ANALYSE DU PROBLÈME BALANCE À N PLATEAUX

Algorithme et Complexité

Université de Lorraine - ENSEM ISN 2A

Déric Augusto França de Sales

32219632

deric.sales@ufv.br

01 janvier 2023

1 le problème Balance à deux Plateaux

Traditionnellement le problème est constituée d'une balance d'assiettes, qui doit être équilibrée à partir d'une quantité M de poids à N plateaux. Pour illustrer, nous allons d'abord considérer un système où la balance comporte deux plateaux qui doivent être équilibrés. Les plateaux sont équidistants du centre de la balance et physiquement les poids exercent un couple sur le bras de la balance qui supporte le plateau (figure 1). Le système ne sera en équilibre lorsque le poids du côté gauche sont en équilibré avec le poids du côté droit. Si les poids ne sont pas équilibrés, les couples générés par les deux côtés ne s'annuleront pas et le côté le plus lourd descendra. Ainsi, le problème vise à positionner les M corps qui présentent des poids différents sur le côté droit ou gauche de la balance, en recherchant l'équilibre maximal.

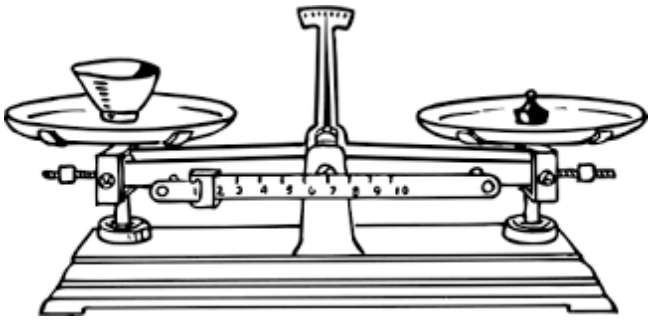

Figura 1: Représentation d'une balance de plateaux.

Il ne sera pas toujours possible d'obtenir un équilibre parfait (cas où les poids du côté droit et du côté gauche sont égaux), mais il cherche à réduire la différence entre les poids qui se trouvent des deux côtés de la balance. Ainsi, en modélisant le problème mathématiquement, on aura :

$$w_i = [w_1, w_2, ..., w_m] \tag{1}$$

$$x_i = \begin{cases} 0 & \text{si } w_i \text{ est placé sur le côté gauche de l'échelle;} \\ 1 & \text{si } w_i \text{ est placé sur le côté droit de l'échelle.} \end{cases} \tag{2}$$

Où w_i (1) c'est le vecteur de m poids, et x_i (2) le vecteur représentant la direction (droite ou gauche) dans laquelle le poids est positionné sur la balance. Une solution idéale est trouvée lorsque les poids sont parfaitement équilibrés entre les côtés droit et gauche de la balance, illustrée par la relation 3.

$$\sum_{i=1}^{m} w_i x_i = \sum_{i=1}^{m} w_i (1 - x_i) \tag{3}$$

Étant donné un ensemble arbitraire de n poids, il n'y a aucune garantie qu'une solution au problème existe. Une solution existe toujours si, au lieu d'équilibrer les balances, le but est de minimiser la différence entre les poids totaux des plateaux de gauche et de droite. Ainsi, à travers le vecteur 1, l'objectif sera la minimisation de δ, comme le montre la relation 4, en considérant la contrainte que tous les poids de 1 doivent être positionnés sur la balance.

$$\delta = \left| \sum_{i=1}^{m} w_i x_i - \sum_{i=1}^{m} w_i (1 - x_i) \right| \tag{4}$$

