DS2

Exercice I : Analyse d'algorithmes

Étant donné deux entiers naturels a et b, on cherche à calculer leur produit p.

Partie 1 : Méthode élémentaire

Cette première méthode consiste à décomposer le produit ab comme la somme $b+b+\cdots+b$ (avec a termes). Elle n'utilise que des additions.

```
1  def enfant(a, b) :
2      p, q = 0, a
3      while q > 0 :
4          p = p + b
5          q = q - 1
6      return p
```

1. Indiquer la spécification de l'algorithme.

Démonstration.

- Signature : `enfant(a : int, b : int)` $\to$ `p : int`.
- Pré-condition : aucune.
- Post-condition : `p = ab`

□

2. Donner, en justifiant, un *variant* qui prouve la terminaison de cet algorithme.

Démonstration.
La variable `q` définit une suite strictement décroissante d'entiers positifs (grâce à la mise à jour de ligne 5).

La variable `q` est donc un variant de la boucle `while`.
Ceci prouve la terminaison de l'algorithme.

□

3. Choisir un *invariant de boucle* judicieux parmi les propositions suivantes :

a) $(I_1) : p = ab$

b) $(I_2) : (p+b)q = ab$

c) $(I_3) : p+qb = ab$

d) $(I_4) : (p+q)b = ab$

Démonstration.
Vérifions que (I_3) est un invariant de la boucle `while`.
Démontrons par récurrence : $\forall i \in \mathbb{N}, \mathscr{P}(i)$ où $\mathscr{P}(i)$: à l'issue du $i^{\text{ème}}$ tour de boucle, $p+qb = ab$.

▸ **Initialisation** :
Avant la boucle `while` la variable p contient 0 et la variable q contient a. Ainsi :

$$p+qb \;=\; 0+ab \;=\; ab$$

D'où $\mathscr{P}(0)$.

- **Hérédité** : soit $i \in \mathbb{N}$.
  Supposons $\mathscr{P}(i)$ et démontrons $\mathscr{P}(i+1)$ (*i.e.* à l'issue du $(i+1)^{\text{ème}}$ tour de boucle, $p+qb=ab$).
  - × Par hypothèse de récurrence, à l'issue du $i^{\text{ème}}$ tour de boucle, $p+qb=ab$.
    On note p' (resp. q') la valeur de p (resp. q) à l'issue du $(i+1)^{\text{ème}}$ tour de boucle. Il s'agit donc de démontrer :
    $$p'+q'b \;=\; ab$$
  - × Or, d'après les lignes 4 et 5 :
    $$p'+q'b \;=\; (p+b)+(q-1)\,b \;=\; p+\cancel{b}+qb-\cancel{b} \;=\; p+qb$$
    Ainsi, d'après l'hypothèse de récurrence :
    $$p'+q'b \;=\; p+qb \;=\; ab$$

  D'où $\mathscr{P}(i+1)$.

Par principe de récurrence : $\forall i \in \mathbb{N}$, $\mathscr{P}(i)$.

La proposition (I_3) est donc un invariant de la boucle `while`. □

4. Prouver la correction de cet algorithme.

Démonstration.

- D'après la question *2.*, l'algorithme se termine.
- D'après la question précédente, la proposition (I_3) est un invariant de la boucle `while`.
  En particulier, à l'issue de la boucle `while`, la variable q contient 0. D'où :
  $$ab \;=\; p+qb \;=\; p+0\times b \;=\; p$$
  Ceci assure la correction partielle de l'algorithme.

L'algorithme se termine et est partiellement correct. Il est donc correct. □

Partie 2 : Méthode du paysan russe

La méthode du paysan russe est un très vieil algorithme déjà décrit (sous une forme légèrement différente) sur un papyrus égyptien rédigé vers 1650 av. J.-C. En comparaison de l'algorithme décimal classique, elle présente l'intérêt de n'utiliser que la table de multiplication par 2.

```
1  def paysan(a, b) :
2      p, x, y = 0, a, b
3      while x > 0 :
4          if x % 2 == 1 :
5              p = p + y
6          x = x // 2
7          y = y * 2
8      return p
```

5. Calculer 18×13 à l'aide de cet algorithme. On donnera dans un tableau les valeurs successives des variables p, x, y.

Démonstration.
Voici les valeurs successives de p, x et y.

p	0	0	26	26	26	234
x	18	9	4	2	1	0
y	13	26	52	104	208	416

On obtient : $18 \times 13 = 234$.

