DS 1 – Initiation à l'algorithmique et à la programmation (26/10/16)

Aucun document et aucune machine autorisés

Exercice 1 (13 pts) : saisie contrôlée de nombres

La saisie d'un entier par un utilisateur peut se faire en utilisant l'instruction `readInt()`. Le problème est que des exceptions sont générées dès que l'utilisateur saisit par mégarde autre chose que des chiffres, mettant ainsi fin de manière prématurée au programme. Pour remédier à ce problème, nous souhaitons écrire une fonction `robustReadInt()` qui vérifie la saisie de l'utilisateur et renvoie un entier si la saisie est correcte.

Q1 (2). Ecrire la fonction `char2Int` qui convertit un caractère représentant un chiffre en l'entier correspondant. Rappelez-vous l'exercice de TP consistant à convertir des minuscules en majuscules … il n'est pas nécessaire de connaître la table ASCII pour faire cette question !

```
void testChar2Int() {
  assertEquals(0, char2Int('0'));
  assertEquals(8, char2Int('8'));
}
```

Q2 (3). Ecrire la fonction `isNumber` qui vérifie que tous les caractères d'une chaîne donnée sont des chiffres.

```
void testQueDesChiffres() {
  assertTrue (isNumber("128"));
  assertFalse(isNumber(" 128"));
  assertFalse(isNumber("12e2"));
}
```

Q3 (4). Ecrire la fonction `convertStringToInt` qui convertit une chaîne de caractères en entier. On supposera que tous les caractères de la chaîne sont des chiffres. Vous pourrez utiliser la fonction `char2Int` pour convertir un caractère contenant un chiffre en l'entier correspondant et la fonction mathématique `double pow(double nombre, int exposant)` pour calculer $\texttt{nombre}^{\texttt{exposant}}$.

```
void testConvertStringToInt() {
  assertEquals(128, convertStringToInt("128"));
  assertEquals(  0, convertStringToInt("0"));
  assertEquals( 42, convertStringToInt("42"));
}
```

Q4 (4). Ecrire la fonction `int robustReadInt()` qui demande à l'utilisateur de saisir un entier tant qu'il n'est pas correct et renvoie cet entier.
Conseil : il est pratique d'appeler les fonctions `isNumber` et `convertStringToInt` afin d'écrire la fonction `robustReadInt`.

BONUS : ré-écrivez la fonction `convertStringToInt` sans utiliser la fonction `pow` :)

Exercice 2 (11 pts) : jeu de conversions

Dans cet exercice nous allons écrire des fonctions qui affichent la conversion de dollars australiens (AUD) en roupies indiennes (INR). Actuellement, le taux de change est d'environ 50.7 INR pour 1 AUD.

Q1 (3). Ecrivez l'algorithme (ie. `void algorithm()`) qui affiche la table de conversion en roupie de 1 AUD à 10 AUD comme l'illustre la trace d'exécution ci-dessous :

```
Taux du dollar australien en roupie indienne ? 50,7
1 AUD = 50.7 INR
2 AUD = 101.4 INR
3 AUD = 152.10000000000002 INR
...
10 AUD = 507.0 INR
```

Q2 (3). Ecrivez la fonction conversion qui construit une chaîne contenant les `nb` (avec `nb` appartenant à [1, 10]) premières lignes de la table de conversion. Si `nb` est plus petit que 1, la fonction ne doit construire qu'une seule ligne. Si `nb` est plus grand que 10, la fonction ne doit construire que les 10 premières lignes.

```
void testConversion() {
  assertEquals("1 AUD = 50.7 INR\n", conversion(50.7, -1));
  assertEquals("1 AUD = 50.7 INR\n", conversion(50.7, 1));
  assertEquals("1 AUD = 50.7 INR\n2 AUD = 101.4 INR\n", conversion(50.7, 2));
}
```

Q3 (5). Ecrivez un troisième algorithme (ie. `void algorithm()`) qui affiche les `k` premières lignes d'une table de conversion (`k` est fourni par l'utilisateur et est compris entre 1 et 40). La table de conversion est maintenant structurée selon les critères suivants :

- le pas vaut 1 entre 1 et 10 AUD,
- le pas vaut 10 entre 10 et 100 AUD,
- le pas vaut 100 entre 100 à 1000 AUD,
- le pas vaut 1000 à partir de 1000 AUD.

