INF7440 : Devoir #2

(À remettre *au plus tard* lundi 29 oct. 2007, à 13h00)

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

Soit a un nombre entier positif (i.e., une *constante* $a \geq 1$). Soit n le nombre d'éléments de `s` — l'argument de la fonction `proc` (qui retourne un entier) dans l'algorithme 1.

```
PROCEDURE proc( s: sequence{Nat} ) r: Nat
DEBUT
  n ← longueur(s)
  SI n ≤ a ALORS
    r ← traiterCasSimple( s )
  SINON
    r1 ← traiterCasSimple( s[a+1:n] )
    s' ← procAux( s[1:n-a] )
    r2 ← proc( s' )
    r ← r1 + r2
  FIN
FIN

PROCEDURE traiterCasSimple( s: sequence{Nat} ) r: Nat
DEBUT
  r ← 0
  POUR i ← 1 A longueur(s) FAIRE
    r ← r + s[i]
  FIN
FIN

PROCEDURE procAux( s: sequence{Nat} ) s': sequence{Nat}
DEBUT
  s'[1] ← 0
  POUR i ← 2 A longueur(s) FAIRE
    s'[i] ← s'[i-1] + s[i]
  FIN
FIN
```

Algorithme 1: Algorithme générique à analyser : `proc` est la fonction principale.

a. Donnez les équations de récurrence associées à l'analyse de la complexité temporelle de la procédure `proc`, en supposant qu'on veuille compter *le nombre d'additions* (additions explicites, i.e., les «+» qui apparaissent explicitement dans les diverses procédures) — vous pouvez toutefois ignorer les «+» (et «-») utilisés pour les calculs d'index.

b. Quelle est la solution générale *exacte* de ces équations de récurrence? Justifiez votre réponse en utilisant la méthode par substitution. Vous pouvez, si vous le désirez, 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 a.

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

2. Ballade à vélos pour un groupe de touristes (40 pts)

Description du problème à résoudre

On veut organiser une ballade à vélos avec un groupe de `m` touristes. Pour ce faire, on a à notre disposition `n` vélos, de différentes grandeurs, et on veut assigner un vélo à chacun des cyclistes.

Plus précisément, les vélos sont de quatre (4) tailles différentes possibles : petit, moyen, grand et très grand. Le nombre de vélos pour chaque taille est donné par un tableau `nbVelos`, où `nbVelos[PETIT]` indique le nombre de petits vélos, `nbVelos[MOYEN]` le nombre de vélos de taille moyenne, `nbVelos[GRAND]` le nombre de grands vélos et `nbVelos[TRES_GRAND]` le nombre de très grands vélos.[1]

Les tailles des cyclistes, en cm, sont données par le tableau `cyclistes` — i.e., `cyclistes[1]`, `cyclistes[2]`, ..., `cyclistes[m]`.

Idéalement, on considère qu'un vélo est de taille appropriée — ne générant *aucun inconfort* pour le cycliste — si les conditions indiquées dans le Tableau 1 sont satisfaites.

Taille du cycliste (en cm)	**Taille de vélo**	**Taille idéale (en cm)**
$t < 150$	Petit	140
$150 \leq t < 170$	Moyen	160
$170 \leq t < 190$	Grand	180
$190 \leq t$	Très grand	200

Tableau 1: Correspondances idéale entre taille de cycliste et taille de vélo.

On sait que le nombre de vélos, indépendamment de leurs tailles, est supérieur ou égal au nombre de cyclistes — bref, on a suffisamment de vélos pour tout le monde. Toutefois, rien n'assure qu'une personne donnée pourra obtenir un vélo de la taille qui lui est la plus appropriée. Lors de l'assignation des vélos aux différentes personnes, on veut donc tenter de «*minimiser l'inconfort*» (total) de l'ensemble des cyclistes. On considère que l'inconfort d'un cycliste pour un vélo est défini comme suit :

- Si le vélo assigné au cycliste est d'une taille qui respecte les conditions du Tableau 1, alors on considère que l'inconfort est nul (0).
- Si le vélo assigné au cycliste n'est pas de la bonne taille, telle que spécifiée par les conditions du Tableau 1, alors on considère que l'inconfort est la valeur absolue de la différence entre la taille du cycliste et la taille considérée comme idéale pour la taille du vélo (troisième colonne du Tableau 1).

