# Examen du 17/06/2010 (2h – conso)

TOP: Techniques and tOols for Programming

Première année

**Documents interdits, à l'exception d'une feuille A4 à rendre avec votre copie.**

La notation tiendra compte de la présentation et de la clarté de la rédaction.

★ **Questions de cours.** (4pt)

▷ **Question 1:** (2pts) Décrivez en quelques mots chacun des algorithmes suivants : tri à bulle, tri par sélection, tri fusion et tri par insertion.

▷ **Question 2:** (1pt) À quelles classes de complexité (en notation $\Theta$) appartiennent les algorithmes 1 et 2 suivants ?

algorithme 1

```
1 pour i = 1 à n faire
2   pour j = 1 à n faire
3     x += 3
```

algorithme 2

```
1 pour i = 1 à n faire
2   pour j = 1 à n faire
3     x += 3
4 pour i = 1 à n faire
5   y = x + 5
```

▷ **Question 3:** (1pt) Qu'est ce que le backtracking ?

**Réponse**

C'est une technique algorithmique permettant d'effectuer des recherches combinatoires bien plus efficacement qu'un parcours exaustif. L'idée de base est de construire peu à peu les solutions en ne parcourant que des sous-solutions valides. Dès qu'une sous-solution est invalide, on arrête l'exploration de cette branche, ce qui permet d'éviter le parcours de nombreuses solutions complètes qui seraient invalidées par la présence de la sous solution courante. Ouf.

Pour mettre en œuvre le backtracking, on utilise généralement une récursion (pour construire peu à peu la solution) avec une boucle ou similaire pour parcourir tous les choix possibles à une étape donnée de la construction de notre solution. Pour chaque choix, s'il mène à une sous-solution valide également, on opère un appel récursif pour explorer les sous-solutions plus grandes que l'on peut construire avec celle que l'on a pour l'instant.

**Fin réponse**

★ **Exercice 1: Code récursif mystère** (5pt).

▷ **Question 1:** (1pt) On considére le code de droite et les tableaux `tab={3, 4, 1, 2}` et `tab2={17, 12, 51, 20}`. Explicitez les appels `mystification(tab)` et `mystification(tab2)` en indiquant le détail de chaque appel récursif.

**Réponse**

```
mystification([3,4,1,2], 0)
  3 > 2 -> return mystification([3,4,1,2], 1)
               4 > 2 -> return 1+mystification([3,4,1,2], 2)
                            1 > 2  -> return 1+(1+mystification([3,4,1,2], 3))
                                      1+mystification([3,4,1,2], 4)
```

**Fin réponse**

```
1  public int mystification(int[ ] tab) {
2      return mystification(tab, 0);
3  }
4
5  public int mystification(int[ ] tab, int i) {
6      if (i == tab.length-1) {
7          return 0;
8      } else {
9          if (tab[i] > tab[tab.length-1]) {
10             return 1+mystification(tab, i+1);
11         } else  {
12             return mystification(tab, i+1);
13         }
14     }
15 }
```

▷ **Question 2:** ($\frac{1}{2}$pt) Que semble calculer ce code ?

**Réponse**

Cette fonction calcule le nombre de valeurs contenues dans le tableau et qui sont strictement plus grande que la dernière valeur du tableau.

**Fin réponse**

▷ **Question 3:** ($\frac{1}{2}$pt) Montrez la terminaison de la fonction récursive `mystification`.

**Réponse**

La fonction "s'arrête" quand `i` est égal à `tab.length-1`. Au premier appel, `i = 0`. Puis à chaque appel la valeur de `i` croît strictement. Elle atteindra donc inévitablement la valeur `tab.length-1`.

**Fin réponse**

▷ **Question 4:** ($\frac{1}{2}$pt) Quelle est la complexité de `mystification` (en nombre d'appels récursifs) ?

Le paramètre de récursion, $i$, croit linéairement. Il faut donc naïvement $O(n)$ étapes pour traiter un problème de taille $n$.

**Fin réponse**

▷ **Question 5:** ($\frac{1}{2}$pt) Est-il possible de dérécursiver directement cette fonction ? Pourquoi ?

**Réponse**

Non, car elle n'est pas terminale : il y a des calculs à la remontée (en ligne 10).

