MOOC Init Prog Java
Exercices semaine 7

Exercice 23 : tri de Shell (révisions)

On cherche ici à implémenter une méthode de tri assez efficace en pratique, surtout pour des tableaux de taille faible à moyenne.
Il s'agit du tri dit "de Shell" du nom de son inventeur.

Le tri de Shell est un tri par insertion où l'on ne compare pas un élément à son prédecesseur immédiat. On le compare plutôt à son prédecesseur à distance `k`.

L'algorithme est le suivant (indices de 1 à taille) :

```
Pour k de taille/2 à 1, en le divisant par 2
    Pour i de k+1 à taille
        j <- i-k
        Tant que j > 0
            Si t[j] > t[j+k]
                échanger t[j] et t[j+k]
                j <- j-k
            Sinon
                j <- 0
```

Exercice 24 : Sapin et guirlande (révisions)

Il s'agit ici d'écrire un programme `Sapin.java` affichant un sapin décorés de guirlandes.

Partie 1

Commencez par écrire une méthode permettant de stocker un triangle, dessiné avec un symbole donné, dans un tableau à deux dimensions. La hauteur du triangle (entre `8` et `35`) ainsi que le symbole à utiliser seront demandés à l'utilisateur.

Le programme redemandera ces données à l'utilisateur tant qu'elles ne sont pas correctes.

Ecrivez ensuite une méthode affichant ce tableau.

Exemple d'exécution:

```
Quel symbole voulez-vous pour le triangle ? *
Combien de lignes (de 8 a 35)? 8

        *
       ***
      *****
     *******
    *********
   ***********
  *************
***************
```

Partie 2

Il s'agit maintenant de modifier la partie précédente pour permettre l'affichage d'un sapin décoré de guirlandes.

L'utilisateur devra introduire :

- la hauteur souhaitée (toujours entre `8` et `35`);
- le symbole utilisé pour dessiner les épines du sapin (1 caractère);
- le motif de guirlande à utiliser pour décorer le sapin (une chaîne entre `3` et `25` caractères). Les caractères utilisés pour les guirlandes devront différer de celui utilisé pour les épines.

Le programme redemandera les données nécessaires tant qu'elles ne sont pas correctes.

Les règles à utiliser pour la pose des guirlandes sont les suivantes :

- les guirlandes seront posées de gauche à droite, une ligne sur 2 de l'arbre en commençant par la seconde ligne (depuis le haut);
- des guirlandes seront placées autant de fois que possible sur une ligne, mais à chaque

fois séparées par des épines. Le nombres d'épines séparant les guirlandes est choisi aléatoirement pour être soit `2` soit `3` (vous pourrez utiliser `Math.random`);
- si la guirlande est trop longue pour être placée entièrement sur une ligne, elle sera "pliée" et continuera à décorer la ligne suivante de droite à gauche.

Voici un exemple d'exécution produit par cette partie du programme :

```
Quel symbole voulez-vous pour les épines du sapin? *
Combien de lignes (de 8 a 35)? 8
Quelles guirlandes voulez-vous mettre (taille de 3 a 25 caractères et elles
ne peuvent pas contenir le même caractère que celui utilisé pour les épines)?
0-0
       *
      0-0
     *****
    0-0**0-
   ********0
  0-0**0-0**0
 ***********0-
0-0***0-0**0-0*
```

Pour faire plus réaliste, vous pourrez enfin ajouter un tronc à votre sapin. La taille du tronc devra être proportionnelle à celle du triangle.

Voici un exemple d'affichage (hauteur 10, avec `'x'` pour les épines et `"1234"` pour les guirlandes) :

```
         x
        123
       xxxx4
      1234xx1
     xxxxxx432
    1234xxx1234
   xxxxxxxxxxxxx
  1234xx1234xx123
 xxxxxxxxxxxxxxxx4
1234xx1234xxx1234xx
        |||
        |||
        |||
```

Exercice 25 : Master-Mind(tm) (révisions)

Positionnement du problème

Le but ici est de réaliser un Master-Mind(tm) élémentaire.
Le programme tirera une combinaison, demandera à l'utilisateur de la retrouver sur la base de ses réponses successives.

