Examen du 19/03/2011 (2h)

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.** (2pt)

▷ **Question 1:** ($\frac{1}{2}$pt) Définissez en français (sans équation) les notations O, Ω et Θ utilisées pour dénoter la complexité algorithmique en insistant sur leurs relations les unes avec les autres.

▷ **Question 2:** ($\frac{1}{2}$pt) Quel est le rapport entre les notations que vous venez de définir et les temps de calcul dans le meilleur des cas, le pire des cas et le cas moyen ?

▷ **Question 3:** ($\frac{1}{2}$pt) Définissez les tests (1) white box (2) de régression.

Réponse

- **Whitebox :** C'est une technique de tests, une façon d'imaginer les tests à faire pour remplir mes objectifs ; J'écris mes tests en lisant le source (et je teste donc les cas limites de l'implémentation)
- **Régression :** Quand j'ai trouvé une erreur (un bug) dans mon programme, je dois écrire un test cherchant à le reproduire afin de m'assurer que ce bug ne refera pas surface lors d'une modification ultérieure du code.

Fin réponse

▷ **Question 4:** ($\frac{1}{2}$pt) Définissez en quelques mots le principe des tris (1) par insertion (2) par sélection en explicitant en français (sans équation) leurs invariants.

★ **Exercice 1: Backtracking pour «Des chiffres et des lettres»** (D'après Eugène Asarin – 5pts)

Étant donné un tableau de lettres, on cherche à trouver le mot le plus long composé de certaines de ces lettres. On s'interdit également d'utiliser les lettres du tableau plus d'une fois. Par exemple, pour `lettres=ERUIGHURH` le mot le plus long est RIGUEUR.

On suppose donné un objet dictionnaire `dict` doté d'une méthode booléenne `contains(w)` qui répond si la chaîne `w` est un mot valide du dictionnaire.

▷ **Question 1:** (2pt) Programmez en Java une fonction récursive répondant au prototype suivant, et qui renvoie un mot de k lettres du dictionnaire, composé du préfixe passé en paramètre et suivi de lettres du tableau utilisées au plus une fois chacune. Si plusieurs mots répondant à cette définition existent, renvoyez le premier d'entre eux que vous calculez.

```
String chercher(String prefixe, char[] lettres, int k)
```

Vous pouvez supposer que la proposition `lettres.length>k` est vraie. `chercher(PA,"SETATTAH",7)` pourrait par exemple renvoyer «PATATES». Si un tel mot n'existe pas il faut renvoyer `null`.

Indications : Cette fonction est de complexité exponentielle ; aucun mot ne contient la lettre '*', qui peut donc être utilisée comme marqueur temporaire.

Réponse

Le remplissage du dictionnaire est tout à fait améliorable :)

```
import java.util.Hashtable;

public class Mot {
   Hashtable<String,String> dict = new Hashtable<String,String>();
   public String chercher(String prefix, char[] lettres, int k) {
       if (k==0) { // fini, c'est assez long on a notre mot
         if (dict.containsKey(prefix))
           return prefix; // cool on a notre mot
         else
           return null; // damnit, on a genere du caca
      }
       // Si on est ici, c'est que c'est pas assez long. Ajoutons la lettre suivante alors.
       // On teste tour a tour de prendre chacune des lettres possibles a cette position avant de faire les suivantes
      for (int i=0;i<lettres.length;i++) {
          if (lettres[i] != '*') { // lettre pas encore prise pour les positions precedentes
              char c = lettres[i];
              lettres[i]='*'; // on la prend
              String res = chercher(prefix+lettres[i],next,k-1);
              if (res != null)
                  return res;
              lettres[i]=c; // on restaure le tableau dans l'etat ou on l'a trouve
```

```
            }
        }
        return null;
    }

    public Mot() {
        char[] lettres = new char[] { '1','2','3'};

        // Ahem, cette partie n'est pas faite. Mais c'est pas dans l'enonce, non plus
        dict.put("31","");

        // On lance la demo (sur des chiffres au lieu de lettres, sans raison apparente)
        System.out.println("found: "+chercher("",lettres,2));
    }

    public static void main(String[] args) {
        new Mot();
    }
}
```