**Fin réponse**

▷ **Question 6:** (2pt) Dérécursivez cette fonction en appliquant les méthodes vues en cours (en une ou plusieurs étapes). Explicitez ce que vous faites et pourquoi.

**Réponse**

La première étape est de rendre cette fonction récursive terminale. Pour cela, on écrit une fonction ayant un argument supplémentaire, et on y fait lors de la descente les opérations que l'on aurait dû faire lors de la remontée. Ceci n'est bien sûr possible que parce que l'opération à modifier est une addition, qui est associative et commutative.

```
1  public int mystification(int [ ] tab) {
2    return mystification(tab,0,0);
3  }
4  public int mystification(int [ ] tab, int i, int accumulateur) {
5    if (i==tab.length-1) {
```

```
    return accumulateur;
  } else {
    if (tab[i] > tab[tab.length-1]) {
      return mystification(tab, i+1, accumulateur+1);
    } else {
      return mystification(tab, i+1, accumulateur);
    }
  }
}
```

Une fois rendue terminale, on peut dérécursiver cette fonction de la façon [simple] vue en cours.

```
public int mystification(int [ ] tab) {
  int accumulateur = 0;
  int i = 0;
  while (i!=tab.length-1) {
    if (tab[i] > tab[tab.length-1]) {
      accumulateur ++;
      i++;
    } else {
      i++;
    }
  }

  return accumulateur
}
```

On peut évidement factoriser du code et écrire i++ une seule fois.

**Fin réponse**



★ **Exercice 2: Encore un code récursif (mais pas mystère normalement)** (7pt).

La multiplication de deux nombres entiers positifs peut se faire de la manière suivante :

- Écrivez le multiplicateur et le multiplicande l'un à côté de l'autre.
- Formez une colonne en dessous de chacun des opérandes en itérant la règle suivante jusqu'à ce que le nombre sous le multiplicateur soit égal à 1 : divisez par deux le nombre sous le multiplicateur, sans tenir compte du reste éventuel, et doublez par addition le nombre sous le multiplicande.
  Par exemple, pour multiplier 19 par 45, vous obtenez la suite de nombres de droite.
- Finalement, rayez tous les nombres de la colonne du multiplicande correspondant à multiplicateur pair (ceux pour les multiplicateurs 22 et 2). Il ne reste plus qu'à additionner les nombres restants : $19 + 76 + 152 + 608 = 855$.

| | |
|---|---|
| 45 | 19 |
| 22 | 38 |
| 11 | 76 |
| 5 | 152 |
| 2 | 304 |
| 1 | 608 |

Une autre façon de présenter ce calcul est la suivante :

$$\begin{aligned} 45 \times 19 &= (1 + 44) \times 19 = 19 + 22 \times 2 \times 19 = 19 + 22 \times 38 \\ &= 19 + 11 \times 2 \times 38 = 19 + 11 \times 76 \\ &= 19 + (1 + 5 \times 2) \times 76 = 19 + 76 + 5 \times 152 \\ &= \dots \end{aligned}$$

▷ **Question 1:** (3pt) Écrivez un algorithme (récursif) calculant la multiplication de cette manière.

▷ **Question 2:** (1pt) Montrez la terminaison de votre algorithme. Est il récursif terminal ?

▷ **Question 3:** (1pt) Calculez la complexité temporelle de votre code (en nombre d'appels) dans le cas moyen, dans le meilleur cas et dans le pire cas.

▷ **Question 4:** (2pt) Dérécursivez votre code.

★ **Exercice 3: Preuve de programmes** (4pt).

Considérez le code de la fonction ci-contre calculant la factorielle de façon itérative. Calculez la plus faible précondition (Weakest Precondition, **WP**) nécessaire pour que la post-condition soit :

$$Q \triangleq res = n!$$

Les règles de calcul des préconditions sont rappelées en annexe.

```
int res;
void Factorial (int n) {
  int f = 1;
  int i = 1;
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
  res = f;
}
```

**Réponse**

Il faut, comme toujours, partir du bas de l'algorithme. Calculons tout d'abord la WP permettant à la boucle while d'avoir Q en post-condition. L'invariant de la boucle est assez simple dans ce cas : $I \triangleq (f = i!) \wedge (i \in [0, n])$ Le variant est $n - i$

WP(while, Q) = I, avec les obligations habituelles. Considérons d'abord les deux dernières obligations, habituellement plus simples.

