IF1 - Corrigé du partiel du 26 novembre 2005

Exercice 1.

1.1. Représentation en base 8 du nombre s'écrivant `7B4E60A8` en base 16.

On passe par la représentation binaire du nombre dans laquelle des espaces séparent la représentation binaire des différents chiffres hexadécimaux :

```
0111 1011 0100 1110 0110 0000 1010 1000
```

Le regroupement des chiffres par 3 à partir de la droite donne la représentation binaire des chiffres de la repésentation octale du nombre :

```
01 111 011 010 011 100 110 000 010 101 000.
```

On en déduit la réprésentation octale du nombre : `17323460250`

1.2. Soit $n = 35 + \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{64} + \frac{1}{1024}$

1.2.1. Le nombre n est finiment représentable en base 2 par `100011,1010110001` car

- $35 = 32 + 2 + 1 = 2^5 + 2^1 + 2^0$, ce qui donne en binaire 100011
- $\frac{1}{2} = 2^{-1}$, $\frac{1}{8} = 2^{-3}$,$\frac{1}{32} = 2^{-5}$, $\frac{1}{64} = 2^{-6}$ et $\frac{1}{1024} = 2^{-10}$,
  ce qui donne en binaire pour $\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{64} + \frac{1}{1024}$ la représentation $0,1010110001$.

1.2.2. Le nombre $n + \frac{7}{10}$ n'est pas finiment représentable en base 2. En effet, $\frac{7}{10} = 0,7$ ne peut pas s'écrire comme somme de puissances négatives de 2. Sa représentation sous forme binaire obtenue par multiplications successives par 2 est infinie :

```
0,10110011001100110011001100110011 ... 0011 ... 0011 ...
```

1.3. Simuler l'exécution de la séquence Java suivante :

```
int m = 3, n = 2, p;
float f, g;
p = 2 * m + n;    m = n + 3;    n = 5;
Deug.println(p); Deug.println(m); Deug.println(p);
m = p / n;     f = p / n;
Deug.println(m); Deug.println(f);
g = n;   f = p / g;
Deug.println(f);
```

On obtient :

```
8    <-- p = 2 * 3 + 2 = 8
5    <-- m prend la valeur actuelle de n (c-à-d 2 augmenté de 3) soit 5
5    <-- n prend 5 comme nouvelle valeur
1    <-- p et n étant des int, la division est entière (quotient de 8/5 = 1)
1.0  <-- même chose, mais le résultat est de type float
1.6  <-- ici, g de type float => une division non entière (8/5 = 1.6)
```

de la variable d après exécution de la séquence suivante et en déduire une séquence plus simple équivalente :

```
boolean a, b, c, d;
  .....
if (!b) c = !a;
else c = true;
if (c)
   if (!a) d = !b;
   else d = true;
else d = false;
```

La table suivante résume ce qui se passe en fonction des valeurs de a et de b :

a	b	!b	c	!a	d
false	*false*	*true*	*true*	*false*	*true*
false	*true*	*false*	*true*	*true*	*false*
true	*false*	*true*	*false*	*false*	*false*
true	*true*	*false*	*true*	*false*	*true*

On peut donc remplacer cette séquence par `d = (a == b);`

———

Exercice 2.

On dispose de m pièces de 50 centimes, de n pièces de 20 centimes et de p pièces de 10 centimes (p non nul).

2.1. Expression arithmétique pour la somme maximale payable (les parenthèses peuvent être omises puisque la multiplication est prioritaire sur l'addition) :

```
(50 * m) + (20 * n) + (10 * p)
```

2.2. Expression logique exprimant le fait qu'une somme s n'a aucune chance de pouvoir être payée.

L'expression suivante traduit le fait que s ne peut être payée si elle n'est pas multiple de 10 ou si elle est supérieure à la valeur maximale dont l'expression a été donnée précédemment (ou si elle est négative).

```
((s % 10) != 0) || (s > (50 * m) + (20 * n) + (10 * p)) || (s < 0)
```

Une condition nécessaire pour qu'une somme soit susceptible d'être payée est la négation de la précédente, c'est-à dire

```
!(((s % 10) != 0) || (s > (50 * m) + (20 * n) + (10 * p)) || (s < 0))
```

ou encore

```
((s % 10) == 0) && (s <= (50 * m) + (20 * n) + (10 * p)) && (s >= 0)
```

2.3. Stratégie gloutonne pour le paiement.

- le nombre de pièces de 50 centimes qu'on va utiliser est au plus le nombre de pièces de ce type dont on dispose et au plus le quotient de la division de s par 50.
  Il est donc égal à $n50 = min(m, s/50)$ où min renvoie la valeur du plus petit de ses deux arguments.

de ce type dont on dispose et au plus le quotient de la division de la somme restant à payer après utilisation des pièces de 50 centimes.
Il est donc égal à $n20 = min(n, (s - 50 * n50)/20)$.

- les pièces de 10 centimes doivent permettre de payer la somme restante, c'est-à-dire $s - (50 * n50 + 20 * n20)$.
  Si cette somme est supérieure à $10 * p$, le paiement n'est pas possible et sinon, le nombre de pièces de 10 centimes à utiliser est $(s - (50 * n50 + 20 * n20))/10$.

