INF4100–40 : Devoir 1
Hiver 2008

À remettre *au plus tard* le jeudi 7 février 2008, à 16h30
Pénalité de retard de 10 % par jour (complet ou partiel)

1. Sommations et estimés Θ (20 pts)

Soit les sommations suivantes :

$$a) \quad \sum_{i=1}^{10}(2i + i^2 + 3) \qquad b) \quad \sum_{i=0}^{n+1} n \cdot (2^i + 4ni)$$

$$c) \quad \sum_{i=1}^{n}\sum_{j=1}^{i}(3i - j) \qquad d) \quad \sum_{i=1}^{n}\sum_{j=1}^{n}\frac{n}{2^j}$$

i) Évaluez de façon exacte chacune de ces sommations.

ii) Donnez un estimé Θ (le plus simple et précis possible) de chacune de ces sommations.

Note : «Simple et précis» signifie que, par exemple, il faut indiquer $\Theta(1)$ et non $\Theta(3)$, ou $\Theta(n^2)$ et non $\Theta(2n^2 + n)$.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1} - 1}{r - 1} \text{ et } r \neq 1)$$

Indice 1: Rappels utiles sur les sommations (pour $r \neq 0$ et $r \neq 1$)

2. Définitions de O et Ω (10 pts)

Pour chacune des fonctions (f_1, f_2, f_3) indiquées plus bas, donnez un estimé O ou Ω (selon ce qui est demandé) *le plus précis et le plus simple possible*. Dans chaque cas, *justifiez* votre réponse en référant de façon détaillée à la définition de O ou Ω et à la fonction f_i — en d'autres mots, identifiez les constantes N et c telles que la définition appropriée (O ou Ω) s'applique pour la fonction f_i indiquée.

$$\begin{array}{lcll} f_1(n) & = & \lg\,(4n^3 \cdot 3^{n+2}) + \lg\ n^3 & \in O(?) \\ f_2(n) & = & 3^{-2n+1} + 2n^2 + 6n + 2^{100} & \in O(?) \\ f_3(n) & = & 2^{2n+1} + 3n^3 + 6n & \in \Omega(?) \end{array}$$

Pour appliquer les définitions, simplifiez les expressions et utilisez «$\leq$» (ou $\geq$) et «$=$» pour établir une relation entre les diverses expressions. Notez que lorsque vous établissez des relations entre expressions, il faut indiquer, à chaque étape, la relation la plus précise ($\leq/\geq$ ou $=$) qui s'applique. Par exemple, pour $\lg\,(n^2+1)$, on a les relations (étapes) suivantes :

$$\begin{array}{lcll} \lg\,(n^2+1) & \leq & \lg(n^2+n^2) & (\text{pour } n \geq 1) \\ & = & \lg(2n^2) & \\ & = & \lg\ 2 + \lg\ n^2 & \\ & = & \lg\ 2 + 2\ \lg\ n & \\ & \leq & \lg\ n + 2\ \lg\ n & (\text{pour } n \geq 2) \\ & = & 3\ \lg\ n & \end{array}$$

D'où l'on conclut que $\lg\,(n^2+1) \in O(\lg\ n)$ (pour $N = 2$ et $c = 3$).

Indice 2: Indice pour appliquer les définitions de O et Ω

$$\begin{array}{lcll} \lg\ n & \leq & n & (\text{pour } n \geq 1) \\ n & \leq & n^k & (\text{pour } n \geq 1 \text{ et } k \geq 2) \\ n & \leq & a^n & (\text{pour } n \geq 1 \text{ et } a \geq 2) \end{array}$$

Indice 3: Rappels utiles sur certaines relations entre fonctions de n

3. Analyse d'un algorithme itératif (20 pts)

Soit l'algorithme suivant (n'essayez pas de le comprendre ;) :

```
PROCEDURE fonct( n: Nat ): Nat
DEBUT
   r ← 1
   POUR i ← 1 A n FAIRE
     POUR j ← 1 A 2*i FAIRE
       r ← r + 1
       s ← 1
       t ← 0
       TANTQUE t ≤ n FAIRE
         s ← s * r
         t ← t + 2
       FIN
     FIN
   FIN
   RETOURNER( s )
FIN
```

a. Quelle précondition faut-il poser sur `n` pour simplifier l'analyse? Combien de fois le corps (le code à *l'intérieur*) de la boucle `TANTQUE` sera-t-il alors exécuté (exprimé en fonction de `n`)?

b. Considérons les opérations d'affectation (←) comme étant les opérations *élémentaires* pour cet algorithme — *en incluant aussi* les affectations aux variables de contrôle des boucles `POUR`. Déterminez alors le *nombre exact* d'opérations élémentaires effectuées par cet algorithme.

