INF4100–40 : Devoir 3
Hiver 2008

À remettre *au plus tard* le lundi 14 avril 2008, à 13h30
Pénalité de retard de 10 % par jour (complet ou partiel)

1. Code de Huffman (15 pts)

Soit un texte, composé uniquement des neuf lettres a, b, ..., i, et comportant un nombre total de 1 000 caractères, où chaque caractère apparaît le nombre de fois suivant :

Caractère	**Occurences**
a	120
b	45
c	80
d	95
e	300
f	150
g	60
h	40
i	110

a. Définissez un code de Huffman pour encoder ce texte. Plus précisément, donnez l'encodage de chacun des caractères ansi que l'arbre binaire (*trie*) du code résultant.

   Note : Comme dans les exemples vus en classe, le sous-arbre de fréquence inférieure doit correspondre au sous-arbre gauche, représentant le bit 0.

b. Combien de bits seront requis pour encoder l'ensemble du document, en utilisant votre code de Huffman?

c. Combien de bits auraient été requis pour encoder ce même document si on avait utilisé un code où les neufs lettres utilisaient le même nombre de bits?

2. Algorithmes sur les graphes (20 pts)

a. Un multigraphe orienté valué est un graphe qui permet l'existence de plusieurs arcs entre deux sommets, tel qu'illustré dans la figure 1.

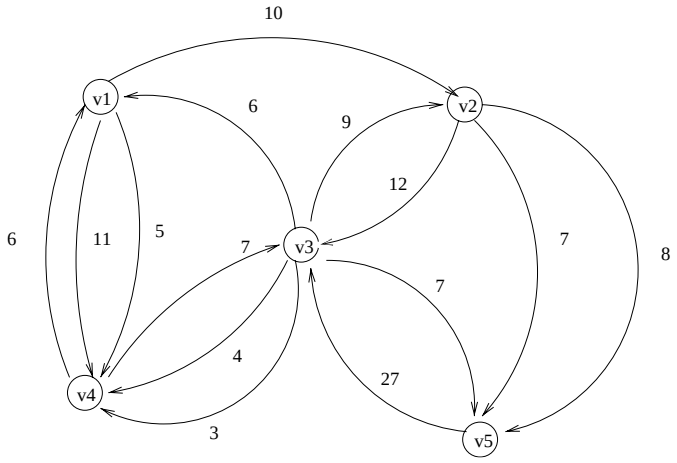

Figure 1: Un multi-graphe valué orienté avec cinq (5) sommets.

Un tel graphe peut être représenté par une matrice d'adjacence où chaque entrée de la matrice est *un ensemble de poids*, par exemple, $G[1,4] = \{5, 11\}$, $G[3,4] = \{3, 4\}$, $G[2,1] = \{\}$, $G[1,2] = \{10\}$, etc.

(i) De quelle façon devrait-on modifier l'algorithme de Floyd pour qu'il fonctionne aussi pour un multigraphe, représenté par une matrice d'adjacence avec ensemble de poids? Décrivez vos modifications en écrivant une version révisée de l'algorithme de Floyd vu en classe.

(ii) En utilisant votre algorithme de Floyd révisé, produisez la matrice des distances minimum entre chacun des sommets du graphe de la figure 1. Indiquez chacune des matrices intermédiaires $D^{(k)}$, pour $0 \leq k \leq 5$.

b. Dans le graphe de la figure 1, supposons que chaque arc soit remplacé par une arête de façon à obtenir *un multigraphe valué non orienté*. Les arcs deviennent donc implicitement bi-directionnelles, par exemple, on aurait donc trois arêtes (de poids 7, 4 et 3) entre `v3` et `v4`, deux arêtes (de poids 9 et 12) entre `v2` et `v3`, etc.

(i) En utilisant l'algorithme de Prim, produisez un arbre couvrant minimum (arbre minimum de recouvrement). Utilisez le sommet `v1` comme sommet de départ. Indiquez clairement dans quel ordre chacun des arcs est ajouté à l'arbre couvrant minimum, soit en donnant la liste des sous-arbres, soit en numérotant de façon appropriée les arcs de l'arbre résultant.

(ii) En utilisant l'algorithme de Kruskal, trouvez un arbre couvrant minimum. Indiquez clairement dans quel ordre chacun des arcs est ajouté à l'arbre couvrant minimum, soit en donnant la liste des forêts, soit en numérotant de façon appropriée les arcs de l'arbre résultant.

