DS2 (CCINP 2023 TPC)

Gestion de Tests dans une entreprise

Une entreprise d'e-commerce vend des meubles tous identifiés par une référence et par un QR code [(1)]. Tous les clients sont identifiés par leur numéro de sécurité sociale. Tous les achats s'effectuent à l'aide d'un numéro de carte de crédit. Cette entreprise met en œuvre différents tests afin d'éviter les erreurs de numéros de sécurité sociale, de numéros de carte de crédit ou de QR codes.

Partie I - Tests de code de sécurité sociale

En France, le numéro de sécurité sociale correspond au numéro d'inscription au répertoire national d'identification des personnes physiques (RNIPP). Il est formé du numéro d'inscription (NIR) à 13 chiffres et d'une clé de contrôle à 2 chiffres. Le NIR, créé à partir de l'état civil, est composé de la façon suivante :

- Sexe (le 1er chiffre) ;
- Année de naissance (les deux chiffres suivants) ;
- Mois de naissance (les deux chiffres suivants) ;
- Lieu de naissance (les cinq chiffres ou caractères suivants - 2 chiffres [(2)] du code du département de naissance, suivis de 3 chiffre du code de commune officiel de l'Insee [(3)]) ;
- Numéro d'ordre permettant de distinguer les personnes nées au même lieu à la même période (les 3 chiffres suivants).

Les deux derniers chiffres, compris entre 01 et 97, permettent de déterminer la clé, appelée aussi « clé de contrôle », qui permettra de contrôler l'exactitude du numéro de sécurité sociale.

Pour obtenir cette clé, on détermine tout d'abord, le reste de la division par 97 du nombre formé par les 13 premiers chiffres. La clé correspond au résultat de ce nombre retranché de 97.
Exemple : soit le numéro de sécurité sociale à 13 chiffres : « 2 910175018 002 ». Le reste de la division de 2910175018002 par 97 est égal à 29. La clé est constituée du résultat : $97 - 29 = 68$. Le numéro de sécurité sociale complet est donc : « 2 910175018002 68 ».

Dans cette partie, le numéro de sécurité sociale de 13 chiffres est une chaîne de caractères composée uniquement de chiffres avec des espaces de séparation entre les différents éléments constituant ce numéro. On ne prendra pas en compte le cas de la Corse.

Ne pas oublier qu'il est toujours possible de transformer un nombre entier en une chaîne de caractères composées de chiffres (fonction `str`) et réciproquement (fonction `int`), (**annexe 3**).

1. Écrire la fonction `num_secu` qui, à partir de la chaîne de caractères d'un numéro de sécurité sociale, donne le numéro sous la forme d'un entier. Le programme devra parcourir la chaîne de caractères représentant le numéro de sécurité sociale en supprimant les caractères d'espacement, puis la transformer en un nombre entier. Cette fonction a un paramètre de type `string` et retourne une valeur de type `int`.
Exemple :

```
>>> num_secu('2 91 01 75 018 002')
2910175018002
```

(1). Un QR code (Quick Response code) désigne un type de code-barres en deux dimensions, lequel se compose de modules noirs disposés dans un carré à fond blanc (voir figure 1).
(2). Pour simplifier le problème, nous supposons que les deux départements corses 2A et 2B sont représentés par le code 20 comme avant 1976.
(3). Institut national de la statistique et des études économiques.

Démonstration.
On propose la fonction suivante.

```
1  def num_secu(code) :
2      ''num_secu(code : str) -> int''
3      code_se = ''
4      for c in code :
5          if c != ' ' :
6              code_se = code_se + c
7      return int(code_se)
```

Détaillons les éléments de cette fonction.

- **Début de la fonction**
  On commence par préciser la structure de la fonction :
  - × cette fonction se nomme `num_secu`,
  - × elle prend en paramètre d'entrée la chaîne de caractères `code`,
  - × elle renvoie l'entier `int(code_se)`.

```
1  def num_secu(code) :
```

```
7      return int(code_se)
```

  La variable `code_se` qui contiendra le numéro de sécurité sociale **sans caractères d'espacement**, toujours sous forme de chaîne de caractères, est initialisée à la ligne 3 : la chaîne de caractères vide.

```
3      code_se = ''
```

- **Structure itérative**
  Les lignes 4 à 6 consistent à supprimer les espaces de la chaîne de caractères `code`.
  Pour cela, on parcourt chaque caractère de `code` grâce à une structure itérative (boucle `for`) et, si le caractère rencontré n'est pas un caractère d'espacement, on le concatène à la chaîne `code_se`.

```
4      for c in code :
5          if c != ' ' :
6              code_se = code_se + c
```

- **Fin de la fonction**
  À l'issue de cette boucle, la variable `code_se` contient le numéro de sécurité sociale **sans caractères d'espacement**, toujours sous forme de chaîne de caractères. Il reste donc à le convertir en un entier (type `int` grâce à la fonction `int` (détaillée en **annexe 3**).

```
7      return int(code_se)
```

Commentaire

Afin de permettre une bonne compréhension des mécanismes en jeu, on a détaillé la réponse à cette question. Cependant, écrire correctement la fonction **Python** démontre la bonne compréhension et permet d'obtenir la totalité des points alloués à cette question. Il en est de même dans les questions suivantes.

□

2. Écrire la fonction `clef` qui détermine la valeur de la clé d'un numéro de sécurité sociale. Cette fonction a un paramètre de type `int` et retourne un élément de type `int`.
Exemple :
```
>>> clef(2910175018002)
68
```

Démonstration.
On propose la fonction suivante.

```
1  def clef(num) :
2      ''clef(num : int) -> int''
3      reste = num % 97
4      return 97 - reste
```

En effet, d'après l'énoncé, pour obtenir la clef du numéro de sécurité sociale `num` :

1) on détermine le reste de la division euclidienne de ce numéro par 97. C'est ce calcul qui est effectué en ligne 3.

```
3      reste = num % 97
```

2) on retranche à 97 l'entier obtenu. C'est le calcul effectué en ligne 4.

```
4      return 97 - reste
```

Commentaire

On rappelle que, comme précisé en **annexe 3**, la commande `a % b` permet d'obtenir le reste dans la division euclidienne de `a` par `b`. Ainsi, par exemple, l'appel `16 % 3` renvoie `1`, car :

$$\begin{cases} 16 = 3 \times 5 + 1 \\ 0 \leqslant 1 < 3 \end{cases}$$

□

3. Écrire la fonction `num_secu_complet` qui détermine le numéro complet de sécurité sociale. Cette fonction a un paramètre de type `int` et retourne un élément de type `int`.
Exemple :
```
>>> num_secu_complet(2910175018002)
291017501800268
```

Démonstration.
On propose la fonction suivante.

```
1  def num_secu_complet(num) :
2      ''num_secu_complet(num : int) -> int''
3      return 100 * num + clef(num)
```

Commentaire

Attention à ne pas faire de confusions entre opérateurs.

- L'opérateur + entre deux entiers (variables de type `int`) est l'opérateur usuel de somme entre deux entiers naturels. Ainsi :

```
In [1] :   2910175018002 + 68
Out[1] :   2910175018070
```

Cet appel ne permet donc pas d'obtenir le numéro de sécurité sociale complet. Pour concaténer la clé 68 à la fin du numéro 2910175018002, il faut donc commencer par multiplier ce dernier par 100 avant de lui ajouter la clé. C'est ce qui est fait en ligne 3.

```
3        return 100 * num + clef(num)
```

- L'opérateur + entre deux chaînes de caractères (variables de type `str`) est l'opérateur de concaténation entre deux chaînes de caractères. Ainsi :

```
In [2] :   '2910175018002' + '68'
Out[2] :   '291017501800268'
```

□

4. Écrire la fonction `test_num_secu` qui détermine si un numéro de sécurité sociale est correct. Cette fonction a un paramètre de type `string` et retourne un élément de type `bool`.
Exemple :

```
>>> test_num_secu('2 910175018002 68')
True
>>> test_num_secu('2 910175018002 93')
False
```

Démonstration.
On propose la fonction suivante.

```
1  def test_num_secu(code) :
2      ''test_num_secu(code : str) -> bool''
3      code_int = num_secu(code)
4      code_sans_cle = code[:-2]
5      num = num_secu(code_sans_cle)
6      num = num_secu_complet(num)
7      return num == code_int
```

Détaillons le contenu de cette fonction.

