TP0 - Mise en jambes

L'objectif de ce TP est de vous confronter à la résolution d'un problème algorithmique simple en Java et de montrer que sans *formalisation* du problème à résoudre il est très difficile d'avoir des garanties sur la qualité de la solution obtenue.

1 Le problème à résoudre

On veut définir une méthode `public boolean subSeq(Object[] t1, Object[] t2)` qui détermine si toutes les occurrences des éléments de `t1` (sauf éventuellement une) apparaissent dans `t2` et dans le même ordre. A noter que `t2` peut contenir des occurrences d'éléments qui n'apparaîssent pas dans `t1` [1]. Les listes sont quelconques : elles ne sont pas triées et peuvent contenir des doublons.

2 La démarche

Remarque générale : **notez le temps que vous passez au développement et celui que vous passez sur les tests**.

1. Lancez Eclipse 4.3 Scala (Menu>Eclipse>Eclipse4.3Scala) et créez un nouveau workspace pour ACF
2. Dans Eclipse importez l'archive `/share/m1info/ACF/TP0/TP0ACF.zip` :

   File>Import>Existing Projects into Workspace>Select archive file

   Ceci crée un projet `TP_0_ACF` dans votre workspace.
3. Exécutez les tests automatiques de la fonction `subSeq` (qui pour l'instant retourne toujours `true`) : dans le projet `TP_0_ACF`, dans le package `p1`, exécutez la classe `Main`.
4. Dans la console figure le nombre de tests réussis.
5. Écrivez des tests (vous pouvez par exemple utiliser le fichier JUnit `MainTest.java`). Rappel, pour créer "rapidement" un tableau en Java :

   ```
   Integer[] t1 = new Integer[] { 1, 2, 3 };
   ```
6. Programmez la fonction `subSeq` en Java. Si vous préférez raisonner sur des listes, vous pouvez convertir les tableaux en listes : `List<Object> l1= new ArrayList<Object>(Arrays.asList(t1));`.
7. Testez votre fonction avec **vos tests** JUnit.
8. Une fois que vous êtes satisfaits de votre solution, ré-exécutez les tests automatiques (gracieusement offerts ☺).
9. Si le test automatique signale des tests échoués, complétez vos tests pour découvrir les cas problématiques et reprenez le développement.

1. Ce type de problème algorithmique simple vient du domaine de la détection d'intrusion dans les réseaux. Les outils de détection d'intrusion recherchent dans les *logs* des *signatures d'attaques* qui sont des séquences d'évènements réseaux représentatives d'une attaque particulière. La recherche d'une signature dans un log correspond à la recherche de sous-liste. Pour qu'une signature soit détectée dans un log, tous ses évènements doivent apparaître dans le même ordre dans le log. Pour certaines signatures, un certain nombre d'évènements les constituant peuvent ne pas apparaître dans le log. C'est le cas ici où on autorise au maximum un évènement de la signature à être manquant dans le log. Par exemple, une signature d'attaque peut être `[su, cp, exec]` et le log `[cd, cd, exec, su, cd, cd, exec, cd, cd]`. Avoir des garanties sur cette fonction est important car une fonction trop permissive (donne false plus souvent que ce qui est attendu) risque d'accepter des logs dans lesquels l'attaque est présente. A l'inverse, une fonction trop restrictive (donne true plus souvent que ce qui est attendu) donnera trop de faux positifs : des logs corrects mais rejetés.