INF7440 — Conception et analyse d'algorithmes
Examen final (Automne 2004)

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

Remarques :

- L'examen comporte trois (3) questions obligatoires (1, 2 et 3), comptant respectivement pour 25, 25 et 30 points. L'examen comporte aussi deux *questions bonus* (max. 15 points), optionnelles, dont une spécifique à votre groupe-cours (jour ou soir). Il est donc possible d'obtenir une note supérieure à 100 % (supérieure à 80).

- 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 `CO-SYNC` (idem mais avec exécution synchrone).

- 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 posez 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 $\Theta(\lg\ n)$ et de coût optimal $\Theta(n)$ (qui utilise donc $\frac{n}{\lg\ n}$ processeurs) :

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

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

- Les algorithmes que vous écrivez *n'ont pas besoin d'être optimaux* (en travail ou en coût). Toutefois, si on vous demande d'écrire un algorithme parallèle, cet algorithme devra évidemment être *asymptotiquement plus rapide* que l'algorithme séquentiel correspondant.

- En MPD, on a les équivalences suivantes :
  - `int(true) == 1`
  - `int(false) == 0`

1. Algorithmes parallèles pour calcul des inversions (25 points)

Soit `A` un tableau (vecteur) de `n` nombres entiers. On veut concevoir une procédure qui permet de déterminer le nombre d'*inversions* d'éléments *adjacents* de `A`, i.e., le nombre d'éléments adjacents qui ne sont pas correctement ordonnés :

```
procedure nbInversions( int A[*], int n ) returns int nb
# POSTCONDITION
#   nb = SUM( 1 <= i < n SUCH THAT A[i] > A[i+1] :: 1 )
```

Quelques exemples :

- `nbInversions( (1, 2, 2, 2), 4 )` $\Rightarrow 0$
- `nbInversions( (1, 2, 3, 4, 3, 2, 1, 0), 8 )` $\Rightarrow 4$
- `nbInversions( (8, 7, 6, 4, 5, 3, 1, 2), 8 )` $\Rightarrow 5$

a. (15 points) Version parallèle récursive :

- Écrivez une version *parallèle* et récursive de la procédure `nbInversions` basée sur une stratégie diviser-pour-régner.
- Donnez les équations de récurrence décrivant le temps d'exécution de cette procédure. (N'oubliez pas de spécifier les conditions appropriées sur `n`.) Quel est alors le temps d'exécution résultant (complexité asymptotique)?
- Combien de processeurs sont requis (nombre exact)?
- Quel est le coût de votre algorithme? Est-il de coût optimal?
- Le *travail* effectué par votre algorithme est-il inférieur, égal ou supérieur au coût? Justifiez brièvement votre réponse.

b. (10 points) Version PRAM CREW :

- Écrivez une version *parallèle* mais non récursive de la procédure `nbInversions`. Cette version parallèle doit évidemment être *asymptotiquement plus rapide* qu'une version de cette procédure qui serait séquentielle ; elle doit aussi respecter le modèle CREW.
- Quel est le temps d'exécution (asymptotique)?
- Combien de processeurs sont requis (nombre exact)?
- Quel est le coût de votre algorithme? Est-il de coût optimal?

2. Programmation dynamique (25 points)

Une personne veut investir un certain montant d'argent — M dollars (avec M un multiple de 1000) — dans des fonds à rendement fixé. Des possibilités d'investissement dans n fonds différents s'offrent à cette personne, tel qu'indiqué dans le tableau suivant — le montant investi dans un fonds doit nécessairement être une des valeurs indiquées (1000 $ ou 5000 $), le bénéfice résultant de cet investissement étant alors celui indiqué dans la table :

Montant investi →	1000 $	5000 $
Montant obtenu du fonds 1	1050 $	6000 $
Montant obtenu du fonds 2	1200 $	5600 $
...	...	...
Montant obtenu du fonds n	1150 $	5800 $

Le but de cet exercice est de calculer le *montant maximum* que l'on peut obtenir en plaçant M dollars dans ces divers fonds. On s'intéresse *uniquement* au montant maximum et *non* à une description précise des investissements à effectuer pour obtenir le montant maximum. Notons qu'il n'y a aucune contrainte quant à la répartition entre les divers fonds ; ainsi, il serait permis, si cela était avantageux, de tout investir dans le même fonds.

Notations. Notons M le montant dont dispose l'investisseur, n le nombre de fonds dans lesquels il peut investir, $R[i, j]$ le revenu du fond i si on y investit j dollars (donc j vaut 0, 1000 ou 5000), et $S(K, i)$ le montant maximum que l'on peut obtenir en investissant K dollars dans les fonds 1 à i. Le but de l'exercice est donc, étant donnés M, n and R, d'écrire un algorithme pour calculer $S(M, n)$.

a. (10 points) Donnez soit une formule récursive, soit un algorithme récursif pour calculer $S(M, n)$. Prenez garde dans votre formulation au fait que M peut valoir moins de 5000.

b. (15 points) Écrivez un algorithme non-récursif, de type "*programmation dynamique ascendante*", permettant de calculer $S(M, n)$.