□

6. Donner, en justifiant, un *variant* qui prouve la terminaison de cet algorithme.

Démonstration.
La variable `x` définit une suite strictement décroissante d'entiers positifs. En effet, à chaque tour de boucle, la variable `x` est mise à jour en remplaçant sa valeur par son quotient dans sa division euclidienne par 2.

La variable `x` est un variant de la boucle `while`. Ceci assure la terminaison de l'algorithme.

□

7. Vérifier que $ab = p + xy$ est un *invariant de boucle* et en déduire la correction.

Démonstration.
Démontrons par récurrence : $\forall i \in \mathbb{N}, \mathscr{P}(i)$ où $\mathscr{P}(i)$: à l'issue du $i^{\text{ème}}$ tour de boucle, $ab = p+xy$.

- **Initialisation**
  Avant le 1^{er} tour de boucle, la variable p (resp. x, resp. y) est initialisée à 0 (resp. a, resp. b). Ainsi :
  $$p + xy \;=\; 0 + ab \;=\; ab$$
  D'où $\mathscr{P}(0)$.
- **Hérédité** : soit $i \in \mathbb{N}$.
  Supposons $\mathscr{P}(i)$ et démontrons $\mathscr{P}(i+1)$ (*i.e.* , à l'issue du $(i+1)^{\text{ème}}$ tour de boucle, $ab = p+xy$)
  - Par hypothèse de récurrence, à l'issue du $i^{\text{ème}}$ tour de boucle : $p + xy = ab$. On note p' (resp. x', resp. y') les valeurs de p (resp. x, resp. y), à l'issue du $(i+1)^{\text{ème}}$ tour de boucle.
  - Deux cas se présentent :
    × si x est impair (autrement dit, si le reste de la division euclidienne de x par 2 est 1), alors il existe $k \in \mathbb{N}$ tel que : $x = 2k + 1$. Dans ce cas, les mises à jour sont les suivantes :
    - $p' = p + y$
    - $x' = \left\lfloor \frac{x}{2} \right\rfloor = \left\lfloor \frac{2k+1}{2} \right\rfloor = \left\lfloor k + \frac{1}{2} \right\rfloor = k = \frac{x-1}{2}$
    - $y' = 2y$

    Ainsi :
    $$p' + x'\,y' \;=\; p + y + \frac{x-1}{2} \times 2y \;=\; p + y + xy - y \;=\; p + xy \;=\; ab$$
    La dernière égalité est obtenue par hypothèse de récurrence.

× si x est pair (autrement dit, si le reste de la division euclidienne de x par 2 est 0), alors il existe $k \in \mathbb{N}$ tel que : $x = 2k$. Dans ce cas, les mises à jour sont les suivantes :

- $p' = p$
- $x' = \left\lfloor \frac{x}{2} \right\rfloor = \left\lfloor \frac{2k}{2} \right\rfloor = \lfloor k \rfloor = k = \frac{x}{2}$
- $y' = 2y$

Ainsi :

$$p' + x'y' \;=\; p + \frac{x}{2} \times 2y \;=\; p + xy \;=\; ab$$

La dernière égalité est obtenue par hypothèse de récurrence.

D'où $\mathscr{P}(i+1)$.

Par principe de récurrence : $\forall i \in \mathbb{N}$, $\mathscr{P}(i)$.

On en déduit que la proposition $ab = p + xy$ est un invariant de la boucle `while`.

À l'issue de la boucle `while` : $x = 0$. Alors :

$$ab \;=\; p + xy \;=\; p + 0 \times y \;=\; p$$

Ceci assure la correction partielle de l'algorithme.

L'algorithme se termine d'après la question précédente, et est partiellement correct. Il est donc correct. □

8. On note $T(a)$ le nombre d'itérations de la boucle `while`.

a) Que vaut $T(0)$? Si $a \geqslant 1$, justifier : $T(a) = 1 + T(\lfloor \frac{a}{2} \rfloor)$.

