EA4 – Éléments d'algorithmique II
Examen de 1re session – 17 mai 2024
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 (∗) marquent des questions plus difficiles que les autres. Elles seront hors barème.
Les indications de durée sont manifestement trop optimistes et ne sont là qu'à titre comparatif entre les différents exercices.
Vous êtes libres de choisir le langage utilisé pour décrire les algorithmes demandés, du moment que la description est suffisamment précise et lisible.

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

Exercice 1 : sous-tableau maximal *(50 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)+1) :
         somme = sum(T[i:j])
         if somme > max : max = somme
   return max
```

Justifier sa correction et déterminer sa complexité en temps, en précisant quelles opérations élémentaires sont pertinentes pour évaluer cette complexité.

2. Modifier légèrement `sstm_naif(T)` pour obtenir une complexité en temps strictement meilleure et une complexité en espace constante.

Recherche d'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 `T[`i`:`j`]` de `T` avec `deb` $\leqslant i \leqslant$ `mil` $< j \leqslant$ `fin`.

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).

Recherche d'une solution de type « programmation dynamique ».

Pour tout indice $k \leqslant n$, 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 2 : le tri du bibliothécaire (Library Sort) *(1 heure 10)*

Le tri du bibliothécaire est un tri inspiré du tri par insertion, et d'une pratique courante pour quiconque gère une bibliothèque : laisser des espaces libres sur chaque étagère pour pouvoir insérer de nouveaux livres sans avoir à décaler d'autres livres que ceux initialement présents sur la même étagère ; bien sûr, au fur et à mesure des ajouts, il devient nécessaire d'acheter de nouvelles bibliothèques, et de répartir à nouveau les livres sur l'ensemble des étagères disponibles pour recréer des espaces sur chacune.
Considérons un tableau `T`, dont on supposera par souci de simplification que la longueur n est une puissance de 2, et que les éléments sont des entiers tous distincts.

1. Rappeler quelle est la complexité, au pire, au mieux et en moyenne, de la i^{e} étape du tri par insertion de `T`.

Pour améliorer cette complexité, l'algorithme de tri du bibliothécaire construit, à partir de `T`, un tableau `B` de longueur αn, avec $\alpha > 1$, contenant les n éléments de `T` triés en ordre croissant, et $(\alpha-1)n$ cases vides réparties dans tout le tableau. L'algorithme procède en plusieurs tours, dont le k^{e} a pour but d'achever le tri des 2^k premiers éléments de `T`. Chacun de ces tours comprend deux phases :

(a) un tableau $\texttt{B}_k$ (suffisamment grand) est initialisé, à partir du tableau $\texttt{B}_{k-1}$ obtenu au tour précédent (où par convention $\texttt{B}_0$ `= [T[0]]`), avec les 2^{k-1} premiers éléments de `T` triés en ordre croissant et espacés aussi régulièrement que possible ;

(b) chacun des 2^{k-1} éléments suivants de `T` est inséré dans $\texttt{B}_k$, en décalant si besoin (vers la droite) certains des éléments déjà présents si la case où l'élément doit s'insérer n'est pas vide.

Pour simplifier l'analyse de l'algorithme, on supposera ici plus précisément que la phase (a) initialise $\texttt{B}_k$ en laissant exactement deux cases libres avant chaque élément, par exemple pour $k = 3$ et $\texttt{B}_2$ contenant les $2^2 = 4$ éléments `17`, `32`, `37` et `53`, on initialise $\texttt{B}_3$ à :

2. Simuler le tri du bibliothécaire pour le tableau `T = [7, 2, 6, 8, 4, 5, 1, 3]`.

3. Quelle doit être au minimum, en fonction de k, la taille du tableau $\texttt{B}_k$ pour que la phase (b) puisse toujours être réalisée sans débordement ? En particulier, quelle est la taille du tableau résultat `B` ? Dans la suite, on utilisera un unique tableau de cette taille, pour y stocker les $\texttt{B}_k$ successifs (complétés par des cases vides le cas échéant).

4. Écrire une fonction `disperse(B, k)` *aussi efficace que possible* permettant de réaliser la phase (a) de la k^{e} étape en modifiant *en place* le contenu du tableau `B` passé en paramètre. *Autrement dit, en supposant que le tableau* `B` *est initialement égal à* $\texttt{B}_{k-1}$ *(complété par des cases vides), modifier son contenu pour obtenir* $\texttt{B}_k$ *(complété par des cases vides).*

5. Quelle est la complexité en temps de `disperse(B, k)` ? Quelle est donc la complexité cumulée des phases (a) de toutes les étapes du tri ?

