INF4100 — Analyse et conception d'algorithmes
Examen final (Hiver 2008)

Durée: (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Nom: ______________________________

Code permanent: | | | | | | | | | | | | |

1	/10
2	/10
3	/10
4	/10
5	/15
6	/10
Bonus	/10
Total	/65

Directives:

a. Utilisez les versos pour vos brouillons ou, le cas échéant et à moins d'indication contraire, **pour les réponses aux questions bonus**. Dans ce dernier cas, signalez-le clairement et explicitement, au recto, en indiquant BONUS.

1. Sac à dos 0–1 version programmation dynamique (10 pts)

On a vu dans le devoir 3 qu'une solution de programmation dynamique *naïve* pour le problème du sac à dos 0–1 remplit inutilement certaines cases de la matrice utilisée pour la mémorisation des solutions des sous-problèmes. Par contre, un remplissage *paresseux* remplit uniquement les cases qui sont effectivement utilisées pour le calcul du résultat final.
Soit `n` un entier positif, `W` une *constante* positive (qui ne dépend pas de `n`) et $p_1, \ldots, p_{\mathtt{n}}$ des valeurs positives (non-nulles) définissant les poids des `n` items : `int poids[n]` $= (p_1, p_2, \ldots, p_{\mathtt{n}-1}, p_{\mathtt{n}})$

a. Pour quelles valeurs de p_i ($i = 1, \ldots, \mathtt{n}$) est-ce qu'un remplissage paresseux conduirait à remplir *le nombre maximum* de cases de la matrice B — en comptant aussi les éléments de la $0^{ième}$ ligne ou de la $0^{ième}$ colonne qui doivent être évalués?

b. Combien de cases (*nombre exact*) seraient alors remplies — en comptant aussi les éléments de la $0^{ième}$ ligne ou de la $0^{ième}$ colonne qui doivent être évalués ?

2. Déplacement d'un robot sur une grille (10 pts)

Le code MPD ci-bas présente la solution de programmation dynamique au problème no. 4 du devoir 2 pour le calcul du déplacement d'un robot sur une grille $n \times n$:

```
procedure oplus( int x, int y ) returns int r
{
  if   ( x == 0 | y == low(int) | x + y <= 0 ) { r = 0;     }
  else                                         { r = x + y; }
}

procedure energieMeilleurTrajetPD( int e[*,*], int n ) returns int r
{
  int E[n,n];
  E[1,1] = oplus( ENERGIE_INITIALE, e[1,1] );
  for [i = 2 to n] {
    E[i,1] = oplus( E[i-1,1], e[i,1] )
    E[1,i] = oplus( E[1,i-1], e[1,i] );
  }
  for [i = 2 to n, j = 2 to n] {
    E[i,j] = max( oplus(E[i-1,j], e[i,j]), oplus(E[i,j-1], e[i,j]) );
  }
  r = E[n,n];
}
```

a. Quelles modifications faudrait-il effectuer à ce programme si l'énoncé du problème était modifié comme suit et qu'on voulait toujours déterminer le niveau maximum d'énergie possible pour le robot à la fin de son parcours :

- Le robot doit partir du coin supérieur gauche (position `[1,1]`) et doit se rendre *sur n'importe quelle case située sur la frontière inférieure* (`[n,*]`) *ou sur la frontière droite* (`[*,n]`). En d'autres mots, le robot n'est pas obligé d'atteindre la case `[n,n]` mais peut «sortir» (terminer) dès qu'il atteint une case située sur une de ces deux bordures.

b. Quelle serait alors la complexité asymptotique de la procédure dans son ensemble?

c. **Bonus** Est-il possible de réduire, de façon *asymptotique*, l'espace mémoire occupé par le tableau de mémorisation (tableau `E`)? Si oui, indiquez quelles modifications il faudrait effectuer à la procédure ou, encore mieux, quel serait le code MPD résultant?

3. Encodage de Huffman (10 pts)

Soit un texte contenant n caractères différents — $c_1, c_2, \ldots, c_n$ — où le nombre d'occurrences des caractères est défini comme suit :

$\texttt{nbOccurrences}(c_1) = 1$
$\texttt{nbOccurrences}(c_2) = 1$
$\texttt{nbOccurrences}(c_i) = \texttt{nbOccurrences}(c_{i-1}) + \texttt{nbOccurrences}(c_{i-2}) \quad$ pour $n > 2$

a. Dessinez l'arbre produit par l'application de l'algorithme `huffman` sur un tel ensemble de caractères pour $n = 6$. Vous devez indiquer clairement le poids associé à chaque noeud (interne ou feuille). Vous devez aussi respecter les **contraintes suivantes** : le sous-arbre de poids inférieur doit être *à gauche* (bit 0) ; si deux sous-arbres ont le même poids, celui le moins profond doit être *à gauche* ; si même poids et même profondeur, celui avec le caractère *d'index le plus grand* doit être à gauche.

b. Soit n une valeur entière positive (pas nécessairement égale à 6!). Quel sera alors l'encodage de c_i, pour $i = 1, \ldots, n$?

