INF4100–40 : Devoir 2
Hiver 2008

À remettre *au plus tard* le jeudi 13 mars 2008, à 16h30
Pénalité de retard de 10 % par jour (complet ou partiel)

1. Forme générale d'une équation de récurrence (10 pts)

Soit `C` un nombre entier positif (i.e., une *constante* $C \geq 1$). Soit n le nombre d'éléments de `s` un tableau d'entiers — `s` est l'argument de la fonction `resoudre` (qui retourne un entier) dans l'algorithme 1 (n'essayez pas de le comprendre ;).

```
const int C = ...;

procedure resoudreSimple( ref int s[*], int inf, int sup ) returns int r
{
  r = 0;
  while ( inf <= sup & inf >= s[sup] ) {
    r += s[inf]
    inf += 1;
  }
}

procedure resoudreRec( ref int s[*], int inf, int sup ) returns int r
{
  if ( sup-inf+1 <= C ) {
    r = resoudreSimple( s, inf, sup );
  } else {
    int r1 = resoudreSimple( s, inf, inf+C-1 );
    int r2 = resoudreSimple( s, inf+C, min(inf+2*C-1, sup) ); # Pour ne pas depasser sup.
    int r3 = resoudreRec( s, inf+C, sup );
    r = r1 + r2 + r3;
  }
}

procedure resoudre( int s[*], int n ) returns int r
{ r = resoudreRec( s, 1, n ); }
```

Algorithme 1: Algorithme à analyser.

a. Donnez les équations de récurrence associées à l'analyse de la complexité temporelle de la procédure `resoudreRec`, en supposant qu'on veuille compter *le nombre total de «comparaisons»* effectuées par l'algorithme — donc en incluant aussi celles effectuées par `resoudreSimple`. Notez que, dans la boucle `while` de `resoudreSimple`, chaque vérification de la conjonction peut être comptée comme une (1) comparaison.

b. Quelle est la solution *exacte* de ces équations de récurrence? Justifiez votre réponse en utilisant *la méthode par substitution*. Vous pouvez, si approprié, imposer des conditions particulières sur n. Toutefois, si vous le faite, indiquez-le *explicitement*. Note : on veut une solution exacte *en fonction de n et de* `C`.

c. Quelle est la complexité *asymptotique* de la procédure `resoudre`? Est-ce qu'elle dépend de `C`? Expliquez brièvement.

2. Application du théorème général (15 pts)

Utilisez le théorème général de Neapolitan & Naimipour pour obtenir une solution asymptotique la plus précise possible pour chacune des récurrences suivantes — dans tous les cas, on suppose que $T(1) \in \Theta(1)$:

a. $T(n) \leq 4T(n/4) + 1$

b. $T(n) = 3T(n/3) + n$

c. $T(n) \geq 2T(n/4) + n$

d. $T(n) = 8T(n/2) + n^3$

e. $T(n) = 3T(n/4) + 2^{10}$

3. Opérations sur des séquences (25 pts)

Langages fonctionnels

Un langage de programmation fonctionnel — on dit aussi langage *applicatif* — est un langage fondé essentiellement sur l'utilisation de valeurs et de fonctions — au sens mathématique du terme, c'est-à-dire *sans effet de bord* et sans «*variable*». Des programmes écrits dans ces langages reposent donc uniquement *sur l'évaluation d'expressions.*

De nombreux langages fonctionnels existent : Lisp/Scheme, ML/SML, Miranda, Haskell, Erlang, etc. Dans tous ces langages, les fonctions sont des «citoyens de première classe» ; il est donc possible, et naturel, d'appeler une fonction avec un argument... *qui est lui-même une fonction.* De plus, la plupart des langages fonctionnels utilisent les listes — appelées aussi «séquences» — comme structures de données de base. Les algorithmes s'expriment donc souvent à l'aide d'opérations sur des séquences.