Quelques exemples d'affectations possibles et d'inconfort associé sont présentés dans le Tableau 2.

[1] Dans le squelette de programme MPD qui vous est fourni, ces constantes sont définies par un type `enum` tel que `int(PETIT)=0`, ..., `int(TRES_GRAND)=3`.

Taille du cycliste	Taille du vélo	Inconfort
145	Petit	0
145	Moyen	15
165	Moyen	0
165	Petit	25
165	Très grand	35

Tableau 2: Exemples d'assignation de vélo et de niveau «d'inconfort».

Pour simplifier le problème, dans ce qui suit, nous nous limiterons *à trouver le coût (inconfort) total minimal* d'une affectation des vélos aux différents cyclistes, sans chercher à identifier comme telle cette affectation (i.e., il n'est pas nécessaire d'identifier explicitement quel cycliste reçoit quel vélo).

Toujours pour simplifier le problème, on suppose aussi que les tailles des cyclistes sont ordonnées, de la plus petite taille à la plus grande (voir la pré-condition plus bas).

Ce que vous devez faire

Pour cet exercice, vous devez développer et coder (en MPD) trois versions différentes d'une fonction qui détermine le coût optimal (minimum) pour l'affectation d'un ensemble de vélos à un groupe de cyclistes. L'interface (y compris les pré-conditions) de ces diverses fonctions est présentée dans l'extrait de Code MPD 1 (p. 5).

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

1. Un algorithme récursif, basé sur l'approche diviser-pour-régner : `coutAffectationVelosRec`.
2. Un algorithme basé sur l'approche de programmation dynamique : `coutAffectationVelosPD`.
3. Un algorithme vorace : `coutAffectationVelosVorace`. Notez que cette version, contrairement aux deux autres, n'a pas nécessairement à toujours identifier le coût minimal — l'important est que cet algorithme soit *vorace* et asymptotiquement très efficace, pas qu'il soit optimal.

Dans les trois cas, tel qu'indiqué précédemment, il suffit de trouver le *coût* de l'affectation minimum, et non pas d'identifier l'affectation elle-même.

À remettre

Vous devrez remettre votre programme final (fichier `affectation-velos.mpd` : voir plus bas) en exécutant la commande suivante sur la machine `arabica` :

`oto rendre_tp tremblay INF7440` <*codePerm*> `affectation-velos.mpd`

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

Questions additionnelles

Finalement, vous devez aussi répondre aux questions suivantes :

a. Expliquez (possiblement à l'aide d'un exemple, c'est-à-dire, avec des valeurs spécifiques pour les différents arguments) pourquoi l'algorithme récursif *n'est pas efficace*.

b. Quelle est la complexité asymptotique de votre algorithme de programmation dynamique?

c. À l'aide d'un exemple, donc avec des valeurs spécifiques pour les différents arguments, montrez que l'algorithme vorace ne produit pas toujours une solution optimale — en d'autres mots, montrez à l'aide d'un exemple que la solution produite par l'algorithme vorace n'est pas nécessairement aussi bonne que celle produite par l'un des autres algorithmes.