- En ligne 3, on commence par stocker dans la variable `code_int` le numéro de sécurité sociale `code` sous forme d'entier (type `int`) grâce à la fonction `num_secu` de la question 1.

```
3        code_int = num_secu(code)
```

- En ligne 4, on stocke ensuite dans la variable `code_sans_cle`, le numéro de sécurité sociale sans la clé, sans changer son type (comme la variable `code`, la variable `code_sans_cle` est de type `str`).

```
4        code_sans_cle = code[:-2]
```

- On convertit ensuite en ligne 5 la variable `code_sans_cle` en un objet de type `int`, toujours avec la fonction `num_secu` de la question 1.

```
5        num = num_secu(code_sans_cle)
```

- Cela permet, en ligne 6, de déterminer le numéro de sécurité sociale correct complet défini à partir de la variable `num`. Pour cela, on utilise la fonction `num_secu_complet` de la question 3.

```
6        num = num_secu_complet(num)
```

- Ainsi, pour vérifier que le numéro de sécurité sociale `code` est correct, il suffit de tester si les entiers `code_int` et `num` sont égaux. On renvoie donc le booléen `num == code_int`.

```
7        return num == code_int
```

□

Partie II - Test de numéro de carte de crédit

Pour savoir si un numéro de carte de crédit est valide, on utilise très souvent l'algorithme de Luhn [(4)]. Comme pour le numéro de sécurité sociale, il y a une clé appelée somme de contrôle (checksum en anglais) qui fait partie du numéro d'une carte de crédit. Ce numéro est un entier composé de 16 chiffres. Le dernier chiffre est la clé qui permet de contrôler l'exactitude du numéro.

Le principe de l'algorithme de Luhn est le suivant. On commence toujours par le chiffre se trouvant le plus à droite. Ce chiffre sera le premier élément de la liste dites des « indices impairs ». Puis on complète cette liste en prenant un chiffre sur deux du numéro de carte bancaire, toujours en le lisant de la droite vers la gauche.
Pour la liste des chiffres « d'indices pairs », on commence par le deuxième chiffre le plus à droite du numéro de la carte de crédit, on se déplace de la droite vers la gauche comme pour la liste précédente et on construit la liste, en prenant un chiffre sur deux. Pour les nombres de cette liste des indices pairs, on double tous les chiffres. Si un nombre est supérieur à 9, on réalise la somme des deux chiffres qui le composent (exemple si on obtient 16, on additionne 1 et 6 pour avoir 7). Par conséquent, tous les nombres des deux listes sont composés uniquement de chiffres compris entre 0 et 9. On calcule alors la somme totale des chiffres de ces deux listes. Si cette somme est un multiple de 10, alors le numéro de la carte de crédit est valide.

Exemple : soit 4762 un nombre (on se limite à 4 chiffres mais le raisonnement est identique pour un nombre à 16 chiffres). Appliquons-lui la formule de Luhn. On commence par le chiffre 2, celui se trouvant le plus à droite. Le nombre 4762 se transforme en deux listes correspondant aux indices impairs et pairs soit $[2, 7]$ et $[6, 4]$. Puis en deux autres listes, la liste des indices impairs inchangés $[2, 7]$ et la liste des indices pairs doublés $[12, 8]$. La somme des éléments de ces deux listes est égale à 20 car la liste des indices pairs $[12, 8]$ se réduit en la liste $[3, 8]$ du fait de la sommation des chiffres constituant le nombre 12. La somme des éléments des deux listes obtenues $[2, 7]$ et $[3, 8]$ est bien égale à 20. Ce résultat est un multiple de 10, le nombre 4762 est donc correct au sens de l'algorithme de Luhn.

On aurait pu raisonner sur une seule liste et obtenir le même résultat. 4762 se transforme en la liste $[8, 7, 12, 2]$ puis en la liste $[8, 7, 3, 2]$ après réduction. La somme des chiffres $8 + 7 + 3 + 2$ est égale à 20.

(4). L'algorithme de Luhn, ou code de Luhn, ou encore formule de Luhn est aussi connu comme l'algorithme « modulo 10 ».

5. Écrire une fonction `num_en_liste` qui transforme un nombre entier en une liste de chiffres. Cette fonction a un paramètre de type `int` et retourne un élément de type `list`.
Exemple :

```
>>> num_en_liste(4532015112830465)
[4, 5, 3, 2, 0, 1, 5, 1, 1, 2, 8, 3, 0, 4, 6, 5]
```

Démonstration.
On propose la fonction suivante.

```
1  def num_en_liste(num) :
2      ''num_en_liste(num : int) -> list''
3      chaine = str(num)
4      return [int(c) for c in chaine]
```

Détaillons le contenu de cette fonction.

- On commence par convertir le nombre entier `num` en chaîne de caractères à l'aide de la fonction `str` rappelée en **annexe 3**. On stocke le résultat obtenu dans la variable `chaine`.

```
3      chaine = str(num)
```

- On renvoie alors la liste des caractères de la chaîne `chaine` en les ayant au préalable convertis en entier (grâce à la fonction `int`). Pour cela, on définit cette liste par compréhension.

```
4      return [int(c) for c in chaine]
```

Commentaire

On pouvait également proposer la fonction suivante qui exploite l'écriture décimale de l'entier `num`.

```
1  def num_en_liste(num) :
2      n = num
3      L = []
4      while n != 0 :
5          r = n % 10
6          L = [r] + L
7          n = (n - r) // 10
8      return L
```

□

6. Écrire une fonction `tuple_pairs_impairs` qui détermine un tuple représentant la liste des chiffres d'indice pair et la liste des chiffres d'indice impair d'un numéro de carte de crédit. Le chiffre le plus à droite de ce numéro est considéré comme le premier chiffre d'indice impair. Cette fonction a un paramètre de type `int` et retourne un tuple composé de deux éléments de type `list`.
Exemple :

```
>>> tuple_pairs_impairs(4532015112830465)
([6, 0, 8, 1, 5, 0, 3, 4], [5, 4, 3, 2, 1, 1, 2, 5])
```

Démonstration.
On propose la fonction suivante.

```
1  def tuple_pairs_impairs(num) :
2      ''tuple_pairs_impairs(num : int) -> (L1 : list, L2 : list)''
3      L = num_en_liste(num)
4      n = len(L)
5      L1 = []
6      L2 = []
7      for k in range(n) :
8          if (n-1 - k) % 2 == 0 :
9              L1.append( L[n-1-k] )
10         else :
11             L2.append( L[n-1-k] )
12     return (L1, L2)
```

Détaillons les éléments de cette fonction.

- On commence par convertir le numéro de carte de crédit `num` sous forme de liste à l'aide de la fonction `num_en_liste` définie en question précédente, et on stocke dans la variable `n` la longueur de la liste `L` obtenue.

```
3      L = num_en_liste(num)
4      n = len(L)
```

- On initialise ensuite :
  - × la liste `L1` qui contiendra la liste des chiffres d'indice pair (en commençant par la droite),
  - × la liste `L2` qui contiendra la liste des chiffres d'indice impair (en commençant par la droite).

```
5      L1 = []
6      L2 = []
```

- On met ensuite à jour les listes `L1` et `L2` à l'aide d'une structure itérative (boucle `for`).
  Pour tout $ttk \in [\![0, n-1]\!]$, on considère l'indice `n-1-k` (on rappelle qu'on doit parcourir les éléments de la liste `L` de la droite vers la gauche).
  - × si cet indice est pair, alors on concatène l'élément d'indice `n-1-k` de la liste `L` à la liste `L1`.

```
8          if (n-1 - k) % 2 == 0 :
9              L1.append( L[n-1-k] )
```

  - × si cet indice est impair, alors on concatène l'élément d'indice `n-1-k` de la liste `L` à la liste `L2`.

```
10         else :
11             L2.append( L[n-1-k] )
```

- On renvoie enfin le tuple contenant les listes `L1` et `L2`.

```
12     return (L1, L2)
```

Commentaire

On pouvait également définir la liste des chiffres d'indice pair et la liste des chiffres d'indice impair par compréhension. On obtient alors le script suivant.

```
def tuple_pairs_impairs_2(num) :
    L = num_en_liste(num)
    n = len(L)
    L1 = [L[n-1-k] for k in range(n) if (n-1-k) % 2 == 0]
    L2 = [L[n-1-k] for k in range(n) if (n-1-k) % 2 == 1]
    returnn (L1, L2)
```

□

7. Écrire une fonction `cree_dico` qui, à partir d'un numéro de carte de crédit, crée un dictionnaire avec deux clés nommées `'pair'` et `'impair'`. La clé `'pair'` est constituée de la liste des nombres d'indice pairs du numéro de la carte de crédit et la clé `'impair'` de la liste des nombres d'indice impairs.
Exemple :

```
>>> cree_dico(4532015112830465)
{'pair' : [6, 0, 8, 1, 5, 0, 3, 4], 'impair' : [5, 4, 3, 2, 1, 1, 2, 5]}
```

Démonstration.

```
def cree_dico(num) :
    ''cree_dico(num : int) -> dict''
    T = tuple_pairs_impairs(num)
    return {'pair' : T[0], 'impair' : T[1]}
```

□

8. Écrire une fonction `traitement_nb_pairs` qui multiplie par 2 tous les chiffres de la liste associée à la clé `'pair'`. Si un chiffre est supérieur à 9 , il faut réaliser la somme des deux chiffres qui le composent. Cette fonction a un paramètre de type dictionnaire et retourne un dictionnaire.
Remarque : la partie correspondant à la clé `'impair'` n'est pas modifiée par le traitement de cette fonction.
Exemple :

```
>>> un_dico = cree_dico(4532015112830465)
>>> traitement_nb_pairs(un_dico)
{'pair' : [3, 0, 7, 2, 1, 0, 6, 8], 'impair' : [5, 4, 3, 2, 1, 1, 2, 5]}
```

Démonstration.
On propose la fonction suivante.

```
def traitement_nb_pairs(dico) :
    ''traitement_nb_pairs(dico : dict) -> dico : dict''
    L = dico['pair']
    n = len(L)
    for i in range(n) :
        nb = 2 * L[i]
        q = nb // 10
        r = nb % 10
        L[i] = q + r
    dico['pair'] = L
    return dico
```

Détaillons les éléments de cette fonction.

