EA4 – Éléments d'algorithmique
Examen de 1re session – 20 mai 2014
Durée : 3 heures

Aucun document autorisé excepté une feuille A4 manuscrite
Appareils électroniques éteints et rangés

Préliminaires : *Ce sujet est constitué de 5 exercices totalement indépendants qui peuvent être traités dans l'ordre de votre choix. Il est donc vivement conseillé de lire l'intégralité du sujet avant de commencer. Le sujet est long, le barème en tiendra compte. Il est bien entendu préférable de ne faire qu'une partie du sujet correctement plutôt que de tout bâcler.*

Pour tous les exercices demandant d'écrire des algorithmes, vous êtes libres du langage utilisé, du moment que la description est suffisamment précise : python, pseudo-code, français, java... Veillez tout de même à préserver la lisibilité. Pour cela, il pourra être judicieux de faire appel à des fonctions auxiliaires au nom évocateur pour réaliser certaines opérations simples sans entrer dans les détails d'implémentation.

Quel que soit le langage choisi, vous pouvez utiliser comme structure de base, sans redéfinir leur fonctionnement, les tableaux et les listes chaînées, ainsi que des arbres binaires pour lesquels les champs ou fonctions `fils_gauche`, `fils_droit`, `pere`, `valeur`, `est_vide` *seront supposés définis. En revanche,* ***vous n'avez pas droit aux dictionnaires (python)****.*

Exercice 1 : *Premier de cordée*

On dit qu'un tableau `T` d'entiers est *unimodal* s'il est constitué d'une première partie croissante, suivie d'une deuxième décroissante, chacune pouvant éventuellement être vide ; autrement dit, si `T` est de longueur n, il existe $m \in [\![0, n-1]\!]$ tel que :

$$T[0] < T[1] < \cdots < T[m] \quad \text{et} \quad T[m] > T[m+1] > \cdots > T[n-1].$$

1. Proposer un algorithme `est_unimodal(T)` qui renvoie `true` si `T` est unimodal et `false` sinon.

2. De quelle(s) opération(s) élémentaire(s) est-il raisonnable de tenir compte pour évaluer la complexité en temps d'un tel algorithme ? Quelle est, avec ces conventions, (l'ordre de grandeur de) la complexité de votre algorithme ?

3. Proposer un algorithme `minimum(T)` *de complexité optimale* qui renvoie le plus petit élément de `T`. Justifier la correction de cet algorithme, et préciser sa complexité.

4. Comment tester *en temps constant* si l'élément en position i dans `T` en est le maximum ? En déduire un algorithme `maximum(T)` *de complexité optimale* qui renvoie le plus grand élément de `T`. Justifier la correction de cet algorithme, et préciser sa complexité.

5. Décrire en détail un algorithme de complexité $O(n \log n)$ dans le pire des cas pour transformer un tableau `T` en tableau unimodal. Justifier sa correction et sa complexité.

Exercice 2 : *L'ABR était presque parfait*

On rappelle qu'un arbre binaire est dit *parfait* si tous ses niveaux sont remplis ; on dira qu'il est *presque parfait* si tous ses niveaux sont remplis sauf éventuellement le dernier (on n'imposera ici pas de contrainte supplémentaire sur la répartition des nœuds sur ce niveau).

1. Rappeler l'algorithme `creation_ABR(T)` permettant de construire un ABR à partir d'un tableau `T` d'entiers, ainsi que sa complexité.
2. Quel est le résultat de cet algorithme pour les tableaux suivants ?
   - $\mathtt{T}_1 =$ | 5 | 6 | 3 | 4 | 1 | 7 | 2 |
   - $\mathtt{T}_2 =$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
   - $\mathtt{T}_3 =$ | 4 | 2 | 6 | 1 | 3 | 5 | 7 |
3. Dans un ABR parfait, quel élément se trouve nécessairement à la racine ? En déduire un algorithme `ABR_parfait(T)`, récursif et de complexité strictement meilleure que celui de la question 1, permettant de construire un ABR presque parfait à partir d'un tableau `T` d'entiers ***trié***. Justifier sa complexité.
4. En déduire un algorithme efficace qui transforme un ABR quelconque en ABR presque parfait. Justifier sa complexité.

