INF7440 — Conception et analyse d'algorithmes
Examen final (Automne 2002)

Durée: 18h00 – 21h00 (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Remarques :

- L'examen comporte trois (3) problèmes (donc, en gros, une heure par problème). Les deux premiers valent 10 points chacun, alors que le dernier vaut 15 points (un algorithme séquentiel et un autre parallèle).
- Vous devez écrire vos algorithmes, surtout lorsqu'il s'agit de segment de code avec parallélisme, en utilisant une notation semblable, le plus possible, à la notation MPD. Toutefois, dans le cas des instructions `co`, vous pouvez utiliser une notation *moins stricte* que la notation MPD acceptée par le compilateur.

  Plus précisément, le compilateur MPD *n'accepterait pas* l'instruction `co` suivante, parce que le corps de l'instruction `co` n'est pas une invocation :

```
co [i = 1 to n]
  x[i] = y[i] + z[i];
oc
```

  Le compilateur obligerait plutôt à introduire une procédure auxiliaire comme suit :

```
procedure plus( int x, int y ) returns int r
{ r = x + y; }

...
co [i = 1 to n]
  x[i] = plus( y[i], z[i] )
oc
```

  Dans le cadre de l'examen, la première notation (avec code arbitraire dans le corps d'un `co`, donc sans procédure auxiliaire) sera acceptée.

- Pour simplifier la présentation et l'analyse des algorithmes que vous écrirez, vous pourrez supposer, lorsque cela sera utile, que la taille du problème est une puissance d'un nombre approprié, par ex., $n = 2^k$.
- Toutes les opérations arithmétiques de base, y compris l'exponentiation (par ex., `2**n`), sont considérées comme des opérations d'exécutant en temps $\Theta(1)$.

1. Sous-chaîne commune la plus longue (10 pts)

Soit *sch* et *ch* deux chaînes de caractères. Notons par $sch \in ch$ le fait que *sch* est une sous-chaîne de *ch*, c'est-à-dire que tous les caractères de *sch* apparaissent dans *ch* dans le même ordre, mais pas nécessairement de façon contiguë. Par ex., `"abcc"` $\in$ `"xabcdecf"`, `"cis"` $\in$ `"chiens"`, ou encore `""` $\in$ `"xabcdef"` (la chaîne vide est une sous-chaîne de n'importe quelle chaîne).

Soit alors deux chaînes de caractères ch_1 et ch_2. On désire déterminer la *longueur* de la plus longue sous-chaîne de caractères qui est commune à ces deux chaînes. En d'autres mots, on cherche à trouver la longueur de la chaîne *sch* telle que :

- $sch \in ch_1 \wedge sch \in ch_2$
- $\texttt{longueur}(sch) = \texttt{max}\{sch' \in ch_1 \wedge sch' \in ch_2 :: \texttt{longueur}(sch')\}$

Un algorithme récursif permettant de résoudre ce problème est le suivant :

```
char ch1[n1];
char ch2[n2];

procedure maxSousChaineCommune( int n1, int n2 ) returns int len
{
  if (n1 == 0 | n2 == 0) {
    len = 0;
  } else if (ch1[n1] == ch2[n2]) {
    len = 1 + maxSousChaineCommune(n1-1, n2-1);
  } else {
    len = max( maxSousChaineCommune(n1-1, n2),
               maxSousChaineCommune(n1,   n2-1) );
  }
}

int d1 = maxSousChaineCommune(n1, n2);
write( "longueur de sous-chaine commune maximum =", d1 );
```

a. Illustrez à l'aide d'un exemple (c'est-à-dire, avec des valeurs spécifiques pour `ch1` et `ch2`) pourquoi cet algorithme récursif n'est pas efficace.

b. Écrivez un algorithme *non récursif* et *asymptotiquement plus efficace* pour résoudre ce problème. Quelle est la complexité asymptotique de votre algorithme?

Note : Si vous ne réussissez pas à écrire un algorithme *non récursif*, vous pouvez écrire un algorithme récursif en autant qu'il soit asymptotiquement plus efficace que celui présenté plus haut. Toutefois, vous n'aurez qu'une partie des points pour la partie algorithme.

c. De quelle façon pourriez-vous paralléliser l'algorithme de la partie b.? (Vous n'êtes pas obligé de donner le code ; indiquez simplement, en mots, comment vous procéderiez.) Que deviendrait alors le temps d'exécution?