4. Files de priorité (10 pts)

Soit la procédure suivante `proc0`, où les `assert` spécifient les pré-conditions, et où le prédicat `enOrdre` assure que les `N` éléments de chaque `Liste ls[i]` sont ordonnés, du plus petit au plus grand :

```
type Tableau = [N] int;   # Tableau de N elements, N une constante.

procedure proc0( Tableau ls[*], int k, res int r[*] )
{
  assert( ub(ls) == k );
  for [i = 1 to k] { assert( ub(ls[i]) == N & enOrdre(ls[i], N) ); }
  assert( ub(r) == k*N );

  cap FilePriorite fp = create FilePriorite( k );

  int pos[k] = ([k] 1);
  for [i = 1 to k] { fp.inserer( ls[i][1], i ); }

  int i = 1;
  while ( ~fp.estVide() ) {
    int l;
    fp.retirerMin( r[i], l );
    i += 1;
    if ( pos[l] < N ) {
      pos[l] += 1;
      fp.inserer( ls[l][pos[l]], l );
    }
  }
}
```

a. Que fait cette procédure? Expliquez brièvement.

b. Supposons que la file de priorité soit mise en oeuvre à l'aide d'un *monceau* (*heap*). Quelle serait alors la complexité asymptotique de la procédure `proc0`?

c. **Bonus** Supposons que les tailles des tableaux passés en argument sont spécifiées par un argument `int taille[*]` plutôt que par une constante `N`. *a*) De quelle façon faudrait-il modifier l'algorithme? *b*) Quelle serait alors la complexité asymptotique, toujours pour une file de priorité mise en oeuvre à l'aide d'un monceau?

5. Problème de programmation dynamique (15 pts)

Une usine dispose de gros tuyaux, tous du même diamètre mais de longueurs différentes. Les longueurs des tuyaux, en mètres, sont données par le tableau suivant — les tuyaux ne sont pas nécessairement de longueur différente, donc il est possible, pour $i \neq j$, que `tuyaux[`i`]` = `tuyaux[`j`]` :

```
type Longueur = int;
Longueur tuyaux[NB_TUYAUX];
```

Les clients de l'usine peuvent commander des tuyaux de diverses longueurs. Lorsqu'aucun tuyau de la longueur demandée n'est disponible, il faut alors assembler des tuyaux, et ce à l'aide de *joints* — l'usine ne possède pas de scie, donc impossible de couper un tuyau. Chaque joint mesure un (1) mètre.
Exemple, pour `tuyaux[5] = (2, 3, 2, 3, 1)` : si un client désire un tuyau de 7m, alors on utilisera deux tuyaux de 3m reliés par un joint.

Le bénéfice retiré de la production et vente d'un tuyau de longueur `L` dépend du nombre de joints utilisés : alors que les tuyaux sont produits directement par l'usine, les joints doivent être achetés à l'extérieur. Le bénéfice net résultant de la vente d'un tuyau de longueur `L` est le suivant :

- Chaque mètre du tuyau résultant formé de «vrais tuyaux» rapporte 5,00 $ par mètre.
- Chaque joint (1m chacun) coûte 1,00 $.

Pour l'exemple précédent, le bénéfice net associé à la production d'un tuyau de 7m avec la sélection indiquée serait alors de 29 $ — 6m de tuyaux (bénéfice brut 30 $) et 1m de joint (coût de 1 $).

On désire définir une fonction, ayant l'interface suivante, qui détermine le bénéfice maximum pour produire un tuyau de longueur `L` :

```
procedure benefice( Longueur tuyaux[*], int n, Longueur L ) returns int benef
# NOTE:
#   benef = −∞ s'il n'est pas possible d'obtenir la longueur désirée
```

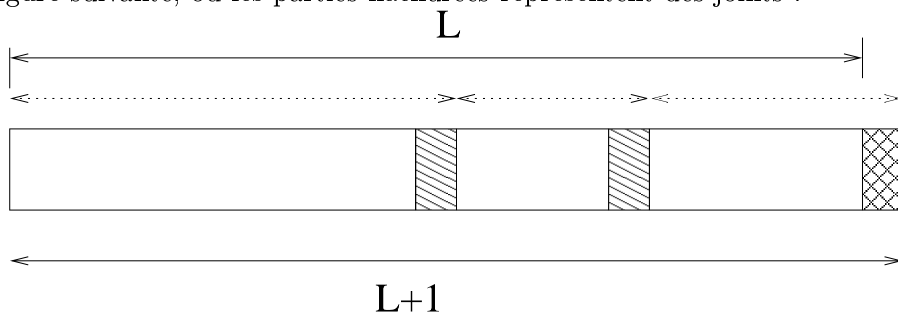
a. Décrivez, de façon informelle ou à l'aide d'équations récursives, ce que représente en termes du problème la structure de donnée auxiliaire B utilisée pour la mémorisation des solutions intermédiaires, donc ce que représente $B[i, j]$.

b. Écrivez, en MPD, une version de la fonction `benefice` basée sur la *programmation dynamique* (sans récursion).

c. Donnez le complexité asymptotique de votre procédure.

