EA4 – Éléments d'algorithmique II
Examen de 1re session – 15 mai 2018
Durée : 3 heures

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

Le sujet est trop long. Ne paniquez pas, le barème en tiendra compte. Il est naturellement préférable de ne faire qu'une partie du sujet correctement plutôt que de tout bâcler.

Lisez attentivement l'énoncé.
Sauf mention contraire explicite, les réponses doivent être justifiées.

Exercice 1 : complexité des fonctions récursives *(15 min)*

On considère les fonctions récursives suivantes :

```
def somme_1(T) :
    return 0 if len(T) == 0 else T[0] + somme_1(T[1:])

def somme_2(T) :
    m = len(T)//2
    return len(T) if m==0 else somme_2(T[:m]) + somme_2(T[m:])

def somme_3(T) :
    m = len(T)//2
    return len(T) if m==0 else somme_3(T[:m]) + 1

def somme_4(T) :
    return 0 if len(T)==0 else sum(elt for elt in T) + somme_4(T[1:])
```

On note $A_i(n)$ le nombre d'additions effectuées par `somme_`i pour une entrée de taille n.
Pour chaque fonction :

a. dire (sans justification) ce que `somme_`i calcule,

b. donner la relation de récurrence naturellement satisfaite par $A_i(n)$,

c. en déduire (sans justification) l'ordre de grandeur de $A_i(n)$.

Exercice 2 : hachage *(15 min)*

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

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

$\mathtt{H}_2$ 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 alors l'insertion 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 ?

Exercice 3 : point fixe *(15 min)*

Le but de cet exercice est de déterminer si un tableau `T` de n entiers ***tous distincts*** possède un point fixe, c'est-à-dire un indice f tel que `T[`f`]`$= f$.

1. Décrire un algorithme correct (et optimal) pour `T` quelconque. Justifier son optimalité.

2. On suppose maintenant que `T` est trié.

a. Supposons que `T` possède un point fixe f (inconnu a priori). Comment peut-on déterminer en temps constant la position d'un indice i par rapport à f ?

b. En déduire un algorithme optimal dans le cas où `T` est trié. Quelle est sa complexité ? Justifier l'optimalité.

Exercice 4 : des vis et des écrous *(15 min)*

Catastrophe ! Alors que tout était méticuleusement bien classé, votre chat a semé la pagaille dans votre stock de vis et d'écrous, qui gisent maintenant en vrac sur l'établi (et un peu par terre). Comment remettre de l'ordre ? Impossible de comparer directement deux vis ou deux écrous – les différences sont trop minimes. La seule solution consiste à tenter d'assembler une vis et un écrou pour comparer leurs diamètres : soit la vis ne rentre pas dans l'écrou, soit ils sont parfaitement ajustés, soit la vis flotte dans l'écrou. Comment peut-on dans ces conditions trier ***rapidement*** vis et écrous ? *(on suppose donc ici, implicitement, qu'il existe un ordre total sur les vis et les écrous, donné uniquement par le diamètre, en négligeant les différences de pas de vis)*

Cet algorithme est-il optimal en moyenne ? *(on ne considèrera que le cas particulier de n vis et n écrous de n calibres différents)*

Exercice 5 : élections à la majorité absolue *(1 heure)*

Le but de cet exercice est de comparer plusieurs solutions au problème suivant :

> *Étant donné une liste de n éléments, déterminer s'il existe parmi eux un élément* ***majoritaire****, c'est-à-dire apparaissant strictement plus que* $\frac{n}{2}$ *fois.*

Certaines solutions suggérées sont des applications directes des algorithmes vus en cours. Il vous est demandé de les utiliser, ***pas de les réécrire***. Vous pouvez énoncer sans démonstration (mais précisément) les résultats de complexité vus en cours.

Dans la suite, `T` désigne un tableau de taille n. Les différents algorithmes demandés devront retourner l'élément majoritaire s'il existe, et `None` sinon.