2. Sac-à-dos fractionnaire (10 pts)

On désire résoudre le problème du sac à dos fractionnaire, où le sac peut contenir un poids maximum W et où les poids et bénéfices des `n` éléments à mettre dans le sac sont donnés par les tableaux suivants :

- `p[1:n]` : les poids des différents items.
- `b[1:n]` : les bénéfices attribués aux différents items.

Pour simplifier, on suppose que les tableaux `p` et `b` sont déjà ordonnés en fonction des rapports bénéfice par unité de poids des divers items, c'est-à-dire que la propriété suivante est satisfaite :

$$\frac{\texttt{b[1]}}{\texttt{p[1]}} \geq \frac{\texttt{b[2]}}{\texttt{p[2]}} \geq \ldots \geq \frac{\texttt{b[n-1]}}{\texttt{p[n-1]}} \geq \frac{\texttt{b[n]}}{\texttt{p[n]}}$$

a. Concevez un algorithme parallèle de style PRAM, écrit en notation MPD, pour résoudre cette version du problème du sac à dos fractionnaire. Votre algorithme doit évidemment être plus rapide que l'algorithme séquentiel mais n'a pas besoin d'être de coût optimal.

   En plus de retourner le bénéfice total, la quantité de chaque item à inclure dans le sac doit aussi être produite. Plus précisément, votre algorithme doit donc réaliser le corps de la procédure ayant l'en-tête suivante :

```
procedure remplirSac( real p[*], real b[*], int n, real W,
                      ref real benefTotal, ref real qtes[*] )
# PRECONDITION
#   ALL( 1 <= i < n :: b[i]/p[i] >= b[i+1]/p[i+1] )
#   W > 0
# POSTCONDITION
#   ALL( 1 <= i <= n :: qtes[i] <= poids[i] )
#   SUM( 1 <= i <= n :: qtes[i] ) = W
#   benefTotal = SUM( 1 <= i <= n :: (qtes[i]/poids[i])*b[i] )
```

b. Quelles sont les caractéristiques de votre algorithme (justifiez brièvement vos réponses)?

   (i) Nombre (exact) de processeurs requis.
   (ii) Temps d'exécution (complexité asymptotique).
   (iii) Coût (complexité asymptotique).
   (iv) Optimalité (votre algorithme est-il optimal, en termes de coût, par rapport à un algorithme séquentiel pour ce même problème)?

Suggestion : Vous pouvez utiliser, sans la définir, la procédure suivante (temps : $\Theta(\lg\ n)$; coût : $\Theta(n)$) :

```
optype OpBinaire = (real, real) returns real;

procedure calculerPrefixes( real x[*], ref real prefixes[*], int n, cap OpBinaire op0 )
```

3. Évaluation d'un polynôme (15 pts)

Soit un tableau `p` de taille `n` représentant un polynôme (entier) de degré `n-1`.
Plus précisément, soit le tableau `p` déclaré comme suit : `int p[n];`
Ce tableau va alors représenter le polynôme suivant :

```
p[1] * x**(n-1)  +  p[2] * x**(n-2)  +  ...  +  p[n-1] * x  +  p[n]
```

On désire obtenir une procédure permettant d'évaluer un polynôme `p` en un point `x`, procédure ayant l'en-tête suivante :

```
procedure evaluer( int p[*], int n, int x ) returns int s
#  Entrees
#    p = tableau des coefficients du polynome
#    n = taille du tableau
#    x = point ou on evalue le polynome
#  Sortie:
#    s =  valeur de p au point x
```

a. Version récursive séquentielle (10 points) :

   (i) Écrivez, dans la notation MPD, une version *séquentielle* et *récursive* de la procédure `evaluer` (basée sur une stratégie diviser-pour-régner dichotomique).

   (ii) Donnez les équations de récurrence décrivant le temps d'exécution de cette procédure.

   (iii) Quelle sera la complexité asymptotique du temps d'exécution de votre procédure?

b. Version récursive parallèle (5 points) :

   (i) Écrivez, dans la notation MPD, une version *parallèle* de la procédure `evaluer`.

   (ii) Combien de processeurs sont requis (nombre exact) par votre algorithme?

   (iii) Quel sera le temps d'exécution (asymptotique)?

   (iv) Quel sera le coût de l'algorithme?

Note : Un *bonus* (2 points) sera accordé pour une version parallèle *non récursive* (moyennement difficile). Un autre *bonus* équivalent sera accordé pour une version parallèle de coût optimal (pas mal plus difficile).