Exercice 2 (6 points)

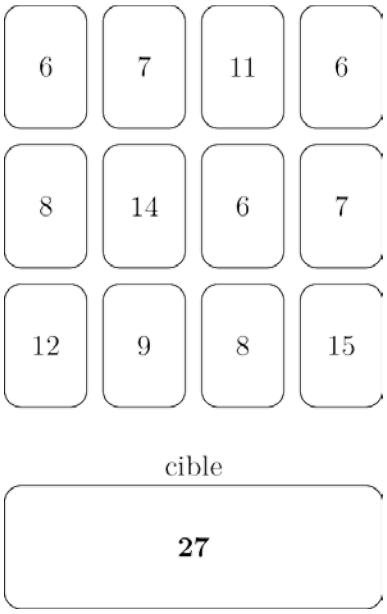
Cet exercice porte sur l'algorithmique, la récursivité et les files.

On cherche à implémenter en Python le jeu de cartes **soupe de nombres** créé par *Javier Dominguez Cruz* et paru aux éditions *EDGE*.

Il s'agit d'un jeu de calcul mental où les joueurs doivent réaliser un nombre à l'aide d'une succession d'opérations mathématiques élémentaires : addition, soustraction, multiplication et division. Par soucis de simplification, dans tout cet exercice, la seule opération utilisée sera **l'addition**.

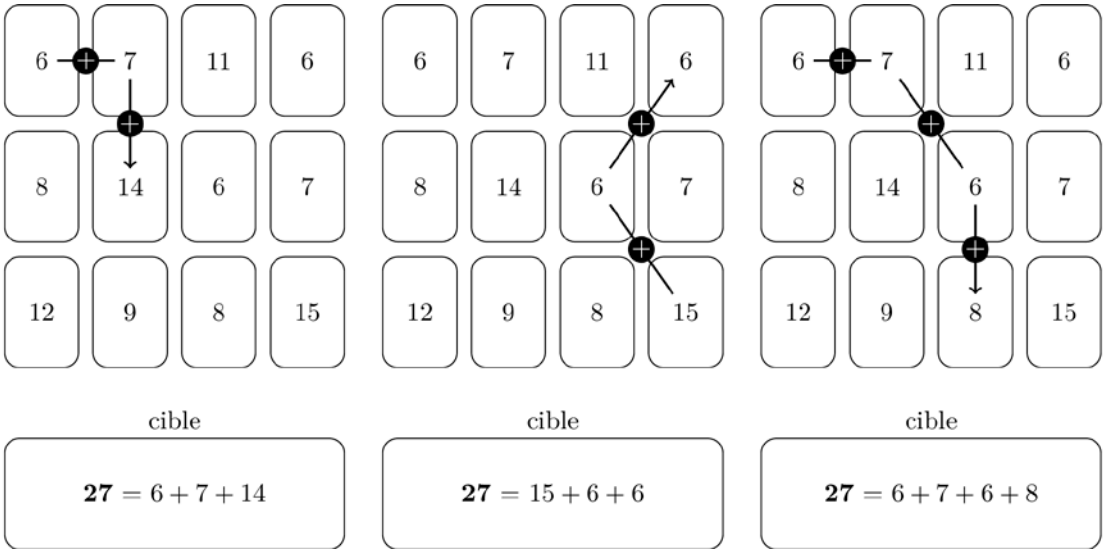
Tout d'abord, on dispose d'un jeu de cartes sur lesquelles est inscrit un nombre. On sélectionne aléatoirement 12 cartes que l'on dispose sous la forme d'une grille de 3 lignes et 4 colonnes. Dans la suite, une telle grille sera appelé *plateau* de jeu. Ensuite, plusieurs manches sont jouées en choisissant un nombre cible aléatoire.

Un exemple de manche est présenté dans le schéma suivant.



L'objectif est de trouver une plus courte chaîne valide (celle dont le nombre de cartes est le plus petit possible) dont la somme vaut la cible (cette chaine peut être constituée d'un seul nombre). Pour constituer une chaine, deux cartes consécutives de cette chaîne doivent être voisines dans la grille, horizontalement, verticalement ou diagonalement. Une même carte ne peut pas être utilisée plusieurs fois dans une chaine.

Avec le plateau ci-dessus, on peut obtenir le nombre 27 de plusieurs manières. Le schéma suivant présente trois exemples de chaînes pour obtenir 27. Les deux premières, de longueur 3, sont les plus courtes possibles.



On remarque que les enchainements suivants ne sont pas des chaines :

- $12 + 15$ car les cartes ne sont pas voisins ;
- $9 + 9 + 8$ car la carte contenant le nombre 9 est utilisée plusieurs fois.