- On commence par stocker dans une variable `L` la liste associée à la clé `'pair'` dans le dictionnaire `dico`. On stocke ensuite sa longueur dans une variable `n`.

```
3       L = dico['pair']
4       n = len(L)
```

- On parcourt alors les indices des éléments de la liste `L` à l'aide d'une structure itérative (boucle `for`). Au $\texttt{i}^{\text{ème}}$ tour de boucle :
  - × on commence par multiplier l'élément d'indice `i` de `L` par 2, et on stocke le résultat dans la variable `nb`.

```
6               nb = 2 * L[i]
```

  - × on stocke ensuite dans une variable `q` le quotient de la division euclidienne de `nb` par 10. La variable `q` contient donc le chiffre des dizaines de l'écriture de `nb` en base décimale.

```
7               q = nb // 10
```

  - × on stocke ensuite dans une variable `r` le reste de la division euclidienne de `nb` par 10. La variable `r` contient donc le chiffre des unités de l'écriture de `nb` en base décimale.

```
8               r = nb % 10
```

  - × enfin, on met à jour le $\texttt{i}^{\text{ème}}$ élément de `L` pour qu'il contienne la valeur de `q + r`, c'est-à-dire la somme des chiffres composant le nombre `nb`.

```
9               L[i] = q + r
```

  Remarquons que si le nombre `nb` est inférieur ou égal à 9, alors son quotient dans sa division euclidienne par 10 est 0, et son reste est `nb`. Dans ce cas, le nombre `nb` n'est donc pas modifié, comme spécifié dans l'énoncé.
- Enfin, on met à jour la valeur associée à la clé `'pair'` dans le dictionnaire `dico` pour qu'elle contienne la liste `L` mise à jour.

```
10      dico['pair'] = L
```

□

9. Écrire une fonction `test_num_carte_credit` qui utilise l'algorithme de Luhn pour savoir si un numéro de carte de crédit est correct. Vous devez utiliser la fonction `traitement_nb_pair` pour sa réalisation. Cette fonction a un paramètre de type `int` et retourne une valeur de type `bool`.
Exemple :

```
>>> test_num_carte_credit(4532015112830465)
True
```

Démonstration.
On propose la fonction suivante.

```
1  def test_num_carte_credit(num) :
2      ''test_num_carte_credit(num : int) -> bool''
3      dico = cree_dico(num)
4      dico = traitement_nb_pairs(dico)
5      S = sum(dico['pair']) + sum(dico['impair'])
6      return S % 10 == 0
```

Détaillons les éléments de cette fonction. On suit l'algorithme de Luhn.

- On commence par créer le dictionnaire contenant la liste des chiffres d'indice pair (en partant de la droite) du numéro de carte de crédit (associée à la clé `'pair'`), et la liste des chiffres d'impair (en partant de la droite) du numéro de carte de crédit (associée à la clé `'impair'`). Pour cela, on utilise la fonction `cree_dico` de la question **??**.

```
3        dico = cree_dico(num)
```

- On modifie ensuite la liste des chiffres d'indice pair conformément à l'algorithme de Luhn. Pour cela, on utilise la fonction `traitement_nb_pairs` de la question **??**.

```
4        dico = traitement_nb_pairs(dico)
```

- On somme ensuite tous les chiffres des deux listes obtenues et on stocke le résultat dans une variable `S`.

```
5        S = sum(dico['pair']) + sum(dico['impair'])
```

- Enfin, pour vérifier que le numéro de carte de crédit est correct, il suffit de tester si l'entier `S` est un multiple de 10. Autrement dit, on doit renvoyer :
  - × le booléen `True` si le reste de la division euclidienne de `S` par 10 est nul,
  - × le booléen `False` sinon.

  Il s'agit donc de renvoyer le booléen `S % 10 == 0`.

```
6        return S % 10 == 0
```

□

Partie III - Tests de QR code

Les QR codes ont été inventés en 1994, par Masahiro Hara, un ingénieur de l'entreprise japonaise Denso-Wave. Cette invention a permis d'assurer le référencement des pièces détachées dans les usines Toyota. Les QR codes sont constitués essentiellement de pixels noirs et blancs codés dans le format RGB (**annexe 1**). Cependant, il existe des QR codes bicolores mais avec un jeu de couleurs très contrastées. Les QR codes peuvent être partiellement raturés ou déchirés car un de leurs avantages est qu'ils peuvent accepter un certain taux d'erreurs, entre 7% et 30% suivant la version du QR code. Il existe 40 versions qui peuvent stocker entre 10 et 7089 caractères numériques. Nous nous restreignons ici à la version 1 qui utilise une matrice de $21 * 21$ pixels pour sa représentation.

En fait sur la **figure 1**, l'image du QR code correspond à une matrice de 420*420 pixels (**programme P1**), alors que la matrice initiale d'un QR code de version 1 ne compte que 21*21 pixels. Pour qu'un QR code soit plus visible, on a créé la notion de module qui correspond à un bloc de pixels identiques pour représenter un pixel du QR code initial. C'est un mécanisme de zoom pour que le QR code soit visible. Un module a une taille de 20*20 pixels. Chaque module représente globalement une valeur binaire : 1 pour le blanc et 0 pour le noir. Attention : ne pas confondre la taille d'un QR code (ici, 420*420) avec la taille d'un module (ici, 20*20) [la valeur 420 correspond à 21*20]. Enfin, un QR code est constitué de différents éléments : des motifs de positionnement (3 blocs de 7×7 pixels), des motifs de synchronisation (6 zones blanches de séparation et 11 pixels de couleur blanc et noir), des motifs de format d'information et une zone comprenant les données utilisateurs avec des motifs de correction (**figure 2**).

Dans la suite de cette partie, nous n'utiliserons que les QR codes de version 1.

Fig. 1 QR code version 1

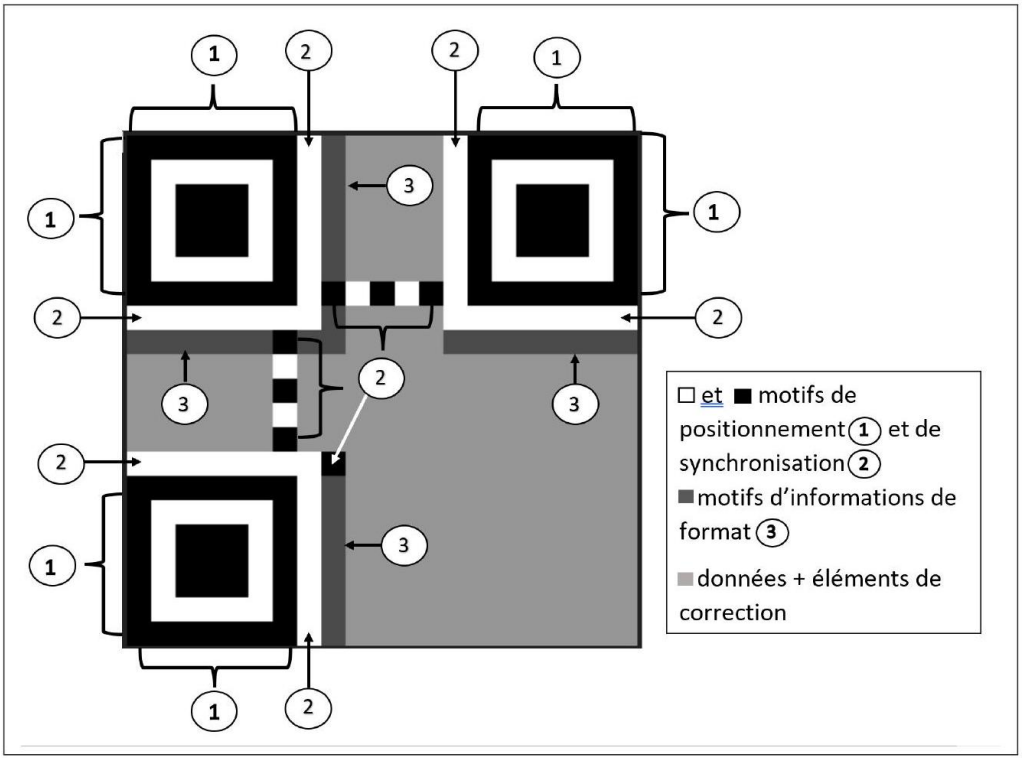

Fig. 2 Organisation d'un QR code

10. Écrire une fonction `init` qui réalise l'initialisation d'une liste de dimension n où chaque élément est également une liste de dimension n. Cette liste de listes représente ainsi une matrice de taille $n \times n$. Cette fonction a un paramètre de type `int` et retourne une liste de listes qui représente un QR code initialisé avec des valeurs 0.
Exemple :

```
>>> init(4)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Démonstration.

