EA4 – Éléments d'algorithmique
Examen de 1re session – 27 mai 2015
Durée : 3 heures

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

Préliminaires : *Ce sujet est constitué de 8 exercices, répartis en 3 parties indépendantes. Dans les parties II et III, il est préférable de commencer par le premier exercice. Les autres exercices sont indépendants les uns des autres et 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. Les indications de durée sont sans doute trop optimistes et ne sont là qu'à titre comparatif entre les différents exercices.

Sauf dans les exercices de la partie I, toutes les réponses doivent être justifiées, même si ce n'est pas précisé. Sauf mention contraire explicite, les complexités demandées sont les ordres de grandeur (Θ) des complexités en temps des algorithmes concernés.

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.

Partie I. Questions de cours

Exercice 1 : algorithmes de tri *(20 min)*

1. Rappeler, sans démonstration, les complexités en temps
   (a) dans le pire des cas
   (b) dans le meilleur des cas
   (c) en moyenne

   des algorithmes de tri par sélection, tri par insertion, tri fusion, tri rapide et tri par tas.

Soit `T` le tableau suivant :

8	15	18	7	11	4	13

2. Détailler l'exécution du tri fusion sur `T`. Compter (exactement) les comparaisons effectuées.

3. Détailler le déroulement du tri par tas sur `T` *(ne pas hésiter à passer à une représentation sous forme d'arbre)*. Compter les comparaisons effectuées.

4. Détailler le déroulement de la recherche de la médiane de `T` à l'aide de `QuickSelect`. Compter les comparaisons effectuées.

Exercice 2 : hachage *(15 min)*

On considère deux tables de hachage $\texttt{H}_1$, et $\texttt{H}_2$ de longueur 11, dans laquelle on place des valeurs entières, en utilisant la fonction de hachage $h(x) = x \mod 11$.

$\texttt{H}_1$ est une table à adressage fermé, pour laquelle les collisions sont résolues par chaînage.

$\texttt{H}_2$ est une table à adressage ouvert, avec résolution des collisions par sondage linéaire.

Répondre aux questions suivantes pour chacune des deux tables :

a. Insérer successivement les entiers 26, 7, 34, 20, 38, 49, 18, 15.

b. Combien de comparaisons nécessite la recherche de la valeur 37 dans la table ?

c. Comment supprimer la valeur 49 de la table ?

d. Comment se passe ensuite la recherche de la valeur 37 ?

Partie II. Recherche de motifs

Exercice 3 : recherche naïve *(10 min)*

1. Écrire une fonction `test_occurrence(m, t, i)` retournant `True` si un motif `m` apparaît à la position `i` d'un texte `t` (`m` et `t` étant représentés par des tableaux de caractères).

2. Écrire une fonction `recherche_naive(m, t)` effectuant une recherche naïve du motif `m` dans `t` à l'aide de `test_occurrence(m, t, i)`.

3. Quelle(s) opération(s) est-il raisonnable de prendre en compte pour le calcul de la complexité (en temps) des algorithmes précédents ? En déduire la complexité de l'algorithme `recherche_naive(m, t)`. Justifier.

Exercice 4 : motifs d'un texte fixé à l'avance *(20 min)*

Rappel : on appelle *suffixe d'indice i* de `t` le facteur (sous-tableau) `t[`i`:]`. La *table des suffixes* de `t` est le tableau contenant les entiers 0 à `len(t)` -1, triés selon l'ordre $\prec$ suivant :

$$i \prec j \text{ si } \texttt{t[}i\texttt{:]} \text{ précède } \texttt{t[}j\texttt{:]} \text{ pour l'ordre lexicographique.}$$

On suppose maintenant que la table des suffixes `S` d'un texte `t` a été construite, et on souhaite l'utiliser pour améliorer l'efficacité des recherches de motifs dans `t`.

1. Décrire un algorithme permettant de comparer lexicographiquement deux mots `m1` et `m2`. Quelle est sa complexité ?

2. Décrire un algorithme `recherche_efficace(m, t)` de complexité strictement meilleure que `recherche_naive(m, t)` pour déterminer si `m` apparaît dans `t`. Quelle est sa complexité ?

3. Comment modifier l'algorithme précédent pour obtenir le nombre d'occurrences de `m` dans `t` sans augmenter (l'ordre de grandeur de) la complexité ?

4. Comment trouver le motif de longueur k le plus fréquent du texte `t`, en temps $\Theta(k\texttt{len(t)})$? Quelle est la complexité en mémoire de cet algorithme ?

Exercice 5 : algorithme de Rabin et Karp *(40 min)*

L'algorithme de Rabin et Karp améliore l'algorithme `recherche_naive(m, t)` de l'exercice 3, *sans nécessiter de précalcul ni de mémoire auxiliaire.* Il repose sur l'utilisation d'une fonction de hachage.

Soit $b \geqslant 2$ un entier fixé ; pour tout entier $k \geqslant 1$, on considère la fonction $h_k : \mathbb{N}^k \to \mathbb{N}$ telle que :

$$h_k(x_{k-1}, \ldots, x_0) = x_{k-1} \cdot b^{k-1} + x_{k-2} \cdot b^{k-2} + \cdots + x_2 \cdot b^2 + x_1 \cdot b + x_0$$

1. Soit `t` un tableau d'entiers, de longueur $\ell \geqslant k$. Décrire un algorithme `initialise_h(t, k)` permettant de calculer $h_k(\texttt{t[0]}, \ldots, \texttt{t}[k-1])$ en $\Theta(k)$ opérations arithmétiques sur des entiers (additions ou multiplications).
2. Si les entiers considérés sont quelconques, ce calcul est-il réellement de complexité linéaire ? Et si tous les calculs sont faits *modulo* un entier q fixé ?