Fin réponse

▷ **Question 2:** (1pt) En utilisant cette fonction `chercher()`, écrivez la fonction `jouer()` qui trouve le mot le plus long composé de lettres du tableau `lettres`. `String jouer(char[] lettres)`

▷ **Question 3:** (2pt) Supposons qu'on dispose aussi d'une fonction booléenne `isPrefix(w)` qui répond si la chaîne w est un préfixe d'un mot valide. Écrivez `String chercher2(prefixe,lettres,k)` plus rapide que `chercher()` en profitant de la fonction `isPrefix(w)` pour ne pas regarder des chaînes non-prometteuses.

★ **Exercice 2: Code récursif mystère** (D'après Baynat, Exercices et problèmes d'algorithmique – 5pt).
Considérez le code mystère suivant.

▷ **Question 1:** (1 pt) Explicitez les appels récursifs effectués pour `puzzle(3)` en prenant garde à ne développer qu'un seul appel récursif par ligne. Calculez le résultat de `puzzle(4)` et `puzzle(5)`. Que semble calculer cette fonction ?

```
public int puzzle(int n) {
  if (i == 0)
    return 2;
  else
    return puzzle(i-1)*puzzle(i-1);
}
```

Réponse

Question 1 :

puzzle(3) =puzzle(2)*puzzle(2)
=puzzle(1)*puzzle(1)*puzzle(2)
=puzzle(0)*puzzle(0)*puzzle(1)*puzzle(2)
=2*puzzle(0)*puzzle(1)*puzzle(2)
=2*2*puzzle(1)*puzzle(2)
=2*2*puzzle(0)*puzzle(0)*puzzle(2)
=2*2*2*puzzle(0)*puzzle(2)
=2*2*2*2*puzzle(2)
=2*2*2*2*puzzle(1)*puzzle(1)
=2*2*2*2*puzzle(0)*puzzle(0)*puzzle(1)
=2*2*2*2*2*puzzle(0)*puzzle(1)
=2*2*2*2*2*2*puzzle(1)
=2*2*2*2*2*2*puzzle(0)*puzzle(0)
=2*2*2*2*2*2*2*puzzle(0)
=2*2*2*2*2*2*2*2
=2^8=256

Question 2 : $puzzle(4) = 2^{2^4}$ $puzzle(5) = 2^{2^5}$ $puzzle(10) = 2^{2^{10}}$

Fin réponse

▷ **Question 2:** ($\frac{1}{2}$pt) Montrez la terminaison de cet algorithme.

Réponse

Le paramètre de récursion, i, est strictement décroissant. Il va donc bien converger vers 0, qui est le cas d'arrêt.

Fin réponse

▷ **Question 3:** (1pt) Démontrez par récurrence ce que calcule cette fonction, en fonction de n.

Réponse

Nous avons donc affaire à la suite

$$\begin{cases} F_0 = 2 \\ F_n = F_{n-1} \times F_{n-1} = F_{n-1}^2 \text{ (si } n > 0) \end{cases}$$

On va démontrer par récurrence que ça calcule $F_n = 2^{2^n}$.

Cette relation est bien vérifiée pour $n = 0$ (car $2^{2^0} = 2^1 = 2$ et $F_0 = 2$), et si on suppose que cette relation est vérifiée pour n, on calcule F_{n+1} à partir de la relation de récurrence :

$$F_{n+1} = F_n^2 = \left(2^{2^n}\right)^2 = 2^{2^{n+1}}$$

On montre donc bien par récurrence que $\forall n \leq 0, F_n = 2^{2^n}$.

Fin réponse

▷ **Question 4:** (1pt) Le nombre de multiplications effectuées par la fonction puzzle est la solution de l'équation de récurrence suivante.

$$m(n) = \begin{cases} 0 & \text{si n=0} \\ 1 + 2 \times m(n-1) & \text{si } n > 0 \end{cases}$$

Montrez par récurrence que $m(n) = 2^n - 1$, et déduisez en la complexité algorithmique de `puzzle`.