3. Sac à dos 0–1 (15 pts)

Une instance du problème pour le sac à dos 0–1 est décrite par un poids maximal W et par un ensemble de n items, chacun de poids `poids[i]` et de bénéfice `benefs[i]`. Pour une telle instance, l'algorithme de programmation dynamique construit une matrice M avec $n+1$ lignes et $W+1$ colonnes. Or, dans de nombreux cas, il n'est pas nécessaire de remplir la matrice au complet. Donc, pour trouver le bénéfice optimal, calculé à la position $M[n, W]$, on pourrait ne remplir que certaines des cases de la matrice, plus précisément, celles correspondant à des sous-problèmes pour lesquels on doit effectivement calculer la solution (y compris les cas de base) en fonction des divers poids des items. Disons d'un tel remplissage de la matrice M qu'il s'agit d'un *remplissage paresseux*.

a. Soit l'instance suivante du problème du sac à dos 0–1 :

- $n = 5$, $W = 15$
- `poids` = [3, 4, 5, 5, 4]
- `benefs` = [15, 25, 15, 35, 25]

(i) Construisez la matrice M de programmation dynamique, mais en ne remplissant que les positions de la matrice qui sont strictement nécessaires (remplissage paresseux).

(ii) Quel sera le bénéfice maximal?

(iii) Quels items devront être inclus dans le sac pour obtenir cette solution optimale?

b. À partir d'une matrice M, remplie de façon paresseuse (incluant les cas de base), écrivez un algorithme (MPD ou pseudocode) qui permet de déterminer les items qui doivent être inclus dans le sac à dos.

Bonus Soit l'instance suivante du problème du sac à dos 0–1 :

- n un entier positif, $W = 2^n - 2$
- `poids` = [1, 2, 4, 8, ..., 2^{n-1}]
- `benefs` = [b_1, ..., b_n]

En d'autres mots, l'item i est de poids 2^{i-1} et le bénéfice associé est b_i, un nombre positif.

(i) En utilisant un remplissage paresseux (en incluant les cas de base, donc les cases de la $0^{ième}$ ligne et de la $0^{ième}$ colonne qui doivent être évalués), combien de cases de la matrice de programmation dynamique M devront être remplies pour cette instance de problème?

(ii) Qu'en concluez-vous quant à la complexité asymptotique (pire cas) de l'algorithme de programmation dynamique pour le problème du sac à dos 0–1?

(iii) Quel sera le bénéfice maximum possible (exprimé en fonction des valeurs b_i)?

Suggestion : Examiner différents cas simples ($n = 3, 4, \ldots$) puis généralisez.

4. Envois de colis (30 pts)

Description du problème

On doit envoyer, par courrier, un ensemble de `m` items. Pour ce faire, on a à notre disposition `n` boîtes, de différentes capacités, et on doit assigner une boîte à chacun des items.

Les capacités des diverses boîtes sont décrites par un tableau `boites`, où `boites[i]` indique la capacité (en grammes), de la boîte `i`. De plus, on suppose que les boîtes sont ordonnées selon leur capacité, c'est-à-dire que `boites[i]` $\leq$ `boites[i+1]` ($1 \leq$ `i` $<$ `n`).

Les poids des items sont donnés par un tableau `items`. Comme pour les boîtes, on suppose que les items sont ordonnés selon leur poids, i.e., `items[i]` $\leq$ `items[i+1]` ($1 \leq$ `i` $<$ `m`).

On considère qu'une boîte est de taille correcte lorsque sa capacité est supérieure ou égale au poids de l'item. Par contre, il est permis d'utiliser une boîte de taille inférieure en autant que la différence entre la capacité de la boîte et le poids de l'item soit inférieure ou égale à 10.

On sait que le nombre de boîtes disponibles, toutes tailles combinées, est suffisant pour empaqueter tous les items, et ce en respectant la condition sur les capacités des boîtes.

Le coût d'envoi d'un item dépend de la boîte et est déterminé comme suit :

- Le coût de base est de 1.00 $ par fraction de 100 g du poids de l'item. Par exemple, le coût de base pour envoyer un item de 250 g est de 3.00 $.
- Si la capacité de la boîte est supérieure ou égale au poids de l'item, alors le coût de l'envoi est déterminé par le coût de base, c'est-à-dire, sans pénalité.
- Si la capacité de la boîte est inférieure au poids de l'item (tout en respectant la condition indiquée plus haut), alors un coût supplémentaire de 0.05 $ par gramme au-delà de la capacité de la boîte est ajouté au cas de base.

