Université Paris Diderot – Paris 7
Introduction à l'informatique et à la programmation (IF1)
Corrigé du partiel du 8 décembre 2007

Prière de ne pas utiliser de rouge dans vos copies, mais de soigner la présentation et de bien détacher chaque exercice. Pour ceux qui ne l'avaient pas remarqué, nous avons utilisé le langage Java en cours, en cours-TD et en TP, et dans l'énoncé "methode" signifie "méthode Java", ou encore "fonction Java".

Exercice 1 **1.1 :** Donner la représentation en base 2 de l'entier 108.
Quelles sont ses représentations en base 8 et en base 16 ?

1.2 : Donner la représentation machine de l'entier -108 comme valeur de type `short` (codée sur 16 bits).

1.3 : On veut stocker un entier compris entre 0 et 2^{60} (utilisés pour la cryptographie). Que se passe-t-il si on utilise le type `int` ? Les types `float` et `double` ? Quel autre type peut-on utiliser ?

1.1 : On fait une série de divisions par 2, et on observe le reste à chaque fois :
$108 = 54*2+\mathbf{0}$; $54 = 27*2+\mathbf{0}$; $27 = 13*2+\mathbf{1}$; $13 = 6*2+\mathbf{1}$; $6 = 3*2+\mathbf{0}$; $3 = 1*2+\mathbf{1}$; $1 = 0*2+\mathbf{1}$
D'où 108 s'écrit 1101100 en base deux (on peut vérifier $108 = 64 + 32 + 8 + 4$).
On groupe les bits 3 par 3 à partir du bit de poids faible : 1 101 100 on obtient 154 en base 8.
On groupe les bits 4 par 4 : 110 1100, on obtient $6C$ en base 16

1.2 : L'entier 108 (positif) codé sur 16 bits s'écrit 0000000001101100. On inverse chaque bit : 1111111110010011 puis on ajoute 1 : 1111111110010100 est la représentation machine de l'entier -108 comme valeur de type `short`.

1.3 : Les entiers de type `int` sont codés sur 4 octets, soit 32 bits, leur valeurs sont comprises entre -2^{31} et $2^{31} - 1$, et les calculs sont faits modulo 2^{32}. Pour stocker un entier entre 0 et 2^{60}, il faut au moins 61 bits. Avec le type `int` on va perdre les bits de poids fort et ne garder que le reste modulo 2^{32}.

Les types `float` et `double` permettent, grâce à l'exposant, de représenter des nombres aussi grands que 2^{60}, mais même le type `double` n'est pas assez précis pour conserver la valeur exacte d'un nombre comme $2^{60} - 5$: il n'y a que 52 bits dans la mantisse. On perd donc de la précision.

On peut utiliser le type `long`, codé sur 64 bits en complément à 2, donc pour toutes les valeurs entre -2^{63} et $2^{63} - 1$.

Exercice 2 Exécuter à la main la séquence d'instructions suivante et en particulier indiquer ce qui est affiché à l'écran.

```
...
int a=3, b=-2, c=5;
float x,y=2;
c=a*b;           b=a+1;
Deug.println(c); Deug.println(b); Deug.println(a);
x=a/2;
Deug.println(x); Deug.println(a/2); Deug.println(a/y);
...
```

Résultats affichés :
`-6` La dernière valeur affectée à `c` est `a*b`, quand `a` vaut 3 et `b` vaut -2
`4` voir la dernière affectation à `b`
`3`
`1.0` Dans `a/2`, les deux opérandes sont de type `int`, donc on obtient le quotient de la division entière, soit 1. Ensuite comme `x` est de type `float`, cette valeur est affichée `1.0`
`1` Division entière encore
`1.5` Ici `y` est de type `float`, donc la division est "à virgule".

Exercice 3 Julien veut préparer son sac pour partir en randonnée. Selon les conditions, il doit emporter plus ou moins de matériel.

