INF7440 — Conception et analyse d'algorithmes
Examen final (Automne 2007)

Durée: 3–3$\frac{1}{2}$ heures **Documentation autorisée:** Documentation personnelle.

Remarques :

- L'examen comporte trois (3) questions obligatoires comptant respectivement pour 10, 20 et 20 points. L'examen comporte aussi une question et des sous-questions au choix ou en *bonus* (max. 10 points). Donc, il donc *possible* d'obtenir une note supérieure à 50 ($\frac{50}{50} = 100$ %).

- Vous devez écrire vos algorithmes, surtout lorsqu'il s'agit de segment de code avec parallélisme, en utilisant une notation semblable à MPD. Toutefois, dans le cas des instructions `co`, vous pouvez utiliser une notation *moins stricte* que celle acceptée par le compilateur, par exemple, instructions `CO` (pouvant contenir des segments arbitraires de code) ou même `CO-SYNC` (idem mais avec exécution synchrone à la PRAM).

- 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 un nombre d'une forme appropriée, par ex., $n = 2^k$, $\frac{n}{\lg\ n} = 2^l$, etc. Toutefois, si vous utilisez une telle hypothèse, vous devez l'indiquer clairement et explicitement

  Pour justifier vos analyses de complexité, vous pouvez aussi, le cas échéant, annoter chacune des parties de votre algorithme avec sa complexité associée puis ensuite synthétiser le tout.

- À moins d'indication contraire, vous pouvez utiliser, sans la définir, la procédure parallèle suivante, de temps $\Theta(\lg\ n)$, de coût et de travail optimal $\Theta(n)$ (utilisant $\frac{n}{\lg\ n}$ processeurs) :

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

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

  Notez que comme l'argument `x` est passé en mode `val` (valeur, *copy-in*, mode par défaut) et que `prefixes` est passé en mode `res` (résultat, *copy-out*), il est donc possible d'effectuer un appel de la forme «`calculerPrefixes( a, a, n, f )`».

- *À moins d'indication contraire*, les algorithmes parallèles que vous écrivez n'ont pas besoin d'être optimaux (ni en travail, ni en coût).

- Rappels sur certaines opération MPD que vous pouvez utiliser :
  - `int(true) == 1 & int(false) == 0`
  - `low(int)` représente le plus petit entier ; `high(int)` représente le plus grand entier.
  - `x ** 2 == x * x`.
  - `lg(n, b)` retourne le logarithme base `b` de `n`, pour `n` une puissance de `b`.

1. Analyse d'un algorithme avec parallélisme récursif (10 points)

Soit l'algorithme avec parallélisme récursif suivant, où `proc1` est le point d'entrée principal :

```
# Une constante non négative K, qui ne dépend pas de n.
const int K = ...;

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

procedure proc2( ref int A[*], int i, int j ) returns int r
{
  int m = j-i+1;

  if (m <= 2**K) {
    int T[m];
    CO [k = 1 to m]
      T[k] =  A[k+i-1] ** 2;
    OC
    calculerPrefixes( T, T, m, g );
    r = T[m];
  } else {
    int mid = (i + j)/2;
    int r1, r2;
    co
       r1 = proc2( A, i,     mid );
    // r2 = proc2( A, mid+1, j   );
    oc
    r = r1 + r2;
  }
}

procedure proc1( int A[*], int n ) returns int r
# PRECONDITION
#   n indique la taille de A ET n est une puissance de 2
{
  r = proc2( A, 1, n );
}
```

a. Expliquez ce que fait la procédure `proc1`.

b. Donnez les équations de récurrence décrivant le temps d'exécution $T(n)$ et déterminez la solution asymptotique.

c. Donnez le nombre (maximum et exact) $P(n)$ de processeurs utilisés.

d. Donnez le coût $C(n)$ de l'algorithme.

e. Indiquez si ce coût est optimal par rapport à un algorithme séquentiel effectuant la même tâche.

2. Calcul parallèle du nombre de zéros (20 points)

Soit `A` un tableau d'entiers de taille `n`. Soit une procédure dont l'en-tête est la suivante :

```
procedure nbZeros( int A[*], int n ) returns int r
# PRECONDITION
#   n = nombre d'éléments de A
#   n > 0
#   n est une puissance de 2
# POSTCONDITION
#   r = nombre de valeurs de A qui sont égales à 0
#
# Formellement :
#   r = NUMBER( 1 <= i <= n SUCH THAT A[i] == 0 :: i )
```

Répondez à deux (2) des trois questions suivantes.