Note : Vous devez indiquer *de façon explicite* la sommation qui représente le nombre d'opérations élémentaires exécutées par l'algorithme. Ensuite, vous devez indiquer les principales étapes qui permettent d'arriver à votre résultat (exact) final.

c. À partir de l'expression obtenue en a. donnez la complexité temporelle (asymptotique) de l'algorithme, exprimée sous forme Θ.

d. Identifiez une *opération barométrique* appropriée pour cet algorithme, déterminez le nombre exact de fois où cette opération barométrique est exécutée, puis trouvez un estimé Θ.

Note : Comme pour b., indiquez clairement l'expression de sommation appropriée, suivie des principales étapes de simplifications et calculs.

e. Supposons que l'on modifie la fonction `fonct` de la façon suivante :

- L'en-tête devient comme suit :

  ```
  PROCEDURE fonct( n: Nat, v: Nat ): Nat
  ```

- L'initialisation de `t` =*avant le* `TANTQUE` devient comme suit :

  ```
  t ← v
  ```

Donnez alors la complexité asymptotique $B(n)$ du «meilleur cas» pour l'algorithme.

4. Algorithmes pour le calcul du nombre de paires ordonnées (20 pts)

Description du problème à résoudre

Soit `A` un tableau (vecteur) de `n` nombres entiers. On veut définir une fonction qui calcule le nombre d'éléments *adjacents* de `A` (donc le nombre de paires d'éléments) qui sont correctement ordonnés :

```
procedure nbPairesOrdonnees( int A[*], int n ) returns int nb
# PRECONDITION
#   n >= 1
# POSTCONDITION
#   nb = SUM( 1 <= i < n SUCH THAT A[i] <= A[i+1] :: 1 )
```

Quelques exemples d'utilisation (mais voir plus bas pour le nom exact de la fonction) :

- `nbPairesOrdonnees( (1, 1, 1), 3 )` $\Rightarrow 2$
- `nbPairesOrdonnees( (1, 2, 1), 3 )` $\Rightarrow 1$
- `nbPairesOrdonnees( (10, 9, 8), 3 )` $\Rightarrow 0$
- `nbPairesOrdonnees( (1, 2, 3, 4, 3, 2, 1, 0, 1), 9 )` $\Rightarrow 4$
- `nbPairesOrdonnees( (8, 7, 6, 4, 5, 3, 1, 2, 3), 9 )` $\Rightarrow 3$
- `nbPairesOrdonnees( (9, 8, 7, 6, 5, 1, 2, 4, 3), 9 )` $\Rightarrow 2$

Ce que vous devez faire

Pour cet exercice, vous devez développer et coder (en MPD) deux (2) versions différentes de la fonction `nbPairesOrdonnees`. Les interfaces (y compris les pré-conditions) de ces fonctions sont les suivantes :

```
global NbPairesOrdonnees

  op nbPairesOrdonneesIter( int A[*], int n ) returns int nb
  # PRECONDITION
  #   n >= 1

  op nbPairesOrdonneesRec ( int A[*], int n ) returns int nb
  # PRECONDITION
  #   n >= 1  ET n est une puissance de 3
  ...
end
```

Plus précisément, les deux versions à développer sont les suivantes :

1. Un algorithme itératif : `nbPairesOrdonneesIter`.
2. Un algorithme récursif, basé sur l'approche diviser-pour-régner : `nbPairesOrdonneesRec`. Plus spécifiquement, vous devez utiliser une décomposition «*trichotomique*» équilibrée — en d'autres mots, vous devez décomposer chaque problème non trivial en trois (3) sous-problèmes de tailles équivalentes (voir la précondition).

Analyse de la complexité temporelle asymptotique

Vous devez déterminer la complexité asymptotique de vos deux algorithmes, en justifiant/expliquant brièvement cette analyse (par exemple, choix et justification d'une opération barométrique, calcul exact, puis approximation asymptotique, du nombre de fois où elle est exécutée, etc.) :

a. Quelle est la complexité temporelle asymptotique de votre algorithme itératif?

b. Quelle équation de récurrence représente la complexité temporelle de votre algorithme récursif? Quelle est la complexité asymptotique résultante (forme Θ explicite)?

À remettre

Le document contenant vos réponses aux autres questions du devoir devra inclure *une copie papier* de votre fichier `nbPairesOrdonnees.mpd` (voir plus bas) ainsi que vos analyses de complexité.
Vous devrez aussi remettre votre fichier `nbPairesOrdonnees.mpd` en exécutant la commande suivante sur la machine `arabica` :

> `oto rendre_tp tremblay INF4100` <*codePerm*> `nbPairesOrdonnees.mpd`
>
> Note : Dans cette commande, <*codePerm(s)*> est soit votre code permanent si vous travaillez seul, soit les codes permanents des deux coéquipiers séparés par une «,» (sans espaces blancs) si vous travaillez avec quelqu'un d'autre, par exemple :

```
oto rendre_tp tremblay INF4100 TREG04127602 nbPairesOrdonnees.mpd
oto rendre_tp tremblay INF4100 TREG04127602,DAVA26628003 nbPairesOrdonnees.mpd
```

Vous pouvez aussi remettre votre programme en utilisant l'interface *Web* pour l'outil Oto : `http://labunix.uqam.ca:8181/~oto/application-web`.