Opérations sur les séquences

De nombreuses opérations sur les séquences sont disponibles. Deux opérations importantes sont les suivantes :

- `map` (aussi appelée `apply`) :

  Soit f une opération unaire (avec un argument) et $S = [s_1, \ldots, s_n]$ une séquence de longueur n. L'application de `map` sur f et sur la séquence S produit alors la séquence R (de longueur n) satisfaisant la propriété suivante :

$$\begin{aligned} R &= \texttt{map}(f, S) \\ &= [f(s_1), \ldots, f(s_n)] \end{aligned}$$

- `reduce` (aussi appelée `fold` ou `inject`) :

  Soit $\odot$ une opération binaire et $S = [s_1, \ldots, s_n]$ une séquence. Une application de `reduce` avec $\odot$ sur S produit alors le résultat r satisfaisant la condition suivante :

$$\begin{aligned} r &= \texttt{fold}(\odot, S) \\ &= ((((s_1 \odot s_2) \odot s_3) \odot \ldots) \odot s_n) \end{aligned}$$

La partie gauche de la figure 1 présente quelques exemples de `map` et `foldr` en Haskell. Quant à la partie droite, elle présente quelques exemples en Ruby [Bla04, TH01] — parce qu'elles sont très utiles (et très puissantes), des opérations de style `map` et `reduce` sont aussi disponibles dans plusieurs langages modernes de programmation qui ne sont pas fonctionnels.

Les opérations `map` et `reduce` sont aussi utiles pour simplifier la programmation de machines parallèles, comme l'ont montré des chercheurs de Google, où de nombreux problèmes sont résolus simplement et efficacement en parallèle (sur des dizaines de milliers de machines!) uniquement à l'aide d'opérations `map` et `reduce` [DG08].

Une autre opération intéressante pour la programmation parallèle est l'opération `scan` [Ble87] :

- `scan` (aussi appelé «calcul des préfixes») :

  Soit $\odot$ une opération binaire et $S = [s_1, \ldots, s_n]$ une séquence. Un calcul des préfixes (un `scan`) avec $\odot$ sur S produit le résultat R (une séquence de longueur n) satisfaisant la condition suivante, donc tel que $R_i = s_1 \odot s_2 \odot \ldots \odot s_i$:

$$\begin{aligned} R &= \texttt{scan}(\odot, S) \\ &= [s_1, s_1 \odot s_2, s_1 \odot s_2 \odot s_3, \ldots, s_1 \odot \ldots \odot s_{n-1}, s_1 \odot \ldots \odot s_{n-1} \odot s_n] \end{aligned}$$

Exemples en Haskell :

```
map abs [-1,-3,4,-12]
=>
[1,3,4,12]

map reverse ["abc","cda","1234"]
=>
["cba","adc","4321"]

foldr (+) 5 [1,2,3,4]
=>
15

foldr max 0 [3,6,12,4,55,11]
=>
55
```

Exemples en Ruby :

```
a = [ "a", "b", "c", "d" ]
a.map {|x| x + "!" }
=>
["a!", "b!", "c!", "d!"]


(5..10).inject {|sum, n| sum + n }
=>
45


(5..10).inject(1) {|product, n| product * n }
=>
151200
```

Figure 1: Exemple d'utilisations de `map` et `fold`/`inject` en Haskell et Ruby.

Ce que vous devez faire

a. Codez (en MPD) les trois (3) opérations, décrites à la figure 2, d'un module `Sequence` — la partie interface, le corps (partiellement défini) et un programme de tests vous sont fournis (site *Web*).

   Contrainte de mise en oeuvre : Les trois opérations doivent être mises en oeuvre à l'aide d'une approche récursive *diviser-pour-régner dichotomique équilibrée* — donc décomposition en deux (2) sous-problèmes de tailles *équivalentes*.

b. Analysez les algorithmes associés à chacune de ces trois opérations.

   Vous devez donner explicitement les équations de récurrence puis trouver la solution asymptotique *en utilisant le thèorème général* et en décrivant brièvement (valeurs de a, b et k) comment le thèorème s'applique.

   Vous pouvez supposer que les opérations `f` (unaire) et `o` (binaire) s'exécutent en temps $\Theta(1)$.

```
op appliquer( cap OpUnaire f ) returns cap Sequence r
# POSTCONDITION
#   r.longueur() = self.longueur(),
#   ALL( 1 <= i <= self.longueur() :: r.elems[i] = f(self.elems[i]) )

op reduire( cap OpBinaire o ) returns int r
# PRECONDITION
#   self.longueur() >= 1
# POSTCONDITION
#   r = self.elems[1] o self.elems[2] o ... o self.elems[self.longueur()]

op tousLesPrefixes( cap OpBinaire o ) returns cap Sequence r
# PRECONDITION
#   self.longueur() >= 1
# POSTCONDITION
#   ALL( 1 <= i <= self.longueur() ::
#     r.elems[i] = self.elems[1] o self.elems[2] o ... o self.elems[i] )
```

Figure 2: Interface des opérations `appliquer`, `reduire` et `tousLesPrefixes` pour un module `Sequence`.