Dorénavant, tous les entiers sont considérés modulo q (inutile de réécrire `initialise_h(t, k)`).

3. Quelle relation existe-t-il entre $h_k(\texttt{t[0]}, \ldots, \texttt{t}[k-1])$ et $h_k(\texttt{t[1]}, \ldots, \texttt{t}[k])$?
4. En déduire un algorithme `affiche_tous_h(t, k)` qui calcule puis affiche *toutes* les valeurs $h_k(\texttt{t}[i], \ldots, \texttt{t}[i+k-1])$ pour $i \in [\![0, \texttt{len(t)} - k]\!]$ avec une complexité *totale* en $\Theta(\texttt{len(t)})$.

Remarque : la fonction précédente peut être appliquée à un tableau `t` de caractères, en considérant la valeur entière du code binaire de chaque caractère – ce que fait la fonction `ord()` de Python, ou une conversion de format en Java.

5. Le principe de l'algorithme de Rabin et Karp est d'utiliser la fonction h_k, pour un k bien choisi, comme un filtre pour limiter le nombre d'appels à `test_occurrence(m, t, i)`. Écrire un algorithme `recherche_RK(t, m)` recherchant le motif `m` dans `t` selon ce principe.
6. Déterminer sa complexité dans le pire des cas, en donnant un exemple de tel pire cas.
7. En faisant l'hypothèse d'uniformité suivante :

   > *b et q peuvent être (et sont) choisis de telle manière que la répartition des images des k-uplets de lettres soit (quasi)-uniforme sur $[\![0, q-1]\!]$*

   déterminer la complexité en moyenne de `recherche_RK(t, m)` en fonction de $\ell = \texttt{len(t)}$, de $k = \texttt{len(m)}$, de q et du nombre p d'occurrences trouvées. En déduire que, sous des hypothèses raisonnables, cette complexité moyenne est en $\Theta(\ell)$.

Partie III. Opérations ensemblistes sur des ABR

On s'intéresse ici à des ensembles d'entiers, représentés par des ABR sans doublon.

Exercice 6 : via les listes triées *(10 min)*

1. Quelle est la complexité des opérations ensemblistes binaires usuelles (union, intersection, différence, test d'inclusion...) lorsque les ensembles sont représentés par des listes triées sans doublon ? Justifier en décrivant le principe général, et en détaillant le cas d'une opération au choix.
2. En déduire la complexité de ces opérations lorsque les ensembles concernés sont représentés par des ABR *si on s'autorise à construire une ou des liste(s) intermédiaire(s).* Justifier, en séparant le cas des opérations dont le résultat est un booléen, et celui des opérations dont le résultat est un ensemble.

Le but des deux exercices suivants est d'obtenir la même complexité (en temps) pour certaines opérations, mais sans passer par la construction de la liste intermédiaire.
Dans la suite, on considère deux ABR `A` et `B` contenant des entiers, chacun supposé sans doublon, dont on note les racines respectivement `a` et `b`, et les sous-arbres $\texttt{A}_\texttt{g}$, $\texttt{A}_\texttt{d}$, $\texttt{B}_\texttt{g}$ et $\texttt{B}_\texttt{d}$.

Exercice 7 : inclusion de `A` dans `B` *(20 min)*

1. À quelle(s) condition(s) (l'ensemble représenté par) `A` est-il inclus dans (l'ensemble représenté par) `B`, dans chacun des 3 cas suivants : $\texttt{a} = \texttt{b}$, $\texttt{a} < \texttt{b}$ et $\texttt{a} > \texttt{b}$?
2. En déduire un algorithme `est_inclus(A, B)` qui retourne `True` si `A` est inclus dans `B`, et `False` sinon.
3. Quelle est sa complexité (en temps) dans le pire des cas ?
4. Quelle est sa complexité dans le meilleur des cas ? Est-ce également le cas pour l'algorithme de l'exercice 6 (question 2) ?
5. Quelle est sa complexité en espace ? Et celle de l'algorithme de l'exercice 6 ?

Exercice 8 : union de `A` et `B` *(30 min)*

1. Rappeler l'algorithme `recherche(A, x)` qui retourne `True` si `A` contient `x`, et `False` sinon (*il est préférable pour la suite d'adopter une présentation récursive*).
2. Modifier `recherche(A, x)` pour écrire un algorithme `split(A, x)` qui coupe `A` en deux selon la valeur `x` : `split(A, x)` retourne un couple d'ABR $\texttt{A}_1$ et $\texttt{A}_2$ contenant respectivement les éléments de `A` strictement inférieurs à `x`, et les éléments de `A` strictement supérieurs à `x`. En particulier, aucun des deux ne contient `x`.
3. Utiliser `split(A, x)` pour écrire un algorithme récursif `union(A, B)` qui retourne un ABR représentant l'union des ensembles représentés par `A` et `B`.
4. Quelle est la complexité (en temps) de `split(A, x)` dans le meilleur des cas ? Et celle de `union(A, B)`, si `A` et `B` sont supposés de même taille n ? Donner un exemple.
5. Quelle est la complexité de `split(A, x)` dans le pire cas, pour un ABR de taille n et de hauteur h ?
6. En déduire que la complexité de `union(A, B)` est au pire en $O(nh)$, où n est la taille de l'un des ABR, et h la hauteur de l'autre.
7. Donner un exemple pour lequel la complexité est effectivement en $\Theta(nh)$.