EA4 – Éléments d'algorithmique II
Examen de 1re session – 10 mai 2019

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. Les indications de durée sont manifestement trop optimistes et ne sont là qu'à titre comparatif entre les différents exercices.

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

Exercice 1 : hachage *(10 min)*

On considère deux tables de hachage $\texttt{H}_1$ et $\texttt{H}_2$ de longueur 13, dans lesquelles on place des valeurs entières, en utilisant la fonction de hachage $h(x) = x \mod 13$.

$\texttt{H}_1$ 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 17, 9, 39, 24, 4, 23, 11, 37.

b. Combien de comparaisons nécessite alors l'insertion de la valeur 22 dans la table ?

c. Comment supprimer la valeur 11 de la table ?

d. Comment se passe ensuite la recherche de la valeur 22 ?

Exercice 2 : minimum local *(15 min)*

Soit `T` un tableau de n éléments comparables, par exemple des entiers positifs. On dit que `T` possède un *minimum local en position* i si $\texttt{T}[i] \leqslant \texttt{T}[i-1]$ et $\texttt{T}[i] \leqslant \texttt{T}[i+1]$ (donc en particulier, $0 < i < n-1$).

1. Déterminer les 5 minima locaux du tableau `T` =

9	7	7	2	1	3	7	5	4	7	3	3	6

.

2. Tout tableau `T` (de longueur suffisante) possède-t-il un minimum local ?

On suppose dorénavant que `T` satisfait la propriété suivante :

$$\texttt{T}[0] \geqslant \texttt{T}[1] \text{ et } \texttt{T}[n-2] \leqslant \texttt{T}[n-1].$$

3. Montrer que sous cette hypothèse, `T` possède au moins un minimum local.

4. Décrire un algorithme *le plus efficace possible* pour déterminer un minimum local d'un tel tableau. Quelle est sa complexité ?

Exercice 3 : sous-tableau maximal *(45 min)*

Dans cet exercice, `T` désigne un tableau de n entiers (positifs *ou négatifs* sinon le problème a fort peu d'intérêt). Le problème du *sous-tableau maximal* consiste à déterminer les indices i et j (avec $i \leqslant j$) maximisant la somme des éléments du sous-tableau `T[`i`:`j`]` (c'est-à-dire de l'indice i à l'indice $j - 1$ inclus). Pour simplifier un peu l'écriture des algorithmes, on se contentera ici de calculer la *somme* du sous-tableau maximal (notée *sstm*(`T`)), et non les indices i et j.

Le but de cet exercice est de comparer plusieurs solutions à ce problème.

1. On considère tout d'abord l'algorithme `sstm_naif(T)` suivant :

```
def sstm_naif(T) :
   max = 0
   for i in range(len(T)) :
      for j in range(i, len(T)) :
         somme = sum(T[i:j])
         if somme > max : max = somme
   return max
```

Justifier sa correction et déterminer sa complexité. Préciser quelles opérations élémentaires sont pertinentes pour évaluer cette complexité.

2. Modifier légèrement `sstm_naif(T)` pour obtenir une complexité strictement meilleure.

Nous allons maintenant rechercher une solution de type « *diviser pour régner* ».

3. Supposons que `T = T1 + T2` ; *sstm*(`T`) dépend-elle uniquement de *sstm*(`T1`) et *sstm*(`T2`) ?

4. Écrire un algorithme de complexité linéaire `sstm_aux(T, deb, mil, fin)` calculant la somme maximale d'un sous-tableau de `T[deb:fin]` *chevauchant la position* `mil`.

5. En déduire un algorithme `sstm_dpr(T, deb, fin)` de type « *diviser pour régner* » pour calculer *sstm*(`T[deb:fin]`).

6. Soit $C(n)$ le nombre d'opérations élémentaires effectuées par `sstm_dpr(T, 0, n)`. Quelle est la relation de récurrence naturellement satisfaite par $C(n)$? En déduire l'ordre de grandeur de $C(n)$ (sans démonstration).

Nous allons enfin étudier une solution de type « *programmation dynamique* ».
Pour tout indice $k < n$, on considère les deux sommes suivantes :

- M_k la somme maximale de tous les sous-tableaux `T[`i`:`j`]` avec $i \leqslant j \leqslant k$ (autrement dit, $M_k = sstm($`T[:`k`]`$)$),
- et S_k la somme maximale de tous les sous-tableaux `T[`i`:`k`]` avec $i \leqslant k$.