2 résoudre le problème en codant un algorithme

Alors, allons y à tout suite analyser le problème en codant un algorithme. L'algorithme en question est chargé d'équilibrer les poids du vecteur W sur les plateaux droite et gauche. La fonction "balance2Plateaux" (joint au dossier, ainsi que les autres fonctions créées) prend en entrée le tableau python W qui contient une liste de nombres représentant les poids à équilibrer.

```
W1 = [3, 2, 1, 4, 2]
W2 = [1,2,3,6]
W3 = [843, -200, 400, 43]
W4 = [3, 10, 15, 5, 2, 1]

sol1 = balance2Plateaux(W1)
sol2 = balance2Plateaux(W2)
sol3 = balance2Plateaux(W3)
sol4 = balance2Plateaux(W4)

print("Solution 1 :"); print(sol1)
print("Solution 2 :"); print(sol2)
print("Solution 3 :"); print(sol3)
print("Solution 4 :"); print(sol4)
```

✓ 0.5s Python

```
Solution 1 :
([2, 1, 2], [4, 3])
Solution 2 :
([1, 2, 3], [6])
Solution 3 :
([-200, 400, 43], [843])
Solution 4 :
([3, 10, 2, 1], [15, 5])
```

Figura 2: Test 1 effectué.

Il est donc possible de constater que la fonction n'est pas efficace car elle ne fonctionne pas bien dans toutes les situations. Dans la solution 1 présentée par la Figure 2, la fonction donne comme réponse l'équilibrage des poids 2,1 et 2 dans un des plateaux de la balance et 4 et 3 dans l'autre plateau, étant donné que dans la solution optimale, l'équilibrage correct serait 4 et 2 dans un des plateaux et 3,1 et 2 dans l'autre, ayant 6 de poids total dans chaque plateau. La solution 2 est optimale, tandis que la solution 4 est également incorrecte. La solution 3 montre que la fonction ne traite pas bien non plus l'utilisation des numéros négatifs.

Ainsi, nous pouvons voir que la première approche adoptée n'est pas très efficace, puisque la plupart du temps elle ne renvoie pas une solution correcte au problème, qui maximise l'équilibre entre les plaques. De plus, la fonction ne fonctionne qu'avec l'équilibrage entre deux plats, mais elle ne peut pas être généralisée à N plats. Pour résoudre ce problème, il est possible d'utiliser la technique du *backtracking* (retour en arrière), qui crée un arbre de possibilités et parcourt les branches de l'arbre en essayant chaque possibilité. Nous pouvons donc étendre le problème à N plats et appliquer la technique du *backtracking* à sa solution.

3 résolution du problème Balance de M poids à N Plateaux à travers d'un algorithme de retour en arrière

Pour résoudre le problème dans le cas de N assiettes sur la balance, nous pouvons créer la variable X, qui sera une matrice chargée de stocker la solution au problème. Dans la matrice X, chaque ligne représente un plateau de balance différent, et chaque colonne représente un

des éléments de poids du vecteur W. La matrice est aussi binaire, comme le vecteur x dans la relation 2. C'est-à-dire que, si dans une case donnée s'il y a un numéro 1, cela signifie que le poids est sur cette échelle, et 0 sinon. Le schéma de la Figure 3 illustre la fonction créée pour résoudre ce problème.