Réponse

Montrer par récurrence que $m(n) = 2^n - 1$ est très simple quand on réécrit m(n) sous la forme $m(n) = \sum_{i=0}^{n-1} 2^i$.

On en déduit donc que la complexité est en $\Theta(2^n)$

Fin réponse

▷ **Question 5:** (1pt) Modifiez l'algorithme pour ramener la complexité dans $\Theta(n)$.
Il n'est pas demandé de démontrer formellement que le nouvel algorithme est effectivement dans cette classe de complexité, ni sa correction.

Réponse

Il suffit bien sûr de ne faire chaque appel récursif une seule fois.

```
public int puzzle(int i) {
  if (i == 0)
    return 2;
  else {
    int a = puzzle(i-1);
    return a*a;
  }
}
```

Fin réponse

▷ **Question 6:** ($\frac{1}{2}$pt) Est-il possible de dérécursiver directement cette nouvelle fonction ? Pourquoi ?

Réponse

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

Fin réponse

★ **Exercice 3: Preuve de programme.** (D'après Baynat, Exercices et problèmes d'algorithmique – 4pts)

Soit l'algorithme itératif ci-contre. Nous allons montrer formellement que la post-condition de cette algorithme est $Q \equiv res = 2^{2^n}$ en utilisant les formules de calcul de la plus faible précondition rappelées en annexe.

```
public int puzzleIter(int i) {
  int res=2;
  int i=0;
  while (i<n) {
    res = res * res;
    i=i+1;
  }
  return res;
}
```

▷ **Question 1:** (1pt) Explicitez l'invariant et le variant de la boucle.

Réponse

$I \equiv i \in [0,n] \ \wedge \ res = 2^{2^i}$
$V = n - i$

Fin réponse

▷ **Question 2:** (3pt) Calculez la plus faible précondition nécessaire pour que cet algorithme calcule 2^{2^n}. Explicitez ce que vous faites et pourquoi.

Réponse

$WP(l2-8, res = 2^{2^n}) = WP(l2-3, WP(while, res = 2^{2^n}))$
$WP(while, res = 2^{2^n}) \equiv I$ avec les trois obligations de preuves suivantes :

1. $(cond = true \wedge I \wedge V = z) \Rightarrow WP(body, I \wedge V < z)$
2. $I \Rightarrow V \geq 0$
3. $(cond = false \wedge I) \Rightarrow res = 2^{2^n}$

La **seconde obligation** de preuve se réécrit en

$$i \in [0,n] \ \wedge \ res = 2^{2^i} \Rightarrow n - i \geq 0$$

Ce résultat est trivial ($i \leq n$ par hypothèse, donc $n - i \geq 0$)

La **troisième obligation** de preuve se réécrit en

$$i \geq n \wedge i \in [0,n] \wedge res = 2^{2^i} \Rightarrow res = 2^{2^n}$$

Ce résultat est également trivial (on obtient $i = n$ en combinant les deux premières hypothèses, ce qui, avec $res = 2^{2^i}$ donne bien la conclusion).

La **première obligation** de preuve se réécrit en

$$(i < n) \wedge (i \in [0,n]) \wedge (res = 2^{2^i}) \wedge (n - i = z) \Rightarrow WP(body, (i \in [0,n]) \wedge (res = 2^{2^i}) \wedge (n - i < z))$$

Calculons tout d'abord le membre droit (la WP).

$WP(l5, l6; (i \in [0,n]) \wedge (res = 2^{2^i}) \wedge (n - i < z))$
$\equiv WP(l5, (i+1 \in [0,n]) \wedge (res = 2^{2^{i+1}}) \wedge (n - i - 1 < z))$
$\equiv (i+1 \in [0,n]) \wedge (res \times res = 2^{2^{i+1}}) \wedge (n - i - 1 < z)$

Il faut maintenant prouver chacun des éléments de cette expression en utilisant les éléments du membre gauche de la première obligation de preuve.