2.4. Programme complet

```
import fr.jussieu.script.*;
class Paiement{
   static boolean verif(int m, int n, int p, int s) {
       if( m < 0 || n < 0 || p < 1)
          return false;
       if (s % 10 != 0 || s > 50 * m + 20 * n + 10 * p || s < 0)
          return false;
       return true;
   }
   public static void main(String[] st){
   int m, n, p, s;
   int n50, n20, n10, reste;
   Deug.print("Nombre de pieces de 50 cts ? "); m = Deug.readInt();
   Deug.print("Nombre de pieces de 20 cts ? "); n = Deug.readInt();
   Deug.print("Nombre de pieces de 10 cts ? "); p = Deug.readInt();
   Deug.print("Somme a payer ? "); s = Deug.readInt();
   if(!verif(m, n, p, s)) {
       Deug.println("Recherche de solution inutile");
       Deug.exit(); }
   // calcul nombre de pieces de 50 cts à utiliser
   n50 = s / 50;
   if (n50 > m) n50 = m;
   reste = s - 50 * n50; // somme restant à payer sans les pieces de 50
   // calcul nombre de pieces de 20 cts à utiliser
   n20 = reste / 20; // nombre de pieces de 20 cts à utiliser
   if (n20 > n) n20 = n;
   reste = reste - 20 * n20; // reste à payer sans les pieces de 50 et de 20
   // calcul nombre de pieces de 10 cts à utiliser
   n10 = reste / 10;
   if (n10 > p) {
       Deug.println("Impossible de payer la somme");
       Deug.exit(); }
   Deug.println("Paiement avec " + n50 + " " + n20 + " " + n10);
   }
}
```

3.1. Comptage des alternances dans un tableau t

```
static int altern(int[] t) {
  // on suppose que le tableau existe et est de taille non nulle
  int alt = 0; // le compteur d'alternances
  int val = t[0]; // la valeur courante pour chercher une alternance
  for(int i = 1; i < t.length; i++)
     if(t[i] != val) { // il y a alternance
        alt = alt + 1; // augmentation du compteur d'alternances
        val = t[i]; // pour rechercher l'alternance suivante
     }
  return alt;
}
```

Une solution plus simple consiste à comparer chacun des éléments du tableau avec son successeur, ce qui conduit à la solution suivante :

```
static int altern(int[] t) {
int res = 0;
for(int i = 0; i < t.length - 1; i++)
   if(t[i] != t[i+1]) res ++;  // res = res + 1;
return res;
}
```

3.2. Taille du tableau étendu correspondant à un tableau factorisé

```
static int taille(int[] t) {
  // on suppose que le tableau existe et est de taille non nulle
  // si le tableau est de taille impaire,
  // ou si t[0] < 1, on renvoie -1
  if ( (t.length % 2 == 1) || (t[0] < 1) )
      return -1;
  int val = t[1]; // valeur courante pour comparer avec valeur suivante
  int size = t[0]; // t sera la taille du tableau sous forme étendue
  for(int i = 2; i < t.length; i = i + 2) {
     // si le couple courant est incompatible, on renvoie -1
     // nombre d'éléments < 1 et valeur égale à la précédente}
     if( (t[i] < 1) || (t[i + 1] == val) )
         return -1;
     size = size + t[i]; // mise à jour de la taille
     val = t[i + 1]; // mise à jour de la valeur du couple
   }
  return size;
}
```

3.3. Écriture dans le tableau t de val, n fois à partir de l'indice ind

```
static void ecrireVal(int[] t, int deb, int n, int val) {
  for(int i = 0; i < n; i++)
     t[deb + i] = val;
}
```

```
static int[] factoriser(int[] t1) {
  // on suppose que le tableau existe et est de taille non nulle
  int n = 2 * (altern(t1) + 1); // la taille du tableau factorisé
  int[] t2 = new int[n]; // allocation du tableau factorisé
  int val = t1[0]; // la prochaine valeur à écrire dans t2
  int nb = 1; // le nombre de fois qu'elle apparaît consécutivement
  int ind = 0; // numéro du prochain couple à écrire dans t2
```

```
        if(t1[i] != val) {
             // alternance atteinte : il faut donc écrire le couple dans t2
             t2 [2 * ind] = nb;  // nombre d'élements identiques
             t2 [2 * ind + 1] = val; // valeur commune des élements
             val = t1[i];    // valeur pour la nouvelle séquence
             nb = 1;        // longueur de la nouvelle séquence
             ind = ind + 1; // numéro du prochain couple
             }
        else  // on reste dans la même séquence
             nb = nb + 1; // on augmente donc sa longueur de 1
    // écriture du dernier couple
    t2 [2 * ind] = nb;  // nombre d'élements identiques
    t2 [2 * ind + 1] = val; // valeur commune des élements
    return t2; // on renvoie la référence du tableau factorisé
  }
```

3.5. Construction du tableau étendu correspondant à un tableau factorisé

```
static int[] etendre(int[] t2) {
   int size =  taille(t2);
   // si t2 n'est pas un tableau factorisé, on renvoie -1
   if (size == -1)
       return null;
   int[] t1  = new int[size]; // allocation du tableau étendu
   int ind = 0; // position prochaine écriture dans t1
   for (int i = 0; i < t2.length; i = i + 2){
       ecrireVal(t1, ind, t2[i], t2[i+1]);
       ind = ind + t2[i];
   }
   return t1;
}
```