Université Denis Diderot - DEUG MASS/MIAS - IF121
Corrigé du partiel du 22 novembre 2003

Exercice 1.

Il s'agissait d'une question de cours traitée ci-après sur l'exemple `110010101000`.

↪ la valeur décimale du nombre est obtenue par :

$$0 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 + 0 \times 2^8 + 0 \times 2^9 + 1 \times 2^{10} + 1 \times 2^{11}$$

ce qui donne : $8 + 32 + 128 + 1024 + 2048$, c'est-à-dire `3240`

↪ pour passer à l'expression en base seize, plutôt que de partir de la forme en base 10 et diviser successivement par seize, on fait des paquets de 4 chiffres dans la forme binaire en commençant par la droite et on exprime chacun de ces paquets comme chiffre hexadécimal : `1100 1010 1000`, soit `CA8`

↪ pour passer à l'expression en base huit, plutôt que de partir de la forme en base 10 et diviser successivement pas huit, on fait des paquets de 3 chiffres dans la forme binaire en commençant par la droite et on exprime chacun de ces paquets comme chiffre de la base huit : `110 010 101 000`, soit `6250`

↪ la représentation en machine sur 16 bits de l'opposé du nombre donné est obtenu en complément à 2, c'est-à-dire que le nombre n est représenté par $2^{16} - n$:

```
   10000000000000000
-         110010101000
--------------------
=  01111001101011000
```

dont on ne garde que les seize bits de droite, soit `1111001101011000`.

D'un point de vue pratique, pour l'obtenir, on recherche le `1` le plus à droite dans le nombre n, et on bascule chacun des bits à sa gauche (tout bit `1` devient un bit `0` et inversement).

Exercice 2. Solution complète, commentée mais donnée « brutalement » :

```
import java.jussieu.script.*;
public class Exo2{
   public static void main(String[] args){
      int a, b; // déclarations des deux variables entières
      double hypo; // variable pour l'hypoténuse du triangle
      double peri; // variable pour le paramètre du triangle
      double aire; // variable pour l'aire du triangle
      Deug.print("Donnez un petit côté : ");
      a = Deug.readInt( );  // lecture d'un petit côté
      Deug.print("Donnez un petit côté : ");
      b = Deug.readInt( );  // lecture de l'autre petit côté
      if ((a <= 0) ||  (b <= 0)) {
          Deug.println("Erreur sur la valeur des côtés");
          Deug.exit(); }   // fin du if
      hypo = Math.sqrt(a*a + b*b); // théorème de Pythagore pour l'hypoténuse
      perim = a + b + hypo; // perimetre = somme des 3 côtés
      aire = (a * b) / 2.0;  // le triangle est un demi-rectangle.
                             // Attention division par 2.0 (donc non entière)
      Deug.println("Le périmètre est " + peri);
      Deug.println("L'aire est " + aire);
   }  // main
}  // class
```

Exercice 3.

3.1. La table de vérité de l'opérateur contient 8 lignes (on représente Vrai par `V` et Faux par `F`) :

a	b	c	maj(a,b,c)
F	F	F	F
F	F	V	F
F	V	F	F
F	V	V	V
V	F	F	F
V	F	V	V
V	V	F	V
V	V	V	V

3.2. On peut par exemple utiliser la forme canonique disjonctive, qui s'écrit en Java , en supposant les variables `a`, `b` et `c` de type `boolean` :

```
((!a)&&b&&c) || (a&&(!b)&&c) || (a&&b&&(!c)) || (a&&b&&c)
```

Des formes plus compactes peuvent en être données comme par exemple :

```
(a&&b) || (a&&c) || (b&&c)
```

« ou mieux », mais pas symétrique :

```
(a&&(b||c)) || (b&&c)
```

ou encore

```
(a&&b) || (c && (a!=b))
```

qui est tout à fait correcte mais ne répond pas exactement à la question car on y utilise un opérateur non booléen

3.3. La solution donnée ci-dessous utilise systématiquement les accolades { et } :

```
....   //  tout ce qu'il faut avant (import, class, main, ....)
// on suppose que a, b et c sont des variables de type boolean initialisées
boolean maj = false;
if (a) {
   if (b) { maj = true; }
   else   { maj = c; }
   }
else if (b) { maj = c; }
....   //  tout ce qu'il faut après
```

Exercice 4.