- Si la température est inférieure à 15 degrés, il doit prendre des vêtements supplémentaires, qui pèsent au total 1,5 kg.
- Si la pluie est annoncée, il emporte une cape de pluie de 0,5 kg, sauf si la température est supérieure à 25 degrés.
- Il prend une gourde (pleine) de 1 kg, et si la température dépasse 30 degrés, il en prend une deuxième.
- Si son ami Max l'accompagne, c'est Max qui apporte le casse-croûte, sinon Julien doit porter 0,4 kg de plus.
- Si la pluie n'est pas annoncée, Julien prolongera un peu la randonnée, il prend donc 0,2 kg d'aliments supplémentaires (sauf si c'est Max qui apporte le casse-croûte).

Pour chaque paramètre (température, présence de Max, pluie), expliquer quel type de donnée Java est adapté à sa représentation (`int, float, double, boolean, char, String`). Écrire une méthode qui prend en arguments la température, le risque de pluie, la présence de Max et qui renvoie le poids du sac à dos.

La température est représentée par un `double` (ou un `float`), la présence de Max et le risque de pluie par des `boolean` ("max vient" et "il peut pleuvoir"). Le poids est représenté par un `double` (ou un `float`).

```
public static double exercice3(double temp, boolean Max, boolean pluie) {
     double poids = 0; // poids du sac à dos
     if (temp<15)
         poids += 1.5; // vêtements
     if (pluie && (temp<=25))
         poids += 0.5; // cape de pluie
     if (temp>30)
         poids += 2; // deux gourdes
     else
         poids += 1; // une gourde
        if (!Max) { // Si Max n'est pas là il faut apporter le casse-croute
         poids += 0.4; // simple casse-croute
         if (!pluie)
             poids += 0.2; // complément
         }
     return poids;
}
```

Exercice 4 *Cet exercice sera traité sans utiliser de tableau*

Écrire un programme qui demande à l'utilisateur un entier `n`, puis lit une suite de `n` entiers, et affiche combien de fois l'utilisateur a entré deux nombres *successifs* égaux. Les nombres seront lus et traités successivement. Par exemple :

- pour 5 puis la suite $5, 3, 3, 5, 7$ le programme doit répondre 1.
- pour 4 puis la suite $1, 5, 5, 5$ le programme doit répondre 2.
- pour 7 puis la suite $8, 6, 6, 1, 6, 6, 6$ le programme doit répondre 3.

```
public static void main(String[] args) {
    int a,b; // avant dernière et dernière valeurs lues
    int fois=0; // compteur pour le nombre de fois que ...
    int n=Deug.readInt(); // nombre d'entiers à lire
    if (n>0) {
         a=Deug.readInt();  // première valeur lue
         for(int i=1; i<n; i++) { // restent n-1 valeurs à lire
             b=Deug.readInt();
             if (a==b)
                  fois++;
             a=b; // On conserve la valeur dans a
         }
         Deug.println("Nombre d'entiers successifs égaux : "+fois);
    }
}
```

Exercice 5 **5.1 :** Écrire une méthode `decale1` qui reçoit en paramètre un tableau d'entiers `t` (à une dimension) et décale les éléments d'une position vers la gauche de façon circulaire (le premier élément prend la place du dernier).

Si le tableau est initialement `t=` | 5 | 2 | 3 | 0 | 8 | 7 | 3 | 1 |,

la méthode le transforme en | 2 | 3 | 0 | 8 | 7 | 3 | 1 | 5 |.

Cette méthode doit modifier le tableau qu'elle reçoit, et ne renvoie rien.

5.2 : On veut écrire de deux façons différentes une méthode `decaleD` qui reçoit en paramètre un tableau d'entiers `t` et un entier `d` positif et décale les éléments de `d` positions vers la gauche de façon circulaire.

Si on appelle `decaleD(t,4)` et que `t` est le tableau donné initialement ci-dessus, il est transformé en | 8 | 7 | 3 | 1 | 5 | 2 | 3 | 0 |.

- *Première version* : écrire une méthode `decaleD` qui utilise la méthode `decale1`.
- *Deuxième version* : écrire une méthode `decaleD2` qui utilise un tableau auxiliaire, où chaque élément sera écrit une seule fois, puis qui recopie ce tableau dans le tableau initial.