Indices :

- Soit la figure suivante, où les parties hachurées représentent des joints :



- Vous pouvez utiliser, sans le définir, un *opérateur* $\oplus$ qui retourne la valeur $-\infty$ lorsque l'un des opérandes est $-\infty$: $-\infty \oplus x = x \oplus -\infty = -\infty$.

a. $B[i,j] =$

b.

```
procedure benefice( Longueur tuyaux[*], int n, Longueur L ) returns int benef
{

}
```

c. $T_{\texttt{benefice}}(n, L) \in \Theta$

6. Algorithme avec *backtracking* pour affectation de tâches (10 pts)

L'extrait de programme MPD présenté à la page suivante (vous pouvez détacher cette page, pour en faciliter la consultation) est une version révisée de l'algorithme avec marche arrière (*backtracking*) pour l'affectation de n tâches à n agents.

a. Soit la matrice de coûts suivante :

	t_1	t_2	t_3
a_1	3	9	5
a_2	3	5	2
a_3	8	8	6

Complétez la table suivante en indiquant, pour chacun des six (6) appels à `mettreAJourMin`, la tâche affectée à chacun des agents, le coût associé à cette affectation, ainsi que la valeur de `coutMin` avant vs. après l'appel.

Tâche a_1	**Tâche a_2**	**Tâche a_3**	**Coût associé**	`coutMin` **avant**	`coutMin` **après**

b. La procédure `trouverAffectationMinBBRec` (bas de la page suivante) est une version *branch-and-bound* de l'algorithme révisé.

Pour chacune des fonctions d'estimation suivantes, dites s'il s'agit d'un estimé intéressant : est-il «correctement optimiste» (permet de trouver la solution optimale et va élaguer certaines branches), «trop optimiste» (permet de trouver la solution mais risque d'élaguer peu de branches) ou «pessimiste» (peut faire *rater* la solution optimale)?

(a) Pour un agent pour lequel on a déjà affecté une tâche, on utilise le coût de la tâche qui lui a été affectée. Pour n'importe quel autre agent, on utilise simplement le coût minimum *parmi toutes les tâches non encore affectées.* L'estimé total est alors la somme des valeurs pour les différents agents.

(b) On retourne simplement comme estimé la valeur `low(int)`, i.e., $-\infty$?

```
# Extrait de programme MPD pour l'affectation de n tâches à
# n agents en utilisant un algorithme avec marche arrière.

procedure mettreAJourMin( Affectation aff, Cout c,
                          ref Affectation affMin, ref Cout coutMin )
{
   if ( c < coutMin ) {
     affMin = aff;
     coutMin = c;
   }
}

procedure acceptable( Affectation aff, Agent a, Tache t ) returns bool r
{
   r = true;
   for [i = a+1 to ub(aff)] {
     r &= aff[i] != t;
   }
}

procedure trouverAffectationMinMARec( Couts cs, Agent a, Affectation aff,
                                      ref Affectation affMin, ref Cout coutMin )
{
  if ( a == 0 ) {
    mettreAJourMin( aff, cout(cs, aff), affMin, coutMin );
  } else {
    for [t = 1 to NB_TACHES st acceptable(aff, a, t)] {
      aff[a] = t;
      trouverAffectationMinMARec( cs, a-1, aff, affMin, coutMin );
    }
  }
}

procedure trouverAffectationMinMarcheArriere( Couts cs ) returns Affectation affMin
{
  Affectation aff;
  Cout coutMin = high(int);
  trouverAffectationMinMARec( cs, ub(cs), aff, affMin, coutMin );
}
```

```
# Extrait de programme MPD pour l'affectation de n tâches à
# n agents en utilisant un algorithme branch-and-bound.

procedure trouverAffectationMinBBRec( Couts cs, Agent a, Affectation aff,
                                      ref Affectation affMin, ref Cout coutMin )
{
  if (a == 0) {
    mettreAJourMin( aff, cout(cs, aff), affMin, coutMin );
  } else {
    for [t = 1 to NB_TACHES st acceptable(aff, a, t)] {
      if ( coutEstime(cs, aff, a, t) < coutMin ) {
        aff[a] = t;
        trouverAffectationMinBBRec( cs, a-1, aff, affMin, coutMin );
      }
    }
  }
}
```