# Devoir sur table

***REMPLIR OBLIGATOIREMENT VOTRE NOM ET PRENOM CI-DESSOUS :***

| | |
|---|---|
| **Nom** | |
| **Prénom** | |

**Jeudi 12 décembre 2024**

**Durée de l'épreuve : 1 heure**

***L'usage de la calculatrice n'est pas autorisé.***

**Le sujet est à rendre avec le devoir**

**Dès que ce sujet vous est remis, assurez-vous qu'il est complet.**
**Ce sujet comporte 10 pages numérotées de 1 à 10.**

# Exercice 1 – Le Tri Fusion (5 points)

Le tri fusion est un algorithme de tri efficace qui utilise la méthode "diviser pour régner". Son but principal est de trier une liste d'éléments en la divisant récursivement en sous-listes plus petites, puis en fusionnant ces sous-listes triées pour obtenir la liste finale triée.

**Partie A : Comprendre et compléter la fonction fusion**

**Question 1 :** Compléter le code sur la feuille

**Question 2 :** Expliquez pourquoi les listes données à la fonction fusion doivent être triées.

```
def fusion(T1: list, T2: list) -> list:
    """
    Fusionne les deux tableaux triés T1 et T2
    """

    T = [0] * (len(T1) + len(T2))
    i1 = i2 = j = 0  # indices des piles

    while i1 < len(T1) and i2 < len(T2):
        if _____________: # Condition à compléter
            T[j] = T1[_______] # Complétez ici
            i1 += 1
        else:
            T[j] = T2[_______] # Complétez ici
            i2 += 1
        j += 1

    while i1 < len(T1):
        T[j] = _______ # Complétez ici
        i1 += 1
        j += 1

    while i2 < len(T2):
        T[j] = _______ # Complétez ici
        i2 += 1
        j += 1

    return _______ # Complétez ici
```

**Partie B : Le tri fusion**

**Question :** Compléter le code sur la feuille

```
def tri_fusion(T: list) -> list:
  """
  Renvoie un tableau T trié
  """

   n = len(T)
   # Cas de base
   if n < 2:
      return T[:]  # Cas de base : une liste de 0 ou 1 élément est déjà triée.

   milieu = n // 2
   T1 = _____ # Complétez ici
   T2 = _____ # Complétez ici

   # Résoudre
   T1 = tri_fusion(_____) # Complétez ici
   T2 = tri_fusion(_____) # Complétez ici

   # Combiner : fusion des tableaux triés
   return fusion(_____, _____) # Complétez ici
```

## Exercice 2 - Cryptage selon le « Code de César » (5 points)

Dans cet exercice, on étudie une méthode de chiffrement de chaînes de caractères alphabétiques. Pour des raisons historiques, cette méthode de chiffrement est appelée "code de César". On considère que les messages ne contiennent que les lettres capitales de l'alphabet "ABCDEFGHIJKLMNOPQRSTUVWXYZ" et la méthode de chiffrement utilise un nombre entier fixé, appelé la clé de chiffrement.

1. Soit la classe CodeCesar définie ci-dessous :

```
class CodeCesar:

    def __init__(self, cle):
        self.cle = cle
        self.alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    def decale(self, lettre):
        num1 = self.alphabet.find(lettre)
        num2 = num1+self.cle
        if   num2 >= 26:
            num2 = num2-26
        if num2 < 0:
            num2 = num2+26
        nouvelle_lettre = self.alphabet[num2]
        return nouvelle_lettre
```

On rappelle que la méthode str.find(lettre) renvoie l'indice (index) de la lettre dans la chaîne de caractères str.

**Représenter le résultat** d'exécution du code Python suivant :

```
code1 = CodeCesar(3)
print(code1.decale('A'))
print(code1.decale('X'))
```

2. La méthode de chiffrement du « code César » consiste à décaler les lettres du message dans l'alphabet d'un nombre de rangs fixé par la clé. Par exemple, avec la clé 3, toutes les lettres sont décalées de 3 rangs vers la droite : le A devient le D, le B devient le E, etc

**Ajouter** une méthode `cryptage(self, texte)` dans la classe `CodeCesar` définie à la question précédente, qui reçoit en paramètre une chaîne de caractères (le message à crypter) et qui retourne une chaîne de caractères (le message crypté).

Cette méthode `cryptage(self, texte)` doit crypter la chaîne `texte` avec la clé de l'objet de la classe `CodeCesar` qui a été instancié.

Exemple :

```
>>> code1 = CodeCesar(3)
>>> code1.cryptage("NSI")
'QVL'
```

3. **Ecrire un programme** qui :

- Demande de saisir la clé de chiffrement
- Crée un objet de classe CodeCesar
- Demande de saisir le texte à chiffrer
- Affiche le texte chiffré en appelant la méthode `cryptage`

Pour rappel, voici la documentation de la fonction input()

**input**()
**input**(*prompt*)

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. Example:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

4. On ajoute la méthode `transforme(texte)` à la classe `CodeCesar` :

```
def transforme(self, texte):
        self.cle = -self.cle
        message = self.cryptage(texte)
        self.cle = -self.cle
        return message
```

On exécute la ligne suivante : print(CodeCesar(10).transforme("PSX"))

**Que va-t-il s'afficher ? Expliquer votre réponse.**

# EXERCICE 3 (5 points)

Une entreprise doit placer des antennes relais le long d'une rue rectiligne. Une antenne relais de portée (ou *rayon*) $p$ couvre toutes les maisons qui sont à une distance inférieure ou égale à $p$ de l'antenne.