Contraintes de mise en oeuvre

Vos algorithmes devront avoir été compilés et exécutés de façon à assurer leur bon fonctionnement. Plus précisément, vous trouverez sur le site *web* du cours divers éléments que vous devez utiliser (évidemment en complétant la principale ressource, soit le fichier `affectation-velos.mpd`) : `http://www.info.uqam.ca/~tremblay/INF7440/Devoirs` :

- `affectation-velos.mpd` : Ressource définissant les diverses versions des fonctions, que vous devez compléter — avec les pré-conditions explicites déjà spécifiées pour la version récursive.
- `tests.mpd`: Ressource définissant diverses suites de tests.[2] 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` [4] 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* [3, 2].[3]
- `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 `coutAffectationVelosRec` dans le fichier `affectation-velos.mpd`.
- `Makefile` : Fichier qui permet d'automatiser la compilation 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, `affectation-velos.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 trois versions (récursive, programmation dynamique et vorace). 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.

[2] 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.

[3] Pour plus de détails sur les cadres de tests dans divers langages, voir le site *Web* `http://www.xunit.org`.

```
global AffectationVelos

  # Type (synonyme) pour la taille des cyclistes.
  type Cycliste = int;


  # Type identifiant les differentes tailles possibles de velos.
  type Velo = enum( PETIT, MOYEN, GRAND, TRES_GRAND );

  # Les bornes des intervalles de taille de cyclistes pour chaque taille de velo.
  Cycliste TAILLES_MIN[PETIT:TRES_GRAND]     = (0,   150, 170, 190);
  Cycliste TAILLES_MAX[PETIT:TRES_GRAND]     = (150, 170, 190, high(int));

  # Les tailles ideales de cycliste pour chaque taille de velo.
  Cycliste TAILLES_IDEALES[PETIT:TRES_GRAND] = (140, 160, 180, 200);


  ############################################################
  # Les trois operations a mettre en oeuvre.
  ############################################################

  op coutAffectationVelosRec
     ( int nbVelos[PETIT:TRES_GRAND], Cycliste cyclistes[*], int m )
     returns int coutMin

  op coutAffectationVelosPD
     ( int nbVelos[PETIT:TRES_GRAND], Cycliste cyclistes[*], int m )
     returns int coutMin

  op coutAffectationVelosVorace
     ( int nbVelos[PETIT:TRES_GRAND], Cycliste cyclistes[*], int m )
     returns int coutMin

  # PRECONDITION
  # Soit n = nbVelos[PETIT] + nbVelos[MOYEN] + nbVelos[GRAND] + nbVelos[TRES_GRAND]
  # Alors
  #   n >= 1,
  #   m >= 1,
  #   n >= m,
  #   ALL( PETIT <= i <= TRES_GRAND ::  nbVelos[i] >= 0 ),
  #   ALL( 1 <= i < m ::  cyclistes[i] <= cyclistes[i+1] )

body AffectationVelos
  ...
  # Partie à compléter...
  ...
end
```

Code MPD 1: En-tête définissant l'interface des opérations à mettre en oeuvre.

Quelques indices

a. La solution récursive *n'est pas une récursion dichotomique* dans le style de la recherche binaire ou du tri par fusion.

b. Pour les versions récursive et de programmation dynamique, il est probablement plus simple[4] de traiter les arguments par l'intermédiaire de deux tableaux : un tableau (ordonné) des tailles de vélos et un tableau (déjà ordonné : cf. pré-condition) des tailles de cyclistes. En d'autres mots, il peut être plus simple d'introduire une structure de données et une (ou plusieurs) procédure(s) auxiliaire(s).

c. En MPD, l'opération `min` permet de trouver la valeur minimum parmi une série de valeurs. Par exemple, `min(10, 20, 14) = 10`.

   L'élément *neutre* de l'opération `min` est la plus grande valeur entière possible. En MPD, cette valeur est dénotée par `high(int)`.

   Donc, pour toute valeur `v`, la propriété suivante est vérifiée :

```
v == min(v, high(int))
```

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.

[4] A. Hunt and D. Thomas. *Pragmatic Unit Testing In Java with JUnit*. The Pragmatic Bookshelf, Raleigh, NC, 2003.

[4] C'est de cette façon, à tout le moins, que j'ai produit ma propre solution.