Démonstration.

- Si $a = 0$, alors la variable `x` est initialisée à 0. Dans ce cas, il n'y aucun tour de boucle `while` car la condition `x > 0` n'est pas vérifiée.

  On en déduit : $T(0) = 0$.

- Si $a \geqslant 1$, alors :
  × la variable `x` est initialisée à a.
  × on effectue alors un tour de boucle où la variable `x` est mise à jour de la façon suivante :

  `x = x//2`

  On en déduit qu'à la fin du premier tour de boucle, la variable `x` contient $\left\lfloor \frac{a}{2} \right\rfloor$.

  Le nombre d'itérations de la boucle `while` nécessaires lorsque $a \geqslant 1$ est donc la somme de :
  × 1 : on effectue un premier tour de boucle,
  × le nombre d'itérations de la boucle `while` lorsque `x` est initialisée à $\left\lfloor \frac{a}{2} \right\rfloor$.

  On en déduit, si $a \geqslant 1$: $T(a) = 1 + T\left(\left\lfloor \frac{a}{2} \right\rfloor\right)$.

□

b) Calculer, pour tout $k \in \mathbb{N}$, la valeur de $T(2^k)$.

Démonstration.
Démontrons par récurrence : $\forall k \in \mathbb{N}$, $\mathscr{P}(k)$ où $\mathscr{P}(k) : T(2^k) = k+1$.

- **Initialisation**
  D'après la question précédente :

$$T(2^0) \;=\; T(1) \;=\; T\left(\left\lfloor \frac{1}{2} \right\rfloor\right) \;=\; T(0) \;=\; 1 \;=\; 0+1$$

  D'où $\mathscr{P}(0)$.
- **Hérédité** : soit $k \in \mathbb{N}$.
  Supposons $\mathscr{P}(k)$ et démontrons $\mathscr{P}(k+1)$ (*i.e.* $T(2^{k+1}) = k+2$)

$$\begin{aligned} T(2^{k+1}) &= 1 + T\left(\left\lfloor \frac{2^{k+1}}{2} \right\rfloor\right) && \textit{(d'après la question précédente)} \\ &= 1 + T\left(\lfloor 2^k \rfloor\right) \\ &= 1 + T(2^k) && \textit{(car } k \in \mathbb{Z}\textit{)} \\ &= 1 + (k+1) && \textit{(par hypothèse de récurrence)} \end{aligned}$$

  D'où $\mathscr{P}(k+1)$.

Par principe de récurrence : $\forall k \in \mathbb{N}$, $T(2^k) = k+1$.

□

c) Montrer finalement : $T(a) = O\big(\ln(a)\big)$.

Démonstration.

- La suite $\big(T(a)\big)_{a\in\mathbb{N}}$ est croissante. En effet, plus l'entier a est grand, plus le nombre de quotients par 2 de cet entier nécessaires pour atteindre 0 est élevé.
- Soit $a \in \mathbb{N}^*$. On note $k = \lfloor \log_2(a) \rfloor \in \mathbb{N}$. Alors :

$$2^k \leqslant a < 2^{k+1} \qquad (*)$$

  Or la suite $\big(T(a)\big)_{a\in\mathbb{N}}$ est croissante. Ainsi :

$$\begin{array}{ccccc} T(2^k) & \leqslant & T(a) & \leqslant & T(2^{k+1}) \\ \shortparallel & & & & \shortparallel \\ k+1 & \leqslant & T(a) & \leqslant & k+2 \end{array}$$

  Donc : $\left\lfloor \dfrac{\ln(a)}{\ln(2)} \right\rfloor + 1 \leqslant T(a) \leqslant \left\lfloor \dfrac{\ln(a)}{\ln(2)} \right\rfloor + 2$. Ainsi, par propriété de la partie entière :

$$\frac{\ln(a)}{\ln(2)} + 1 \;\leqslant\; \left\lfloor \frac{\ln(a)}{\ln(2)} \right\rfloor + 1 \;\leqslant\; T(a) \;\leqslant\; \left\lfloor \frac{\ln(a)}{\ln(2)} \right\rfloor + 2 \;<\; \frac{\ln(a)}{\ln(2)} + 1 + 2$$

Finalement, on trouve bien : $T(a) \underset{a\to+\infty}{=} O\big(\ln(a)\big)$.