1. *Solution 1 (naïve, pour des éléments quelconques)* Décrire un algorithme permettant de résoudre le problème posé sans faire aucune hypothèse sur la nature des éléments de `T`, sans modifier `T`, et avec une complexité en espace constante. Quelle est sa complexité en temps ? Préciser le meilleur et le pire cas, selon que la recherche est fructueuse ou infructueuse.

2. *Solution 2 (pour des éléments comparables)* Proposer une solution plus efficace (en temps) dans le cas où il existe une relation d'ordre sur les éléments de `T`. Préciser les complexités en espace et en temps (au mieux, au pire et en moyenne).

3. *Solution 3 (pour des entiers)*

a. Proposer une solution plus efficace (en temps) dans le cas où les éléments de `T` sont des entiers compris entre 0 et une constante `m`. Préciser les complexités en espace et en temps, dans le meilleur et le pire cas.

b. Généraliser cette solution à des éléments entiers quelconques. Que deviennent les complexités en espace et en temps ?

4. *Solution 4 (pour des éléments quelconques)* Comment le principe de la solution précédente se généralise-t-il à des éléments quelconques ? Préciser les complexités en espace et en temps (au mieux, au pire et en moyenne).

Dorénavant, la seule opération autorisée sur les éléments de `T` est le test d'égalité.

5. *Solution 5 (« diviser pour régner »)*
   a. Soit `T1` et `T2` tels que `T = T1 + T2`. Si `x` est majoritaire dans `T`, qu'en est-il dans `T1` et dans `T2` ? Et réciproquement ?
   b. En déduire un algorithme de type « diviser pour régner » pour résoudre le problème.
   c. Soit $C(n)$ le nombre de comparaisons nécessaires, dans le pire cas, pour un tableau de longueur n. Écrire l'équation satisfaite par $C(n)$, et en déduire son ordre de grandeur.

6. *Solution 6 (presque optimale)*
   Pour améliorer la complexité, on relâche un peu la contrainte, et on cherche un algorithme `peutEtreMajoritaires(T)`, défini pour `len(T)` $\geqslant 2$, qui retourne une liste de deux éléments de `T` (éventuellement identiques) telle qu'***aucun autre*** élément de `T` n'est majoritaire.
   a. Comment obtenir `majoritaire(T)` à partir de `peutEtreMajoritaires(T)` ?
   b. Supposons que `T` possède un élément majoritaire `x`, et que `T[0]` $\neq$ `T[1]`. Que peut-on dire de `T[2:]` ? La réciproque est-elle vraie ?
   c. Supposons que, pour tout i tel que $2i+1 <$ `len(T)`, `T[2*`i`]` $=$ `T[2*`i`+1]`. Si `T` possède un élément majoritaire `x`, qu'en est-il de `T[::2]` ? *(attention au cas particulier* `[x]*(2k)+[y]*(2k-1)`*)*
   d. En déduire un algorithme récursif `peutEtreMajoritaires(T)` utilisant un sous-tableau de `T` de longueur au plus $\lceil \frac{n}{2} \rceil$.
   e. Quelle est la complexité (au pire) de l'algorithme `majoritaire(T)` obtenu ?

Exercice 6 : les arbres à bouc émissaire (scapegoat trees) *(1 heure)*

Un *arbre à bouc émissaire* (ABE dans la suite) est un triplet `(A, n, m)` formé d'un arbre binaire de recherche (ABR) `A` (identifié par sa racine) et de 2 compteurs `n` et `m` tels que $\frac{1}{2}$`m` $\leqslant$ `n` $\leqslant$ `m`. Plus précisément, le compteur `n` stocke la *taille* de `A`, c'est-à-dire son nombre de sommets. La hauteur de `A` doit toujours rester inférieure à $\log_{3/2}$`m`, ce qui peut nécessiter des rééquilibrages, complets ou partiels. Le compteur `m` garde la mémoire de la taille maximale atteinte par `A` depuis son dernier rééquilibrage complet.