Ce que vous devez remettre

- Remise papier : listing de vos trois procédures — **inutile** de me remettre l'ensemble du code du module! — et analyse de la complexité des procédures.
- Remise électronique :

```
oto rendre_tp tremblay INF4100-sequence <codePerm> sequence-body.mpd
```

4. Calcul du déplacement d'un robot sur une grille (20 pts)

Description du problème

Un robot doit se déplacer à travers un espace délimité par une grille carrée à deux dimensions, de taille $n \times n$. Plus précisément, en partant du coin supérieur gauche, le robot doit se déplacer, d'une case à la fois, jusqu'au coin inférieur droit, et chaque déplacement ne peut se faire que d'une case, soit *vers la droite*, soit *vers le bas*.

Au moment du départ, la réserve d'énergie du robot est de 1000 KJ (kilo-joules). Lorsqu'il arrive à une case (y compris lorsqu'il est déposé sur la première case), sa réserve d'énergie est modifiée de la façon suivante, en fonction du nombre indiqué dans la case de la grille :

- Si la valeur est positive, sa réserve d'énergie est augmentée de la valeur indiquée.
- Si la valeur est «$-\infty$», alors il perd entièrement sa réserve d'énergie et ne peut plus se déplacer.
- Si la valeur est négative, sa réserve d'énergie est diminuée de la valeur indiquée. Si sa réserve d'énergie devient inférieure ou égale à zéro (0), alors il ne peut plus se déplacer.

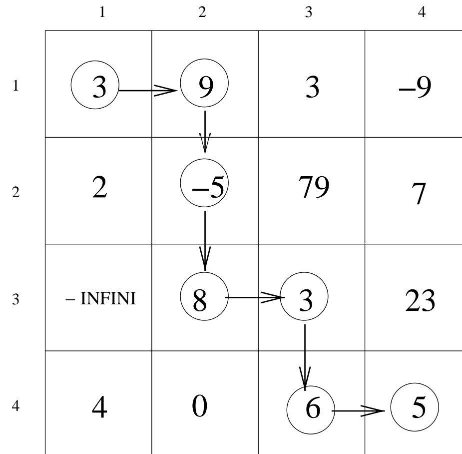

Figure 1: Un exemple d'une série de déplacements sur une grille 4×4.

La figure 1 présente un exemple d'une série possible de déplacements sur une grille 4×4. Le niveau d'énergie du robot, une fois cette série de déplacements effectuée, serait de $1000 + 3 + 9 - 5 + 8 + 3 + 6 + 5 = 1029$. Notons que cette série de déplacements n'est pas optimale : le déplacement sur les cases 3, 9, 3, 79, 7, 23 et 5 aurait conduit à un niveau final d'énergie plus élevé.

Notons par $x \oplus y$ l'opération suivante (*qui n'est pas commutative*) qui représente la façon d'accumuler l'énergie, où x représente un niveau d'énergie possible du robot et y une valeur de la grille :

$$x \oplus y \quad = \quad \begin{cases} 0 & \text{si } x = 0 \\ 0 & \text{si } y = -\infty \\ 0 & \text{si } x + y \leq 0 \\ x + y & \text{si } x + y > 0 \end{cases}$$

Notons $e[i, j]$ le gain ou perte d'énergie à la position $[i, j]$ de la grille, où le coin supérieur gauche correspond à $[1, 1]$ et le coin inférieur droit correspond à $[n, n]$ — n est un nombre entier supérieur ou égal à 1.

Notons $E[i,j]$ le niveau maximum final d'énergie pouvant être atteint par le robot se déplaçant de la case $[1,1]$ jusqu'à la case $[i,j]$ (inclusivement). Ce niveau optimal d'énergie peut être caractérisé par l'équation de récurrence suivante, pour $i > 1, j > 1$ — les cas de base sont omis et ce sera à vous de les expliciter… :

$$E[i,j] \quad = \quad max(E[i-1,j] \oplus e[i,j], E[i,j-1] \oplus e[i,j]) \quad \text{(pour } i > 1 \text{ et } j > 1\text{)}$$

Informellement : pour le cas général de la position $[i,j]$ non située sur une bordure à gauche ou en haut, il y deux façons différentes d'arriver à cette position (de la case du haut, ou de celle de gauche) et il faut donc trouver le maximum parmi ces deux possibilités.