Obligation 2 $\triangleq I \Rightarrow V \geq 0$
$\triangleq (f = i!) \wedge (i \in [0, n]) \Rightarrow n - i \geq 0$
$\Leftarrow i \in [0, n] \Rightarrow n - i \geq 0$ (ce qui est trivialement vrai)

Obligation 3 $\triangleq (E = \texttt{false} \wedge I) \Rightarrow Q$
$\triangleq n = i \wedge (f = i!) \wedge (i \in [0, n]) \Rightarrow f = n!$
$\Leftarrow n = i \wedge (f = i!) \Rightarrow f = n!$ (c'est bon aussi).

Reste la première obligation de preuve, la plus dure.

Obligation 1 $\triangleq (E = \texttt{true} \wedge I \wedge V = z) \Rightarrow \mathbf{WP}(C, I \wedge V < z))$

Commençons par calculer le membre droit de l'implication

$\mathbf{WP}(C, I \wedge V < z) \triangleq \mathbf{WP}(C, (f = i!) \wedge (i \in [0, n]) \wedge n - i < z)$
$\triangleq ((f = i!) \wedge (i \in [0, n]) \wedge n - i < z)_{[i:=i+1; f:=f+1]}$
$\triangleq ((f + 1 = (i + 1)!) \wedge (i + 1 \in [0, n]) \wedge n - i - 1 < z)$

Ce qui nous donne :

Obligation 1 $\triangleq i < n \wedge f = i! \wedge i \in [0, n] \wedge n - i = z \Rightarrow f \times i = (i + 1)! \wedge i + 1 \in [0, n] \wedge n - i - 1 < z$
$\triangleq f = i! \wedge i \in [0, n[ \wedge n - i = z \Rightarrow f \times i = (i + 1)! \wedge i + 1 \in [0, n] \wedge n - i - 1 < z$

Notons $P_1$, $P_2$ et $P_3$ les trois prédicats à droite de l'implication.
- $P_1 \triangleq f \times i = (i + 1)!$ est donné par le fait que $f = i!$ (en prémisse de l'implication) et la définition de la factorielle.
- $P_2 \triangleq i + 1 \in [0, n]$ se déduit de la prémisse $i \in [0, n[$
- $P_3 \triangleq n - i - 1 < z$ se déduit de la prémisse $n - i = z$

On a donc montré que $WP(while, Q) \equiv I$. Reste à finir.

WP(l3-4,I) $\triangleq$ WP(l3-4,$(f = i!) \wedge (i \in [0, n])$)
$\triangleq ((f = i!) \wedge (i \in [0, n]))_{[i:=1, f:=1]}$
$\triangleq ((1 = 1!) \wedge (1 \in [0, n]))$
$\triangleq n \geq 1$

L'élément de droite est toujours vrai par définition de la factorielle, et l'élément de gauche est vrai si et seulement si n est plus grand que 1 (sinon, l'écriture est même invalide).

Donc c'est fini. WP(factoriel, res=n !)$\equiv n \geq 1$

**Fin réponse**

★**Annexe :** Règles de calcul des préconditions.
- $\mathbf{WP}(nop, Q) \equiv Q$
- $\mathbf{WP}(x := E, Q) \equiv Q[x := E]$
- $\mathbf{WP}(C; D, Q) \equiv \mathbf{WP}(C, \mathbf{WP}(\text{D,Q}))$
- $\mathbf{WP}(\texttt{if}\ Cond\ \texttt{then}\ C\ \texttt{else}\ D, Q) \equiv (Cond = \texttt{true} \Rightarrow \mathbf{WP}(C, Q)) \wedge (Cond = \texttt{false} \Rightarrow \mathbf{WP}(D, Q))$
- $\mathbf{WP}(\texttt{while}\ E\ \texttt{do}\ C\ \texttt{done}\ \{\text{inv I var V}\}, Q) \equiv I$ ; Obligations de preuve :
  - $(E = \texttt{true} \wedge I \wedge V = z) \Rightarrow \mathbf{WP}(C, I \wedge V < z))$
  - $I \Rightarrow V \geq 0$
  - $(E = \texttt{false} \wedge I) \Rightarrow Q$