Une combinaison sera représentée par une séquence de `n` chiffres tirés parmi `m` (typiquement `n=4` et `m=6`).
Par exemple, une combinaison pourrait être `3251`.

1. Création de la combinaison à trouver

Dans un fichier `MasterMind.java`, deux entiers `n` et `m` valent respectivement 4 et 6. Déclarez deux tableaux de `n` entiers appelés `laCombinaison`, destiné à stocker la combinaison à deviner, et `combinaison`, destiné à stocker la combinaison proposée par l'utilisateur.

On veut écrire la méthode `tirerCombinaison` qui tire une combinaison de `n` chiffres.
Pour cela, utilisez la méthode

```
static int hasard(int max) {
  return (1 + (int)(Math.random() * max));
}
```

qui tire un nombre au hasard entre 1 et `max` (qui par défaut vaut `m`).

En utilisant cette méthode `hasard`, écrivez la méthode `tirerCombinaison` décrite ci-dessus, ayant comme entête quelque chose comme :

```
static void tirerCombinaison(int[] uneCombinaison ..);
```

et qui affecte une combinaison au hasard à la variable `uneCombinaison`.

Note/Rappel : on utilise ici le fait qu'un tableau de taille fixe est toujours passé par référence et peut donc être directement modifié dans le corps de la méthode.

Remarque : Pour tester votre programme, il sera peut être utile de modifier la méthode `tirerCombinaison` de sorte à ce qu'elle retourne une combinaison connue choisie par vous pour vos tests.
Faites le simplement en écrivant "en dur" la combinaison que vous voulez utiliser pour vos tests.

2. Saisie de la réponse du joueur

On s'intéresse maintenant à pouvoir entrer une réponse du joueur.

Définissez la méthode `demanderCoup`, analogue à `tirerCombinaison`, et qui remplit `combinaison`, par des valeurs demandées à l'utilisateur.

Voici un exemple d'interaction (voir aussi l'exemple complet à la fin de l'énoncé) :

```
Entrez les 4 chiffres  de votre proposition:
1
2
3
4
```

On ne vous demande pas de sophistiquer les saisies au clavier : on suppose ici que l'utilisateur est coopératif et répond correctement (il donne bien 4 entiers).

3. L'os : génération de la réponse

On attaque maintenant la partie la plus difficile du programme : la génération de la réponse.

Pour ceux qui ne connaissent pas le Master-Mind(tm), une réponse est constituée d'indicateurs (au plus `n`) de 2 types :

1. des indicateurs correspondant aux chiffres corrects bien placés, c'est-à-dire les valeurs et positions exactement les mêmes entre les deux séquences.

   Par exemple, si la séquence à deviner est `1213` et que l'on a proposé `4516`, la réponse sera "*`1` correct*", ce qui correspond au deuxième `1`.

   Ce type de réponse sera noté par un '`#`'.

2. des indicateurs correspondant aux chiffres corrects mais mal placés, c'est-à-dire une valeur présente mais pas à la bonne position.

   Par exemple, si la séquence à deviner est `1213` et que l'on a proposé `4562`, la réponse sera "*`1` mal placé*", ce qui correspond au `2`.

   Ce type de réponse sera noté par un '`O`'.

Ces indicateurs sont cumulés pour tous les chiffres de la réponse.

Par exemple, si la séquence à deviner est `1213` et que l'on a proposé `4512`, la réponse sera "`#O`", ce qui correspond respectivement au `1` et au `2`.

Attention ! Un principe de base dans la génération des réponse est qu'un chiffre ne peut servir qu'**une seule fois** dans la réponse.

Par exemple, si la séquence à deviner est `1213` et que l'on a proposé `4156`, la réponse est "`O`" (c'est-à-dire un bon chiffre mais mal placé : le `1`) et non pas "`OO`" (pour le `1` deux fois mal placé) car il n'y a qu'<u>un seul</u> `1` dans la séquence proposée.
Par contre, si la solution proposée est `4151`, la réponse sera bien "`OO`".

Un **deuxième principe** est que l'on n'indique pas quels chiffres sont bons ou bien placés. On indique uniquement leur nombre.

Ainsi si la séquence à deviner est `4213` et que l'on a proposé `5243`, la réponse sera `##O` et non

pas `.#O#`.