Dans la solution la plus simple, il y a une seule boucle et le tableau `tab` n'est parcouru qu'une seule fois :

```
....   //  tout ce qu'il faut avant (import, class, main, ....)
// le tableau tab est supposé initialisé
// déclaration et allocation du tableau position à la même taille que tab
int[] position = new int[tab.length];
for(int i=0; i<tab.length; i++) {
   position[tab[i]] = i;
   }
....   //  tout ce qu'il faut après
```

On peut évidemment faire plus compliqué avec deux boucles, comme dans la solution suivante (on a relâché l'usage des accolades) :

```
....   //  tout ce qu'il faut avant (import, class, main, ....)
// le tableau tab est supposé initialisé
// déclaration et allocation du tableau position à la même taille que tab
int[] position = new int[tab.length];
for(int i=0; i<tab.length; i++)
     for(int j=0; i<tab.length; j++)
          if(tab[j] == i)
               position[i] = j;
....   //  tout ce qu'il faut après
```

La solution peut être améliorée en utilisant l'instruction `break` pour sortir de la boucle intérieure contrôlée par la variable `j` une fois qu'on a trouvé la place de `i` :

```
....   //  tout ce qu'il faut avant (import, class, main, ....)
// le tableau tab est suppose initialisé
// déclaration et allocation du tableau position à la même taille que tab
int[] position = new int[tab.length];
for(int i=0; i<tab.length; i++)
     for(int j=0; i<tab.length; j++)
          if(tab[j] == i){
               position[i] = j;
               break;
               }
....   //  tout ce qu'il faut après
```

Exercice 5.

Dans la solution suivante, on a calculé préalablement la masse totale : le test de nullité de la masse totale est fait par rapport à une valeur *epsilon* transmise en paramètre afin d'éviter un test par rapport à `0.0` (mais ce n'était pas demandé dans l'examen). Dissocier le calcul de la masse totale de celui des coordonnées du barycentre est intéressant s'il y a un grand nombre de points et que la somme de leur masse est nulle.

On suppose par ailleurs que les masses sont toutes positives et que les tableaux `abscisses`, `ordonnees`, `cotes` et `masses` sont de même taille.

```
....   //  tout ce qu'il faut avant (import, class, main, ....)
public static void barycentre(double[] abscisses, double[] ordonnees,
                              double[] cotes, double masses[],
                              double epsilon) {
double abscisse = 0,  // abscisse du barycentre
       ordonnee = 0,  // ordonnée du barycentre
       cote = 0,      // cote du barycentre
       masse = 0;     // masse totale
// calcul de la somme des masses
for(int i = 0; i < abscisses.length; i++) {
      // cumul de la masse du point i dans la masse totale
      masse = masse + masses[i];
      }
if (masse < epsilon) {
      Deug.print("Somme des masses assimilable a 0 : ");
      Deug.println("Pas de calcul du barycentre");
      Deug.exit();
      }
```

```
// calcul des coordonnées du barycentre
for(int i = 0; i < abscisses.length; i++) {
      abscisse = abscisse + masses[i] * abscisses[i];
      ordonne = ordonnee + masses[i] * ordonnees[i];
      cote = cote + masses[i] * cotes[i];
      }
abscisse = abscisse / masse;
ordonnee = ordonnee / masse ;
cote  = cote / masse;
Deug.println("Abscisse du barycentre " + abscisse);
Deug.println("Ordonnée du barycentre " + ordonnee);
Deug.println("Cote du barycentre " + cote);
}
....   //  tout ce qu'il faut après
```

Exercice 6.

Dans ce qui suit, le degré du polynôme représenté par le tableau `pol` est `pol.length-1`.