```python
def init(n) :
    ''init(n : int) -> list(list)''
    return [ [0 for _ in range(n)] for _ in range(n)]
```

□

On donne le programme **P1** suivant : (**annexe 1** pour la description du module *Gestion_ QRCode*).

P1

```
from Gestion_QRCode import *

img = open('./Image/ccinp.png').....# Lecture de l'image
img.show()..............................# Affichage de l'image (figure 1)
largeur, hauteur = img.size.............# La taille de l'image (largeur, hauteur)
position = (largeur,hauteur) ...........# Résultat : (420, 420)
```

11. Écrire la fonction `charge_valeur` qui a pour but de réduire les données de l'image dans une liste de listes de dimension $21 * 21$. Attention : l'image correspondant à un QR code représente une liste de listes de dimension 420*420 dont on veut réduire tous les blocs constitués de 20*20 pixels à un seul pixel pour avoir à partir de l'image une liste de listes de dimension 21*21. Ne pas oublier que tous les pixels d'un bloc sont identiques. Cette fonction a un paramètre de type image et retourne une liste de listes de triplets (couleur des pixels).
Indication : utiliser la fonction `getpixel` du module Python `Gestion_QRCode` (voir sa définition dans l'**annexe 1**).

Démonstration.
On propose la fonction suivante.

```
def charge_valeur(image) :
    ''charge_valeur(image : image) -> reduc : list(list)''
    reduc = init(21)
    for i in range(21) :
        for j in range(21) :
            reduc[i][j] = image.getpixel(20 * i, 20 * j)
    return reduc
```

Détaillons les éléments de cette fonction.

- On commence par initialiser la liste de listes `reduc` qui contiendra un exemplaire de chaque bloc de 20*20 pixels. Pour cela, on utilise la fonction `init` définie en question précédente qui permet de créer une liste de 21 listes à 21 éléments.

```
    reduc = init(21)
```

- On remplit ensuite la liste de listes `reduc` à l'aide d'une structure itérative (boucle `for`).
Comme un bloc de taille $20 * 20$ comporte 400 fois le même pixel, il suffit de stocker la valeur du premier pixel en haut à gauche de ce bloc. Pour le bloc de coordonnées $(\texttt{i}, \texttt{j})$, ce premier pixel en haut à gauche a pour coordonnées $(20 * \texttt{i}, 20 * \texttt{j})$. C'est ce pixel que l'on stocke dans le $\texttt{j}^{\text{ème}}$ élément de la $\texttt{i}^{\text{ème}}$ liste de `reduc`.
Pour cela, on utilise la méthode `getpixel` détaillée dans l'**annexe 1**.

```
    for i in range(21) :
        for j in range(21) :
            reduc[i][j] = image.getpixel(20 * i, 20 * j)
```

- On renvoie enfin la liste de listes `reduc` obtenue.

```
    return reduc
```

□

On prend comme bloc de positionnement celui représenté dans la **figure 3**.

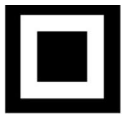

Fig. 3 Bloc de positionnement

12. Écrire une fonction `cree_bloc` qui crée un bloc de positionnement. Cette fonction, qui n'a pas de paramètre, retourne une liste de listes de dimension 7*7.
Exemple :

```
>>> cree_bloc()
[[0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 1, 1, 1, 0],
[0, 1, 0, 0, 0, 1, 0],
[0, 1, 0, 0, 0, 1, 0],
[0, 1, 0, 0, 0, 1, 0],
[0, 1, 1, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 0]]
```

Démonstration.

```
def cree_bloc() :
    ''cree_bloc() -> list(list)''
    A = [0 for _ in range(7)]
    B = [0] + [1 for _ in range(5)] + [0]
    C = [0, 1, 0, 0, 0, 1, 0]
    return [A[:], B[:], C[:], C[:], C[:], B[:], A[:]]
```

Commentaire

Rien dans l'énoncé n'interdit de coder l'ensemble du bloc de positionnement de façon explicite (même si cela est un peu lourd).

```
def cree_bloc() :
    bloc = [[0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 1, 1, 0],
        [0, 1, 0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 1, 0],
        [0, 1, 1, 1, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 0]]
    return bloc
```

□

13. Écrire une fonction `test_bloc` qui teste si un bloc de positionnement (rappel : il y en a trois) est bien représenté pixel par pixel dans un QR code. Cette fonction a 3 paramètres : les coordonnées x et y donnant la position du début d'un bloc de positionnement d'un QR code (toujours les coordonnées du pixel le plus haut et à gauche) et la liste de listes de dimension 21*21 associée au même QR code. Cette fonction retourne un booléen.
Remarque : on cherche à tester si un bloc de positionnement d'un QR code n'a pas subi une modification. Les coordonnées du pixel le plus haut et à gauche pour le premier bloc sont égales à $(0, 0)$, pour le second bloc à $(0, 14)$ et pour le troisième bloc à $(14, 0)$.

Exemples :
```
>>> test_bloc (0, 0, mat1)
True
>>> test_bloc(1, 3, mat1)
False
```

Démonstration.
On propose la fonction suivante.

```
1  def test_bloc(x, y, mat) :
2      ''test_bloc(x : int, y : int, mat : list(list)) -> bool''
3      bloc_pos = cree_bloc()
4      for i in range(7) :
5          for j in range(7) :
6              if mat[x + i][y + j] != bloc_pos[i][j] :
7                  return False
8      return True
```

Détaillons les éléments de cette fonction.

- On commence par stocker dans une variable `bloc_pos` un bloc de positionnement sous forme de liste de listes. Pour cela, on utilise la fonction `cree_bloc` de la question précédente.

```
3      bloc_pos = cree_bloc()
```

- On compare ensuite chaque coefficient du bloc 7*7 de la variable `mat` en partant de l'élément de coordonnée (x, y) (c'est l'élément d'indice `y` de la liste d'indice `x` de `mat`) avec l'élément correspondant dans le bloc de position. Pour cela, on utilise une structure itérative (boucle `for`).
  Si l'élément de `mat` et celui de `bloc_pos` correspondant diffèrent, alors le bloc de `mat` parcouru n'est pas un bloc de positionnement. On renvoie donc `False`.

```
4      for i in range(7) :
5          for j in range(7) :
6              if mat[x + i][y + j] != bloc_pos[i][j] :
7                  return False
```

- Si toutes les comparaisons effectuées dans la boucle sont fausses (autrement dit si tous les coefficients des 2 blocs sont égaux 2 à 2), alors le bloc parcouru est bien un bloc de positionnement. On renvoie donc `True`.

```
8      return True
```

□

14. On considère qu'un QR code est bien positionné lorsque ses 3 blocs de contrôle sont effectivement présents en haut à gauche, en haut à droite et en bas à gauche (comme sur la **figure 1**). Écrire une fonction `test_QRcode` qui permet de tester si un QR code est bien positionné. Cette fonction a pour paramètre une matrice de dimension 21*21 et retourne un booléen.
Exemple :
```
>>> test_QRcode(mat1)
True
```

Démonstration.
Pour qu'un QR code soit bien positionné, il faut qu'il présente un bloc de positionnement :

- en position $(0, 0)$ (en haut à gauche),
- en position $(0, 14)$ (en haut à droite),
- en position $(14, 0)$ (en bas à gauche).

Pour effectuer ces trois tests, on utilise la fonction `test_bloc` définie en question précédente et la conjonction `and`.

