INF7440 — Conception et analyse d'algorithmes
Examen intra (Hiver 2005)

Durée: 90 minutes **Documentation autorisée:** Documentation personnelle.

Remarques :

- Vous devez écrire vos algorithmes en utilisant une notation semblable à MPD.
- Pour simplifier la présentation ou l'analyse des algorithmes, vous pourrez supposer, lorsque cela sera utile, que la taille du problème ou tout autre paramètre utile est un nombre d'une forme appropriée, par ex., $n = 2^k$, $\frac{n}{\lg n} = 2^l$, etc. Toutefois, si vous posez une telle hypothèse, *vous devez l'indiquer clairement et explicitement.*
- À moins d'indication contraire, vous pouvez utiliser le théorème général (n'importe quelle version) pour résoudre vos équations de récurrence.

1. Comparaison d'algorithme (20 points)

Soit les deux (2) algorithmes suivants :

```
# Premier algorithme.
procedure proc1Rec( ref int A[*], ref int B[*], int bInf, int n ) returns int r
{
   r = 0;
   if ( n <= SEUIL ) {  # SEUIL est une constante positive (fixe).
     for [k = bInf to bInf+n-1] {
       r = r + A[k] * B[k];
     }
   } else {
     for [l = 0 to 3] {
       r = r + proc1Rec( A, B, bInf+l*(n/4), n/4 );
     }
   }
}

procedure proc1( int A[*], int B[*], int n ) returns int r
{  r = proc1Rec( A, B, 1, n );  }


############################################################

# Deuxieme algorithme.
procedure proc2( int A[*], int B[*], int n ) returns int r
{
  r = 0;
  for [i = 1 to n/4] {
    r = r + A[i] * B[i];
    for [j = 1 to 3] {
      r = r + A[i+j*n/4] * B[i+j*n/4];
    }
  }
}

############################################################
```

Pour chacun de ces algorithmes, déterminez

a. le temps d'exécution de l'algorithme (complexité asymptotique Θ) ;

b. le nombre exact d'additions.

Justifiez brièvement chacune de vos réponses.

2. Algorithme de programmation dynamique (20 points)

Dans cet exercice, on s'intéresse à un algorithme de programmation dynamique pour la résolution du problème d'optimisation suivant :

> Vous disposez d'un nombre *illimité* de pièces de n types différents. On note $v[i]$ la valeur des pièces de type i. Par exemple, si $n = 3$ et $v[1] = 1$, $v[2] = 3$ et $v[3] = 5$, alors vous disposez de pièces de valeur 1, 3 et 5. Vous devez composer un montant de M à l'aide de ces pièces ($M \geq 1$), pour rendre la monnaie par exemple.
>
> On cherche à déterminer $nbPieces_{min}(M, v, n)$ le *nombre minimum de pièces* nécessaires pour obtenir le montant M à l'aide de pièces ayant les n valeurs $v[1], \ldots, v[n]$.

Par exemple, si $n = 3$, $v[1] = 1$, $v[2] = 3$, $v[3] = 5$ et $M = 9$, il faut au minimum trois ($nbPieces_{min}(M, v, 3) = 3$) pièces pour obtenir le montant 9, lequel peut être obtenu de différentes façons : *i*) une pièce de 5, une pièce de 3 et une pièce de 1 ; *ii*) trois pièces de 3.

L'algorithme de type diviser-pour-régner suivant permet de calculer $nbPieces_{min}(M, v, n)$, mais en temps *non polynomial* :

```
procedure nbPiecesMin( int M, int v[*], int n ) returns int nbPieces
{
  if ( M == 0 ) {
    nbPieces = 0;
  } else {
    nbPieces = high(int) # +∞
    for [ i = 1 to n ] {
      if ( M >= v[i] ) {
        nbPieces = min( nbPieces, nbPiecesMin(M-v[i], v, n)+1 );
      }
    }
  }
}
```

a) Expliquez, à l'aide d'un exemple, pourquoi/comment cet algorithme n'est pas efficace.

b) Écrivez un algorithme de type "programmation dynamique ascendante" (donc itératif, c'est-à-dire, non récursif) qui permet de calculer le nombre minimum de pièces mais de façon asymptotiquement plus efficace que l'algorithme ci-dessus. Donnez, en justifiant brièvement votre réponse, la complexité en temps (asymptotique) de votre algorithme, et sa complexité en espace.

Remarques importantes :

- Il est possible qu'un montant M donné ne puisse pas être obtenu avec les pièces disponibles. Dans ce cas, $nbPieces_{min}$ retournera $+\infty$ (obtenu avec `high(int)` en MPD).
- Vous pouvez supposer que `high(int)+1 == high(int)`.