   L'en-tête de votre algorithme devra être le suivant :

```
procedure beneficeMax( int M, int n, int benefs1000[*], int benefs5000[*] )
          returns int benef
```

   Quelle est la complexité asymptotique de votre algorithme?

c. **(Bonus 5 points)** Écrivez un algorithme *parallèle* s'exécutant en temps logarithmique et de coût optimal permettant de trouver le bénéfice maximal. L'en-tête devra être le même que l'algorithme précédent. Notez que cette solution n'est pas basée sur la programmation dynamique.

Indice :

- Pour la solution de programmation dynamique, ce problème peut être vu comme une *variante* de l'un des problèmes vus en cours pour illustrer la programmation dynamique.

3. Backtracking et Branch-and-bound (30 points)

On s'intéresse au problème de la *couverture minimale d'un graphe* ("*vertex cover*"). Soit $G = (V, E)$ un graphe non orienté dont l'ensemble des sommets est V et l'ensemble des arêtes est E.

Soit u un sommet ; on dit que u *couvre* toutes les arêtes dont il est une des extrémités. Le but de cet exercice est de calculer la *taille du plus petit sous-ensemble* V' *de* V ($V' \subseteq V$) tel que les divers sommets de V' couvrent toutes les arêtes de E — on dit qu'on cherche à calculer *la taille* de la *couverture minimale* de G. Bien qu'il ait été montré que ce problème est NP-difficile, vous devez écrire deux algorithmes exacts pour le résoudre : un algorithme de type *backtracking* et un algorithme de type *branch-and-bound*.

Représentation du graphe. Le graphe comporte `n` sommets et vous est fourni sous la forme d'une matrice d'adjacence `G` dont chaque entrée accepte initialement deux valeurs possibles : `G[i][j]` vaut `0` si il n'existe pas d'arête entre `i` et `j`, ou vaut `1` si il existe une arête entre `i` et `j`.

a. (10 points) Écrivez un algorithme récursif de type *backtracking* permettant de calculer la taille de la couverture minimale du graphe représenté par `G`.

   Indices et contraintes.
   (1) Pour représenter un ensemble de sommets (la couverture de `G` en cours de construction), vous devez utiliser un tableau `Couv` de `n` entiers dans lequel `Couv[i]` vaut `1` si le sommet `i` est dans la couverture courante et `0` sinon.
   (2) Pour mémoriser la taille de la plus petite couverture calculée vous devez utiliser une variable globale `CouvOpt`, initialisée à `INFINI`, i.e., `high(int)`.
   (3) Pour garder trace des arêtes couvertes par la couverture courante, vous avez le droit de modifier la matrice d'adjacence `G` en donnant à la case `G[i][j]` la valeur `-1` si l'arête entre `i` et `j` est couverte.

b. (5 points) Analysez le plus précisément possible la complexité en temps dans le pire des cas de votre algorithme.

c. On suppose maintenant que vous disposez d'un tableau `D` d'entiers tel que `D[i]` est le degré du sommet `i` (nombre d'arêtes dont `i` est une des extrémités), tableau que vous pouvez modifier au cours de l'algorithme. On suppose aussi que vous disposez d'une procédure ayant l'en-tête suivant :

   ```
   procedure SommeMax( ref int D[*], int d, int k ) returns int r
   ```

   Étant donné un tableau `D` de `d` entiers non-négatifs, cette procédure renvoie, en temps linéaire (Θ(`d`)), la valeur maximale que l'on peut obtenir en sommant au plus `k` éléments de `D`.

   (i) (10 points) Expliquez comment utiliser `SommeMax` et `D` pour calculer une *borne inférieure* sur la plus petite couverture de `G` que l'on peut obtenir en étendant la couverture partielle représentée par le tableau `Couv`. On note comme suit la fonction calculant cette borne :

   ```
   procedure Estime( int G[*][*], int n, int D[*], int Couv[*] ) returns int r
   ```

   Analysez la complexité de votre fonction `Estime`.

   (ii) (5 points) En utilisant la fonction `Estime` de la question précédente, écrivez un algorithme de type *branch-and-bound* avec *parcours en largeur de l'arbre des solutions.*

   Remarque. Vous pouvez répondre à cette question sans avoir répondu à la précédente : il vous suffit de supposer que vous avez la fonction `Estime`. Vous pouvez aussi utiliser le type `Sequence` (file) vu en cours, ou un type équivalent, sans le définir.