Le tableau suivant présente quelques exemples de coûts :

Poids de l'item	Capacité de la boîte	Coût d'envoi
150	140	2.50 $
150	150	2.00 $
150	160	2.00 $
200	180	Impossible
200	200	2.00 $
201	200	3.05 $
201	400	3.00 $

Lors de l'assignation des boîtes aux différents items, on veut évidemment tenter de «*minimiser le coût*» (total) de l'ensemble des envois.

Pour simplifier le problème, on suppose qu'on cherche simplement *à trouver le coût total minimal* d'une affectation des boîtes aux différents items, sans chercher à identifier cette affectation (i.e., il n'est pas nécessaire d'identifier explicitement quel item est mis dans quelle boîte).

Ce que vous devez faire

Pour cet exercice, vous devez développer et coder (en MPD) deux (2) versions d'une fonction qui détermine le coût optimal (minimum) pour l'envoi d'un ensemble d'items à l'aide d'un ensemble de boîtes. L'interface (y compris les pré-conditions) de ces diverses fonctions est présentée dans l'extrait de Code MPD 1 (p. 6).

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

1. Un algorithme basé sur l'approche de programmation dynamique : `coutEnvoiPD`.

2. Un algorithme vorace : `coutEnvoiVorace`. Cette version, contrairement à l'autre, n'a pas nécessairement à toujours identifier le coût minimal — l'important est que cet algorithme soit *vorace* et asymptotiquement efficace, pas qu'il soit optimal.

À remettre

Vous devrez remettre votre programme final (fichier `envois-items.mpd` : voir plus bas) en exécutant la commande suivante sur la machine `arabica` :[1]

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

Barème de correction et questions additionnelles

Barème pour chaque version : compilation correcte (20 %), bon fonctionnement sur les tests publics (20 %), bon fonctionnement sur les tests privés (20 %), qualité du code (20 %), analyse asymptotique et/ou exemple (20 %).

Vous devez aussi répondre aux questions suivantes :

a. Quelle est la complexité asymptotique de chacun des vos algorithmes?

b. 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 toujours aussi bonne que celle produite par l'algorithme de programmation dynamique.

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 `envois-items.mpd`) : `http://www.info.uqam.ca/~tremblay/INF4100/Devoirs` :

- `envois-items.mpd` : Ressource définissant le squelette des deux versions des fonctions, que vous devez compléter — avec les pré-conditions explicites spécifiées.
- `tests.mpd`: Ressource définissant les suites de tests.
- `OpsAuxiliaires.mpd`, `MPDUnit.mpd` et `MPDUnit-body.mpd` : Fichiers pour assertions et cadre de tests.
- `Makefile` : Fichier qui permet d'automatiser la compilation et l'exécution des tests.

  Les principales commandes `Makefile` sont les suivantes :

  - «`make`» : compile les fichiers requis par le programme `tests.mpd` — entre autres, `envois-items.mpd` — en ne recompilant que ce qui doit l'être.
  - «`make tests`» : compile les fichiers requis par le programme `tests.mpd` puis exécute le programme de tests — l'exécutable est dans `a.out`.

    Cette commande exécute les tests pour les deux versions (programmation dynamique et vorace). Pour exécuter les tests d'une seule des versions, il suffit d'exécuter directement la commande `a.out` avec l'argument approprié (0 ou 1).
  - «`make clean`» : fait le ménage, c'est-à-dire, supprime les fichiers qui peuvent être générés de façon automatique.

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

```
global EnvoisItems

  # Type (synonyme) pour les poids des items.
  type Poids = int;

  # Type (synonyme) pour les capacites des boites.
  type Capacite = int;

  # Type (synonyme) pour les couts: on utilise des entiers,
  # donc cout en cents plutot qu'en $ (pour eviter d'utiliser des real).
  type Cout = int;


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

  op coutEnvoiPD    ( Capacite boites[*], int n, Poids items[*], int m )
     returns Cout coutMin

  op coutEnvoiVorace( Capacite boites[*], int n, Poids items[*], int m )
     returns Cout coutMin

##############################################################################
##############################################################################
body EnvoisItems
```

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