Architecture des Ordinateurs - TP 1 : Opérateurs bit à bit et nombres non signés

Le but de ce TP est d'utiliser les opérateurs bit à bit, les fonctions disponibles dans l'API Java et de programmer certainement de ces fonctions. Nous aurons besoin de certaines de ces fonctions d'entrées/sorties (pour le projet final).

Remarque 1 :

- Il n'y a pas de type non signé en JAVA. Seuls les types signés sont implémentés.
- Les entiers 32 bits sont de type **int** et les entiers 64 bits de type **long**. Une constante de ce type se termine par le suffixe L.
- L'opération de conversion de type est obtenue par l'utilisation de nom du type entre parenthèses devant l'expression à convertir.
- L'aide des fonctions de l'API JAVA est disponible sur le site Oracle :
  - Caractère :`docs.oracle.com/javase/7/docs/api/java/lang/Character.html`
  - Entier 32 bits `docs.oracle.com/javase/7/docs/api/java/lang/Integer.html`
  - Entier 64 bits `docs.oracle.com/javase/7/docs/api/java/lang/Long.html`
- Vous utiliserez votre environnement de développement préféré.

Exercice 1 : Test des opérateurs bit à bit

Soit un nombre codé sur 64 bits. Nous souhaitons accéder à trois champs : le bit de poids fort b_{63}, les bits de rangs intermédiaires b_{62} à b_{52} et les bits de rangs b_{51} à b_0. Soit deux variables N_1 et N_2 initialisées aux valeurs **0x**405F 5000 0000 0000 et N_2=**0x**BFD8 0000 0000 0000.

Soit des variables x_i,y_i avec $0 < i \leq 2$.

Question 1 : Affichez en binaire, octal, et en hexadécimal N_1 et N_2 en utilisant les fonctions de la classe *long*. Recherchez dans la javadoc les fonctions nécessaires.

Question 2 : Pour afficher plus clairement la valeur en hexadécimal, nous utiliserons la fonction suivante :

```
1  public static String ensembleHexa(long n){
2      int i;
3      String s="";
4      for(i=1;i<17;i++){
5          s=s+Character.forDigit((int)(n>>>60),16);
6          if ((i & 3)==0) s=s+"_";
7          n = n << 4;
8      }
9      return s;
10 }
```

Analysez et testez cette fonction pour l'affichage des nombres N_1 et N_2. Vous rechercherez dans l'API de la classe caractère le fonction *forDigit*. Que se passe t-il si nous supprimons l'appel de cette fonction ci-dessus ?

Question 3 : Modifiez la fonction *ensembleHexa* pour faire afficher le nombre en binaire par paquet de 4 bits et non pas par paquet de 4 digits hexadécimaux. La fonction *forDigit* est elle indispensable ici ?

Question 4 : Testez les opérateurs bit à bit `&` , `|` , `^` entre variables N_1 et N_2 et affichez les résultats en hexadécimal. Vérifiez que vous avez compris les valeurs obtenues.

Question 5 : Placez les différents champs N_2 dans trois variables ($0 < i \leq 2$). et affichez les champs de poids fort et intermédiaire en décimal et le champs de poids faible en hexadécimal.

Question 6 : Placez à 0 les différents champs de N_1 à 0 dans les variables y_i ($0 < i \leq 2$) t affichez les champs de poids fort et intermédiaire en décimal et le champs de poids faible en hexadécimal.

Question 7 : Placez un à un les champs de x_i dans y_i et affichez les trois valeurs en hexadécimal. Vérifier la validité des résultats.

Question 8 : Que fait le code ci-dessous. Expliquez ce qui se passe à chaque ligne ?

```
N1 = N1^N2;N2 = N2^N1; N1 = N1^N2;
```

Exercice 2 : Fonctions et opérateurs bit à bit

Question 1 : Testez les fonctions *RotateRight* et *RotateLeft* de l'API en décalant les nombres N_1 et N_2 de 1 rang puis de 8 rangs.