Commentaire

- Détaillons $(*)$. Soit $a \in \mathbb{N}^*$. On note $k = \lfloor \log_2(a) \rfloor$. Alors, par définition de la partie entière :

$$\begin{array}{lccccc} & k & \leqslant & \log_2(a) & < & k+1 \\ \text{donc} & k\ln(2) & \leqslant & \ln(a) & < & (k+1)\ln(2) \quad \textit{(car : } \ln(2) \geqslant 0\textit{)} \\ \text{d'où} & 2^k & \leqslant & a & < & 2^{k+1} \quad \textit{(par stricte croissance de } \exp \textit{ sur } \mathbb{R}\textit{)} \end{array}$$

- Notons de plus que le fait que la suite $\big(T(a)\big)_{a\in\mathbb{N}}$ soit croissante n'est pas automatique. En toute généralité, le contenu d'une boucle `while` ou `for` pourrait contenir des cas où l'on sort plus tôt de la boucle (c'est par exemple le cas si la boucle contient des instructions `return`.

□

9. Comparer l'efficacité des deux algorithmes.

Démonstration.

- D'après la question précédente : $T(a) = \underset{a\to+\infty}{O}\big(\ln(a)\big)$.
- Cherchons $T'(a)$ le nombre d'itérations de la boucle `while` dans la fonction `enfant`.
  - × La variable `q` est initialisée à a.
  - × On effectue ensuite autant de tours de boucle que nécessaire pour que la variable `q` atteigne la valeur 0.
    Or, à chaque tour de boucle, la valeur stockée dans la variable `q` diminue de 1.

  On effectue donc : $T'(a) = a$ tours de boucle.

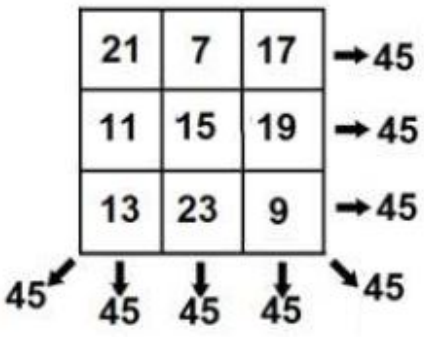
Ainsi : $T'(a) = \underset{a\to+\infty}{O}(a)$.

Le 2nd algorithme `paysan` est donc plus efficace asymptotiquement que le 1er.

□

Exercice II : Programmation et complexité

Soit $n \in \mathbb{N}^*$. Un carré magique d'ordre n est une matrice carrée d'ordre n qui contient des nombres entiers strictement positifs. Ces nombres sont disposés de sorte que les sommes sur chaque ligne, les sommes sur chaque colonne et les sommes sur chaque diagonale soient égales. La valeur de ces sommes est appelée *constante magique*.



Carré magique d'ordre 3 et de constante magique 45.

Pour représenter une matrice carrée d'ordre n, on utilisera une liste qui contient n listes toutes de même longueur n. Pour l'exemple précédent, on a donc :

- `M=[[21,7,17],[11,15,19],[13,23,9]]`,
- `M[1]=[11,15,19]`,
- `M[1][0]=11`.

Dans tout le sujet, on considèrera que les matrices carrées sont représentées par de telles listes et on ne vérifiera ni que la matrice est bien carrée, ni que les coefficients sont bien des entiers strictement positifs.

Partie A : Sommes d'éléments sur une matrice carrée

10. Écrire une fonction `somme_ligne` qui prend en paramètre une matrice carrée `M` sous forme de liste de listes et un entier `i`, et qui renvoie la somme des termes de la ligne d'indice `i` de la matrice `M`.

Par exemple, `somme_ligne([[1,2,3],[4,5,6],[7,8,9]], 1)`, renvoie : $4+5+6=15$.