Exercice 3 : *Les comptes d'Huffman*

L'algorithme de Huffman construit un arbre binaire complet décrivant un code (binaire) pour les caractères présents dans le texte à compresser :

- chaque feuille contient un caractère `c` ;
- le mot de code de `c` est obtenu en suivant le chemin de la racine vers la feuille, en codant `'0'` chaque pas vers un fils gauche, et `'1'` chaque pas vers un fils droit.

Pour que l'encodage du texte soit réalisé de manière efficace, on souhaite accéder au mot de code de chaque caractère en temps (quasi) constant. Cela nécessite la création d'un dictionnaire associant à chaque caractère son mot de code.

1. Proposer une structure de données pour le dictionnaire répondant à ce cahier des charges.
2. Décrire un algorithme aussi efficace que possible qui transforme l'arbre de Huffman en dictionnaire de Huffman. Justifier sa complexité.

Exercice 4 : *Le tri flou, c'est très clair*

On considère un tableau `T` de données à trier dont la valeur n'est pas connue de manière exacte, mais seulement par un encadrement : `T` est donc un tableau de couples représentant des intervalles $\mathtt{T[}i\mathtt{]} = [\mathtt{T[}i\mathtt{][0]}, \mathtt{T[}i\mathtt{][1]}]$ tels que le i^{e} élément x_i de l'ensemble appartient à $\mathtt{T[}i\mathtt{]}$:

$$\mathtt{T[}i\mathtt{][0]} \leqslant x_i \leqslant \mathtt{T[}i\mathtt{][1]}.$$

Le *tri flou* d'un tel tableau de longueur n consiste à réordonner les éléments de `T` en un tableau `S` tel qu'il existe $(x_0, x_1, \ldots, x_{n-1})$ vérifiant :

$$\forall i,\ x_i \in \mathtt{S[}i\mathtt{]} \quad \text{et} \quad x_0 \leqslant x_1 \leqslant \cdots \leqslant x_{n-1}.$$

1. Effectuer un tri flou des intervalles suivants :

$$[11, 15],\ [19, 21],\ [5, 8],\ [10, 13],\ [17, 20],\ [2, 4],\ [14, 16],\ [3, 6],\ [17, 18],\ [9, 12]$$

On dit qu'un problème A est ***moins dur*** qu'un problème B si tout algorithme qui résout le problème B peut être modifié pour résoudre le problème A sans augmentation de complexité.

2. Expliquer pourquoi le problème du tri flou est moins dur que le problème du tri usuel.

3. Que peut-on dire lorsque les intervalles à trier ont une intersection globale non vide ?

On souhaite tirer parti de cette propriété pour obtenir un algorithme de tri flou, basé sur le tri rapide (QuickSort), mais de complexité moindre lorsque les instances du problème sont plus simples. Pour cela, on va modifier l'étape de partitionnement du tri rapide. Une fois l'intervalle `pivot` choisi, on considère un sous-intervalle ***non vide*** de `pivot`, appelé `chevauchement`, ayant la propriété suivante :

`chevauchement` est égal à l'intersection des intervalles de `T` qui l'intersectent. *(*)*

En particulier, si un intervalle `I` appartenant à `T` ne contient pas `chevauchement`, alors ils sont disjoints.

4. Quels sont les deux intervalles `chevauchement` possibles pour l'exemple de la question 1, si `pivot`$= [11, 15]$?

L'ensemble `T` d'intervalles est alors partitionné en trois sous-ensembles :
- les `petits` intervalles, dont tous les éléments sont plus petits que ceux de `chevauchement`,
- les intervalles `moyens`, qui intersectent `chevauchement`,
- les `grands` intervalles, dont tous les éléments sont plus grands que ceux de `chevauchement`.