6.1. La première solution donnée ci-dessous ne contient aucune subtilité. Pour chaque valeur de i , elle calcule α^i sans utiliser de résultat de calcul préalable :

```
public static double valeurPolynome (double[] pol, double alpha) {
     double valeur = pol[0];
     for(int i = 1; i < pol.length; i++)
          valeur = valeur + (pol[i] * Math.pow(alpha, i));
     return valeur;
}
```

La solution suivante n'utilise pas la fonction **Math.pow**. Elle calcule α^i par multiplications successives de α en conservant dans la variable `puissance` la valeur de α^i pour calculer α^{i+1} :

```
public static double valeurPolynome (double[] pol, double alpha) {
     double valeur = pol[0];
     double puissance = 1;
     for(int i = 1; i < pol.length; i++) {
          puissance = puissance * alpha;
          valeur = valeur + pol[i] * puissance ;
          }
     return valeur;
}
```

Enfin, la dernière solution, la plus performante, connue sous le nom de méthode Horner, évalue le polynôme par lecture des coefficients par degrés décroissants et non plus par degrés croissants comme dans les deux solutions précédentes :

```
public static double valeurPolynome (double[] pol, double alpha) {
      double valeur = 0;
      for(int i = pol.length - 1; i >= 0; i--)
           valeur = (valeur * alpha) + pol[i];
      return valeur;
}
```

6.2. La solution ci-dessous est sans aucune subtilité : elle calcule successivement la valeur du polynôme en tous les réels multiples de `0,001`, en partant de `0,0`, au mooyen d'un boucle `for`. Initialement on a $P(0) > 0$. La première fois qu'elle trouve une valeur x tel que $P(x) \leq 0$, on a encore $P(x - 0,001) > 0$, donc le zéro du polynôme est compris entre $(x - 0,001)$ et x.

```
//  on suppose définie la fonction valeurPolynome précédente
double [] pol;
....   // on suppose le tableau pol
       // contient les coefficients du polynôme
double val,     // valeur du polynôme
       x = 0;   // pour calculer la solution
for(i = 1; i <= 1000; i++) {
    x = i * 0.001; // ou  encode : x = x + 0.001;
    val = valeurPolynome(pol, x);
    if(val <= 0) {
          break;   // on sort de la boucle : la solution est trouvée
          }
    }
Deug.println("Le zéro du polynôme est " + x + "à 0,001 près par excès");
....   //  tout ce qu'il faut après
```

Une autre écriture avec la même stratégie est évidemment possible avec une boucle `while`.

Il existe également d'autres façons de procéder telle que la dichotomie mais c'est une autre histoire.

Barème appliqué :

	exo1	exo2	exo3	exo4	exo5	exo6	total
total	7	13	11	12	11	13	67
décomposition	1+2+2+2	—	3+4+4	—	—	—	—

Quelques erreurs couramment rencontrées et facilement évitables quand on écrit des programmes :

↪ utilisation de `=` à la place de `==` pour l'opérateur d'égalité et de $\times$ à la place de `*` pour la multiplication ;

↪ utilisation de `^` pour l'évaluation à la puissance (l'opérateur `^` est un opérateur réalisant le ET bit-à-bit sur ses deux opérandes) ;

↪ utilisation de caractères inexistants sur un clavier tels que α, $\sum$, x_G, a^2, $\frac{1}{m}$, $\sqrt{a^2 + b^2}$

↪ l'opérateur `/` appliqué à deux entiers fait une division entière (`3/2` donne `1` alors que `3/2.0` donne `1.5`) ;

↪ ne pas oublier les parenthèses `()` dans les appels de fonction sans paramètre ;

↪ une valeur de retour de fonction de la forme `return x,y,z` n'a pas de sens (ou probablement pas celui espéré par son auteur)
Une séquence de la forme `return x ; return y ; return z ;` n'a quant à elle aucun sens (ce qui suit un `return` ne sera par définition pas exécuté) ;

↪ l'instruction `if (a && b > 0)` ne permet pas de tester que a et b sont tous les deux positifs ;

↪ l'affectation d'une valeur de retour à une fonction

```
static boolean fonc(int a, int b, int c)
```

sous la forme

```
fonc(a, b, c) = valeur ;
```

est évidemment incorrecte.

↪ ne pas utiliser de variables non initialisées dans des expressions ;

↪ lorsqu'il est dit

« Écrire une fonction `barycentre`, qui recevant en paramètres quatre tableaux `abscisses`, `ordonnees`, `cotes` et `masses` »

cela signifie que les différents tableaux sont supposés être complètement définis (allocations et initialisations donc réalisées) avant l'entrée dans la fonction (il n'y a donc rien de ce type à faire dans la fonction à écrire).

↪ ce n'est pas une faute, mais si `a` est de type `boolean`, le test `if(a==true)` est avantageusement remplacé par `if(a)`