Afin d'implémenter ce jeu en Python, on représente le plateau de nombres par la liste de ses lignes, chacune d'entre elles étant représentée par une liste d'entiers. Ainsi, le plateau précédent sera représenté par :

```
1 plateau = [
2     [ 6,  7, 11,  6],
3     [ 8, 14,  6,  7],
4     [12,  9,  8, 15]
5 ]
```

Le but est de proposer un code qui simule le jeu et propose une des meilleures solutions. On va considérer une extension des règles précédentes où les plateaux sont de taille quelconque.

Soit le code Python suivant :

```
1 v = plateau[1][2]
```

1. Donner la valeur de v obtenue.

On souhaite écrire une fonction `plateau_init` telle que `plateau_init(n, m, cartes)` où :

- `n` et `m` sont deux entiers naturels non nuls et
- `cartes` est une liste d'au moins $n \times m$ entiers

renvoie un plateau à `n` lignes et `m` colonnes comportant des nombres choisis au hasard parmi `cartes` sans remise.

La fonction `shuffle` du module `random` mélange aléatoirement une liste en la modifiant en place.

2. Compléter le programme suivant :

```
1 def plateau_init(n, m, cartes):
2     shuffle(cartes)
3     plateau = ...
4     for i i... ra...ge(...):
5         plateau.append([cartes[i+j*n] for j in
range(m)])
6     return ...
```

3. Compléter la fonction `cartes_voisines` telle que `carte_voisines(n, m, i, j)`, où `(i, j)` sont les coordonnées d'une carte dans un plateau à `n` lignes et `m` colonnes, renvoie la liste des coordonnées des cartes voisines de `(i, j)`.

```
1 def cartes_voisines(n, m, i, j):
2     voisines = []
3     for i2 in range(i-1, i+2):
4         for j2 in ...:
5             if (i2, j2) != (i, j) and \
6                 i2 in range(n) and \
7                 ...:
8                 voisines.append( ... )
9     return ...
```

On va représenter une chaîne comme une liste de coordonnées des cartes correspondantes. Ainsi, la chaîne $6 + 7 + 6 + 8$ présente dans le schéma précédent sera représentée par la liste Python suivante :

```
1 chaine = [ (0,0), (0,1), (1,2), (2,2) ]
```

Soit l'enchainement suivant :

```
1 e1 = [ (2,1), (1,0), (0,0), (0,1) ]
2 e2 = [ (1,1), (1,2), (2,0) ]
3 e3 = [ (0,2), (0,3), (1,4) ]
```

4. Donner la valeur associée à e1, e2 et e3 (si la chaine existe)

5. Écrire une fonction `chaine_evalue` qui prend en paramètres un plateau et une chaine, et renvoie la valeur associée à une telle chaîne.

   Avec la chaine précédente `chaine_evalue(plateau, chaine)` renverra l'entier `27`.

Pour réaliser l'exploration et obtenir une chaine la plus courte possible, on va considérer un parcours en largeur depuis chaque carte du plateau.

6. Expliquer pourquoi il est pertinent d'utiliser un parcours en largeur plutôt qu'un parcours en profondeur pour réaliser l'exploration permettant d'obtenir une chaine la plus courte possible.

Afin de réaliser ce parcours, on suppose définie une structure de donnée `file` munie des quatre opérations suivantes :

- `file_init` telle que `file_init()` renvoie une nouvelle file vide ;
- `file_ajoute` telle que `file_ajoute(f, x)` ajoute `x` dans la file `f` ;
- `file_retire` telle que `file_retire(f)` retire l'élément de `f` le plus anciennement ajouté, si la file est vide l'exception `ValueError` sera émise ;
- `file_est_vide` telle que `file_est_vide(f)` renvoie un booléen indiquant si la file `f` est vide.

7. Recopier et compléter les lignes incomplètes de la fonction `explore` qui réalise le parcours.

```
 1 def explore(plateau, cible):
 2     n, m = len(plateau), len(plateau[0])
 3     a_visiter = ...
 4     for i in range(n):
 5         for j in range(m):
 6             file_ajoute(a_visiter, [(i,j)])
 7
 8     while ...:
 9         chemin = file_retire(a_visiter)
10         if chaine_evalue(plateau, chemin) == ...:
11             return chemin
12         dernier = chemin[-1]
13         i0, j0 = dernier
14         for i, j in cartes_voisines(n, m, i0, j0):
15             if (i, j) not in ...:
16                 file_ajoute(a_visiter,
17                     chemin + [ (i,j) ])
18
19     # Pas de solutions
20     return None
```

8. Expliquer, sans écrire de code mais en décrivant votre solution, comment modifier la fonction `explore` afin qu'elle renvoie la liste de tous les chemins solutions.