Dans un premier temps, nous allons décrire un algorithme de tri flou en admettant l'existence de `chevauchement`. Plus précisément, on supposera que `trouve_chevauchement(T, pivot)` calcule un tel intervalle en temps linéaire en la longueur de `T`.

Pour simplifier, on ne cherchera pas à effectuer les opérations en place.

5. Écrire une fonction `compare(I, J)` qui renvoie -1, 0 ou 1 selon la position de l'intervalle `I` par rapport à l'intervalle `J`.

6. Écrire précisément la fonction `partition(T, pivot)` qui partitionne le tableau `T` par rapport à l'intervalle `pivot`.

7. Décrire un algorithme de type QuickSort réalisant le tri flou d'un tableau d'intervalles.

8. Quelle est sa complexité dans le pire des cas ?

9. Quelle est sa complexité si les intervalles à trier ont une intersection non vide ?

10. En vous basant sur la complexité en moyenne de QuickSort, borner la complexité en moyenne de votre algorithme.

Nous allons maintenant écrire la fonction `trouve_chevauchement(T, pivot)`.

11. L'algorithme décrit aux questions 6 et 7 fonctionne-t-il toujours si `chevauchement` est remplacé par `pivot` ?

12. Écrire une fonction `trouve_chevauchement(T, pivot)` qui renvoie un sous-intervalle (non vide) `chevauchement` de `pivot` ayant la propriété (*).

On ne demande pas que `chevauchement` soit « optimal » parmi les sous-intervalles de `pivot` ayant cette propriété (par exemple du point de vue du nombre d'intervalles qui l'intersectent). En revanche, on exige que la complexité de la fonction `trouve_chevauchement(T, pivot)` soit au plus linéaire en la longueur de `T`.

Exercice 5 : *L'Arnaque*

La société X propose un service de sauvegarde et d'archivage longue durée très onéreux, pour des données de très grand volume (imaginons des centaines de téraoctets).
L'entreprise Y manipule de gros volumes de données qu'elle ne veut absolument pas perdre. Elle s'adresse donc à la société X pour en sauvegarder des copies. On va supposer que ces données sont constituées de très nombreux fichiers d'un gigaoctet (disons des centaines de milliers).
L'entreprise Y souhaite s'assurer que son argent ne sera pas dépensé pour rien : si jamais la société X est remplie d'escrocs, l'éventualité d'un procès gagné par Y contre X ne consolerait que mollement la société Y de la perte de ses données ; elle veut avoir l'assurance qu'elles pourront être restaurées en cas de panne matérielle dans les locaux de Y.
L'entreprise Y demande donc à X d'effectuer des simulations de restauration de données. Le commercial de la société X leur propose le mode alternatif décrit dans le paragraphe suivant.

> *« Étant donné les volumes en question, les tests de restauration seraient trop compliqués à mettre en place. Nous vous recommandons plutôt de nous demander chaque jour la valeur de hachage par la fonction* `md5` *d'un fichier de votre choix parmi la centaine de milliers de fichiers soumis. »*

1. Où est l'arnaque ? Faudrait-il choisir une autre fonction de hachage ?

Le commercial concède que le mécanisme qu'il propose ne prouve pas grand-chose. Il propose une version améliorée : chaque jour, Y doit demander à X la valeur de hachage par la fonction `md5` d'un fichier de son choix parmi la centaine de milliers de fichiers soumis, *précédé d'une séquence d'un kilooctet, également choisie par* Y. Si F et B sont respectivement le fichier et le bloc aléatoire choisis par Y, la preuve que doit fournir X est donc :

$$\texttt{md5}(B \oplus F),$$

où $\oplus$ désigne la concaténation.

2. Est-ce mieux ? Expliquer.

Une semaine plus tard, le commercial de la société X rappelle la société Y :

> *« Nous avons mis en place le protocole dont nous avions convenu. Cependant, pour des raisons techniques, nos informaticiens ont préféré placer le bloc aléatoire après le fichier et non avant. Nous espérons que cela ne vous posera pas de problème.»*

3. Pourquoi cela peut-il légitimement renforcer les soupçons de la société Y quant à l'honnêteté de la société X ?