Exercices : solutions de la série #6

Solution à l'exercice 0.

La solution prendrait la forme suivante, car le corps d'une instruction `co` ne peut contenir qu'une invocation de fonction ou procédure (donc ne peut pas être une expression) :

```
procedure sommeItem( int a, int b ) returns int c
{ c = a+b; }

procedure somme( int a[*], int b[*], ref int c[*], int n )
{
  co [i = 1 to n]
    c[i] = sommeItem(a[i], b[i]);
  oc
}
```

Caractéristiques :

- $T(n) \in \Theta(1)$.
- $P(n) = n$.
- $C(n) \in \Theta(n)$.
- $W(n) \in \Theta(n)$.

L'algorithme est donc optimal tant en travail qu'en coût.

Solution à l'exercice 1.

Voici deux solutions possibles :

a. Une première solution où on introduit une procédure auxiliaire qui reçoit en argument les deux bornes de l'intervalle du tableau à traiter.

```
procedure somme_( int a[*], int b[*], ref int c[*], int i, int j )
{
  if (i == j) {
    c[i] = a[i] + b[i];
  } else {
    int m = (i+j) / 2;
    co
        somme_(a, b, c, i, m);
    //  somme_(a, b, c, m+1, j);
    oc
  }
}

procedure somme( int a[*], int b[*], ref int c[*], int n )
{
  somme_(a, b, c, 1, n);
}
```

b. Une deuxième solution qui utilise des tranches de tableaux. Dans ce cas, il faut noter l'utilisation du mode `res`, puisque le mode `ref` ne peut pas être utilisé pour des tranches de tableaux.

```
procedure somme( int a[*], int b[*], res int c[*], int n )
{
  if (n == 1) {
    c[1] = a[1] + b[1];
  } else {
    int mid = n/2;
    co
       somme( a[1:mid],   b[1:mid],   c[1:mid],   mid );
    // somme( a[mid+1:n], b[mid+1:n], c[mid+1:n], n-mid );
    oc
  }
}
```

Caractéristiques (pour les deux variantes) :

- $T(n) \in \Theta(\lg\ n)$.
- $P(n) = n$.
- $C(n) \in \Theta(n\ \lg\ n)$.
- $W(n) \in \Theta(n)$.

L'algorithme est donc optimal en travail, mais pas en coût.

Solution à l'exercice 2.

Deux solutions :

a. Solution avec passage complet des tableaux et transmission des bornes de l'intervalle à traiter :

```
procedure somme_seq( int a[*], int b[*], ref int c[*], int i, int j )
{
  for [k = i to j] {
    c[k] = a[k] + b[k];
  }
}

procedure somme( int a[*], int b[*], ref int c[*], int n )
{
  co [i = 1 to NBPROCS]
    somme_seq(a, b, c, inf(i, n), sup(i, n));
  oc
}
```

b. Solution avec tranches de tableaux (encore une fois, il faut alors utiliser le mode `res` pour le tableau `c`) :

```
procedure somme_seq( int a[*], int b[*], res int c[*] )
{
  for [k = 1 to ub(c)] {
    c[k] = a[k] + b[k];
  }
}

procedure somme( int a[*], int b[*], res int c[*], int n )
{
  co [i = 1 to NBPROCS]
    somme_seq( a[inf(i, n):sup(i, n)], b[inf(i, n):sup(i, n)],
               c[inf(i, n):sup(i, n)] );
  oc
}
```

Caractéristiques :

- $T(n) \in \Theta(n)$.
- $P(n) = \texttt{NBPROCS} \in \Theta(1)$.
- $C(n) \in \Theta(n)$.
- $W(n) \in \Theta(n)$.

L'algorithme est donc optimal tant en travail qu'en coût.

Solution à l'exercice 3.

```
sem nbItemsGeneres = 0;

process generer {
  int i;
  for [i = 1 to n] {
    elems[i] = int(random(1, n));
    V(nbItemsGeneres);      # Signaler
  }
}

process verifier {
  int i;
  for [i = 1 to n] {
    P(nbItemsGeneres);      # Attendre
    estPair[i] = (elems[i] % 2) == 0;
  }
}
```

Solution à l'exercice 4.

```
sem nbOccsLibre = 1;

procedure nbOccurences( int cle, int a[*], int i, int j ) {
  if (i == j) {
    if (a[i] == cle) {
       P(nbOccsLibre);
       nbOccs++;
       V(nbOccsLibre);
    }
  } else {
    int m = (i + j)/2;

    co nbOccurences( cle, a, i, m );
    // nbOccurences( cle, a, m+1, j );
    oc
  }
}
```

Caractéristiques :

- $T(n) \in \Theta(n)$ — le pire cas survient lorsque `a[i] == cle`, pour tous les `i` possibles, auquel cas un goulot d'étranglement est créé pour l'accès au sémaphore.
- $P(n) = n$.
- $C(n) \in \Theta(n^2)$.

L'algorithme n'est donc pas optimal.

Solution à l'exercice 5.

```
procedure trouverMax_( int a[*], int i, int j )
{
  int maxLocal = a[i];

  for [k = i+1 to j] {
    if (a[k] > maxLocal) { maxLocal = a[i]; }
  }
  P(maxGlobalLibre);
  if (maxLocal > maxGlobal) { maxGlobal = maxLocal; }
  V(maxGlobalLibre);
}

procedure trouverMax( int a[*], int n ) {
  maxGlobal = a[1];
  co [i = 1 to NBPROCS]
    trouverMax_( a, inf(i, n), sup(i, n) );
  oc
}
```