Si vous répondez aux trois questions, je choisirai les deux meilleures parmi les trois.
Si vous répondez correctement aux trois questions, un bonus pourra vous être accordé (≤ 5 pts).

Indice : Voir les rappels sur le langage MPD à la page 1.

a. Utilisez le module `DPR2` vu dans le devoir 4 (voir feuille jointe à l'examen pour la description et la mise en oeuvre de ce module) pour écrire une mise en oeuvre de la fonction `nbZeros`, et ce en vous assurant que l'algorithme parallèle résultant soit *optimal en coût*.

b. Écrivez une mise en oeuvre de la fonction `nbZeros` qui, pour générer du parallélisme, utilise uniquement des instructions `co` simples ou des appels à la procédure `calculerPrefixes` (qui est déjà parallèle et dont l'interface est décrite à la première page) — vous n'avez donc pas le droit ici d'utiliser explicitement du parallélisme récursif. Le temps d'exécution de la fonction résultante *doit être au pire de complexité logarithmique*.

c. Écrivez une mise en oeuvre de la fonction `nbZeros` qui utilise une stratégie diviser-pour-régner *trichotomique* — c'est-à-dire avec division en trois (3) sous-problèmes — *mais sans utiliser de récursion explicite*. Vous devez donc plutôt utiliser un processus itératif de β-réduction logarithmique (simulation *ascendante* itérative de la récursion).

3. Ramassage de pièces sur un échiquier (20 points)

On a un échiquier, de taille $n \times n$, sur lequel reposent des pièces de monnaie. En partant de la case du coin supérieur gauche, on désire se déplacer, d'une case à la fois, jusqu'à la case du coin inférieur droit, et ce en ramassant le plus grand nombre possible de pièces. Toutefois :

- Les déplacements d'une case à une autre case adjacente sont soumis à la contrainte suivante : le déplacement ne peut être que d'une case *vers la droite* ou d'une case *vers le bas*.
- Lorsqu'on arrive sur une case qui contient la valeur 0, alors *on perd toutes les pièces déjà accumulées* — donc, c'est comme s'il y avait un immense trou sur cette case : (

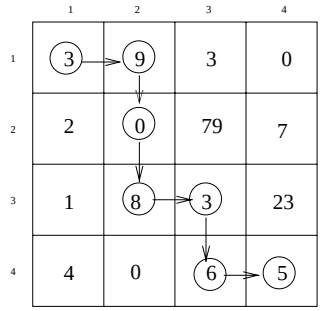

Figure 1: Un exemple d'une série de déplacements sur un échiquier 4×4

La figure 1 présente un exemple d'une série possible de déplacements respectant ces contraintes sur un échiquier de taille 4×4. La valeur indiquée sur chaque case spécifie le nombre de pièces sur la case. Le *bénéfice* de cette série de déplacements serait alors de $8 + 3 + 6 + 5 = 22$ — à cause de la case $[2, 2]$ avec un 0 qui fait perdre les 12 pièces accumulées depuis le départ. Notons qu'on constate facilement que cette série de déplacements n'est pas optimale.

Notons par $P[i, j]$ le nombre de pièces à la position $[i, j]$, où le coin supérieur gauche correspond à $[1, 1]$ et le coin inférieur droit correspond à $[n, n]$.

Le bénéfice maximum pour se rendre de la case $[1, 1]$ jusqu'à la case $[i, j]$ (inclusivement), noté $B[i, j]$, peut être caractérisé par les équations récursives suivantes, pour $i > 1, j > 1$ — les cas de base sont omis et ce sera à vous de les expliciter... :

$$B[i,j] \quad = \quad \begin{cases} 0 & \text{si } P[i,j] = 0 \\ P[i,j] + max(B[i-1,j], B[i,j-1]) & \text{si } P[i,j] \neq 0 \end{cases} \quad \text{(pour } i > 1 \text{ et } j > 1\text{)}$$

Informellement : pour le cas général de la position $[i, j]$ non situé sur une bordure à gauche ou en haut, il y deux façons différentes d'arriver à cette position (de la case du haut, ou de celle de gauche) et il faut donc trouver le maximum parmi ces deux possibilités.

