Université de Limoges **Année universitaire 2010-2011**
Faculté des Sciences et Techniques

Master mathématiques 1ère année
Cryptis - Acsyon
Algorithmique et programmation
Examen du 7 janvier 2011

Durée 2h. Documents manuscrits et documents distribués durant le semestre autorisés.
Le barême est indicatif et pourra être révisé.

Exercice 1. [6 points] On s'intéresse à l'algorithme de tri de tableaux baptisé *quicksort*. Une version récursive de cet algorithme a été présentée en cours.

1. Proposer une version *non récursive* de cet algorithme en utilisant une pile. On pourra réutiliser directement l'algorithme `scinder` du cours. La pile contiendra des couples d'entiers représentants les indices délimitant les sous-tableaux à trier.

2. Que pouvez-vous dire de la taille maximale atteinte par la pile dans votre algorithme ? Dans quelle situation obtient-on une pile de taille maximale ? Justifier brièvement.

3. Pouvez-vous proposer une version de cet algorithme dans laquelle la taille de la pile est dans $O(\log_2(n))$? On ne cherchera pas à justifier précisément, mais on pourra argumenter informellement.

Exercice 2. [6 points] On considère un tableau T contenant n éléments d'un certain type appelé `Objet`. Par définition, un *élément majoritaire* de T est un élément apparaissant au moins $\lfloor n/2 \rfloor + 1$ fois dans T. On suppose que la seule primitive disponible est le test d'égalité entre deux éléments de type `Objet`. En particulier, il n'existe pas de relation d'ordre sur les éléments du type `Objet` et on ne peut pas trier les éléments de T. Pour simplifier le raisonnement, **on suppose dans la suite que $n = 2^k$, avec $k \in \mathbb{N}$.**

1. Proposer un algorithme récursif de type «diviser pour régner» pour déterminer si un tableau admet un élément majoritaire. On pourra par exemple couper le tableau en deux parties égales ; quelles informations peut-on obtenir à partir d'une application récursive de l'algorithme aux deux parties ?

2. Estimer la complexité asymptotique dans le pire des cas de votre algorithme, en nombre de tests d'égalité d'éléments de type `Objet` à effectuer.

Exercice 3. [8 points] On considère des listes chainées construites de la manière suivante :

```
typedef struct cellule {
  unsigned int val;
  struct cellule *suiv;
} CELLULE;
typedef CELLULE* LISTE;
```

On souhaite programmer un algorithme permettant de trier une telle liste par valeurs croissantes du champ `val`. Pour cela, on utilise la méthode suivante :

- On répartit les éléments de la liste dans un tableau de 255 listes, en fonction de la valeur du dernier octet du champ `val` (octet correspondant aux bits de poids faible).

- On fusionne les éléments des 255 listes en une seule liste, de sorte qu'ils soient triés par ordre de dernier octet croissant.

- On recommence ensuite avec le second octet, puis le troisième, et enfin le quatrième.

Si les insertions dans les listes se font toujours en tête, on se convaincra que la liste obtenue in fine est triée par ordre croissant. Cet algorithme s'appelle le tri par compartiment. Il permet de trier la liste avec $O(n)$ opérations, ce qui est meilleur que les algorithmes vus en cours. Néanmoins, il ne s'applique pas à n'importe quel type d'objets, puisqu'il exploite la structure des entiers, contrairement aux algorithmes vu en cours, qui n'utilise que la relation d'ordre. Il est donc de portée moins générale.
Exemple. Pour simplifier, on travaille avec des entiers sur 3 chiffres décimaux au lieu d'octet, dont les valeurs vont de 0 à 3. On part de la liste (chainée de gauche à droite):

110 102 321 123 012 013 200

Les éléments sont distribués dans des listes en fonction de leur dernier chiffre :

0	200 110
1	321
2	012 102
3	013 123

Ces listes sont fusionnées en partant de la dernière pour obtenir:

110 200 321 102 012 123 013

On distribue en fonction du second chiffre :

0	102 200
1	013 012 110
2	123 321
3	

Ce qui donne:

200 102 110 012 013 321 123

Enfin, on distribue par rapport au chiffre de tête :

0	013 012
1	123 110 102
2	200
3	321

D'où finalement:

012 013 102 110 123 200 312

1. Écrire la fonction C dont le propotype suit :

   `void distribuer(LISTE L, int i, LISTE T[256])`

   et qui contruit un tableau `T` de LISTE tel que `T[j]` contiennent les éléments de L dont le `i`-ième octet du champ `val` vaut `j`. La fonction ne devra pas recopier les éléments de la liste `L` (pas d'allocation de nouvelles `CELLULE`), mais simplement les répartir dans le tableau résultat.

2. Écrire une fonction `fusion` prenant en entrée un tableau tel que celui renvoyé par la fonction `distribuer` et qui construit une liste formée des éléments du tableau triés par valeur de `i`-ième octet croissant (comme précédemment, sans allocation de nouvelles cellules).

   `LISTE fusion(LISTE T[256])`

3. Écrire la fonction de tri, dont le prototype suit :

   `void tri (LISTE *L)`