Nom : *CORRECTION*
Groupe :

Projet Intégrateur – informatique
travail individuel n. 4 – 4 avril 2008

Retour sur la calculabilité des grands nombres et la complexité des algorithmes.

On considère la portion de code Java suivante :

```
1       public long methodeA (int n) {
2             long resultat = 1 ;
3             for (int i = 1 ; i <= n ; i++) {
4                   resultat *= i ;
5             }
6             return resultat ;
7       }

9       public long methodeB (int n) {
10            long resultat = 1 ;
11            for (int i = 1 ; i <= n ; i++) {
12                  resultat *= 2 ;
13            }
14            return resultat ;
15      }

17      public double methodeC (int n) {
18            double terme = 1 ;
19            double resultat = terme ;
20            for (int i = 1 ; i <= n ; i++) {
21                  terme = (double)methodeB(i) / methodeA(i) ;
22                  resultat += terme ;
23            }
24            return resultat ;
25      }
```

1. Donnez, sous une expression mathématique, la fonctionnalité de chacune de ces trois méthodes. *(3)*

`methodeA(n)`=	`methodeB(n)`=	`methodeC(n)`=
$\begin{cases} n! & si\ n \geq 0 \\ 1 & sinon. \end{cases}$	$\begin{cases} 2^n & si\ n \geq 0 \\ 1 & sinon. \end{cases}$	$\sum_{i=0}^{n} \frac{2^n}{n!}$ *si* $n \geq 0$

2. Donnez, sous une expression mathématique (fonction de `n`), le nombre approximatif d'opérations exécutées par l'appel à ces méthodes, avec `n` entier strictement positif. *(3)*

`methodeA(n)`	`methodeB(n)`	`methodeC(n)`
$O(n)$	$O(n)$	$O(\sum_{i=0}^{n} i) = O(n^2)$

3. L'algorithme de `methodeC(int n)` est correct, pourtant l'exécution de cette méthode Java avec `n=100` retourne `NaN` (*Not a Number*). Pourquoi ? *(2)*

La valeur entière $21! \equiv 5.10^{19}$ *est supérieure à la valeur maximale codée par un* `long`.
Dans `methodeC(100)`*, l'appel à la* `methodeA(i)` *ne retourne donc plus une valeur cohérente dès que* $i \geq 21$*. En l'occurence, elle retourne* 0 *pour certaines valeurs de* i*. À une telle itération, la variable* `terme` *reçoit alors la valeur* `NaN` *(division par* 0*) ; par suite, la variable* `resultat` *vaut désormais* `NaN` *(addition par* `NaN`*) jusqu'au terme de la boucle.*

4. Proposez un correctif à `methodeC` (indiquez les numéros de lignes de code à changer). Combien d'opérations sont maintenant exécutées avec ce nouvel algorithme ? *(2)*

On peut modifier le type de retour de `methodeA` *en* `double` *car il permet de coder des nombres (flottants) de plus grandes tailles. Mais cette solution n'est pas satisfaisante, car elle introduit rapidement des erreurs d'arrondi. Il est préférable de calculer directement les termes de la somme* `methodeC` *de manière récursive, en remplaçant la ligne* `21` *par :*

`21 :  terme = (double)2/i ;`

En prime, cette modification entraîne une réduction de la complexité de `methodeC(n)` *à* $O(n)$.

Des erreurs relevées dans les copies, en réponse à la question 3 :

1. *«le type* `long` *code des valeurs de plus grande taille que* `double`*» :*
   NON : `long` et `double` sont codés sur 64 bits, mais `long` codent des entiers et `double` des nombres à virgules flottantes. Par conséquent, on ne peut coder des **entiers** supérieurs à 2^{63} avec `long`. En revanche, on peut enregistrer dans un `double`, une valeur approchée, en notation scientifique avec un nombre de chiffres significatifs limité.

2. *«*`methodeC(100)` *effectue un trop grand nombre d'opérations» :*
   NON : Il ne fallait pas mélanger les **nombre et temps** de calculs (qui sont ici négligeables) avec les **valeurs** retournées par ces mêmes calculs (qui sont elles «quasi-infinies»). 100^2 ce n'est pas un grand nombre d'opérations pour un PC ordinaire... le calcul est vite fait pour une machine pouvant effectuer chaque opération élémentaire en 1 nanoseconde (10^{-9} seconde). On considère généralement qu'un algorithme quadratique, c'est-à-dire un algorithme de complexité de l'ordre de n^2 opérations, est **rapide** : même pour $n = 100.000$, il faut à peine 1 minute pour l'exécuter. En revanche, il existe des algorithmes de complexité bien supérieure, telle que le nombre d'opérations devient rapidement inimaginable : typiquement les algorithmes de complexité exponentielle $O(10^n)$ ou factorielle $O(n!)$ ne peuvent être exécutés que pour de très petites valeurs de n : 10^{163} nanosecondes, par exemple, c'est supérieur au temps écoulé depuis le Big-Bang ! ($59! \equiv 10^{80}$ c'est aussi le nombre estimé d'atomes dans l'univers.) Ce phénomène s'appelle l'**explosion combinatoire** et il incite à toujours se poser la question de la réalisabilité d'un algorithme.

3. *«ligne* `21` *: la conversion en* `(double)` *est incorrecte :»*
   NON : cette instruction est correcte et nécessaire. En effet, en Java, l'opérateur `/` entre deux entiers (`int` ou `long`) retourne le quotient entier de ces deux entiers. En revanche, il retourne le quotient fractionnaire quand il est appliqué à un nombre à virgule flottante (`float` ou `double`). Ainsi, l'instruction de la ligne `21` effectue cette suite d'opérations :
   (a) calcul de `methodeB(i)` puis conversion de cette valeur en `double` : 2^i
   (b) calcul de `methodeA(i)` : $i!$
   (c) division fractionnaire de 2^i (`double`) par $i!$ (`long`)
   (d) la valeur du quotient, de type `double`, est affecté à la variable `terme`.

   Sans la conversion du numérateur en `double`, le quotient `terme` serait entier et égal à 0 pour toute valeur de $i > 2$ (car alors $2^i < i!$).