```
        """Renvoie True si le processus est terminé, False sinon,
        en se basant sur le reste à faire."""
        ..........................................
```

c) La fonction `tourniquet` ci-dessous implémente l'algorithme décrit dans l'exercice.
Elle prend en paramètre une liste d'objets `Processus` donnés par ordre d'arrivée et un nombre entier positif correspondant au quantum. La fonction renvoie la liste des PID dans l'ordre de leur exécution par le processeur.
Recopier et compléter sur la copie le code manquant.

```
def tourniquet(liste_attente, quantum):
    ordre_execution = []
    while liste_attente != []:
        # On extrait le premier processus
        processus = liste_attente.pop(0)
        processus.change_etat("En cours d'exécution")
        compteur_tourniquet = 0
        while ..................... and .....................:
            ordre_execution.append(...........)
            processus.execute_un_cycle()
            compteur_tourniquet = compteur_tourniquet + 1
        if ......................:
            processus.change_etat("Suspendu")
            liste_attente.append(processus)
        else:
            processus.change_etat(...........)
    return ordre_execution
```

Exercice 3 — 4 points

Cet exercice porte sur l'algorithmique, la programmation orientée objet et la méthode diviser-pour-régner

L'objectif de cet exercice est de trouver les deux points les plus proches dans un nuage de points pour lesquels on connaît les coordonnées dans un repère orthogonal.

On rappelle que la distance entre deux points A et B de coordonnées $(x_A\,;\,y_A)$ et $(x_B\,;\,y_B)$ est donnée par la formule : $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

Les coordonnées d'un point seront stockées dans un tuple de deux nombres réels.

Le nuage de points sera représenté en Python par une liste de tuples de taille n, n étant le nombre total de points. On suppose qu'il n'y a pas de points confondus (mêmes abscisses et mêmes ordonnées) et qu'il y a au moins deux points dans le nuage.

Pour calculer la racine carrée, on utilisera la fonction `sqrt` du module `math`, pour rappel :

```
>>> from math import sqrt
>>> sqrt(16)
```

```
4.0
```

1. Cette partie comprend plusieurs questions générales :
   a) Donner le rôle de l'instruction de la ligne 1 du code précédent.
   b) Expliquer le résultat suivant :

```
>>> 0.1 + 0.2 == 0.3
False
```

   c) Expliquer l'erreur suivante :

```
>>> point_A = (3, 4)
>>> point_A[0]
3
>>> point_A[0] = 2
Traceback (most recent call last):
    File "<console>", line 1, in <module>
TypeError : 'tuple' object does not support item assignment
```

2. On définit la classe `Segment` ci-dessous :

```
1 from math import sqrt
2 class Segment:
3     def __init__(self, point1, point2):
4         self.p1 =  point1
5         self.p2 = point2
6         self.longueur = ..............  # à compléter
```

   a) Recopier et compléter la ligne 6 du constructeur de la classe `Segment`.

La fonction `liste_segments` donnée ci-dessous prend en paramètre une liste de points et renvoie une liste contenant des objets `Segment` qu'il est possible de construire à partir de ces points. On considère **les segments [AB] et [BA] comme étant confondus et ajoutera un seul objet dans la liste**.

```
1 def liste_segments(liste_points):
2     n = len(liste_points)
3     segments = []
4     for i in range(..............):
5         for j in range(.............., n):
6             # On construit le segment à partir des points i et j.
7             seg =  ..............................................
8             segments.append(seg)  # On l'ajoute à la liste
9     return segments
```

   b) Recopier la fonction sans les commentaires et compléter le code manquant.

c) Donner en fonction de n la longueur de la liste `segments`. Le résultat peut être laissé sous la forme d'une somme.

d) Donner, en fonction de n, la complexité en temps de la fonction `liste_segments`.

3. L'objectif de cette partie est d'écrire la fonction de recherche des deux points les plus proches en utilisant la méthode diviser-pour-régner.

On dispose de deux fonctions : `moitie_gauche` (respectivement `moitie_droite`) qui prennent en paramètre une liste et qui renvoient chacune une nouvelle liste contenant la moitié gauche (respectivement la moitié droite) de la liste de départ. Si le nombre d'éléments de celle-ci est impair, l'élément du center se trouve dans la partie gauche.

Exemples :

```
>>> liste = [1, 2, 3, 4]
>>> moitie_gauche(liste)
[1, 2]
>>> moitie_droite(liste)
[3, 4]
```

```
>>> liste = [1, 2, 3, 4, 5]
>>> moitie_gauche(liste)
[1, 2, 3]
>>> moitie_droite(liste)
[4, 5]
```

a) Écrire la fonction `plus_court_segment` qui prend en paramètre une liste d'objets `Segment` et renvoie l'objet `Segment` dont la longueur est la plus petite.

On procédera de la façon suivante :

- Tester si le cas de base est atteint, c'est-à-dire lorsque la liste contient un seul segment ;
- Découper la liste en deux listes de tailles égales (à une unité près) ;
- Appeler récursivement la fonction pour rechercher le minimum dans chacune des deux listes ;
- Comparer les deux valeurs récupérées et renvoyer la plus petite des deux.

4. On considère les trois points A$(3\,;\,4)$, B$(2\,;\,3)$ et C$(-3\,;\,-1)$.

a) Donner l'instruction Python permettant de construire la variable `nuage_points` contenant les trois points A, B et C.

b) En utilisant les fonctions de l'exercice, écrire les instructions Python qui affichent les coordonnées des deux points les plus proches du nuage de points `nuage_points`.