1 Récursivité croisée

On parle de récursivité *croisée* lorsque deux méthodes s'appellent l'une l'autre récursivement.
Les méthodes suivantes sont censées donner la parité d'un nombre entier : cela sera-t-il le cas pour toutes les valeurs entières positives ? Si ce n'est pas le cas, proposer une correction.

```
import fr.jussieu.script.Deug;

class PairImpair{

    static boolean pair (int n){
        if (n==0)
            return true;
        else
            return impair(n-1);
    }
    static boolean impair (int n){
        if (n==1)
            return true;
        else
            return pair(n-1);
    }

    public static void main (String[] args) {
        int p = Deug.readInt();
        Deug.println("pair? "+ pair(p) + " impair? " + impair(p));
    }
}
```

2 Exponentiation

En utilisant la propriété : $x^n = x.x^{n-1}$ (lorsque $n > 0$) et $x^0 = 1$, écrire en Java :

1. une méthode qui calcule x^n (où x et n sont des entiers), par la méthode itérative (sans appel récursif) ;
2. une méthode qui calcule x^n (où x et n sont des entiers), en utilisant la récursion.

3 Exponentiation rapide

L'algorithme dit d'*exponentiation rapide* permet de calculer la $n^{\text{ème}}$ puissance d'un nombre plus efficacement qu'en le multipliant n fois par lui-même. Il repose sur les deux faits suivants :

$$\begin{aligned} x^{2n} &= (x^n)^2 \\ x^{2n+1} &= x\,(x^n)^2 \end{aligned}$$

1. Écrire en Java une méthode récursive prenant deux entiers x et n en argument, et renvoyant en résultat x^n, en appliquant récursivement celle des deux égalités qui correspond.
2. Supposons que l'on calcule 2^{65} à l'aide de votre méthode. Donner la suite des appels récursifs effectués, avec leur imbrication.
3. Combien y a t-il d'appels à la méthode provoqués lors du calcul de 2^{16} ? 2^{32} ? 2^{64} ? et par la méthode de l'exercice précédent ?

4 Fibonacci

La suite de Fibonacci, due à Leonardo Pisano, Léonard de Pise dit Fibonacci, est définie de façon à calculer le nombre d'individus d'une population évoluant de la façon suivante :
– on suppose que les individus ne meurent jamais ;
– au début il n'y a qu'un couple d'individus (non pubère) ;
– les individus deviennent pubères au bout de deux mois ;
– un couple d'individus pubères engendrent chaque mois un couple d'individus non pubères.

1. Combien de couples existe t-il au bout de n mois ?
2. Trouver la récurrence, et écrire en Java la version récursive du calcul de $F(n)$.
3. Comparer ce type de génération avec celle des hyménoptères.

5 Pyramide

Dans cette exercice, toutes les méthodes demandées doivent être *récursives*.

1. Écrire une méthode `String repete(int n, String s)` renvoyant la chaîne de caractères s répétée n fois : `repete(3, "bla")` donne `"blablabla"`.
2. Écrire une méthode `void pyramide( int n, String s)` qui écrit sur la première ligne, 1 fois la chaîne `s`, sur la deuxième, 2 fois la chaîne `s`, et ainsi de suite jusqu'à la dernière ligne, où il y aura `n` fois la chaîne `s`. Ainsi `pyramide(5, "bla");` donnera

   ```
   bla
   blabla
   blablabla
   blablablabla
   blablablablabla
   ```

3. Quand on lance `pyramide(n,s)`, combien y a-t-il d'appels à `pyramide` ? Combien y a-t-il d'appels à `repete` ? Dessiner l'arbre des appels pour `n` = 3.
4. Comment faire pour obtenir la pyramide *inverse* de la précédente ? Toujours en utilisant uniquement des appels récursifs...

6 Ackermann-Péter

On définit la fonction d'Ackermann-Péter de la façon suivante :

$$A(m, n$$

On dispose également d'un constructeur par défaut qui permet de créer une liste vide. On considère les deux méthodes suivantes :

```
static Liste temp (Liste l1, Liste l2){
   if (l1.estVide()) return l2;
   else l2.ajouteEnTete(l1.premier());
   return temp(l1.suivant(),l2);
}

static Liste mystere (Liste l){
   return temp(l,  new Liste());
}
```

1. que renvoie un appel à `mystere(new Liste())` ?
2. que renvoie un appel à `mystere(l)`, si `l` est une liste à un élément de valeur 12 ?
3. que renvoie un appel à `mystere(l)`, si `l` est une liste à deux éléments : le premier de valeur 17 et le second de valeur 28 ?
4. que calcule la méthode **mystere** ?

8 Sous-suites maximales

Une *sous-suite* d'un mot u est un mot w obtenu à partir de u en effaçant des lettres et en respectant l'ordre. Par exemple, ara est une sous-suite des mots $quadratique$, $marchandage$ et $alcatraz$, mais n'est pas une sous-suite de raa.
On cherche une méthode qui retourne la longueur maximale d'une sous-suite commune à deux mots u et v donnés.

1. Dans un premier temps, on cherche à en donner une version récursive. On note $LSSM(i, j)$ la longueur maximale d'une sous-suite commune au mot formé des i premières lettres de u avec celui formé des j premières lettres de v.

   Que vaut $LSSM(i, 0)$? $LSSM(0, j)$?

   On note u_i la i-ème lettre du mot u. Supposons que $u_i = v_j$.
   Exprimer $LSSM(i, j)$ en fonction de $LSSM(i - 1, j - 1)$.

   Supposons maintenant que $u_i \neq v_j$.
   Exprimer $LSSM(i, j)$ en fonction de $LSSM(i - 1, j)$ et de $LSSM(i, j - 1)$.

   En déduire une méthode récursive naïve pour le problème. Pour $u = abac$ et $v = cbc$, dessiner l'arbre des appels récursifs et donner les états successifs de la pile.
2. Implémenter une version itérative puis une version récursive *dynamique* de cette méthode, en mémorisant les valeurs des sous-problèmes dans un tableau à deux dimensions. Construire le tableau pour les deux versions lorsque $u = abac$ et $v = cbc$.