1. Décrire comment doivent évoluer `n` et `m` lors des opérations d'insertion et de suppression d'un élément dans `A`.
2. Décrire un algorithme `reequilibre(r)` permettant de rééquilibrer un ABR de racine `r`, c'est-à-dire de le transformer en un ABR *presque parfait* (c'est-à-dire dont toutes les feuilles sont à profondeur maximale, et les autres niveaux complètement remplis).
3. Quelle est la complexité (en temps) de l'algorithme précédent ?

Lors d'une suppression, si les conditions définissant un arbre à bouc émissaire ne sont plus remplies, l'ABR subit un rééquilibrage complet.

4. Décrire l'algorithme `suppression(A, n, m, x)` permettant de supprimer l'élément `x` de l'ABE `(A, n, m)`, et retournant le nouvel ABE.

Lors de l'insertion d'un élément `x`, si la hauteur de l'ABR dépasse la limite $\log_{3/2}$`m`, il subit un rééquilibrage *partiel* : un *bouc émissaire* `b` est cherché parmi les ancêtres de la feuille créée pour contenir `x`, puis le sous-arbre de racine `b` est rééquilibré. Le bouc émissaire `b` est choisi avec la propriété `taille(fils)` $> \frac{2}{3}$ `taille(b)`, où `fils` est le fils de `b` qui est un ancêtre de `x`.

5. Justifier l'existence d'un bouc émissaire.

6. Écrire un algorithme `taille(r)` permettant de calculer la taille du sous-arbre de racine `r` ; quelle est sa complexité ?

7. Écrire un algorithme `boucEmissaire(feuille)` permettant de trouver le bouc émissaire le plus proche de la `feuille` créée pour contenir `x`.

8. Quelle est la complexité de cet algorithme ?

9. Décrire l'algorithme `insertion(A, n, m, x)` permettant d'insérer l'élément `x` dans l'ABE `(A, n, m)`, et retournant le nouvel ABE.

On peut montrer le résultat suivant (admis) :

> *À partir d'un arbre à bouc émissaire vide, le coût cumulé des rééquilibrages nécessaires à une succession de n insertions ou suppressions est en $\Theta(n \log n)$.*

10. Dans quel(s) sens peut-on dire que les opérations de recherche, d'insertion et de suppression d'un élément dans un arbre à bouc émissaire sont de complexité logarithmique ?

Exercice 7 : *(hors-barème)*

La société ΦLOU 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 GOGO manipule de gros volumes de données qu'elle ne veut absolument pas perdre. Elle s'adresse donc à la société ΦLOU pour en sauvegarder des copies. Supposons que ces données sont constituées de très nombreux fichiers d'un gigaoctet (disons des centaines de milliers).
L'entreprise GOGO souhaite s'assurer que son argent ne sera pas dépensé pour rien : si jamais la société ΦLOU est remplie d'escrocs, l'éventualité d'un procès gagné par GOGO contre ΦLOU consolerait fort peu la société GOGO de la perte de ses données ; elle veut avoir l'assurance qu'elles pourront être restaurées en cas de panne matérielle dans ses locaux.
L'entreprise GOGO demande donc à ΦLOU d'effectuer régulièrement des simulations de restauration de données. Le commercial de la société ΦLOU leur propose le mode alternatif 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, GOGO doit demander à ΦLOU l'image par `md5` d'un fichier de son choix parmi les fichiers soumis, *précédé d'une séquence aléatoire d'un kilooctet* (également choisie par GOGO) : si `F` et `B` sont respectivement le fichier et le bloc aléatoire choisis par GOGO, la preuve que doit fournir ΦLOU est donc `md5`$(\texttt{B} \oplus \texttt{F})$, où $\oplus$ désigne la concaténation.

2. Est-ce mieux ? Expliquer.

Une semaine plus tard, le commercial de la société ΦLOU rappelle la société GOGO :

> *« 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 GOGO quant à l'honnêteté de la société ΦLOU ?