Démonstration.

```
def somme_ligne(M, i) :
    n = len(M)
    S = 0
    for j in range(n) :
        S = S + M[i][j]
    return S
```

Commentaire

On pouvait également utiliser une compréhension de liste :

```
def somme_ligne(M, i) :
    n = len(M)
    L = [ M[i][j] for j in range(n) ]
    return sum(L)
```

□

11. Écrire une fonction `somme_colonne` qui prend en paramètre une matrice `M` sous forme de liste de listes et un entier `j`, et qui renvoie la somme des termes de la colonne d'indice `j` de `M`.

Par exemple, `somme_colonne([[1,2,3],[4,5,6],[7,8,9]], 0)` renvoie : $1+4+7=12$.

Démonstration.

```
def somme_colonne(M, j) :
    n = len(M)
    S = 0
    for i in range(n) :
        S = S + M[i][j]
    return S
```

Commentaire

On pouvait également utiliser une compréhension de liste :

```
def somme_colonne(M, j) :
    n = len(M)
    L = [ M[i][j] for i in range(n) ]
    return sum(L)
```

□

12. Écrire une fonction `somme_diag1` qui prend en paramètre une matrice `M` sous forme de liste de listes, et qui renvoie la somme des termes de la diagonale de `M`.

Par exemple, `somme_diag1([[1,2,3],[4,5,6],[7,8,9]])` renvoie : $1+5+9=15$.

Démonstration.

```
def somme_diag1(M) :
    n = len(M)
    S = 0
    for i in range(n) :
        S = S + M[i][i]
    return S
```

Commentaire

On pouvait également utiliser une compréhension de liste :

```
def somme_diag1(M) :
    n = len(M)
    L = [ M[i][i] for i in range(n) ]
    return sum(L)
```

□

13. Écrire une fonction `somme_diag2` qui prend en paramètre une matrice `M` sous forme de liste de listes, et qui renvoie la somme des termes de l'antidiagonale de `M` (celle qui va du coin en haut à droite jusqu'au coin en bas à gauche).

Par exemple, `somme_diag2([[1,2,3],[4,5,6],[7,8,9]])` renvoie : $3+5+7=15$.

Démonstration.

```
def somme_diag2(M) :
    n = len(M)
    S = 0
    for i in range(n) :
        S = S + M[i][n-1-i]
    return S
```

Commentaire

On pouvait également utiliser une compréhension de liste :

```
def somme_diag2(M) :
    n = len(M)
    L = [ M[i][n-1-i] for i in range(n) ]
    return sum(L)
```

□

14. Déterminer la complexité de ces quatre dernières fonctions en fonction de n.

Démonstration.
On considère comme fonction taille, la fonction :

$$\begin{array}{rccc} \text{taille} & : & \mathscr{M}_n(\mathbb{R}) & \to & \mathbb{N}^* \\ & & M & \mapsto & n \text{ (l'ordre de } M\text{)} \end{array}$$

Soit $n \in \mathbb{N}^*$. Soit $M \in \mathscr{M}_n(\mathbb{R})$.
On considère la somme (d'entiers) comme opération élémentaire.

- Dans chacune des quatre fonctions précédentes, on effectue 1 opération élémentaire par tour de boucle `for`.
- On effectue de plus toujours n tours de boucle par construction de ces boucles `for`.

Ainsi, en notant $C(n)$ la complexité de ces fonctions pour une matrice M de $\mathscr{M}_n(\mathbb{R})$, on obtient : $C(n) = n$.

Ces quatre fonctions ont donc une complexité linéaire. Autrement dit : $C(n) = \underset{n\to+\infty}{O}(n)$. □

Partie B : Carré magique

15. Écrire une fonction `carre_magique` qui prend en paramètre une matrice `M` sous forme de liste de listes, et qui renvoie `True` si `M` est un carré magique et `False` sinon.

Par exemple :

- l'appel `carre_magique([[1,2,3],[4,5,6],[7,8,9]])` renvoie : `False`,
- l'appel `carre_magique([[21,7,17],[11,15,19],[13,23,9]])` renvoie : `True`.