6. Écrire une fonction `cherche(x, B, fin)` *aussi efficace que possible* permettant de déterminer la position à laquelle l'élément `x` doit être inséré dans `B`, en supposant que la dernière case remplie de `B` est en position `fin-1`. Si deux cases vides sont possibles, on choisira la première.

7. Quelle est la complexité de `cherche(x, B, fin)` ? Quelle est donc la complexité cumulée de toutes les recherches effectuées lors du tri ?

8. Écrire une fonction `insere(x, B, i)` *aussi efficace que possible* qui insère l'élément `x` à la position `i` dans `B` en décalant des éléments vers la droite si nécessaire.

9. Décrire explicitement l'algorithme `triBibliothecaire(T)` à l'aide des fonctions précédentes.

10. Quelle est *dans le meilleur cas* la complexité de `insere(x, B, i)` ? Quelle est, toujours dans le meilleur cas, la complexité cumulée de toutes les insertions effectuées lors du tri ? Quelle est donc la complexité dans le meilleur cas de l'algorithme de tri du bibliothécaire ? Justifier que cette borne est atteinte.

11. Quelle est *dans le pire cas* la complexité de `insere(x, B, i)` ? Quelle est, toujours dans le pire cas, la complexité cumulée de toutes les insertions effectuées lors du tri ? Quelle est donc la complexité dans le pire cas de l'algorithme de tri du bibliothécaire ? Justifier que cette borne est atteinte.

12. La complexité en moyenne est plus difficile à évaluer. Supposons comme d'habitude que le tableau `T` est choisi uniformément parmi les permutations de taille n. Considérons une certaine étape k, et notons D_i le nombre de décalages nécessaires pour la i^{e} insertion dans $\texttt{B}_k$.

a. $(*)$ On appelle *rouges* les cases de $\texttt{B}_k$ dont la position vaut 1 modulo 3. Montrer que la probabilité qu'une case rouge (fixée) soit occupée est inférieure à $\frac{1}{2}$.

b. En déduire que $\mathbb{P}(D_i > 3\ell) \leqslant 1/2^\ell$ pour tous i et ℓ.

c. En déduire que $\mathbb{P}(D_i > 6 \log n) \leqslant 1/n^2$ pour tout i.

d. $(*)$ Soit $D = \max_{1 \leqslant i \leqslant n} D_i$. Montrer que $\mathbb{P}(D > 6 \log n) \leqslant 1/n$.

e. $(*)$ En déduire que $\mathbb{E}(D) \in O(\log n)$.

f. Quelle est donc la complexité en moyenne du tri du bibliothécaire ?

Exercice 3 : les arbres à bouc émissaire (Scapegoat Trees) *(1 heure)*

Un *arbre à bouc émissaire* (ABE dans la suite) est un arbre binaire de recherche (ABR) enrichi pour permettre un auto-équilibrage. Plus précisément, c'est un triplet `(A, n, m)` formé d'un arbre binaire de recherche `A` (identifié par sa racine) et de deux compteurs `n` et `m` tels que $\frac{1}{2}\,\mathtt{m} \leqslant \mathtt{n} \leqslant \mathtt{m}$; 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 $\log_{3/2} \mathtt{m}$, 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. Comment `n` et `m` doivent-ils évoluer lors des opérations d'insertion et de suppression d'un élément dans `A` ?

2. Décrire un algorithme `reequilibre(r)` le plus efficace possible 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 renvoyant le nouvel ABE.

5. Quelle est sa complexité lorsqu'aucun rééquilibrage n'est nécessaire ? Et lorsqu'un rééquilibrage est réalisé ?

Lors de l'insertion d'un élément `x`, si la hauteur de l'ABR dépasse la limite $\log_{3/2} \mathtt{m}$, 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 $\mathtt{taille(fils)} > \frac{2}{3}\,\mathtt{taille(b)}$, où `fils` est le fils de `b` qui est un ancêtre de `x`.

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

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

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

9. Quelle est la complexité de cet algorithme ?

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

11. Quelle est sa complexité lorsqu'aucun rééquilibrage n'est nécessaire ?

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 $\Theta(n \log n)$.*

12. 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 ?