```
def test_QRcode(mat) :
    ''test_QRcode(mat : list(list)) -> bool''
    return test_bloc(0,0,mat) and test_bloc(0,14,mat) and test_bloc(14,0,mat)
```

Lors de la lecture d'un QR code par un appareil dédié (scanner, caméra ou autre) le processus de lecture permet de placer un QR code dans l'une des quatre positions possibles, comme illustré dans la **figure 4**. Cela dépend bien évidemment de l'orientation du QR code lors de sa lecture.

On se propose de faire tourner un QR Code par rotation successive de 90° afin qu'il puisse se trouver dans la bonne position comme celui de la **figure 1**.

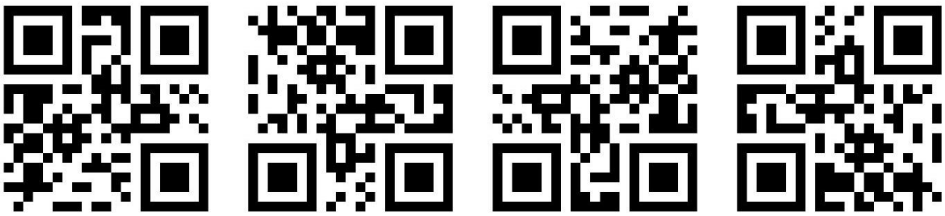

Fig. 4 Les 4 positions possibles lors de la lecture d'un QR code

15. Écrire une procédure [(5)] `tourHoraire` qui réalise une rotation de 90°, dans le sens des aiguilles d'une montre, des 4 éléments du QR code. La fonction a trois paramètres, les coordonnées x et y d'un élément de la liste de listes et une liste de listes de dimension 21*21.
Exemple :
`>>> tourHoraire(0, 1, mat1)`

Démonstration.
On effectue la rotation présentée en figure 5 sur les tables 1 et 2, entre les élements de coordonnées (x, y), $(y, n - x)$, $(n - x, n - y)$ et $(n - y, x)$. On effectue ces modifications directement sur la liste de listes en paramètre d'entrée puisqu'on souhaite que la fonction `tourHoraire` renvoie `None`.

```
def tourHoraire(x, y, mat) :
    ''tourHoraire(x : int, y : int, mat : list(list)) -> None''
    n = len(mat) - 1
    pix = mat[x][y]
    mat[x][y] = mat[n-y][x]
    mat[n-y][x] = mat[n-x][n-y]
    mat[n-x][n-y] = mat[y][n-x]
    mat[y][n-x] = pix
    return None
```

(5). Une procédure est une fonction qui retourne la valeur None mais cette valeur n'est pas destinée à être utilisée ou à être capturée.

Commentaire

- Il est indispensable de stocker l'une des 4 valeurs à permuter dans une variable auxiliaire. Dans le cas contraire, l'une des quatre valeurs à permuter sera écrasée par une autre. Détaillons cela en considérant la fonction suivante.

```
1  def tourHoraire(x, y, mat) :
2      ''tourHoraire(x : int, y : int, mat : list(list)) -> None''
3      n = len(mat) - 1
4      mat[x][y] = mat[n-y][x]
5      mat[n-y][x] = mat[n-x][n-y]
6      mat[n-x][n-y] = mat[y][n-x]
7      mat[y][n-x] = mat[x][y]
8      return None
```

Illustrons la transformation effectuée par cette fonction sur la table 1 de la figure 5 et les coordonnées `x` = 0 et `y` = 1. On souhaite donc permuter les lettres **b**, **h**, **i** et **o**.

a	**b**	c	d
e	f	g	**h**
i	j	k	l
m	n	**o**	p

× À l'issue de l'exécution de la ligne 4, on stocke dans l'élément de coordonnées $(0, 1)$ la dernière valeur en date contenue dans l'élément de coordonnées $(3 - 1, 0) = (2, 0)$, c'est-à-dire **i**. Ainsi, à ce stade, la variable `mat` contient :

a	**i**	c	d
e	f	g	**h**
i	j	k	l
m	n	**o**	p

× À l'issue de l'exécution de la ligne 5, on stocke dans l'élément de coordonnées $(2, 0)$ la dernière valeur en date contenue dans l'élément de coordonnées $(3 - 0, 3 - 1) = (3, 2)$, c'est-à-dire **o**. Ainsi, à ce stade, la variable `mat` contient :

a	**i**	c	d
e	f	g	**h**
o	j	k	l
m	n	**o**	p

× À l'issue de l'exécution de la ligne 5, on stocke dans l'élément de coordonnées $(3, 2)$ la dernière valeur en date contenue dans l'élément de coordonnées $(1, 3 - 0) = (1, 3)$, c'est-à-dire **h**. Ainsi, à ce stade, la variable `mat` contient :

a	**i**	c	d
e	f	g	**h**
o	j	k	l
m	n	**h**	p

Commentaire

× À l'issue de l'exécution de la ligne 5, on stocke dans l'élément de coordonnées $(1,3)$ la dernière valeur en date contenue dans l'élément de coordonnées $(0,1) = (0,1)$, c'est-à-dire **i**. Ainsi, à ce stade, la variable `mat` contient :

a	**i**	c	d
e	f	g	**i**
o	j	k	l
m	n	**h**	p

Ne pas avoir stocké le contenu initial de `mat[x][y]` a empêché une mise à jour correcte de `mat[y][n-x]`.

- Notons également que l'ordre des affectations est importante pour les mêmes raisons. □

On se limite à un exemple d'une liste de listes de dimension 4*4 pour expliquer le fonctionnement, mais ce serait la même chose pour une liste de listes de dimension 21*21. Si on prend les 4 éléments (b, h, o, i) de la liste de listes **table 1** de la **figure 5**, b doit se trouver, après une rotation de 90°, à la place de l'élément h, l'élément h à la place de l'élément o, l'élément o à la place de l'élément i et l'élément i à la place de l'élément b. Le résultat de la transformation est illustré dans la **table 2** de la **figure 5**. Le mécanisme doit s'exécuter de la même manière sur les autres éléments de la **table 2**. Le résultat de la transformation finale est illustré dans la **table 3** de la **figure 5**.

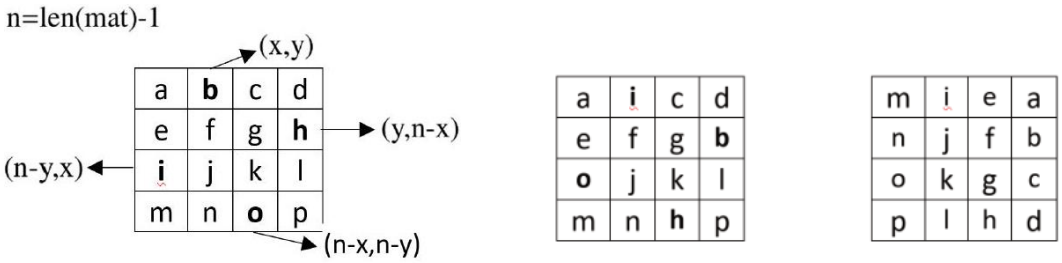

Table 1 **Table 2** **Table 3**

Fig. 5 Simple rotation

16. Écrire la procédure `rotationHoraire` qui réalise la rotation de 90° d'un QR code. Cette procédure a un seul paramètre, une liste de listes de dimension 21*21.
Par exemple, dans la **figure 4** cette fonction réalisera la première rotation de 90° du QR code.

Démonstration.
On applique une rotation de 90° aux éléments de la liste de listes en paramètre. Pour cela, on utilise la fonction `tourHoraire` définie en question précédente.

```
1  def rotationHoraire(mat) :
2      ''rotationHoraire(mat : list(list)) -> None''
3      n = len(mat)
4      for i in range(n//2) :
5          for j in range(i, n-i-1) :
6              tourHoraire(i, j, mat)
7      return None
```

Commentaire

On prendra garde à ne pas appliquer la fonction `tourHoraire` à tous les éléments de la variable `mat`. En effet, si c'était le cas, on effectuerait en fait 4 rotations de 90° sur chaque élément. La fonction `rotationHoraire` serait alors la fonction... identité. □

17. Connaissant les 4 positions possibles lors de la lecture d'un QR code par un appareil dédié, écrire la procédure `QRcode_posi` qui positionne correctement un QR code. Cette procédure a un seul paramètre, une liste de listes de dimension 21*21.
Indication : utiliser les fonctions `rotationHoraire` et `test_QRcode`.

Démonstration.