4. [G. Tremblay] Algorithme PRAM sur un arbre binaire (Bonus 10 points)

La procédure `calculerProfondeur` (Algorithme 1, vu en cours) permet de calculer la profondeur des noeuds d'un arbre binaire par l'intermédiaire d'un calcul parallèle de préfixes sur un circuit eulérien associé à l'arbre.

Un arbre binaire est un *arbre binaire de recherche* lorsque tous les enfants du sous-arbre gauche d'un noeud ont une valeur inférieure ou égale à celle du noeud, alors que tous les enfants du sous-arbre droit ont une valeur supérieure ou égale, et ce récursivement. Une recherche d'une valeur dans l'arbre peut alors s'effectuer à l'aide d'un procédure récursive telle que la suivante :

```
procedure estPresent( PtNoeud racine, int v ) returns bool r
{
  if      ( racine^.valeur == v )   { r = true; }
  else if ( v < racine^.valeur )    { r = estPresent( racine^.gauche, v ) }
  else                              { r = estPresent( racine^.droite, v ) }
}
```

Écrivez une procédure permettant de déterminer si un arbre binaire représenté par un tableau `noeuds` (comptant `n` noeuds) est bien *un arbre binaire de recherche*. L'interface de cette procédure sera la suivante :

```
procedure determinerSiArbreDeRecherche( ref PtNoeud noeuds[*], int n )
```

Après son exécution, le champ `tmp` de chaque noeud de l'arbre devra indiquer si le sous-arbre ayant ce noeud pour racine est bien un arbre binaire de recherche, c'est-à-dire `tmp == 1` si c'est bien un arbre binaire de recherche, `tmp == 0` autrement. Si vous le désirez, vous pouvez aussi supposer à la place que `tmp` est un champ de type `bool` plutôt que `int`.

```
type PtNoeud = ptr Noeud;
type Noeud = rec( int valeur, tmp, numProcesseur; PtNoeud gauche, droite, parent; )
procedure plus( int x, int y ) returns int r { r = x + y; }

procedure calculerProfondeur( ref PtNoeud noeuds[*], int n )
{
  const int N = 3*n;
  NoeudCircuitEuler euler[N];
  construireCircuitEuler( noeuds, euler, n );

  co [i = 1 to n]  euler[A(i)].tmp = copier(  1 );
  // [j = 1 to n]  euler[B(j)].tmp = copier(  0 );
  // [k = 1 to n]  euler[C(k)].tmp = copier( -1 );
  oc
  calculerPrefixes( euler, N, plus );
  co [i = 1 to n]  noeuds[i]^.profondeur = copier( euler[C(i)].tmp );
  oc
}
```

Algorithme 1: Algorithme pour calcul de la profondeur des noeuds d'un arbre

5. [C. Chauve] Arbre des suffixes. (Bonus 10 points)

On se place dans le cadre d'un alphabet binaire Σ = `{0, 1}`. Étant donné un mot binaire `T`, de longueur `n`, et un mot binaire `M`, de longueur `m`, on dit qu'il existe une *occurrence approchée* de `M` en position `i` dans `T` si le sous-mot `T[i..i+m-1]` de `T` diffère de `M` en *au plus une lettre*. Par exemple, si `T = 0.0.1.0.1.1.0.0.1.1.1.0.0` et `M = 0.1.1.1`, on a trois occurrences approchées de `M` en `T`, en positions `2`, `4` et `8`.

On suppose que l'arbre des suffixes de `T`, noté `AS(T)`, a été calculé. Décrivez un algorithme utilisant `AS(T)` pour rechercher rapidement les positions de toutes les occurrences approchées de `M` dans `T`. Analysez la complexité de votre algorithme.

Vous pouvez décrire informellement votre algorithme et sa complexité ou, si vous êtes plus à l'aise, vous pouvez utiliser la structure de données suivante, codant un sommet de l'arbre des suffixes, pour décrire votre algorithme. On rappelle que

- chaque arête de l'arbre des suffixes est étiquetée par un segment de `T`, et les champs `deb` et `fin` encodent les bornes de ce segment,
- un sommet d'`AS(T)` reconnaît le suffixe `i` de `T`, c'est-à-dire `T[i..n]`, si la concaténation des arêtes du chamin allant de la racine à ce sommet est égal à `T[i..n]`.

```
type Sommet    = ptr SommetRec;
type SommetRec = rec(
  int suffixe;      // Indice du suffixe reconnu en ce sommet (0 si aucun)
  int deb;          // Indice de debut de l'arete vers le parent (0 si racine)
  int fin;          // Indice de fin de l'arete vers le parent (0 si racine)
  Sommet parent;    // Parent (null si racine)
  Sommet fils[0:1]; // Branchement
);
```

Algorithme 2: Sommet d'un arbre des suffixes

Vous pouvez alors supposer qu'`AS(T)` vous est donné par sa racine, dénotée `R`.