7. Exprimer récursivement M_k et S_k en fonction de M_{k-1}, S_{k-1} et `T[`$k-1$`]`.

8. En déduire un algorithme `sstm_dyn(T)` de complexité linéaire calculant *sstm(T)*.

9. Justifier l'optimalité de `sstm_dyn(T)`.

Exercice 4 : tri flou d'intervalles *(1 heure)*

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 `T[`i`]` $=$ [`T[`i`][0]`, `T[`i`][1]`] tels que le i^{e} élément x_i de l'ensemble appartient à `T[`i`]` :

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

Le *tri flou* d'un tel tableau de longueur n consiste à réordonner les éléments de `T` en un tableau `S` vérifiant la propriété suivante :

Il existe des représentants x_i de chaque intervalle `S[i]` tels que $x_0 \leqslant x_1 \leqslant \cdots \leqslant x_{n-1}$.

1. Effectuer un tri flou du tableau d'intervalles suivants :

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

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 (et de même complexité en général) 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.

Par exemple, avec $[19, 21]$ comme pivot pour le tableau `T` de la question 1, le seul `chevauchement` possible serait $[19, 20]$: il possède bien la propriété (*), car il n'intersecte que $[19, 21]$ et $[17, 20]$, dont l'intersection est précisément $[19, 20]$ lui-même.

4. Quels sont les deux intervalles `chevauchement` possibles 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 contiennent `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`. On utilisera `trouve_chevauchement`.

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 : ABR d'intervalles (encore) *(30 min)*

Un *ABR d'intervalles* est une structure de données généralisant la notion d'ABR au cas où les éléments stockés dans les nœuds sont des intervalles – donc des éléments sans relation d'ordre *total* naturelle.

Chaque intervalle est identifié par ses bornes ***inf*** et ***sup***. La borne ***inf*** sert de clé de tri : tout intervalle contenu dans le sous-arbre gauche d'un nœud contenant l'intervalle $[a, b]$ a une borne ***inf*** inférieure à a, et tout intervalle contenu dans son sous-arbre droit a une borne ***inf*** (strictement) supérieure à a.

1. Construire un ABR d'intervalles par insertion successive des intervalles suivants :

$$[20, 36],\ [29, 99],\ [12, 25],\ [16, 64],\ [25, 50],\ [3, 7],\ [37, 42]\,.$$

2. On s'intéresse à l'existence dans l'ABR d'un intervalle contenant un certain entier `x`. Montrer que l'algorithme ci-dessous est incorrect :

```
def recherche_avec_bug(noeud, x) :
   if noeud == None : return False
   (inf, sup, fils_gauche, fils_droit) = noeud
   if x >= inf and x <= sup : return True
   elif x < inf : return recherche_avec_bug(fils_gauche, x)
   else : return recherche_avec_bug(fils_droit, x)
```

Pour pouvoir écrire un algorithme de recherche efficace, on rajoute une information complémentaire dans chaque nœud : la plus grande borne supérieure d'un intervalle du sous-arbre, `maxi`. Chaque nœud est donc un quintuplet `(inf, sup, maxi, fils_gauche, fils_droit)`.

3. Compléter l'ABR d'intervalles de la question 1 avec les valeurs `maxi` pour chaque nœud. Que se passe-t-il ensuite lors de l'insertion de l'intervalle $[7, 77]$?

4. Plus généralement, écrire une fonction `insertion(noeud, I)` insérant un intervalle `I` dans le sous-arbre de racine `noeud` d'un ABR d'intervalles. Quelle est sa complexité ?

5. Considérons un entier x et un nœud `n = (inf, sup, maxi, fg, fd)` dont le fils gauche est `fg = (infg, supg, maxg, fgg, fdg)`. Que peut-on dire si $x >$ `maxi` ? si $x <$ `inf` ? Montrer que si $x \geqslant$ `inf` et $x \leqslant$ `maxg`, alors x appartient à (au moins) un intervalle de l'ABR.

6. En déduire un algorithme de recherche (correct et) efficace. Quelle est sa complexité ?