```
def QRcode_posi(mat) :
    '' QRcode_posi(mat : list(list)) -> None''
    for i in range(3) :
        if test_QRcode(mat) :
            return None
        else :
            rotationHoraire(mat)
```

□

Partie IV - Gestion réseau

Cette entreprise possède de nombreux magasins dans le monde entier. Des serveurs ont été placés dans tous les pays et sont nommés par des lettres.

Les différentes informations envoyées dans le réseau circulent de serveur en serveur. Les serveurs sont représentés par des nœuds, **figure 6**, et plusieurs routes sont possibles entre chacun d'eux. Le poids associé aux arêtes correspond à la valeur du temps de transmission entre deux nœuds du graphe multiplié par un facteur correctif. L'entreprise souhaite optimiser les temps de transmission entre deux nœuds du réseau en utilisant l'algorithme de Dijkstra.

L'algorithme de Dijkstra permet de déterminer les plus courts chemins à partir d'un sommet unique $s \in S$ vers les autres sommets d'un graphe pondéré orienté ou non $G = (S, A)$, avec S un ensemble de sommets et A un ensemble d'arêtes qui sont des paires de sommets. Toutes les arêtes de G sont de poids positif.

Principe de l'algorithme de Dijkstra

Entrée :

- $G(S, A)$: un graphe pondéré,
- d : le sommet de départ à partir duquel on veut déterminer les plus courts chemins aux autres sommets,
- P : construction d'un sous-graphe tel que la distance entre un sommet de P depuis d soit définie et soit un minimum dans le graphe G,
- Parent : tableau pour noter les sommets par où on passe. Parent est utilisé comme le tableau des précédents de chaque sommet, initialisé avec un élément n'appartenant pas à S,
- M : tableau où les indices représentent les sommets du graphe : 0 désigne le sommet "A", 1 désigne le sommet "B", 2 désigne le sommet "C", etc...

  Les éléments de ce tableau sont corrigés au fur et à mesure de l'algorithme afin d'obtenir les distances les plus courtes du sommet de départ à un sommet du graphe.

Début :

