Examen du 26 mars 2009 (2h)

TOP: Techniques and tOols for Programming

Première année

La notation tiendra compte de la validité des réponses, mais aussi de la présentation et de la clarté de la rédaction.

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

★ **Questions de cours.** (2pts)

▷ **Question 1:** 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 ($\frac{1}{2}$pt).

▷ **Question 2:** 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 ? ($\frac{1}{2}$pt)

▷ **Question 3:** À quelles classes de complexité (en notation Θ) appartiennent les algorithmes 1 et 2 suivants ? ($\frac{1}{2}$pt)

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 4:** Qu'est ce que le backtracking ($\frac{1}{2}$pt) ?

★ **Exercice 1: Preuve de programmes** (5pts).

On suppose que P est un prédicat défini pour tous les entiers. On considère le code ci-dessous.

▷ **Question 1:** Calculez la précondition la plus faible pour que ce code effectue une recherche linéaire bornée, ie la post-condition suivante :

$x \in \mathbb{N} \wedge p \leq x \wedge x \leq q$

$\wedge \quad x < q \Rightarrow P(x)$

$\wedge \quad x = q \Rightarrow (\forall i \in \mathbb{N}, p \leq i \wedge i \leq q \Rightarrow \neg P(i))$

```
1 x = p;
2 y = q;
3 tant que x != y faire
4   si P(x) alors y = x
5           sinon x = x+1;
6 fait
```

On rappelle les règles de calcul des préconditions suivantes :

1. $\mathbf{WP}(nop, Q) \equiv Q$
2. $\mathbf{WP}(x := E, Q) \equiv Q[x := E]$
3. $\mathbf{WP}(C; D, Q) \equiv \mathbf{WP}(C, \mathbf{WP}(\mathrm{D,Q}))$
4. $\mathbf{WP}(\texttt{if } Cond \texttt{ then } C \texttt{ else } D) \equiv (Cond = \texttt{true} \Rightarrow \mathbf{WP}(C, Q)) \wedge (Cond = \texttt{false} \Rightarrow \mathbf{WP}(D, Q))$
5. $\mathbf{WP}(\texttt{while } E \texttt{ do } C \texttt{ done } \{\mathrm{inv\ I\ var\ V}\}, \mathrm{Q}) \equiv I$

   Plus les obligations de preuves suivantes :
   - $(E = \texttt{true} \wedge I \wedge V = z) \Rightarrow \mathbf{WP}(C, I \wedge V < z))$
   - $I \Rightarrow V \geq 0$
   - $(E = \texttt{false} \wedge I) \Rightarrow Q$

★ **Exercice 2: Code récursif mystère** (5pts).

Considérez le code mystère ci-contre.

▷ **Question 1:** Explicitez les appels récursifs effectués pour `puzzle(4)` ($\frac{1}{2}$pt).

▷ **Question 2:** Quelle somme cette fonction calcule-t-elle ? ($\frac{1}{2}$pt)

Exprimez le calcul réalisé par la fonction sous forme d'un $\sum_{?}^{?}$

```
1  private int puzzle(int i, int j) {
2    if (i == 0)
3      return 0;
4    if (j % 2 == 1)
5      return j+puzzle(i-1,j+1);
6    else
7      return   puzzle(i  ,j+1);
8  }
9  public int puzzle (int i) {
10   return puzzle(i,1);
11 }
```

▷ **Question 3:** Calculez le résultat de la fonction puzzle pour i=1, i=2, i=3, i=4, i=5 et i=6. Que semble calculer cette fonction (en plus de la somme vue plus haut) ? ($\frac{1}{2}$pt)

▷ **Question 4:** Montrez la terminaison de cet algorithme ($\frac{1}{2}$pt).

▷ **Question 5:** Quelle est la complexité algorithmique de `puzzle` (en nombre d'appels récursifs) ? ($\frac{1}{2}$pt)

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

▷ **Question 7:** Dérécursivez cette fonction en appliquant les méthodes vues en cours (en une ou plusieurs étapes). Explicitez ce que vous faites et pourquoi (2pts).

★ **Exercice 3: Identification d'algorithmes de tris** (4pts).

La colonne la plus à gauche constitue les données d'entrée du problème. La colonne la plus à droite représente les données de sortie, c'est-à-dire les données triées alphabétiquement. Chacune des autres colonnes représente une étape intermédiaire de l'un des algorithmes de tri listés ci-dessous.