5.1 :

```
public static void decale1(int[] t) { // Exercice 5
    int aux= t[0]; // on met de côté le premier élément
    for(int i=0; i<t.length-1;i++)
         t[i]=t[i+1];
    t[t.length-1]=aux;
}
```

5.2 :

```
public static void decaleDprem(int[] t, int d) {
    for(int i=1;i<=d;i++)
             decale1(t);
}
public static void decaleDdeux(int[] t, int d) {
    int[] tab=new int[t.length]; // création du tableau auxiliaire, de même taille
    for(int i=0; i<t.length;i++)
         tab[i]=t[ (i+d)%t.length ]; // décalage modulo t.length
    for(int i=0; i<t.length;i++)
         t[i]=tab[i];  // puis recopie
}
```

Exercice 6 On veut ici modéliser le jeu de l'Awélé (ou Awalé). Il est composé d'une suite de cases qui contiennent des graines, et est représenté par un tableau d'entiers. Chaque élément du tableau représente le nombre de graines contenues dans la case correspondante. Ainsi une case vide sera associée à un élément du tableau qui vaut zéro.

6.1 : Écrire une méthode `vide` qui renvoie le nombre de cases vides dans un état du jeu donné.

6.2 : Écrire une méthode `repartirFin` qui reçoit un tableau d'entiers `t` et un indice `i`, qui enlève les graines la case d'indice `i` et les répartit dans les cases suivantes à raison d'une graine par case. Si on arrive à la fin du tableau et qu'il reste des graines à répartir, on les jette.
Si le tableau est initialement `t=` | 5 | 2 | 3 | 0 | 8 | 7 | 3 | 1 | , est que l'indice est 4, on obtient | 5 | 2 | 3 | 0 | 0 | 8 | 4 | 2 | .
On ne demande pas de vérifier que l'indice `i` est acceptable.

6.3 : Écrire une méthode `jouer` qui reçoit un tableau d'entiers `t` et demande à l'utilisateur de lui donner le numéro d'une case. La méthode vérifie que le numéro donné est bien un indice du tableau, et que la case correspondante contient au moins une graine.
- Si c'est bien le cas, la méthode appelle `repartirFin`.
- Si ce n'est pas le cas, la méthode demande à l'utilisateur de modifier son choix, jusqu'à ce qu'il donne un choix correct.

6.4 : Écrire une méthode `repartirTour` identique à `repartirFin`, sauf que si on arrive à la fin du tableau et qu'il reste des graines à répartir, on continue à partir du début du tableau, et ainsi de suite.

6.5 : Écrire une méthode `repartirSauf` identique à `repartirTour`, sauf qu'elle ne repose jamais de graine dans la case qui a été vidée au début (le cas échéant, elle passe à la suivante).

6.1 :

```
public static int vide(int[] t) {
    int cpt=0; // compte les cases vides
    for(int i=0; i<t.length;i++)
         if (t[i]==0)
             cpt++;
    return cpt;
}
```

6.2 :

```
public static void repartirFin(int[] t,int i) {
    int gr=t[i]; // nombre de graines à répartir
    t[i]=0;
    for(int j=i+1;j<t.length;j++)
         if (gr>0) { // S'il reste des graines à répartir
             gr--;
             t[j]= t[j]+1;
         }
    /* VARIANTE :
       for(int j=i+1; (j<t.length) && (j<i+1+gr) ;j++)
           t[j]--;
    */
}
```

6.3 :

```
public static void jouer(int[] t) {
    int i;
    do {
         Deug.println("Donnez un numéro de case");
         i=Deug.readInt();
    } while ((i<0) || (i>=t.length) || (t[i]==0));
        // L'évaluation paresseuse est indispensable ici : double || et pas |
    repartirFin(t,i);
}
```

6.4 :

```
public static void repartirTour(int[] t,int i) {
    int gr=t[i]; // nombre de graines à répartir
    t[i]=0;
    for(int j=i+1;j<i+1+gr;j++)
         t[j%t.length]= t[j%t.length]+1;
}
```

6.5 :

```
public static void repartirSauf(int[] t,int i) {
    int gr=t[i]; // nombre de graines à répartir
    t[i]=0;
    int j=i;
    while(gr>0){ // tant qu'il reste des graines à répartir
         j= (j+1)%t.length; // case suivante, en faisant le tour
         if (j!=i){ // pour les cases autres que i
             gr--;
             t[j]=t[j]+1;
         }
    }
}
```