```
P ← ∅
M[d] ← 0   \\ la distance de d à lui-même est égale à 0
M[s] ← +∞ pour chacun des sommets du graphe autre que d
Tant qu'il existe un sommet qui ne soit pas dans P
    Choisir un sommet s hors de P de plus petite distance M[s]
    Ajouter s à P
    Pour chaque sommet u hors de P mais voisin de s
        si M[u] > M[s] + poids(s,u)
            M[u] = M[s] + poids(s,u)
            Parent[u] = s    \\ le sommet s est le prédécesseur du sommet u
    Fin du pour
Fin tant que
```

Fin

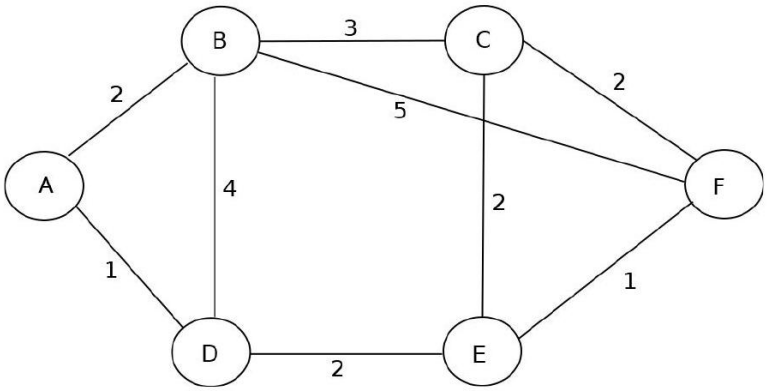

Fig. 6 Organisation des serveurs

En utilisant l'algorithme de Dijkstra, on souhaite déterminer tous les plus courts chemins, en terme de temps de communication, en partant du sommet A du graphe de la **figure 6**.

Il est plus simple de réaliser l'exécution de l'algorithme de Dijkstra avec un tableau particulier que l'on nomme $M+$. Ce tableau est dans le **DR**. Une première colonne précise l'évolution des distances d'un sommet spécifique depuis le sommet de départ lors de l'exécution de l'algorithme. Chaque ligne correspond à une étape de l'algorithme. Chaque ligne donne les valeurs des distances courantes des sommets depuis le sommet de départ avec éventuellement une mise à jour si une distance est plus petite que celle déjà calculée.

Dans le tableau $M+$, l'élément $B\,(2_A)$ correspond à la colonne choisie qui est B, à la valeur 2 du chemin et au sommet précédent A. Sur la ligne de l'élément $B\,(2_A)$ on choisit la valeur ③ de la colonne E pour l'étape suivante.

18. Compléter les 3 lignes manquantes du tableau $M+$ sur le **DR**.

Démonstration.
Détaillons la mise à jour des différentes variables de l'algorithme de Dijkstra, ce qui permettra de compléter au fur et à mesure le tableau $M+$. En effet, les 6 dernières colonnes du tableau $M+$ correspondent aux différentes mises à jour des 6 éléments de la liste M.

- **Initialisation**
  Au début de l'algorithme, dans notre exemple :
  - × la variable d contient 0 : l'indice du sommet A qui est choisi comme sommet de départ. On cherche donc les plus courts chemins entre A et les autres sommets.
  - × La liste P contient la liste vide.
    Elle contiendra l'indice des sommets dont on a visité tous les voisins.
  - × le tableau *Parent* contient, par exemple, le tableau à 6 éléments suivant :

$$[-1, -1, -1, -1, -1, -1]$$

    On initialise en effet chaque élément de *Parent* avec un élément qui n'est pas un indice d'un des sommets du graphe G. On a ici choisi -1 mais cela pourrait être n'importe quel élément autre que 0, 1, 2, 3, 4 et 5.
    Cette liste contiendra, pour chaque sommet, son prédecesseur dans le plus court chemin le reliant à A.

× la liste M (à 6 éléments : autant que de sommets du graphe considéré) vérifie :

- $M[d] \leftarrow 0$
- pour tous les autres sommets s du graphe G : $M[s] \leftarrow \infty$.

Comme ici la variable d contient 0, la liste M est initialisée à :

$$[0, \infty, \infty, \infty, \infty, \infty]$$

Ce qui correspond bien à la première ligne du tableau $M+$.

- **Principe de relâchement**
L'opération de relâchement d'un arc (s, u) consiste en un test permettant de savoir s'il est possible, en passant par s d'améliorer le plus court chemin de d jusqu'à u.
Si oui, il faut mettre à jour $M[u]$.
Plus précisément, on effectue le test suivant :

$$\boxed{M[u] \overset{?}{>} M[s] + poids(s, u)}$$

× Si ce test est négatif, on n'effectue pas de mise à jour.

× Si ce test est positif, cela signifie qu'un chemin de plus petite taille a été trouvé pour passer du sommet d au sommet u :

- on passe de d à s (ce chemin est de poids $M[s]$),
- on termine ce chemin en utilisant l'arc (s, u) de poids $poids(s, u)$.

Il convient alors d'effectuer la mise à jour :

$$M[u] \leftarrow M[s] + poids(s, u)$$

Par ailleurs, on veut sauvegarder l'information selon laquelle dans le plus court chemin (en date) reliant d à u, le prédecesseur de u est s. Pour cela on met à jour la liste $Parent$:

$$Parent[u] \leftarrow s$$

- **Fin de l'algorithme**
L'algorithme se termine lorsque tous les sommets du graphe ont été explorés, c'est-à-dire lorsque la liste P contient tous les indices des sommets du graphe G (ici les entiers de 0 à 6), mais pas forcément dans l'ordre.
À l'issue de ce programme :

× la liste $Parent$ contient, pour chaque sommet, l'indice de son prédecesseur dans le plus court chemin qui le relie à d.

× la liste M contient, pour chaque sommet, la longueur du plus court chemin le reliant à d.

- Détaillons maintenant l'exécution de ce programme dans le cas du graphe de l'énoncé, avec sommet de départ A.
  - × On rappelle qu'initialement :
    - la liste P est vide,
    - $Parent = [-1, -1, -1, -1, -1, -1]$,
    - $M = [0, \infty, \infty, \infty, \infty, \infty]$

    Il existe des sommets qui n'ont pas encore été visités (tous les indices des sommets de G n'apparaissent pas dans P).
  - × On choisit alors le sommet, dont l'indice n'est pas dans P, de plus petite distance à A. Autrement dit, le sommet d'indice s pour lequel $M[s]$ est le plus petit.
    Il s'agit ici du sommet A (d'indice $s = 0$).
    - On ajoute alors s à P. Ainsi :
    $$P = [0]$$
    - On explore les sommets voisins de A dont l'indice n'est pas dans P.
      Il s'agit des sommets B et D.
      - ▸ Pour le sommet B (d'indice 1), on remarque :
      $$M[1] = \infty \ > \ 2 = M[0] + poids(0, 1)$$
      On met donc à jour $M[1]$ pour qu'il contienne $M[0] + poids(0, 1) = 2$. De plus, à ce stade, le prédecesseur de B dans le plus court chemin qui le relie au sommet A est A. On met donc à jour $Parent[1]$ pour qu'il contienne 0 (l'indice du sommet A).
      - ▸ Pour le sommet D (d'indice 3), on remarque :
      $$M[3] = \infty \ > \ 1 = M[0] + poids(0, 3)$$
      On met donc à jour $M[3]$ pour qu'il contienne $M[0] + poids(0, 3) = 1$. De plus, à ce stade, le prédecesseur de D dans le plus court chemin qui le relie au sommet A est A. On met donc à jour $Parent[3]$ pour qu'il contienne 0 (l'indice du sommet A).

    À la fin de ce premier tour de boucle, on a donc :
    $$\begin{aligned} M &= [0, 2, \infty, 1, \infty, \infty] \\ Parent &= [-1, 0, -1, 0, -1, -1] \end{aligned}$$

    Il existe toujours des sommets du graphe G qui n'ont pas été visités (tous les sommets de G n'apparaissent pas dans P).
  - × On choisit alors le sommet, dont l'indice n'est pas dans P, de plus petite distance à A. Autrement dit, le sommet d'indice s, différent de 0, pour lequel $M[s]$ est le plus petit.
    Il s'agit ici du sommet D (d'indice $s = 3$).
    - On ajoute alors s à P. Ainsi :
    $$P = [0, 3]$$
    - On explore les sommets voisins de D dont l'indice n'est pas dans P.
      Il s'agit des sommets B et E.
      - ▸ Pour le sommet B (d'indice 1), on remarque :
      $$M[1] = 2 \not> 5 = M[3] + poids(3, 1)$$
      On ne met donc à jour ni M, ni $Parent$.

- ▸ Pour le sommet E (d'indice 4), on remarque :

$$M[4] = \infty \ > \ 3 = M[3] + \mathit{poids}(3,4)$$

On met donc à jour $M[4]$ pour qu'il contienne $M[3] + \mathit{poids}(3,4) = 3$. De plus, à ce stade, le prédecesseur de E dans le plus court chemin qui le relie au sommet A est D. On met donc à jour $\mathit{Parent}[4]$ pour qu'il contienne 3 (l'indice du sommet D).

À la fin de ce deuxième tour de boucle, on a donc :

$$\begin{aligned} M &= [0, 2, \infty, 1, 3, \infty] \\ \mathit{Parent} &= [-1, 0, -1, 0, 3, -1] \end{aligned}$$

Il existe toujours des sommets du graphe G qui n'ont pas été visités (tous les sommets de G n'apparaissent pas dans P).

× On choisit alors le sommet, dont l'indice n'est pas dans P, de plus petite distance à A. Autrement dit, le sommet d'indice s, différent de 0 et 3, pour lequel $M[s]$ est le plus petit.
Il s'agit ici du sommet B (d'indice $s = 1$).

- On ajoute alors s à P. Ainsi :

$$P = [0, 3, 1]$$

- On explore les sommets voisins de B dont l'indice n'est pas dans P.
Il s'agit des sommets C et F.

- ▸ Pour le sommet C (d'indice 2), on remarque :

$$M[2] = \infty \ > \ 5 = M[1] + \mathit{poids}(1,2)$$

On met donc à jour $M[2]$ pour qu'il contienne $M[1] + \mathit{poids}(1,2) = 5$. De plus, à ce stade, le prédecesseur de C dans le plus court chemin qui le relie au sommet A est B. On met donc à jour $\mathit{Parent}[2]$ pour qu'il contienne 1 (l'indice du sommet B).

- ▸ Pour le sommet F (d'indice 5), on remarque :

$$M[5] = \infty \ > \ 7 = M[1] + \mathit{poids}(1,5)$$

On met donc à jour $M[5]$ pour qu'il contienne $M[1] + \mathit{poids}(1,5) = 7$. De plus, à ce stade, le prédecesseur de F dans le plus court chemin qui le relie au sommet A est B. On met donc à jour $\mathit{Parent}[5]$ pour qu'il contienne 1 (l'indice du sommet B).

À la fin de ce troisième tour de boucle, on a donc :

$$\begin{aligned} M &= [0, 2, 5, 1, 3, 7] \\ \mathit{Parent} &= [-1, 0, 1, 0, 3, 1] \end{aligned}$$

Il existe toujours des sommets du graphe G qui n'ont pas été visités (tous les sommets de G n'apparaissent pas dans P).

× On choisit alors le sommet, dont l'indice n'est pas dans P, de plus petite distance à A. Autrement dit, le sommet d'indice s, différent de 0, 3 et 1, pour lequel $M[s]$ est le plus petit.
Il s'agit ici du sommet E (d'indice $s = 4$).

- On ajoute alors s à P. Ainsi :

$$P = [0, 3, 1, 4]$$

- On explore les sommets voisins de E dont l'indice n'est pas dans P.
  Il s'agit des sommets C et F.
  - ▸ Pour le sommet C (d'indice 2), on remarque :

$$M[2] = 5 \not> 5 = M[4] + poids(4, 2)$$

    On ne met donc à jour ni M, ni $Parent$.
  - ▸ Pour le sommet F (d'indice 5), on remarque :

$$M[5] = 7 \ > \ 4 = M[4] + poids(4, 5)$$

    On met donc à jour $M[5]$ pour qu'il contienne $M[4] + poids(4, 5) = 4$. De plus, à ce stade, le prédecesseur de F dans le plus court chemin qui le relie au sommet A est E. On met donc à jour $Parent[5]$ pour qu'il contienne 4 (l'indice du sommet E).

  À la fin de ce quatrième tour de boucle, on a donc :

$$\begin{aligned} M &= [0, 2, 5, 1, 3, 4] \\ Parent &= [-1, 0, 1, 0, 3, 4] \end{aligned}$$

Il existe toujours des sommets du graphe G qui n'ont pas été visités (tous les sommets de G n'apparaissent pas dans P).

× On choisit alors le sommet, dont l'indice n'est pas dans P, de plus petite distance à A. Autrement dit, le sommet d'indice s, différent de 0, 3, 1 et 4, pour lequel $M[s]$ est le plus petit.
Il s'agit ici du sommet F (d'indice $s = 5$).

- On ajoute alors s à P. Ainsi :

$$P = [0, 3, 1, 4, 5]$$

- On explore les sommets voisins de F dont l'indice n'est pas dans P.
  Il s'agit du sommet C.
  - ▸ Pour le sommet C (d'indice 2), on remarque :

$$M[2] = 5 \not> 6 = M[5] + poids(5, 2)$$

    On ne met donc à jour ni M, ni $Parent$.

  À la fin de ce cinquième tour de boucle, on a donc :

$$\begin{aligned} M &= [0, 2, 5, 1, 3, 4] \\ Parent &= [-1, 0, 1, 0, 3, 4] \end{aligned}$$

Il existe toujours des sommets du graphe G qui n'ont pas été visités (tous les sommets de G n'apparaissent pas dans P).

× On choisit alors le sommet, dont l'indice n'est pas dans P, de plus petite distance à A. Il ne reste que le sommet d'indice $s = 2$ (le sommet C) dont l'indice n'est pas dans P.

- On ajoute alors s à P. Ainsi :

$$P = [0, 3, 1, 4, 5, 2]$$

- On explore les sommets voisins de C dont l'indice n'est pas dans P. Il n'y en a pas.
  On n'effectue donc pas de mise à jour.

Tous les sommets du graphe G ont été visités. L'algorithme se termine.

On peut maintenant remplir le tableau $M+$ avec les mises à jour obtenues à chaque tour de boucle. On obtient le tableau suivant.

	A	**B**	**C**	**D**	**E**	**F**
étape initiale	0	∞	∞	∞	∞	∞
$A(0)$	-	2	∞	①	∞	∞
$D\,(1_A)$	-	②	∞	-	3	∞
$B\,(2_A)$	-	-	5	-	③	7
$E\,(3_D)$	-	-	5	-	-	④
$F\,(4_E)$	-	-	⑤	-	-	-
$C\,(5_B)$	-	-	-	-	-	-

□

19. Donner la valeur du plus court chemin entre "A" et "F". Expliquer comment on obtient cette valeur à l'aide du tableau $M+$. Expliciter le chemin le plus court trouvé pour aller de "A" à "F".

Démonstration.

- Après exécution de l'algorithme de Dijkstra, la variable M contient la liste suivante :

$$[0,\ 2,\ 5,\ 1,\ 3,\ 4]$$

La longueur du plus court chemin entre A et F (sommet d'indice 5) est contenu dans l'élément $M[5]$.

On en déduit que le plus court chemin entre A et F est de longueur 4.

- Après exécution de l'algorithme de Dijkstra, la variable *Parent* contient, pour chaque sommet, son prédécesseur dans le plus court chemin le reliant à A.
Ainsi, pour déterminer le plus court chemin reliant A à F, il suffit de chercher le prédecesseur de F (contenu dans $Parent[5]$), puis le prédécesseur de son prédécesseur, etc. jusqu'à atteindre le sommet A.
On rappelle qu'après exécution de l'algorithme :

$$Parent\ =\ [-1,\ 0,\ 1,\ 0,\ 3,\ 4]$$

On obtient que :

× l'indice du prédécesseur de F (sommet d'indice 5) est contenu dans $Parent[5]$. Il s'agit de l'indice 4. Ainsi le prédécesseur de F est E.

× l'indice du prédécesseur de E (sommet d'indice 4) est contenu dans $Parent[4]$. Il s'agit de l'indice 3. Ainsi le prédécesseur de E est D.

× l'indice du prédécesseur de D (sommet d'indice 3) est contenu dans $Parent[3]$. Il s'agit de l'indice 0. Ainsi le prédécesseur de D est A.

Le plus court chemin entre A et F est donc : $A - D - E - F$.

□

Partie V - Requêtes SQL

L'entreprise possède une base de données nommée `Gestion_Entreprise` constituée de trois tables : clients, produits et ventes. Les contenus de ces tables se trouvent en **annexe 2**.

La **table 'clients'** est constituée de 5 champs :

- *id* : de type `INTEGER` – clé primaire auto-incrémentée ;
- *num_secu* : de type `INTEGER` - entier de 15 chiffres ;
- *nom* : de type `TEXT` ;
- *prenom* : de type `TEXT` ;
- *num_CB* : de type `INTEGER`.

La **table 'produits'** est constituée de 5 champs :

- *id* : de type `INTEGER` – clé primaire auto-incrémentée ;
- *ref_produit* : de type `INTEGER` ;
- *nom_produit* : de type `TEXT` ;
- *qrcode* : de type `TEXT` ;
- *prix* : de type `DECIMAL`.

La **table 'ventes'** est constituée de 3 champs :

- *date* : de type `TEXT` ;
- *ref_produit* : de type `INTEGER` ; – clé étrangère, pointe vers la clé primaire *id* de la table *produits* ;
- *num_clients* : de type `INTEGER` ; – clé étrangère, pointe vers la clé primaire *id* de la table *clients*.

  Les dates dans cette table sont définies par une chaîne de 10 caractères suivant le format *année-mois-jour*. Exemples de dates : '2019-06-01', '2022-12-30'.

20. Écrire, en SQL, la requête (**1**) qui permet d'obtenir le numéro de la carte de crédit de toutes les personnes référencées dans la base de données de l'entreprise dont le numéro de sécurité sociale commence par 2. On utilisera le caractère '_' comme séparateur des milliers. Par exemple 10000000 sera réécrit comme 10_000_000.

Démonstration.