① Tri à bulle, ② Tri fusion, ③ Tri par insertion, ④ Tri par selection, ⑤ QuickSort (en prenant le premier élément du sous-tableau comme pivot), ⑥ ShellSort.

(in)	A	B	C	D	E	F	(out)
gens	aveu	dard	gens	dard	gens	aveu	aveu
lama	base	file	lama	file	lama	base	base
zoom	miel	gens	pain	char	pain	char	char
pain	char	inox	inox	base	zoom	dard	dard
inox	file	lama	dard	aveu	dard	file	file
tape	rage	pain	file	gens	file	gens	gens
dard	dard	tape	miel	lama	inox	lama	inox
file	inox	zoom	char	zoom	tape	zoom	kilo
miel	tape	miel	kilo	pain	char	pain	lama
char	gens	char	base	inox	kilo	inox	miel
kilo	kilo	kilo	aveu	tape	miel	tape	pain
rage	vous	rage	rage	miel	rage	miel	rage
sort	pain	sort	sort	kilo	aveu	kilo	sort
base	lama	base	tape	rage	base	rage	tape
vous	zoom	vous	vous	sort	sort	sort	vous
aveu	sort	aveu	zoom	vous	vous	vous	zoom
	shell	insert	bubble	quick	fusion	selection	

▷ **Question 1:** Pour chacun des algorithmes, indiquez la colonne représentant une étape intermédiaire. Décrivez également quelle serait l'opération suivante (il n'est pas nécessaire de calculer l'état du tableau après cette opération, mais simplement de décrire en français l'opération réalisée).

Réponse

Barême : 1pt pour insertion et selection, 0,5 pour les autres

Fin réponse

★ **Exercice 4: Tests** (4pts).

Les fonctions ci-dessous appartiennent à une classe permettant de stocker des chaînes de caractères sous forme d'une pile.

▷ **Question 1:** Pour chaque méthode, indiquez les tests qu'il faudrait écrire pour vérifier leur bon fonctionnement. Écrivez simplement les tests à réaliser en français (vous n'avez pas à les écrire explicitement en utilisant JUnit ou une autre technique) (2pts).

▷ **Question 2:** Écrivez en JUnit les tests de la méthode `isEmpty()` (1pt).

▷ **Question 3:** Avez vous utilisé une approche whitebox, greybox, bluebox ou blackbox testing ? Justifiez votre réponse ? (1pt)

```
1  public class MyStack {
2    /** Tests if this stack is empty.
3     * @return True if and only if this stack contains no items; false otherwise
4     */
5    public boolean isEmpty()
6
7    /**
8     * Pushes an item onto the top of this stack.
9     * @param item { the item to be pushed onto this stack
10    * @return The item that was pushed onto the stack
11    */
12   public String push(String item)
13
14   /**
15    * Removes the object at the top of this stack and returns that object as the value of this
16    * function.
17    * @return The object at the top of this stack
18    * @throws EmptyStackException if this stack is empty.
19    */
20   public String pop()
21
22   /**
23    * Looks at the object at the top of this stack without removing it from the stack.
24    * @return the object at the top of this stack
```

```
25     * @throws EmptyStackException if this stack is empty
26     */
27   public String peek()
28 }
```

Réponse

- **isEmpty()**
  - 1. Test that calling empty() on an empty stack returns true
  - 2. Test that calling empty() on a non-empty stack returns false
- **push(item)** :
  - 1. Test that pushing an item on the stack does in fact put it at the head of the stack
  - 2. Adding a null item
  - 3. Pushing elements until you run out of space
- **pop()** : Any 3 of the following 4 :
  - 1. Test that the exception is thrown if the stack is empty
  - 2. Test that the exception is not thrown if the stack is not empty
  - 3. Test that it is indeed the object at the top of the stack that is returned
  - 4. Test that the object at the top of the stack is removed
- **peek()** : Any 3 of the following 4 :
  - 1. Test that the exception is thrown if the stack is empty
  - 2. Test that the exception is not thrown if the stack is not empty
  - 3. Test that it is indeed the object at the top of the stack that is looked at
  - 4. Test that the object at the top of the stack is not removed

Fin réponse