

# TP d'Algorithmique et structures de données

## Python : allocation des ressources

Équipe pédagogique Algo SD

```
|xxxxxxxxxx|  
|.....|  
|.....xx..|  
|.xx....xx..|  
|..x....xx..|  
|.x.....|
```

On va implémenter en python la gestion des ressources similaires à la mémoire dans un ordinateur. À cet effet, on va créer une classe `Ressources`, qui contiendra des méthodes pour réserver et rendre des ressources, ainsi que des méthodes pour visualiser l'utilisation des ressources et de la vérification. On va considérer un univers de  $n$  ressources  $U = \{0, 1, \dots, n - 1\}$  et on garde des plages d'intervalles des ressources disponibles. Par exemple, au moment de l'initialisation de la classe `Ressources`, tous les ressources  $U$  sont disponibles, donc intervalle correspondant est  $[0, n)$ . Si, par exemple, la ressource 5 est la seule qui n'est pas disponible (car allouée), on représente les ressources disponibles en python par la liste d'intervalles `[[0, 5], [6, n]]`.

1. **Initialisation.** Écrire la méthode `__init__` de la classe `Ressources` :

```
def __init__(self, n, intervalles=None):  
    """  
    Initialisation  
    """
```

Le paramètre `n` est le nombre de ressources existantes (la taille de l'univers). On veut le garder dans `self.n`. De plus, on devrait garder une liste d'intervalles de ressources disponibles dans `self.intervalles`. Si le paramètre optionnel `intervalles` n'est pas `None`, on suppose qu'il contient une liste d'intervalles qu'on devrait tout simplement stocker dans `self.intervalles`. S'il est `None`, on suppose que les `n` ressources sont toutes disponibles et on devrait initialiser `self.intervalles` en conséquence.

2. **Visualisation.** On veut visualiser l'état des ressources (disponible/indisponible) avec une chaîne de caractères. Écrire la méthode `__str__` de la class `Ressources` :

```
def __str__(self):  
    """  
    renvoie une chaîne 'visuelle' des ressources libres/utilisées.  
    par exemple, '/x..xxxx.../' indique qu'il y a 10 ressources,  
    les ressources 0, 3-7 sont libres.  
    """
```

**Sorties attendues.** Vous pouvez passer les intervalles ci-dessous au constructeur `__init__` de la classe `Ressources` !

```
str(Ressources(10, [[0, 6], [6, 10]])) -> |xxxxxxxxxx|  
str(Ressources(10, [[2, 4], [6, 8]])) -> |..xx..xx..|
```

3. **Vérification.** Les intervalles stockés dans `self.intervalles` devraient être disjoints (pour éviter des conflits) et les debuts devraient être triés par ordre croissant. De plus, aucun intervalle ne devrait utiliser des ressources inexistantes. Écrire la méthode `verification_invariants`, qui vérifie que les intervalles de `self.intervalles` respectent ces propriétés.

```

def verification_invariants(self):
    """
    renvoie True s'il n'y a pas d'intersections entre les intervalles et
    les intervalles sont triés du plus petit au plus gros indice, False sinon.
    """

```

**Sorties attendues.**

```

self.intervalles = [[0, 6], [6, 10]] -> True
self.intervalles = [[0, 7], [6, 10]] -> False
self.intervalles = [[1, 6], [6, 10], [0,1]] -> False
self.intervalles = [[1, 6], [6, 11]] -> False

```

4. **Disponibilité.** Écrire la fonction suivante, qui teste si la ressource à un certain indice est disponible :

```

def disponible(self, indice):
    """
    Renvoie True si l'indice donne est disponible dans la ressource et
    False sinon.
    """

```

**Sorties attendues.**

```

self.intervalles = [[0, 5], [6, 10]] : la ressource 6 est disponible
self.intervalles = [[0, 5], [6, 10]] : la ressource 5 n'est pas disponible
self.intervalles = [[1, 6], [6, 8]] : la ressource 8 n'est pas disponible

```

5. **Réservation.** Écrire la fonction suivante, qui alloue `ressource_demandées` (un nombre naturel) ressources :

```

def reserve(self, ressources_demandées):
    """
    Enlève le nombre de ressources demandées.
    Renvoie les ressources correspondant aux plages réservées.
    """

```

La difficulté est de gérer le cas, où les ressources demandées ne rentrent pas dans un seul intervalle de ressources libres. Dans ce cas, les ressources demandées seront divisées en plusieurs intervalles. La méthode `reserve` devrait renvoyer une instance de la classe `Ressources`, qui sera initialisée en utilisant les intervalles qui correspondent aux ressources réservées.