Connaissant les positions des maisons dans la rue, l'objectif est de placer les antennes le long de la rue, pour que toutes les maisons soient couvertes, tout en minimisant le nombre d'antennes utilisées.

La rue est représentée par un axe, et les maisons sont représentées des points sur cet axe :

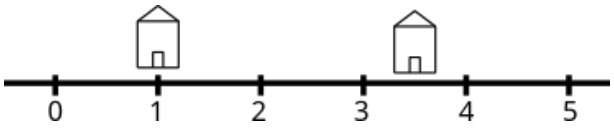



Figure 1. Deux maisons sur une rue, repérée par leur abscisse : 1 et 3,5

Les entités manipulées sont modélisées en utilisant la programmation orientée objet.

```
class Maison:
    def __init__(self, position):
        self._position = position

    def get_pos_maison(self):
        return self._position

class Antenne:
    def __init__(self, position, rayon):
        self._position = position
        self._rayon = rayon

    def get_pos_antenne(self):
        return self._position

    def get_rayon(self):
        return self._rayon
```

1. Donner le code qui crée et initialise deux variables m1et m2 avec des instances de la classe Maison situées aux abscisses 1 et 3,5 (Figure 1).

On ajoute à présent une antenne ayant un rayon d'action de 1 à la position 2,5 :

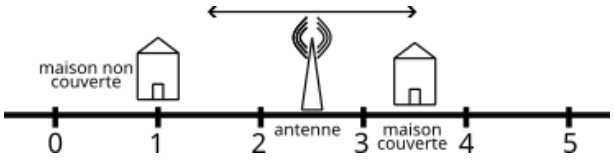



Figure 2. L'antenne placée en 2,5 et de rayon d'action 1 couvre la maison en 3,5 mais pas celle en 1

2. Donner le code qui crée la variable acorrespondant à l'antenne à la position 2,5 avec le rayon d'action 1.

On souhaite modéliser une rue par une liste d'objets de type Maison. Cette liste sera construite à partir d'une autre liste contenant des nombres correspondant aux positions des maisons. La fonction creation_rue réalise ce travail. Elle prend en paramètre une liste de positions et renvoie une liste d'objets de type Maison.

3. Recopier le schéma ci-dessous et le compléter pour donner une représentation graphique de la situation créée par :

   creation_rue([0, 2, 3, 4, 5, 7, 9, 10.5, 11.5])

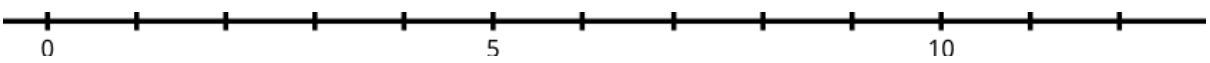



Figure 3. Axe pour représenter le problème avec 9 maisons

4. Compléter le code donné ci-dessous de la fonction creation_rue.

```
1  def creation_rue(pos):
2          pos.sort()
3          maisons = []
4          for p in pos:
5                  m = Maison(p)
6                  maisons.append(...)
7          return ...
```

Pour rappel : la commande tab.sort() trie la liste tab.

La méthode couvre de la classe Antenne prend en paramètre un objet de type Maison et indique par un booléen si l'antenne couvre la maison en question ou non. La méthode peut être utilisée ainsi (en supposant que les objets précédents m1, m2 et a existent).

>>> a.couvre(m1)False

>>> a.couvre(m2)True

5. Compléter la fonction couvre, ci-dessous, en veillant à ne pas accéder directement aux attributs d'une maison depuis la classe Antenne (on pourra utiliser les méthodes get_pos_maison, get_pos_antenne et get_rayon).

   Pour rappel : la fonction valeur absolue se nomme abs()en Python.

```
1 # Méthode à ajouter dans la classe Antenne
2 def couvre(self, maison):
3     # Code à compléter (éventuellement plusieurs lignes)
4     ...
```

## EXERCICE 4 (5 points)

La poussette est un jeu de cartes en solitaire. Cet exercice propose une version simplifiée de ce jeu basée sur des nombres.