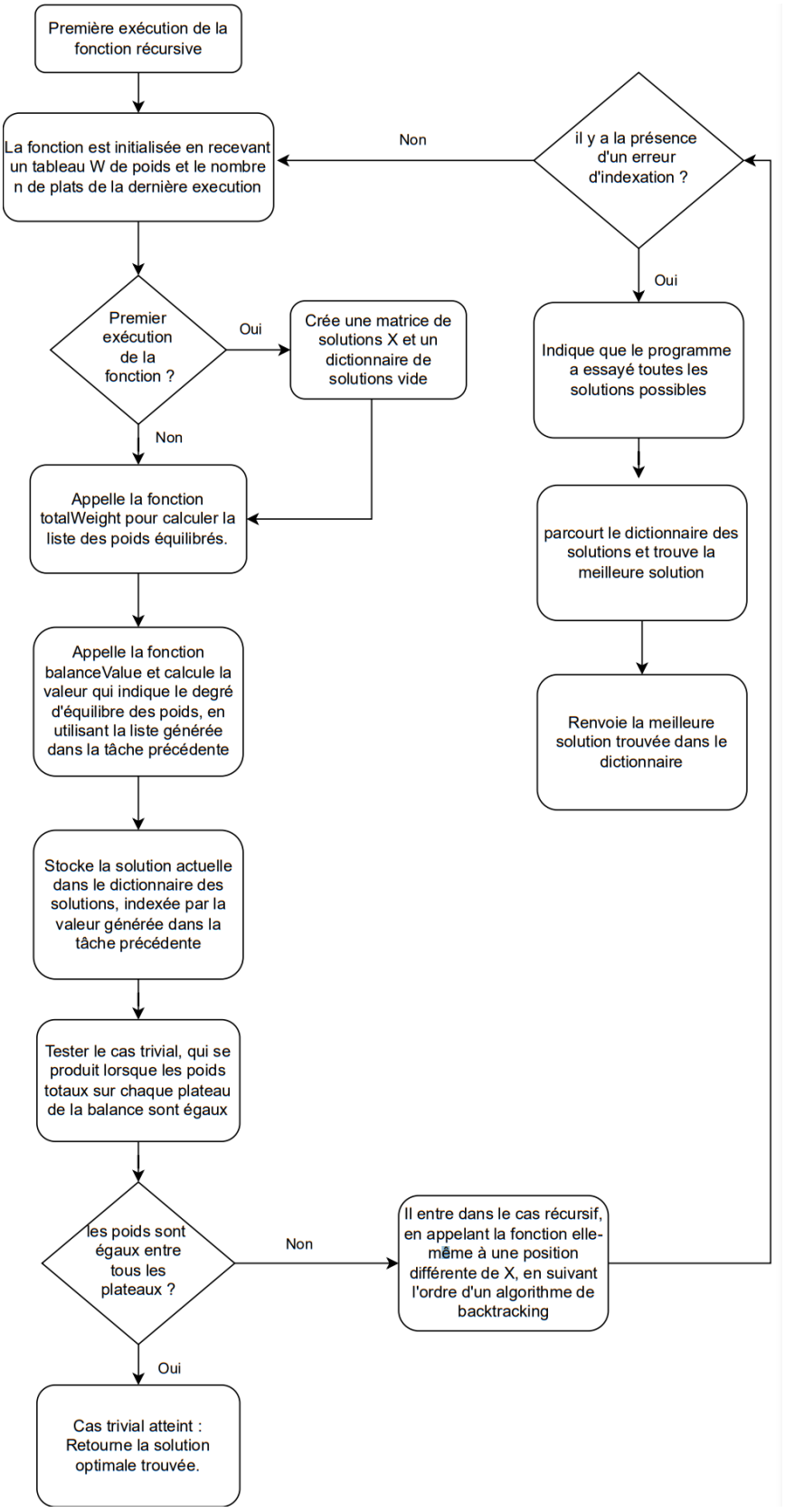

Figura 3: Diagramme de la fonction *backtracking*.

La fonction de retour en arrière effectue ses tâches en utilisant d'autres fonctions. Nous allons d'abord expliquer plus en profondeur ces autres fonctions (elles sont jointes et commentées à la fin du fichier).

La fonction *moveWeight* est chargée de déplacer les poids des plaques lorsqu'elle est appelée. C'est-à-dire qu'en passant la matrice de solution X et une colonne spécifique de cette matrice, qui indique l'un des éléments à déplacer, la fonction déplace l'élément vers la plateau suivante, ce qui est fait en déplaçant le numéro 1 vers la ligne inférieure de la colonne analysée. Ainsi, ce mouvement suit l'ordre d'un algorithme de *backtracking*, en parcourant l'ensemble de l'arbre des solutions une par une.