Supposons maintenant que l'on modifie les règles pour compter le nombre de pièces accumulées lors d'une série de déplacements de la façon suivante : lorsqu'on arrive sur une case qui ne contient aucune pièce — i.e., sur l'exemple de la figure 1, l'une des trois cases avec la valeur 0 — alors on garde simplement les pièces déjà accumulées, sans en ajouter de nouvelles mais sans non plus perdre celles déjà ramassées. Ainsi, pour l'exemple de la figure 1, le déplacement indiqué conduirait alors, avec ces nouvelles règles, à un bénéfice de 34.

Soit une fonction MPD ayant l'interface suivante :

```
procedure piecesMax( int P[*,*], int n ) returns int r
```

Les deux questions qui suivent demandent d'écrire deux mises en oeuvre différentes de cette procédure, et ce pour la version *modifiée* du problème.

a. Donnez une mise en oeuvre, séquentielle et itérative (non récursive), de la fonction `piecesMax` fondée sur la *programmation dynamique* pour résoudre le problème modifié.

   Quel est le temps d'exécution (complexité asymptotique) de votre algorithme?

b. Donnez une mise en oeuvre, séquentielle, de la fonction `piecesMax` utilisant un algorithme vorace (glouton) pour résoudre le problème modifié.

   Quel est le temps d'exécution (complexité asymptotique) de votre algorithme?

   Donnez un exemple, sur un échiquier de la taille de votre choix, où votre algorithme *ne produit pas la solution optimale*.

La sous-question suivante est une question bonus (max. 3 pts).

c. Supposons qu'on modifie à nouveau les règles de déplacement de façon à ce qu'il soit permis de terminer la série de déplacements sur *n'importe quelle case située sur la bordure en bas ou à droite*, donc sur n'importe quelle case située sur la dernière ligne ou sur la dernière colonne — ceci peut être intéressant si les cases qui «suivent» pour se rendre au coin inférieur droit contiennent des zéros (0).

   De quelle façon votre algorithme de programmation dynamique devrait-il être modifié pour résoudre cet autre problème? Est-ce que le temps d'exécution serait différent? Justifiez brièvement.

4. Algorithme *branch-and-bound* (optionnelle) (Bonus max. 5 points)

Soit un type `Arbre` binaire défini par les types suivants — l'invariant indique que le champ `valeur` est toujours non négatif, qu'un noeud interne a deux enfants et qu'une feuille n'en a aucun :

```
type Sorte    = enum( FEUILLE, NOEUD );
type Arbre    = ptr ArbreRec;
type ArbreRec = rec( Sorte sorte; Arbre gauche, droit; int valeur; );
                     # INVARIANT
                     #   valeur >= 0
                     #   sorte = FEUILLE => gauche  = null & droit  = null
                     #   sorte = NOEUD   => gauche ~= null & droit ~= null
```

La procédure suivante permet d'explorer l'ensemble des chemins allant de la racine d'un arbre vers ses feuilles et d'imprimer le coût associé à ce chemin. Toutefois, pour que le chemin (et son coût) soit imprimé, tous les noeuds sur ce chemin doivent satisfaire la condition spécifiée par `satisfaitCond` (opération de type `Condition`) :

```
optype Condition = (Arbre) returns bool;

procedure traverserArbre( Arbre a, int coutCumul, cap Condition satisfaitCond )
{
  if ( satisfaitCond(a) ) {
    coutCumul += a^.valeur;
    if ( a^.sorte == FEUILLE ) {
      printf( "Cout du chemin = %d\n", coutCumul );
    } else {
      traverserArbre( a^.gauche, coutCumul, satisfaitCond );
      traverserArbre( a^.droit,  coutCumul, satisfaitCond );
    }
  }
}
```

Ainsi, la fonction et l'appel suivants permettraient d'imprimer le coût de tous les chemins partant de `racine` vers des feuilles où tous les noeuds le long du chemin ont une `valeur` supérieure à 10 :

```
procedure sup10( Arbre a ) returns bool r
{ r = a^.valeur > 10; }

traverserArbre( racine, 0, sup10 );
```

a. Écrivez un segment de code et modifiez la procédure `traverserArbre` pour faire en sorte que seul le coût du chemin de coût minimal soit imprimé, et ce en utilisant une stratégie de type *branch-and-bound*.

b. Serait-il possible de faire le même genre de modification, toujours avec une stratégie *branch-and-bound*, pour le cas où on voudrait plutôt imprimer uniquement le coût du chemin de coût maximum? Justifiez brièvement.