- $(n - i - 1 < z)$ est trivial avec la prémisse $(n - i = z)$
- $res \times res = 2^{2^{i+1}}$ s'obtient facilement de $res = 2^{2^i}$ car $2^{2^i} \times 2^{2^i} = 2^{2^{i+1}}$
- $(i+1 \in [0,n])$ s'obtient des prémisses $(i < n) \wedge (i \in [0,n])$

Maintenant que les trois obligations de preuves sont remplies, il est possible de conclure, en repartant de l'objectif.

$$\begin{aligned} WP(l2-8, res = 2^{2^n}) &\equiv WP(l2-3, WP(while, res = 2^{2^n})) \equiv WP(l2-3, I) \\ &\equiv WP(l2-3, i \in [0,n] \ \wedge \ res = 2^{2^i}) \\ &\equiv WP(l2, 0 \in [0,n] \ \wedge \ res = 2^{2^0}) \\ &\equiv n \geq 0 \ \wedge \ 2 = 2^{2^0} \end{aligned}$$

La seconde partie est triviale puisque $2^{2^0} = 2^1 = 2$. Reste donc $\boxed{WP(puzzleIter(n), res = 2^{2^n}) = n \geq 0}$

Fin réponse

★ **Exercice 4: Encore un code mystère (mais pas récursif)** (d'après Maylis DELEST – 4pt).

On considère la fonction ci-contre, avec `swap(tab,a,b)`, qui inverse les valeurs des cases `tab[a]` et `tab[b]`, et le tableau A de dimension 8 contenant la séquence {2,8,7,4,3,6,5,1}.

```
1  public int mystere(int[] tab, int cle) {
2    int d,f;
3
4    d=0 ;f=tab.length-1;
5    do {
6      while (d<tab.length && tab[d] <= cle) {
7        d++;
8      }
9      while (f>=0 && tab[f]>cle) {
10       f--;
11     }
12     if (d<f) {
13       swap(tab,d,f);
14       d++;
15       f--;
16     }
17   } while (d<f);
18   return f;
19 }
```

▷ **Question 1:** (2pt) Représentez graphiquement l'effet de l'appel `mystere(A, 6)` sur le tableau A en détaillant les différentes étapes et indiquez la valeur retournée par la fonction.

▷ **Question 2:** (1pt) Quelle est la valeur retournée par la fonction si tous les nombres contenus dans le tableau sont strictement inférieurs à la clé ? Même question si tous les nombres sont strictement supérieurs à la clé.

Réponse

Question 1 :

```
étape 1:  [2, 8, 7, 4, 3, 6, 5, 1]
           d                    f

étape 2:  [2, 8, 7, 4, 3, 6, 5, 1] (boucle ligne 6-7)
              d                 f

étape 3:  [2, 8, 7, 4, 3, 6, 5, 1] (boucle ligne 9-10)
              d                 f

étape 4:  [2, 1, 7, 4, 3, 6, 5, 8] (swap)
                 d           f

étape 5:  [2, 1, 7, 4, 3, 6, 5, 8] (les deux boucles: ne font rien)
                 d           f

étape 6:  [2, 1, 5, 4, 3, 6, 7, 8] (swap)
                    d     f

étape 7:  [2, 1, 5, 4, 3, 6, 7, 8] (les deux boucles)
                          f  d

retourne 5
```

La fonction range au début du tableau toutes les valeurs inférieures ou égales à la clé, et dans une seconde zone à la fin du tableau les valeurs supérieures à la clé. La valeur retournée est l'indice de fin de la première zone.

Question 2 : Si tous les nombres sont inférieurs à la clé, la valeur retournée est N-1. Si tous les nombres sont supérieurs à la clé, la valeur retournée est -1,

Fin réponse

▷ **Question 3:** (1pt) Donnez la complexité maximale de cette fonction en nombre de tests, et justifiez votre réponse.

Réponse

La complexité maximale est O(n). En effet, il y a bien deux boucles imbriquées mais d (resp. f) est croissant (resp. décroissant). La boucle externe contrôle d et f, plus précisément si d¿f la boucle s'arrête donc le nombre de test de cette boucle ne peut dépasser N. Au total, il y aura donc 2*N tests d'où la complexité annoncée.

Fin réponse

★**Annexe :** Règles de calcul des préconditions.

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