Démonstration.
On calcule une première somme (celle sur la diagonale dans la fonction suivante). Puis on lui compare toutes les autres sommes.

```
1  def carre_magique(M) :
2      n = len(M)
3      s = somme_diag1(M)
4      if somme_diag2(M) != s :
5          return False
6      for i in range(n) :
7          if somme_ligne(M, i) != s or somme_colonne(M, i) != s :
8              return False
9      return True
```

□

16. Déterminer la complexité de la fonction `carre_magique` dans le pire cas en fonction de n.

Démonstration.
On considère la même fonction taille qu'en question ***14.***
Soit $n \in \mathbb{N}^*$. Soit $M \in \mathscr{M}_n(\mathbb{R})$.
On considère la comparaison et la somme (d'entiers) comme opérations élémentaires.

- Le pire cas est celui où l'on effectue toutes les comparaisons, c'est-à-dire le cas où M est effectivement un carré magique.
- On compte :
  - × en ligne 3 : $\underset{n\to+\infty}{O}(n)$ sommes pour calculer `somme_diag1(M)` (d'après la question ***14.***).
  - × en ligne 4 :
    - 1 comparaison,
    - $\underset{n\to+\infty}{O}(n)$ sommes pour calculer `somme_diag2(M)` (toujours d'après ***14.***).

    On obtient $1 \times \underset{n\to+\infty}{O}(n) = \underset{n\to+\infty}{O}(n)$ opérations élémentaires.
  - × entre les lignes 6 à 8 :
    - 2 comparaisons par tour de boucle `for`,
    - $\underset{n\to+\infty}{O}(n)$ sommes par comparaison (d'après la question ***14.***),
    - n tours de boucle, par construction de la boucle `for` en ligne 6.

    On obtient $2 \times n \times \underset{n\to+\infty}{O}(n) = \underset{n\to+\infty}{O}(n^2)$ opérations élémentaires.

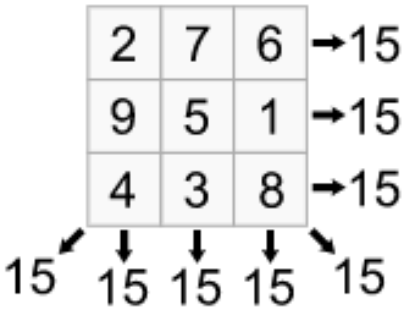
  Ainsi, on dénombre, dans le pire cas $C(n) = \underset{n\to+\infty}{O}(n) + \underset{n\to+\infty}{O}(n) + \underset{n\to+\infty}{O}(n^2) = \underset{n\to+\infty}{O}(n^2)$ opérations élémentaires.

La complexité de cette algorithme est donc quadratique : $C(n) = \underset{n\to+\infty}{O}(n^2)$.

□

Partie C : Carré magique normal

Un carré magique *normal* d'ordre n est un carré magique d'ordre n constitué de tous les entiers positifs compris entre 1 et n^2.



Carré magique normal d'ordre 3.

17. Écrire une fonction `magique_normal` qui prend comme argument une matrice carrée `M` sous forme de liste de listes et qui renvoie `True` si `M` est un carré magique normal et `False` sinon.

Par exemple :

- si `M = [[21,7,17],[11,15,19],[13,23,9]]`, on a : `magique_normal(M) = False`,
- si `M = [[2,7,6],[9,5,1],[4,3,8]]`, on a : `magique_normal(M) = True`.

Démonstration.
Il suffit pour cette question d'écrire une fonction qui teste si tous les éléments de `M` appartiennent à l'ensemble $[\![1, n^2]\!]$ et sont distincts deux à deux. Il reste simplement ensuite à utiliser la fonction `carre_magique` de la question ***15.***

```
def magique_normal(M) :
    n = len(M)
    deja_vu = [False for k in range(n * n + 1)]
    for i in range(n) :
        for j in range(n) :
            chiffre = M[i][j]
            if 0 < chiffre < n * n + 1 and not deja_vu[chiffre] :
                deja_vu[chiffre] = True
            else :
                return False
    return carre_magique(M)
```

□

18. Que vaut la constante magique d'un carré magique normal d'ordre n ?

Démonstration.
Notons c la valeur de la constante magique d'un carré magique normal d'ordre n.