Pour expliquer le fonctionnement de l'algorithme de *backtracking*, nous allons prendre comme exemple un problème de 3 plateaux et 3 poids. Ce problème n'aurait pas de sens, car si nous avons trois plateaux et trois éléments, la seule solution possible serait de mettre un poids sur chaque plateau. Cependant, nous utiliserons cet exemple pour comprendre comment l'ordre d'exécution de l'algorithme est réalisé. La Figure 4 montre un arbre de solutions pour cet exemple, où les chiffres 0, 1 et 2 représentent le déplacement du chiffre 1 dans la matrice X dans chaque colonne. De plus, chaque niveau de l'arbre représente l'un des poids qui seront équilibrés.

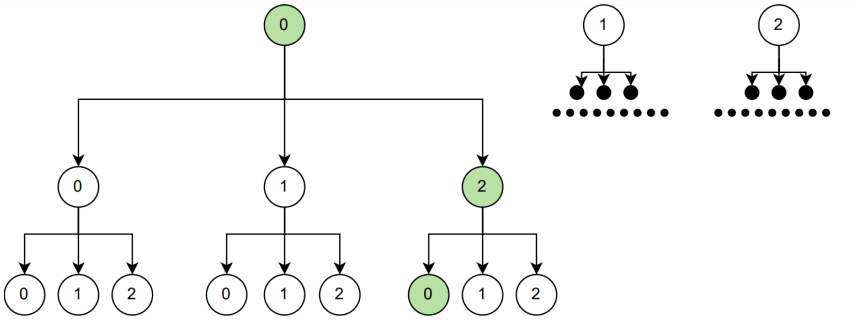

Figura 4: Arbre des solutions simplifié pour n=3 et 3 poids, exemple 1.

Ainsi, l'arbre de la Figure 4 générera la matrice X de la relation (5), où le premier élément du premier niveau de l'arbre a le mouvement 0, ce qui fait que le nombre 1 de la première colonne de la matrice reste dans la première ligne. De même, le deuxième poids aura un mouvement de 2, ce qui fait que le numéro 1 de la matrice X reste dans la troisième ligne, et le troisième poids, à son tour, a également un mouvement de 0, ce qui fait que le numéro 1 de la matrice X dans la dernière colonne reste dans la première ligne. Ainsi, dans cette solution, les poids 1 et 3 seront sur le premier plateau de la balance et le poids 2 sera sur le troisième plateau.

$$X = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{5}$$

Afin de comprendre l'ordre d'exécution de l'algorithme de *backtracking*, nous allons analyser la Figure 5, qui montre l'itération suivante par rapport à la position dans l'arbre des solutions décrit par la Figure 4.

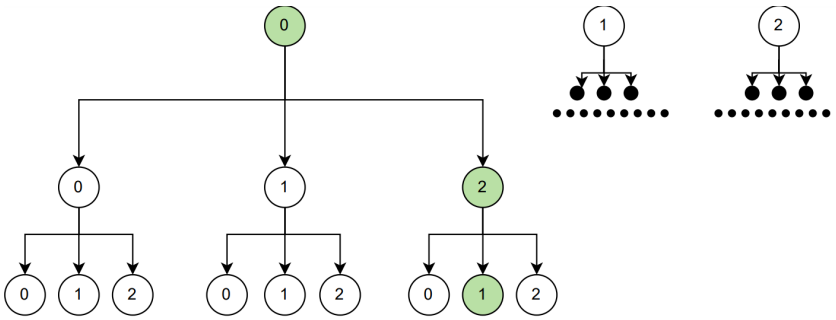

Figura 5: Arbre des solutions simplifié pour n=3 et 3 poids, exemple 2.

En passant à la branche suivante de l'arbre, nous allons générer la matrice des positions X de la relation (6). Ainsi, l'algorithme continue à fonctionner itération par itération, jusqu'à ce qu'il trouve une solution triviale, où les poids sont complètement équilibrés sur toutes les plateaux, ou jusqu'à ce qu'il essaie toutes les solutions et renvoie la meilleure. Ce qui a été expliqué pour n=3 (trois plateaux) et trois poids, vaut pour n plateaux et M poids, appartenant aux numéros naturels.