Voici un exemple de trace d'exécution dans laquelle le nombre entre parenthèse correspond au numéro de la ligne dans la table de conversion :

```
Nombre de lignes (<=40) ? 31
(1)1 AUD = 50.7 INR
(2)2 AUD = 101.4 INR
(3)3 AUD = 152.10000000000002 INR
(4)4 AUD = 202.8 INR
(5)5 AUD = 253.5 INR
(6)6 AUD = 304.20000000000005 INR
(7)7 AUD = 354.90000000000003 INR
(8)8 AUD = 405.6 INR
(9)9 AUD = 456.3 INR
(10)10 AUD = 507.0 INR
(11)20 AUD = 1014.0 INR
(12)30 AUD = 1521.0000000000002 INR
(13)40 AUD = 2028.0 INR
(14)50 AUD = 2535.0 INR
(15)60 AUD = 3042.0000000000005 INR
(16)70 AUD = 3549.0000000000005 INR
(17)80 AUD = 4056.0 INR
(18)90 AUD = 4563.0 INR
(19)100 AUD = 5070.0 INR
(21)200 AUD = 10140.0 INR
```

```
(22)300 AUD = 15210.000000000002 INR
(23)400 AUD = 20280.0 INR
(24)500 AUD = 25350.0 INR
(25)600 AUD = 30420.000000000004 INR
(26)700 AUD = 35490.0 INR
(27)800 AUD = 40560.0 INR
(28)900 AUD = 45630.0 INR
(29)1000 AUD = 50700.0 INR
(31)2000 AUD = 101400.0 INR
```

Exercice 3 (16 pts) : jeu de chiffres et de lettres

Dans cet exercice nous nous intéressons à des mots un peu particuliers. L'alphabet permettant de construire ces mots repose sur des couples constitués d'une lettre de l'alphabet et d'un chiffre. Voici quelques exemples de mots illustrant ce langage : `"a0"`, `"f3d1a4"`, `"a0a0"` …
Dans les questions qui suivent, nous allons définir différentes fonctions permettant de manipuler ce langage particulier.

Q1 (1) : Ecrivez la fonction `nombreCouples` qui détermine le nombre de couples présents dans un mot donné. On suppose que la chaîne est un mot valide du langage (ie. pas la peine de le vérifier).

```
    void testNombreCouples() {
        assertEquals(0, nombreCouples(""));
        assertEquals(1, nombreCouples("a0"));
        assertEquals(3, nombreCouples("b3a1c0"));
    }
```

Q2 (3) : Définissez la fonction `plusPetitQue` indiquant si un couple est plus petit qu'un autre, sachant que la comparaison se base sur l'ordre lexicographique (ie. celui du dictionnaire) et en cas d'égalité sur l'ordre naturel des chiffres.

```
    void testPlusPetitQue() {
        assertTrue(plusPetitQue('a', 1, 'b', 1));
        assertTrue(plusPetitQue('a', 1, 'a', 2));
    }
```

Q3 (2) : Définissez la fonction `plusPetitQue` permettant de comparer deux couples comme dans la question précédente, mais donnés cette fois sous la forme de deux chaînes de caractères. Vous pouvez éventuellement appeler la fonction définie en Q2.

```
    void testPlusPetitQue_String() {
        assertTrue(plusPetitQue("a1", "b1"));
        assertTrue(plusPetitQue("a1", "a2"));
    }
```

Q4 (4) : Définissez la fonction `estPresent` qui vérifie la présence d'un couple dans un mot donné.

```
    void testEstPresent() {
        assertTrue (estPresent("a1b2a2b1", 'a', 2));
        assertTrue (estPresent("a1b2a2b1", 'b', 1));
        assertFalse(estPresent("a1b2a2b1", 'a', 3));
    }
```

Q5 (3) : On souhaite une fonction vérifiant qu'un mot est constitué de couples strictement ordonnés. On supposera que le mot est valide.

```
    void testEstStrictementOrdonne() {
        assertTrue (estStrictementOrdonne("a0a1c2"));
        assertFalse(estStrictementOrdonne("a0a0a1"));
        assertFalse(estStrictementOrdonne("a1a0"));
    }
```

Q6 (3) : Définissez la fonction `fusionner` qui créer un nouveau mot ordonné à partir de deux mots ordonnés. Il est possible de réutiliser la fonction `plusPetitQue`.

```
    void testFusionner() {
        assertEquals("a1a1b0b1c1c2", fusionner("a1b0c2", "a1b1c1"));
        assertEquals("a1a1b0b1c1"  , fusionner("a1b0"  , "a1b1c1"));
        assertEquals("a1b1c1"      , fusionner(""      , "a1b1c1"));
    }
```