Ce que vous devez faire

```
global Robot

  # Operations pour determiner l'energie finale du robot associee au meilleur
  # trajet possible dans la grille e, de taille nXn.

  op energieMeilleurTrajetRec( int e[*,*], int n ) returns int r

  op energieMeilleurTrajetPD ( int e[*,*], int n ) returns int r
```

Figure 3: Interface des opérations pour le calcul des déplacements optimaux d'un robot sur une grille (fichier `robot.mpd`).

Codez (en MPD) les deux (2) opérations, indiquées à la figure 3, d'un module `Robot` — la partie interface et le corps (partiellement défini et intégré dans le même fichier que l'interface) de ce module ainsi qu'un programme de tests vous sont fournis (site *Web*).

Contraintes de mise en oeuvre :

- L'opération `energieMeilleurTrajetRec` doit être réalisée à l'aide d'une approche *diviser-pour-régner récursive* (*mais pas dichotomique!*).
- L'opération `energieMeilleurTrajetPD` doit être réalisée à l'aide d'une approche de *programmation dynamique itérative*, donc sans récursion.

Dans les deux cas, il s'agit uniquement, pour le problème de base (donc en ignorant la question bonus), de déterminer le niveau maximum final d'énergie. Vous pouvez supposer qu'un chemin valide existe, i.e., qu'il existe toujours une façon pour le robot de se rendre du coin gauche supérieur au coin droit inférieur sans tomber en panne d'énergie.

Ce que vous devez remettre

- Remise papier : listing de vos deux procédures — **inutile** de me remettre l'ensemble du code du module!
- Remise électronique :

`oto rendre_tp tremblay INF4100-robot` <*codePerm*> `robot.mpd`

Question bonus pour l'exercice 4 (10 pts)

Soit le type suivant et l'opération suivante, dont l'en-tête est déjà définie dans le fichier `robot.mpd` :

```
# Operation pour produire la liste effective des deplacements
# du trajet optimal.

type Deplacement = enum( DROITE, BAS );

op obtenirDeplacements( res Deplacement ds[*], res int nbDeplacements )
# PRECONDITION
#   Un appel a energieMeilleurTrajetRec/PD a ete effectue.
```

Écrivez la mise en oeuvre de cette opération. On suppose (pré-condition) que lorsque cette opération est appelée, un appel à `energieMeilleurTrajetRec` ou à `energieMeilleurTrajetPD` a été effectué au préalable. Il n'est donc pas nécessaire (ou approprié) de refaire l'ensemble du calcul du trajet optimal. Il faut plutôt, lorsque le calcul initial est effectué, que le résultat soit conservé (mémorisé, mis en cache) de façon à pouvoir ensuite être utilisé lors de l'appel à `obtenirDeplacements`.

Barème de correction

Le barème utilisé pour la correction des différents exercices sera, en gros, le suivant :

1. Équations : 4 pts ; raisonnement et solution 4 pts ; solution asymptotique + explications : 2 pts
2. 3 pts chaque : 1,5 pts pour l'identification des constantes et de la relation ; 1,5 pts pour la solution finale.
3. 15 pts pour `appliquer` et `reduire`, 10 pts pour `tousLesPrefixes` : compilation correcte (20 %), bon fonctionnement sur les tests publics (20 %), bon fonctionnement sur les tests privés (20 %), équations de récurrence et analyse asymptotique (20 %), qualité du code (20 %).
4. 10 pts chacune : compilation correcte (20 %), bon fonctionnement sur les tests publics (30 %), bon fonctionnement sur les tests privés (20 %), qualité du code (30 %).

Note : Pour les procédures des exercices 3 et 4, le barème indiqué ne s'appliquera *que si les contraintes de mise en oeuvre ont été respectées* : décomposition dichotomique équilibrée, diviser-pour-régner/programmation dynamique.

Références

[Bla04] C. Blaess. *Scripts sous Linux—Shell Bash, Sed, Awk, Perl, Tcl, Tk, Python, Ruby...*. Eyrolles, 2004. [QA76.76O63B4899].

[Ble87] G.E. Blelloch. Scans as Primitive Parallel Operations. In *Proc. of the International Conference on Parallel Processing*, pages 355–362, Aug. 1987.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[TH01] D. Thomas and A. Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, Reading, MA, 2001.