Il s'agit maintenant de générer le code qui est capable de donner la réponse correspondant à une proposition de l'utilisateur.
On va pour cela écrire une méthode `compare`.

La méthode `compare` va naturellement prendre deux arguments : les deux combinaisons à comparer qui, dans l'appel, correspondront à la référence à trouver et à la solution proposée par le joueur.

Que doit retourner la méthode `compare` ?
Non seulement si oui ou non (booléen) la solution proposée est la bonne, mais aussi la réponse à donner, c'est-à-dire le nombre de chiffres corrects bien placés et le nombre de chiffres corrects mal placés. Ces deux derniers chiffres seront stockés dans un tableau à deux éléments appellé `reponse`:

le premier élément de `reponse` compte le nombre de chiffres corrects et bien placés, le second compte le nombre de bons chiffres mais mal placés (c'est-à-dire le nombre de chiffres de la proposition compris dans la solution mais à une autre place).

Ceci étant fait, l'entête de la méthode `compare` comportera au moins les paramètres suivants :

```
static boolean compare(int[]combinaison1, int[]combinaison2, int[] reponse);
```

qui modifie directement le tableau `reponse` en mémoire.

Pour construire la réponse de `compare` :

- Commencez par déterminer le nombre de chiffres corrects et bien placés et déterminer la valeur booléenne indiquant si la bonne combinaison a été trouvée ;
- Puis dans un second temps, déterminez le nombre de chiffres corrects mais mal placés.
- **Truc :** Pour éviter de "compter" plusieurs fois le même chiffre, utilisez un tableau de booléens de taille `n` qui "marque" les positions des chiffres de la solution proposée qui ont déjà eu une réponse (bien ou mal placé).

4. Mise en place du tout

Il ne reste plus qu'à assembler le tout !

Pour faciliter encore un peu plus l'écriture du code, écrivez deux méthodes d'affichage:

```
static void afficheCombinaison(int[] combinaison ..);
static void afficheReponse(int[] reponse..);
```

La première méthode affichera une `combinaison` en mettant un espace entre chaque chiffre.
Il vous est possible d'ajouter tous les paramètres vous semblant nécessaires.
Par exemple :

```
1 2 3 4
```

Elle servira à afficher la solution si l'utilisateur ne trouve pas.

La seconde méthode permet d'afficher la réponse à un coup.
Par exemple :

```
#
```

ou

```
OOO
```

ou

```
##OO
```

etc…

Pour finaliser le programme, écrivez, dans la méthode `main` le code suivant :

1. Tirer une combinaison au hasard
2. Si le nombre de coups est plus petit qu'un certain nombre (disons 10), demandez une proposition (incrémentez le nombre de coups), comparez la à la solution et affichez la réponse correspondante.
3. Répétez tant que le nombre de coups joués est plus petit que la constante définie ci-dessus et que le joueur n'a pas trouvé.
4. A la fin, décidez d'afficher le message de réussite ou d'échec en fonction du résultat du joueur (voir ci-dessous).

Un exemple d'exécution est donné plus bas.

Exemples d'exécution

Exemple positif :

```
Pourrez vous trouver ma combinaison de 4 chiffres [compris entre 1 et 6
avec répétitions possibles] en moins de 10 coups ?
Entrez les 4 chiffres de votre proposition:
1
2
3
4
#
Entrez les 4 chiffres de votre proposition:
5
6
1
2
OO
Entrez les 4 chiffres de votre proposition:
1
5
1
5
#O
Entrez les 4 chiffres de votre proposition:
```

```
1
1
6
6
###
Entrez les 4 chiffres de votre proposition:
1
1
6
1
####
Bravo ! Vous avez trouvé en 5 coups
```

Exemple négatif (raccourci) :

```
Pourrez vous trouver ma combinaison de 4 chiffres [compris entre 1 et 6
avec répétitions possibles] en moins de 10 coups ?
Entrez les 4 chiffres de votre proposition:
1
1
1
1
##
Entrez les 4 chiffres de votre proposition:
1
1
2
2
OO
...
Entrez les 4 chiffres de votre proposition:
4
6
1
1
##OO
Désolé vous n'avez pas trouvé...
La bonne réponse était 6 4 1 1.
```