Question 2 : Implémentez ces deux fonctions pour des entiers 64 bits. Le code ne contiendra pas d'itération. Retrouvez ensuite les valeurs ci-dessus.

Question 3 : Que fait la fonction suivante. Dans quel cas retourne-t-elle true et false.

```
public static boolean fonction(int x){
    while ((x%2)==0){
    x = x / 2;
   }
   return  (x==1);
}
```

Proposez une version sans l'opérateur de division, puis une version sans l'itération avec uniquement un opérateur bit à bit et l'expression $x - 1$.

Question 4 : Nous désirons implémenter l'opérateur relationnel $>$ entre deux nombres 64 bits considérés comme non signés. Vérifiez que pour les nombres $FFFF\,FFFF\,FFFF\,FFFF$ et
$7FFF\,FFFF\,FFFF\,FFFF$ cet opérateur ne donne pas le résultat attendu. Proposer une version itérative de cette fonction qui les compare et test les bits deux à deux.

Exercice 3 : Fonction *bitCount*

Question 1 : Testez la fonction *bitCount* de l'API

Question 2 : Implémentez cette fonction sur le modèle du TD. Testez la différence entre l'opérateur $\neq$ et $>$ pour la sortie de l'itération

Question 3 : Analysez et testez ce code (avec pour paramètre une valeur 32 bits). Pourquoi choisir cette implémentation plutôt que celle proposée en TD ?

```
static int bitCount(int i) {
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f;
}
```

Question 4 : Donnez une implémentation similaire pour les entiers longs. Vous modifierez les masques et ajouterez un décalage logique de 32 rangs.

Exercice 4 : Fonctions de la classe Character

Question 1 : Testez les fonctions *forDigit* et *digit* de l'API avec des caractères ou des valeurs hexadécimales et des valeurs non hexadécimales. Quel est le rôle du deuxième paramètre ?

Question 2 : Implémentez ces deux fonctions, sans le deuxième paramètre. Vous testerez les majuscules et les minuscules.

Question 3 : Analysez et testez ce code. Pourquoi choisir cette implémentation ?

```
    static char forDigit(int n){
        char[] digits = { '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' ,
            '8' , '9' , 'a' , 'b' ,'c' , 'd' , 'e' , 'f' };
        return digits[ n & 0xF];
    }
```

Question 4 : Pouvons-nous réaliser le même type d'implémentation pour la fonction *digit* ?

Exercice 5 : Fonction *toString*

Question 1 : Testez la fonction *toString* de l'API. Que se passe t'il pour les conversions de N_1 et de N_2 en base 10 ?

Question 2 : Implémentez cette fonction comme étudié en TD.

Question 3 : Écrivez les *toHexString*, *toBinaryString* et *toOctalString* sans utiliser la division ni le modulo. Vous limiterez l'utilisation de la fonction *forDigit* autant que possible.

Exercice 6 : Fonction *parseLong*

Question 1 : Testez la fonction *parseLong* de l'API. Que se passe t'il pour les conversions de N_1 et de N_2 en base 10 ?

Question 2 : Implémentez cette fonction comme étudié en TD. Vous vérifierez que les résultats de vos fonctions sont corrects pour toutes les valeurs comprise entre **0x**0 et **0x**7FFF FFFF FFFF FFFF. Que se passe-t-il pour les valeurs supérieures à **0x**7FFF FFFF FFFF FFFF ?

Question 3 : Analysez et testez ce code ? Modifiez cette fonction pour une chaine de caractère binaire en supprimant l'appel de la fonction *digit*

```
1         static long  parseHexLong(String s){
2          int i=0, len = s.length(), digit;
3          long result=0;
4          for(i=0;i<len;i++){
5               digit = digit(s.charAt(i));
6               result = (result<<4)|digit;
7          }
8          return result;
9      }
```

Exercice 7 : Autres fonctions

Implémentez les fonctions *NumberOfLeadingZeros*,*NumberOfTralingZeros*, *highestOneBit*, *LowestOneBit* pour des entiers 64 bits (long). Vous vérifierez que les résultats de vos fonctions sont correctes pour toutes les valeurs (zero inclus)