DS2 /54

Exercice I : Analyse d'algorithmes /26

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.

- **1 pt**

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

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

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

- **3 pts : (I_3) est un invariant de la boucle `while`**
  × **1 pt : initialisation**
  × **2 pts : hérédité**

4. Prouver la correction de cet algorithme.

- **1 pt : 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 : $p = ab$.
- **1 pt : l'algorithme se termine (d'après la question précédente) et est partiellement correct (d'après ce qui précède). 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.

- **3 pts : 1 pt par ligne**

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

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

- **1 pt : 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.**

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

- **3 pts : $ab = p + xy$ est un invariant de boucle**
  - × **1 pt : initialisation**
  - × **2 pts : hérédité**
    - **1 pt : cas x pair**
    - **1 pt : cas y impair**
- **1 pt : À l'issue de la boucle `while` : $x = 0$. Donc : $ab = p$. Ceci assure la correction partielle de l'algorithme.**
  De plus il se terme d'après la question précédente. 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 : $T(a) = 1 + T(\lfloor \frac{a}{2} \rfloor)$.

- **1 pt : $T(0) = 0$**
- **1 pt : 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 $T(2^k)$.

- **3 pts : $\forall k \in \mathbb{N}$, $T(2^k) = k+1$**
  - × **1 pt : initialisation**
  - × **2 pts : hérédité**

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

- **1 pt : 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.**
- **1 pt : On note $k = \lfloor \log_2(a) \rfloor \in \mathbb{N}$. Alors : $2^k \leqslant a < 2^{k+1}$** $\qquad (*)$
- **1 pt : par croissance de $\big(T(a)\big)_{a\in\mathbb{N}^*}$ et d'après la question précédente :**

$$k+1 \ \leqslant\ T(a) \ \leqslant\ k+2$$

- **1 pt : par propriété de la partie entière :**

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

9. Comparer l'efficacité des deux algorithmes.

- **1 pt : justification de $T'(a) = \underset{a\to+\infty}{O}(a)$**
- **1 pt : Le 2nd algorithme `paysan` est donc plus efficace asymptotiquement que le 1er**

Exercice II : Programmation et complexité /28

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*.

	21	7	17	→ 45
	11	15	19	→ 45
	13	23	9	→ 45
45 ↙	↓ 45	↓ 45	↓ 45	↘ 45

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 : $4+5+6=15$.

- **3 pts :**
  - × **1 pt : initialisation**
  - × **2 pts : structure itérative**

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

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 : $1+4+7=12$.

- **1 pt**

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

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 : $1+5+9=15$.

- **1 pt**

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

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 : $3+5+7=15$.

- **1 pt**

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

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

- **2 pts :** $C(n) \underset{n\to+\infty}{=} O(n)$

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`.

- **1 pt : initialisation**
- **1 pt : condition sur `somme_diag2`**
- **2 pts : structure itérative**

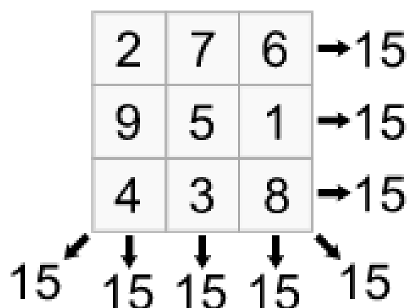
```
def carre_magique(M) :
    n = len(M)
    s = somme_diag1(M)
    if somme_diag2(M) != s :
        return False
    for i in range(n) :
        if somme_ligne(M, i) != s or somme_colonne(M, i) != s :
            return False
    return True
```

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

- **1 pt : 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.**
- **2 pts :** $C(n) \underset{n\to+\infty}{=} O(n^2)$

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`.

- **4 pts : on valorise tout morceau de script pertinent**

```
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 ?

- **2 pts : en notant c la constante magique : $c = \dfrac{n(n^2+1)}{2}$**
  - × **1 pt : Comme la somme sur chacune des lignes du carré est identique, alors la somme de toutes les lignes du carré magique vaut nc.**
  - × **1 pt : 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 :**

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

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

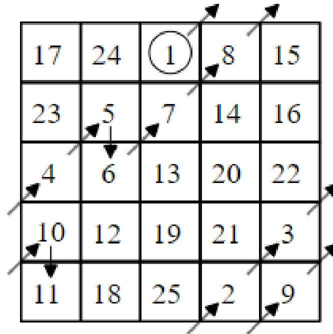
La méthode siamoise est une méthode qui permet de construire un carré magique normal d'ordre n impair. Le principe de cette méthode est le suivant :

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.

- **1 pt**

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

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.

- **2 pts**

```
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.

- **4 pts**

```
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
```