TD2 : Test Driven Development

L'objectif de ce TP est de pratiquer le *Test Driven Development*, ou TDD. Le TDD est un concept utilisé en intégration continue (CI). Vous développerez progressivement une petite bibliothèque permettant de représenter un ensemble d'entiers. Le comportement attendu du programme sera introduit progressivement.
Rappel : Pour chaque nouvelle fonctionnalité, le TDD consiste à :

1) écrire un test qui échoue ;

2) écrire le minimum de code permettant de faire passer le test ;

3) améliorer le code sans modifier son comportement ;

4) vérifier que tous les tests précédents passent toujours.

Il n'est pas demandé d'anticiper les fonctionnalités suivantes. Vous êtes libres de choisir la représentation interne qui vous semble la plus simple.

1 Ensemble d'entiers

On souhaite développer une classe `IntervalSet` représentant un ensemble d'entiers. Elle fournit initialement les opérations suivantes :

```
IntervalSet()
add(a, b)
contains(x)
size()
```

Un ensemble nouvellement créé est vide. Ainsi :

```
s = IntervalSet()
s.size() == 0
s.contains(10) == false
```

L'appel `add(a, b)` ajoute à l'ensemble tous les entiers compris entre `a` et `b`, bornes incluses. On supposera toujours que :

$$a \leqslant b.$$

Ajouter l'intervalle [4,4] revient à ajouter uniquement l'entier 4. Après `s.add(4, 4)`, on doit avoir :

```
s.contains(4) == true
s.contains(3) == false
s.contains(5) == false
s.size() == 1
```

Ajouter l'intervalle [4,4] revient à ajouter uniquement l'entier 4. Après `s.add(3, 5)`, l'ensemble contient exactement les entiers 3, 4 et 5. On doit donc avoir :

```
s.contains(3) == true
s.contains(4) == true
s.contains(5) == true
s.contains(2) == false
s.contains(6) == false
s.size() == 3
```

Un entier présent plusieurs fois dans les intervalles ajoutés ne compte qu'une seule fois dans l'ensemble. Après :

```
s.add(1, 3)
s.add(3, 5)
```

on doit avoir :

```
s.size() == 5
```

L'ensemble contient alors les entiers :

$$\{1,2,3,4,5\}.$$

1.1 Suppression

On ajoute maintenant l'opération :

```
remove(a, b)
```

Elle retire de l'ensemble tous les entiers compris entre a et b, bornes incluses. Retirer un entier absent n'a aucun effet.
Pour une suppression simple, après :

```
s.add(1, 5)
s.remove(3, 3)
```

on doit avoir :

```
s.contains(1) == true
s.contains(2) == true
s.contains(3) == false
s.contains(4) == true
s.contains(5) == true
s.size() == 4
```

On peut aussi supprimer des intervalles. Après :

```
s.add(1, 10)
s.remove(4, 7)
```

l'ensemble contient :

$$\{1,2,3,8,9,10\}.$$

Une suppression peut dépasser l'ensemble de l'interval. Après :

```
s.add(10, 20)
s.remove(0, 15)
```

l'ensemble contient exactement les entiers de 16 à 20.

1.2 Changement d'échelle

Jusqu'à présent, tous les exemples utilisaient de petits intervalles. La classe doit maintenant être capable de représenter des intervalles beaucoup plus grands. Par exemple, le programme suivant doit fonctionner :

```
s = IntervalSet()
s.add(0, 1000000000000)
```

On doit immédiatement pouvoir vérifier :

```
s.size() == 1000000000001
s.contains(0) == true
s.contains(42) == true
s.contains(999999999999) == true
s.contains(1000000000000) == true
s.contains(1000000000001) == false
```

Le programme doit être capable de représenter cet ensemble sans stocker explicitement chacun des 10^{12} entiers. Le comportement suivant doit également être supporté :

```
s = IntervalSet()
s.add(0, 1000000000000)
s.remove(100, 999999999900)
```

Après cette opération :

```
s.contains(0) == true
s.contains(99) == true
s.contains(100) == false
s.contains(500000000000) == false
s.contains(999999999900) == false
s.contains(999999999901) == true
s.contains(1000000000000) == true
s.size() == 200
```

1.3 Travail demandé

À la fin du TP, vous devez disposer :

- d'une implémentation de `IntervalSet` respectant toutes les spécifications ;
- d'une suite de tests automatisés couvrant les comportements introduits pendant le TP ;
- d'un historique de développement permettant d'observer les cycles successifs de TDD.

À la fin du TP, comparez votre première implémentation et votre implémentation finale.

1) Quelle représentation aviez-vous choisie initialement ?

2) Quels tests ont dû être modifiés lors du changement de représentation ?

3) Quels tests sont restés inchangés ?

4) Pourquoi des tests portant uniquement sur le comportement observable facilitent-ils ce type de changement ?

2 Chaîne compacte d'entiers

On souhaite développer une classe `CompactString` représentant une séquence de caractères. La classe fournit les opérations suivantes :

```
  CompactString()
  append(c, n)
  insert(i, c, n)
  charAt(i)
  length()
  count(c)
```

Les indices commencent à zéro.
L'opération `append(c, n)` ajoute n occurrences du caractère `c` à la fin de la chaîne. L'opération `insert(i, c, n)` ajoute n occurrences du caractère `c` à la position `i`. Les caractères précédemment situés à partir de cette position sont décalés vers la droite. On supposera toujours que :

$$n > 0$$

et, pour une insertion :

$$0 \leqslant i \leqslant \texttt{length()}.$$

L'opération `charAt(i)` retourne le caractère situé à l'indice `i`. On supposera que l'indice fourni est valide. L'opération `length()` retourne le nombre de caractères contenus dans la chaîne. Enfin, `count(c)` retourne le nombre d'occurrences du caractère `c` dans la chaîne.
Par exemple :

```
1  s = CompactString()
2  s.append('a', 3)
3  s.append('c', 3)
4  s.insert(3, 'b', 2)
```

La chaîne représente alors : `aaabbccc`.
La classe doit également permettre de manipuler des chaînes contenant un très grand nombre de caractères.
Développez `CompactString` en suivant une démarche TDD.