```
1  SELECT num_CB FROM clients WHERE num_secu > 199_999_999_999_999
```

Commentaire

Bien sûr, on pouvait aussi proposer la requête suivante.

```
1  SELECT num_CB FROM clients WHERE num_secu >= 200_000_000_000_000
```

□

21. Écrire, en SQL, la requête (**2**) permettant d'obtenir le nom et le prénom de toutes les personnes ayant effectué un achat avec un résultat sans doublon.

Démonstration.

```
1  SELECT DISTINCT clients.nom, clients.prenom FROM clients
2  JOIN ventes ON clients.id = ventes.num_client
```

□

22. Écrire, en SQL, la requête (**3**) qui permet d'obtenir les produits associés à chaque numéro de carte de crédit du client et qui ont été vendus entre le 1 juin 2020 et le 30 juillet 2020. On rappelle que SQL compare les variables de type `TEXT` grâce à l'ordre lexicographique.
Par exemple '1989-06-13 $<$ 1999-07-13' est vrai.

Démonstration.

```
SELECT produits.nom_produit, clients.num_CB FROM produits
JOIN ventes ON produits.ref_produit = ventes.ref_produit
JOIN clients ON clients.id = ventes.num_client
WHERE (ventes.date >= 2020-06-01) AND (ventes.date <= 2020-07-30)
GROUP BY clients.num_CB
```

□

Annexe 1

Module "Gestion_QRCode"

fonction `Gestion_QRCode.open(fp)`

× Paramètre :

- `fp` : nom de fichier (chaîne de caractères) représentant une image sous différents formats tels que PPM, PNG, JPEG, GIF, TIFF et BMP.

× Retour :

- retourne une variable qui est un descripteur d'image (un objet image).

Attention : la valeur retournée n'est en aucun cas une liste de listes ou une liste de listes similaire à celle de la bibliothèque *Numpy*.

Exemple :
```
img = Gestion_QRCode.open('uneImage.png')
```

fonction `Gestion_QRCode.Show()`

× Retour :

- retourne la valeur None. (6)

Cette fonction affiche l'image dans n'importe quelle visionneuse d'images.

Exemple :
```
img.show()
```

attribut `Gestion_QRCode.size`
Permet de connaître la taille de l'image en pixels, le résultat est un tuple (largeur, hauteur).

Exemple :
```
print(img.size)
>>> (360, 160) # soit largeur = 360 pixels et hauteur = 160 pixels
```

fonction `Gestion_QRCode.getpixel(x,y)`

× Paramètres :

- x : la coordonnée x du pixel référencé ;
- y : la coordonnée y du pixel référencé.

× Retour :

- retourne les attributs de la couleur du pixel, au format RVB (7)

La valeur retournée est un tuple `(r, v, b)` correspondant à la couleur du pixel : le plus souvent $(0, 0, 0)$ pour un pixel noir et $(255, 255, 255)$ pour un pixel blanc pour un QR code.

Exemple :
```
(r,v,b) = img.getpixel(100,30)
>>> (0, 0, 0) # pour un pixel noir
```

(6). La valeur None est une valeur qui correspond à l'absence de valeur.
(7). Le système `RVB` (`Rouge`, `Vert`, `Bleu`), ou en anglais `RGB` (`Red`, `Green`, `Blue`), permet de coder les couleurs en informatique. Un écran informatique est composé de pixels représentant une couleur au format `RVB`. La composante `R` est codée sur 8 bits de 0 à 255 en décimal. Il en va de même pour les composantes suivantes. Le codage des couleurs va du plus foncé au plus clair.

Annexe 2

Base de données "Gestion_Entreprise"

Le contenu des tables *clients*, *produits* et *ventes* de la base de données *Gestion_Entreprise* est donné ci-après.

Tables `clients`

id	num_secu	nom	prenom	num_CB
1	286128817863441	Eldyn	Sophie	6767342589219928
2	298082934500890	Gomez	Maria	2324563490665454
3	298082934500896	Ruiza	Flor	9889454573204522
4	109086723487917	Kovitz	Boris	6789543778653678
5	175105642102321	Mottreff	Erwan	4745342178563217
6	189027511732543	Settin	Michel	7856432167453492
7	191017511318196	Valérie	Georges	8787564521392354
...	...	...	...	...

Table `produits`

id	ref_produit	nom_produit	qrcode	prix
1	27	Buffet chêne	27.jpg	320
2	102	Chaise rustique	102.jpg	65
3	453	Table ronde	453.jpg	75
4	756	Table ovale	756.jpg	120
5	921	Coffret Bali	921.jpg	170
...	...	...	...	...

Table `ventes`

date	ref_produit	num_client
2020-05-15	2	2
2020-06-17	3	1
2020-06-21	3	7
2020-07-19	4	5
2020-08-19	5	5
2020-09-05	4	6

Annexe 3

Rappels des syntaxes en Python

Définir une liste.	$L = [1, 2, 3]$ >>> $L[0]$ 1
Définir une liste de listes.	$LL = [[1, 2], [3, 4], [5, 6], [7, 8]]$ >>> $LL[1]$ $[3, 4]$
Ajouter un élément à la fin d'une liste.	$L.append(5)$ $LL.append([9, 10])$
Convertir un nombre entier en une chaîne de caractères.	>>> $str(12345)$ $'12345'$
Convertir une chaîne de caractères en un nombre entier. Attention : si la chaîne de caractères contient un espace, l'appel de la fonction int provoque une erreur.	>>> $int\,('12345')$ 12345
$a//b$ donne le quotient de la division euclidienne de a par b.	>>> $10//3$ 3
$a\%b$ donne le reste de la division euclidienne de a par b.	>>> $10\%3$ 1
Définir une chaîne de caractères.	`mot = 'Python'`
Longueur d'une chaîne.	`len(mot)`
Le slicing permet d'extraire des éléments d'une liste ou d'une chaîne.	si $L = [6, 8, 5, 3, 7]$ $L[2:4] - - > [5, 3]$ $L[:3] - - > [6, 8, 5]$ $L[3:] - - > [3, 7]$ $L[::2] - - > [6, 5, 7]$ $L[2:-1] - - > [5, 3]$ $L[-1] - - > 7$ $mot[2:7] - - > 'thon'$

FIN