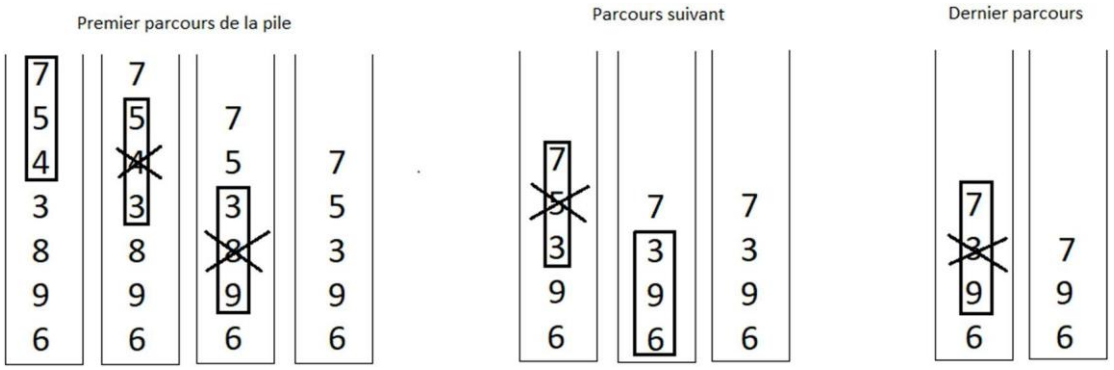
On considère une pile constituée de nombres entiers tirés aléatoirement. Le jeu consiste à réduire la pile suivant la règle suivante : quand la pile contient du haut vers le bas un triplet dont les termes du haut et du bas sont de même parité, on supprime l'élément central.

Par exemple :

- Si la pile contient du haut vers le bas, le triplet 1 0 3, on supprime le 0.
- Si la pile contient du haut vers le bas, le triplet 1 0 8, la pile reste inchangée.

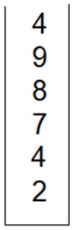
On parcourt la pile ainsi de haut en bas et on procède aux réductions. Arrivé en bas de la pile, on recommence la réduction en repartant du sommet de la pile jusqu'à ce que la pile ne soit plus réductible. Une partie est « gagnante » lorsque la pile finale est réduite à deux éléments exactement.

Voici un exemple détaillé de déroulement d'une partie.

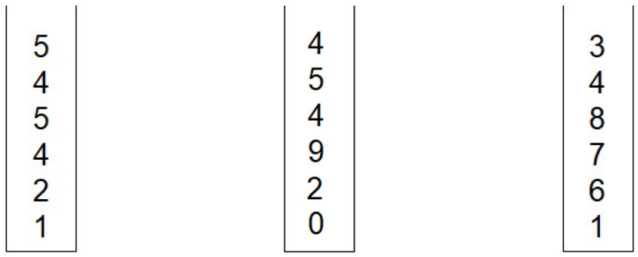


1.

**a.** Donner les différentes étapes de réduction de la pile suivante :




**b.** Parmi les piles proposées ci-dessous, donner celle qui est gagnante.




L'interface d'une pile est proposée ci-dessous. On utilisera uniquement les fonctions figurant dans le tableau suivant :

| Structure de données abstraite : Pile |
|---|
| • `creer_pile_vide()` renvoie une pile vide |
| • `est_vide(p)` renvoie `True` si `p` est vide, `False` sinon |
| • `empiler(p, element)` ajoute `element` au sommet de `p` |
| • `depiler(p)` retire l'élément au sommet de `p` et le renvoie |
| • `sommet(p)` renvoie l'élément au sommet de `p` sans le retirer de `p` |
| • `taille(p)` : renvoie le nombre d'éléments de `p` |

**2.** La fonction reduire_triplet_au_sommet permet de supprimer l'élément central des trois premiers éléments en partant du haut de la pile, si l'élément du bas et du haut sont de même parité. Les éléments dépilés et non supprimés sont replacés dans le bon ordre dans la pile.

**Recopier et compléter sur la copie :**
Le code de la fonction reduire_triplet_au_sommet prenant une pile p en paramètre et qui la modifie en place. Cette fonction ne renvoie donc rien.

```
1  def reduire_triplet_au_sommet(p):
2      a = depiler(p)
3      b = depiler(p)
4      c = sommet(p)
5      if a % 2 != .... :
6          empiler(p, ...)
7      empiler(p, ...)
```

**3.** On se propose maintenant d'écrire une fonction `parcourir_pile_en_reduisant` qui parcourt la pile du haut vers le bas en procédant aux réductions pour chaque triplet rencontré quand cela est possible.

**a.** Donner la taille minimale que doit avoir une pile pour être réductible.

**b.** Recopier et compléter sur la copie :

```
1  def parcourir_pile_en_reduisant(p):
2      q = creer_pile_vide()
3      while taille(p) >= ....:
4          reduire_triplet_au_sommet(p)
5          e = depiler(p)
6          empiler(q, e)
7      while not est_vide(q):
8          ............
9          ............
10      return p
```

**4.** Partant d'une pile d'entiers `p`, on propose ici d'implémenter une fonction récursive jouer renvoyant la pile `p` entièrement simplifiée. Une fois la pile parcourue de haut en bas et réduite, on procède à nouveau à sa réduction à condition que cela soit possible. Ainsi :

- Si la pile `p` n'a pas subi de réduction, on la renvoie.
- Sinon on appelle à nouveau la fonction `jouer`, prenant en paramètre la pile réduite.

Recopier et compléter sur la copie le code ci-dessous :

```
1  def jouer(p):
2      q = parcourir_pile_en_reduisant(p)
3      if ..............  :
4          return p
5      else:
6          return jouer(...)
```