où les clés sont les grandeurs qui
    représentent la qualité de la solution et les clés la solution X.
  '''
  # Initialisant la fonction
  if X.size == 0 :
    lenW = len(W)
    X = np.zeros((n,lenW), dtype=int)
    X[0,:] = np.ones((lenW), dtype=int)

  # calcule la liste des poids équilibrés
  weightList = totalWeight(W, X)
  # enregistre la solution actuelle dans le dictionnaire des solutions
  solutions[balanceValue(weightList)] = np.copy(X)

  # cas trivial (les poids distribués sur chaque plateau sont exactement égals)
  if solution == False:
    elementsAreEqual = np.all(weightList == weightList[0])
    elementsAreInScales = np.all(np.any(X == 1, axis=0))
    if elementsAreEqual and elementsAreInScales:
      return X

  # cas récursif
  else:
    try:
```

```
        X = moveWeight(X, -1)
        return backtracking(W, n, solution, X, solutions)
        # s'il a essayé toutes les possibilités,
        # renvoie la meilleure solution trouvée
      except IndexError:
        if solution == True:
          return solutions[min(solutions.keys())], solutions
        else:
          return solutions[min(solutions.keys())]


def moveWeight(X, col):
  '''
      Reçoit en entrée la matrice de positionnement des poids X et la colonne
      où les poids seront déplacés. Il s'agit d'une fonction récursive où le
      mouvement commence toujours à partir de la dernière colonne jusqu'à la
      première. L'élément de la colonne est déplacé à la rangée du bas,
      représentant un mouvement entre les différentes plaques du de l'échelle.
      Il produit la matrice X avec le poids déplacé dans la colonne indiqué.
  '''
  n = X.shape[0]
  column = X[:, col]
  number1LineIndex = int(np.where(column == 1)[0])

  # cas trivial (déplacement du numéro 1 en X)
  if number1LineIndex <= n-2 :
    # en déplaçant le 1 sur la ligne ci-dessous
    X[number1LineIndex, col] = 0
    X[number1LineIndex + 1, col] = 1
    return X

  # cas récursif
  # (on ne peut pas déplacer le 1, donc on se déplace dans la colonne précédente)
  else :
    # déplacer l'un à la première ligne de la colonne analysée
    X[number1LineIndex, col] = 0
    X[0, col] = 1
    return moveWeight(X, col-1)


def totalWeight(W, X):
  '''
    renvoie le tableau numpy des poids équilibrés entre chaque plaque à partir
    du des vecteurs W de poids et X, qui indique où les poids sont positionnés
    entre chaque échelle.
  '''
# renvoie une liste numpy des poids équilibrés entre chaque plaque à partir de
# la méthode vecteurs W de poids et X
  numRows, numColumns = X.shape
  weightList = np.array([])
  for row in range(numRows):
    plateWeight = 0
    for column in range(numColumns):
      plateWeight += W[column]*X[row][column]
    weightList = np.append(weightList, [plateWeight])
  return weightList


def weightDifference(weightList):
  '''
```

```
    Renvoie le vecteur (tableau numpy) qui indique toutes les différences des
    éléments à partir du tableau des poids.
  '''
# renvoie le vecteur avec les différences entre tous les éléments
  diffList = np.array([], dtype=int)
  weightList = np.copy(weightList)
  weightsToCompare = np.copy(weightList)

  # pour chaque élément
  for index1 in range(weightList.size):
    weightOnPlate = weightList[index1]
    weightsToCompare = np.delete(weightsToCompare, 0)

    # Différence entre l'élément X et tous les autres
    for index2 in range(weightsToCompare.size):
      diff = abs(weightOnPlate - weightsToCompare[index2])
      diffList = np.append(diffList, [diff])

  return diffList


def balanceValue(weightList):
  '''
    Renvoie une valeur qui indique le degré d'équilibre de la solution.
    Cette valeur est généré à partir de la moyenne de la liste des différences
    entre les poids. Prend comme entrée la liste des poids totaux dans chaque
    plateau de la balance. Introduire la liste des poids totaux dans chaque
    plateau de la balance.
  '''
  # renvoie la valeur qui définit le degré d'équilibre de la solution
  diffList = weightDifference(weightList)
  return sum(diffList)/len(diffList)
```