Note : L'outil de correction `oto` sera aussi utilisé pour vérifier le bon fonctionnement de votre programme. Les tests utilisés pour la correction incluront ceux qui vous sont fournis, ainsi que des tests additionnels.

Barème de correction

- Algorithme itératif (5 pts) : compilation correcte (1 pt), bon fonctionnement sur les tests publics (1 pt), bon fonctionnement sur les tests privés (1 pt), analyse asymptotique (1 pt), qualité du code (1 pt).
- Algorithme récursif (15 pts) : compilation correcte (2 pts), bon fonctionnement sur les tests publics (3 pts), bon fonctionnement sur les tests privés (3 pts), équation de récurrence et analyse asymptotique (4 pts), qualité du code (3 pts).

Contraintes de mise en oeuvre

Vos algorithmes devront avoir été compilés et exécutés de façon à assurer leur bon fonctionnement — si votre programme ne compile pas, alors vous allez perdre une (très) grande partie des points : (Vous trouverez sur le site *web* du cours divers éléments que vous devez utiliser (en *complétant* la ressource décrite dans le fichier `nbPairesOrdonnees.mpd`) : `http://www.info.uqam.ca/~tremblay/INF4100/Devoirs` :

- `nbPairesOrdonnees.mpd` : Ressource définissant le squelette pour les deux versions des fonctions, squelette que vous devez compléter — avec les pré-conditions explicites déjà spécifiées.
- `tester-nbPairesOrdonnees.mpd`: Ressource définissant diverses suites de tests.[1] Ces suites de tests sont définies à l'aide de la ressource `MPDUnit`.
- `MPDUnit.mpd` et `MPDUnit-body.mpd` : Les jeux d'essai du fichier `tests.mpd` sont définis à l'aide d'un cadre de tests (*test framework*) pour le langage MPD. Ce cadre de tests joue pour MPD un rôle semblable à celui de `JUnit` [3] pour `Java`, c'est-à-dire permettre l'automatisation (de l'exécution) des tests, y compris les tests de régression, tel que proposé par les approches *eXtreme Programming* [1] et *Test-Driven Development* [2].[2]
- `OpsAuxiliaires.mpd` : Fichier qui contient un certain nombre d'opérations auxiliaires, entre autres, une procédure `assert` qui peut être utilisée pour spécifier des assertions en divers points d'un programme MPD — y compris des pré-conditions : voir `nbPairesOrdonneesRec` dans le fichier `nbPairesOrdonnees.mpd`.
- `Makefile` : Fichier qui permet d'automatiser la compilation du programme et l'exécution des tests. Ce fichier a été créé pour compiler les fichiers indiqués dans le présent document ; si vous ajoutez des fichiers, le fichier `Makefile` devra alors être modifié.

  Les principales commandes pour ce fichier `Makefile` sont les suivantes (exécutées au niveau du *shell* Unix) :

  - «`make`» : compile les fichiers requis par le programme `tests.mpd` — entre autres, `nbPairesOrdonnees.mpd` — en ne recompilant que ce qui doit l'être, c'est-à-dire, ce qui a été modifié ou ce qui dépend d'un fichier qui été modifié.
  - «`make tests`» : compile les fichiers requis par le programme `tests.mpd` (en ne recompilant que ce qui doit l'être) puis exécute le programme de tests — l'exécutable est dans `a.out`.

    Cette commande exécute les tests pour les deux versions (itérative et récursive). Pour exécuter les tests pour une seule des versions, il suffit d'exécuter directement la commande `a.out` avec l'argument approprié.
  - «`make clean`» : fait le ménage, c'est-à-dire, supprime les fichiers qui peuvent être générés de façon automatique.

Références

[1] K. Beck. *Extreme Programming Explained — Embrace Change.* Addison-Wesley, Reading, MA, 2000.

[2] K. Beck. *Test-Driven Development — By Example.* Addison-Wesley, 2003.

[3] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[1] Un *cas de test* permet de tester une fonctionnalité bien précise, avec un objectif de test bien défini et spécifique. Une *suite de tests*, quant à elle, regroupe un certain nombre de cas de tests. Ici, chaque version d'algorithme a sa propre suite de tests.

[2] Pour plus de détails sur les cadres de tests dans divers langages, voir `http://www.xunit.org`.