- Comme la somme sur chacune des lignes du carré est identique, alors la somme de toutes les lignes du carré magique vaut nc.
- Par ailleurs, effectuer la somme sur toutes les lignes du carré revient à sommer toutes les cases du carré. Comme ce dernier comporte tous les entiers de 1 à n^2, la somme de toutes les cases vaut :

$$\sum_{k=1}^{n^2} k = \frac{n^2\left(n^2+1\right)}{2}$$

On en déduit : $nc = \dfrac{n^2\left(n^2+1\right)}{2}$. D'où : $c = \dfrac{n\left(n^2+1\right)}{2}$.

La constante d'un carré magique normale d'ordre n est donc $\dfrac{n\left(n^2+1\right)}{2}$.

□

Partie D : Construction d'un carré magique normal d'ordre impair

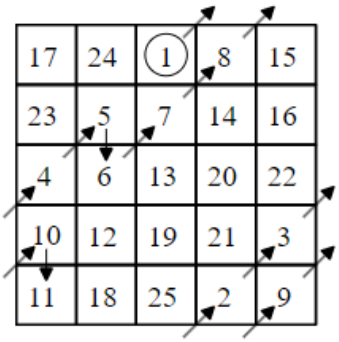
La méthode siamoise est une méthode qui permet de construire un carré magique normal d'ordre n impair. Le principe de cette méthode est le suivant :

i. Créer une matrice carrée d'ordre n, remplie de 0.

ii. Placer le nombre 1 au milieu de la ligne d'indice 0.

iii. Se décaler en diagonale d'une case vers la droite et une case vers le haut pour placer le nombre 2, puis faire de même pour le nombre 3, puis le nombre 4, ... jusqu'à n^2.
Le déplacement doit respecter les deux règles suivantes :

- × si la pointe de la flèche sort du carré, revenir de l'autre côté, comme si le carré était enroulé sur un tore (voir la figure suivante).
- × si la prochaine case contient un entier non nul, se déplacer d'une case vers le bas.



Création d'un carré magique normal d'ordre 5.

19. Écrire une fonction `matrice_nulle` qui prend en paramètre un entier `n`, et qui renvoie la matrice carrée d'ordre n dont tous les coefficients sont nuls, représentée sous forme de liste de listes.

Démonstration.

```
def matrice_nulle(n) :
    return [ [ 0 for j in range(n) ] for i in range(n) ]
```

Commentaire

On peut également proposer le script suivant qui d'utilise pas la compréhension de liste. On privilégiera tout de même celui présenté plus haut.

```
def matrice_nulle(n) :
    Ligne_M = []
    for j in range(n) :
        Ligne_M.append(0)
    M = []
    for i in range(n) :
        M.append(Ligne_M)
    return M
```

□

20. Écrire une fonction `decalage` qui prend en paramètres trois entiers `n`, `i` et `j`, et qui renvoie la prochaine position `(pi, pj)` qui succède à `(i, j)` après décalage vers la droite et le haut.

Démonstration.
On utilise les restes dans la division euclidienne par n pour gérer les conditions au bord du carré : il faut travailler avec des indices appartenant à $[\![0, n-1]\!]$. On aurait également pu procéder par disjonction de cas.

```
def decalage(n, i, j) :
    pi = (i - 1) % n
    pj = (j + 1) % n
    return (pi, pj)
```

□

21. Écrire une fonction `siamoise`, étant donné un entier `n` impair strictement positif en paramètre, renvoie un carré magique normal d'ordre n en utilisant la méthode siamoise.

Démonstration.
On teste à chaque étape si la prochaine case est libre (au départ, toutes les cases contiennent 0) et on ajuste le décalage en conséquence si besoin.

```
def siamoise(n) :
    M = matrice_nulle(n)
    i, j = 0, n // 2
    for k in range(1, n * n + 1) :
        M[i][j] = k
        pi, pj = decalage(p, i, j)
        if M[pi][pj] == 0 :
            i, j = pi, pj
        else :
            i = i + 1
    return M
```

□