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{6}$$

Après avoir compris le fonctionnement de la fonction *moveWeight* qui exécute l'ordre selon un algorithme de *backtracking*, nous allons aborder les autres fonctions utilisées. La fonction *totalWeigh* calcule le poids total dans chaque plateau et renvoie une liste avec tous ces poids totaux. La fonction *weightDifference*, quant à elle, calcule la différence de poids entre tous ces poids totaux et renvoie une liste contenant toutes ces différences. Enfin, la fonction *balanceValue* calcule la moyenne de ces différences, pour générer une valeur qui sera utilisée pour indiquer la qualité de la solution. Cette valeur sera utilisée comme critère de décision entre les solutions trouvées.

Pour tester la fonction finale crée, on peut commencer par tester les mêmes exemples abordées par la Figure 2, avec deux plateaux. On peut conclure que la fonction *backtrecking* donne la bonne réponse pour tout les cas, même avec des numéros négatifs. Si on teste d'autres cas, avec n=3 par exemple, on peut obtenir les résultats suivants :

$$W1 = [3, 2, 1, 4, 2] \qquad W2 = [1, 2, 3, 6] \qquad W3 = [3, 13, 9, 8, 1, 5]$$

$$X1 = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad X2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \qquad X3 = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Nous pouvons noter que la fonction maximise l'équilibre dans tous les exemples, qu'il s'agisse d'une solution triviale ou non. Dans l'exemple de W1, la fonction atteint le résultat trivial ayant un poids total de 4 dans chaque plateau. L'exemple 3, quant à lui, présente un poids total de 13 dans chaque plateau.

4 analyse de la complexité de l'algorithme créé

Afin d'analyser la complexité de l'algorithme créé, nous allons d'abord analyser la complexité des fonctions utilisées dans la fonction de *backtracking*. Tout d'abord, en analysant la complexité de la fonction *totalWeight*, nous pouvons remarquer qu'elle présente deux boucles. L'une qui est exécutée en se référant au nombre de lignes de la matrice X, qui est égal au nombre de plateaux travaillés dans le problème, c'est-à-dire n. L'autre boucle s'exécute sur le nombre de colonnes de X, qui représente le nombre d'éléments à analyser, c'est-à-dire le nombre de poids du vecteur W. Nous appellerons le nombre de poids M. Ainsi, la fonction a un temps d'exécution O(n*M).

La fonction *weightDifference*, à son tour, présentera deux dimensions de boucles se référant à M, elle aura donc une complexité $O(M^2)$. La fonction *balanceValue* présentera également cette même complexité, pour l'exécution de la fonction *weightDifference* à l'intérieur.

La fonction *moveWeight*, quant à elle, présentera une complexité $(M-1)^n$ dans son exécution la plus défavorable, puisqu'elle suivra l'ordre de l'algorithme de *backtracking*. On peut s'en rendre compte en analysant les arbres des figures 4 ou 5, où chaque nœud multiplie le nombre de nœuds précédents. Dans l'exemple présenté dans les figures, nous aurons n = 3 et M = 3. Par conséquent, dans le pire des cas, si l'algorithme parcourt tout l'arbre, il sera exécuté $3^3 = 27$ fois.

```python
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())]
```

Figura 6: Analyse de la complexité de la fonction de *backtracking*.

En analysant la complexité de l'algorithme de *backtracking* à travers la figure 6, nous pouvons voir que le terme avec le plus haut degré est $(M-1)^n$, donc ce sera la complexité de l'algorithme. On peut le simplifier pour obtenir $O(2^n)$, comme on l'écrit le plus souvent. On peut donc noter que pour les très grands termes, le nombre de plateaux de la balance compte plus pour le temps d'exécution de l'algorithme que le nombre de poids à équilibrer.

```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)
```