**Sorties attendues.**

```

res = Ressources(10, [[0, 2], [4, 6], [7, 9]])
print('début:      res =', str(res))
R1 = res.reserve(1)
print('reserve(1): res =', str(res))
R2 = res.reserve(3)
print('reserve(3): res =', str(res))
print('R1  =', str(R1))
print('R2  =', str(R2))
print('res  =', str(res))

```

La sortie attendue du code ci-dessus est :

```

début:      res = |xx..xx.xx.|
reserve(1): res = |xx..xx.x..|
reserve(3): res = |xx.....|

```

```
R1 = |.....x.|  
R2 = |....xx.x..|  
res = |xx.....|
```

6. **Fusion de listes d'intervalles.** Écrire une fonction `fusion` qui, étant donné deux listes triées d'intervalles disjoints, les fusionne en une seule liste triée d'intervalles disjoints. On peut supposer que les intervalles des deux listes sont disjoints.

```
def fusion(inter1, inter2):  
    """  
    Fusion des deux listes d'intervalles inter1 et inter2  
    """
```

**Sorties attendues.**

```
fusion([[3, 5]], [[1, 2], [5, 6]]) -> [[1, 2], [3, 5], [5, 6]]  
fusion([[0, 1], [5, 6]], [[1, 2], [4, 5]]) -> [[0, 1], [1, 2], [4, 5], [5, 6]]
```

7. **Libération.** En utilisant la fonction `fusion` de la question précédente, écrire la méthode `retourne` de la classe `Ressources`, qui libère les intervalles de ressources de la liste `ressources_rendues`. Pensez à simplifier les intervalles : par exemple, `[[1, 2], [2, 3], [3, 4]]` devient `[[1, 4]]`.

```
def retourne(self, ressources_rendues):  
    """  
    remet les plages de ressources donnees dans le systeme.  
    """
```

**Sorties attendues.** Voici un jeu de tests utilisant tous les méthodes. Le code suivant

```
ressources = Ressources(10)  
print("Disponibles :", ressources)  
print("on commence par tout reserver, il ne reste donc plus rien:")  
reservees = [ressources.reserve(c) for c in (2, 2, 3, 2, 1)]  
print("Disponibles :", ressources)  
print("on rend deux ressources:")  
ressources.retourne(reservees[1])  
print("Disponibles :", ressources)  
print("on rend encore deux ressources, mais plus loin:")  
ressources.retourne(reservees[3])  
print("Disponibles :", ressources)  
print("on reserve trois ressources sur les quatres disponibles:")  
print("Reservees   :", ressources.reserve(3))  
print("Disponibles :", ressources)  
print("Les intervalles :", ressources.intervalles)
```

devrait afficher

```
Disponibles : |xxxxxxxxxx|  
on commence par tout reserver, il ne reste donc plus rien:  
Disponibles : |.....|  
on rend deux ressources:  
Disponibles : |.....xx..|  
on rend encore deux ressources, mais plus loin:  
Disponibles : |..xx...xx..|  
on reserve trois ressources sur les quatres disponibles:  
Reservees   : |..x...xx..|  
Disponibles : |.x.....|  
Les intervalles : [[1, 2]]
```

8. On veut maintenant profiter du fait que les intervalles des ressources disponibles sont triés pour optimiser la méthode `disponible` de la classe `Ressources`. Écrire la méthode `disponible_bisect`, qui se comporte comme `disponible` et qui utilise le module `bisect` (voir `bisect`) de la librairie standard de python pour tester, si une ressource donnée est disponible.
9. On désire maintenant associer à une réservation de ressources un intervalle le plus petit possible contenant assez de ressources. Proposer un algorithme et analyser son coût en fonction du nombre d'intervalles disponibles.