EA4 – Éléments d'algorithmique II
Examen de 1re session – 26 mai 2021
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.
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 : minimum et maximum simultanés

Le but de cet exercice est de trouver un algorithme optimal pour déterminer *à la fois* le minimum et le maximum d'un tableau `T` de n éléments comparables (entiers par exemple).

1. Décrire un algorithme `minmax_naif(T)` déterminant *successivement* le minimum puis le maximum de `T` en effectuant, pour chacune de ces deux recherches, le moins de comparaisons possible. Quel est le nombre exact $C_{naïf}(n)$ de comparaisons effectuées par `minmax_naif(T)` pour un tableau de taille n ?
2. Expliquer pourquoi tout algorithme qui résout ce problème a une complexité en temps en $\Omega(C_{naïf}(n))$ – autrement dit, l'ordre de grandeur de $C_{naïf}(n)$ est optimal.

Pour ce problème, la recherche du « meilleur algorithme » consiste donc uniquement à chercher à atteindre la plus petite constante multiplicative possible.

3. Quel est le nombre minimal de comparaisons nécessaire pour $n = 1$? $n = 2$? $n = 3$? $n = 4$? Justifier.

Nous allons tout d'abord rechercher une solution de type « *diviser pour régner* ».

4. Supposons que `T = T1 + T2` ; comment calculer `min(T)` et `max(T)` à partir de `min(T1)`, `max(T1)`, `min(T2)` et `max(T2)` en effectuant le moins de comparaisons possible ?
5. En déduire un algorithme `minmax_dpr(T, deb, fin)` de type « *diviser pour régner* » pour calculer le couple `(min(T[deb:fin]), max(T[deb:fin]))`.
6. Soit $C_{dpr}(n)$ le nombre de comparaisons effectuées par `minmax_dpr(T, 0, n)` pour un tableau de taille n. Quelle relation de récurrence $C_{dpr}(n)$ vérifie-t-il ? Calculer ses premières valeurs (jusqu'à $n = 8$). Proposer un algorithme `cplx_dpr(n)` le plus efficace possible pour calculer $C_{dpr}(n)$.

(On constate que $C_{dpr}(n)$ croît moins vite que $C_{\text{naïf}}(n)$; il n'est pas demandé de le démontrer.)

Nous allons enfin étudier une solution de type « *programmation dynamique* ».

7. Soit $k \leqslant$ `len(T)` $- 2$. Comment calculer `min(T[:`$k+2$`])` et `max(T[:`$k+2$`])` à partir de `min(T[:`k`])`, `max(T[:`k`])`, `T[`k`]` et `T[`$k+1$`]` en effectuant le moins de comparaisons possible ?

8. En déduire un algorithme `minmax_dyn(T)` calculant *simultanément* `min(T)` et `max(T)` selon le principe de la programmation dynamique.

9. Quel est nombre exact $C_{dyn}(n)$ de comparaisons effectuées par `minmax_dyn(T)` pour un tableau de taille n si n est pair ? si n est impair ?

(On peut montrer que $C_{dyn}(n)$ est bien l'optimum, mais c'est assez difficile et non demandé ici.)

Exercice 2 : densité

Dans cet exercice, on considère des tableaux de nombres (non nécessairement entiers), supposés tous distincts pour simplifier.

On définit l'élément *le plus isolé* d'un tel tableau `T` comme celui qui maximise la distance à son plus proche voisin. Par exemple, dans `T = [ 5.4, 3.2, 4.7, 1.5, 0.9, 1.8 ]`, l'élément le plus isolé est `3.2`, à distance `1.4` de son plus proche voisin, `1.8`.

1. Décrire un algorithme naïf `plus_isole_naif(T)` permettant de résoudre ce problème sans modifier le tableau `T` et avec mémoire auxiliaire constante.

2. Proposer un algorithme `plus_isole_efficace(T)` plus efficace que l'algorithme naïf. Justifier sa correction et sa complexité (au pire et en moyenne).

La distance `r` entre l'élément le plus isolé et son plus proche voisin permet de définir une mesure de *densité* : la densité autour d'un élément `x` est le nombre d'éléments de `T` à distance au plus `r` de `x`. Elle vaut donc 1 pour l'élément le plus isolé de `T` et au moins 2 (voire beaucoup plus) pour tous les autres éléments de `T`.

3. Définir un algorithme `densite_tab_trie(T, x, r)` le plus efficace possible pour déterminer la densité de `T` autour de l'élément `x` en supposant que `T` est trié. Quelle est sa complexité ?

4. Supposons qu'on souhaite calculer les densités autour d'un nombre m d'éléments d'un même tableau `T` *initialement non trié*. Pour quels ordres de grandeur de m est-il judicieux de trier `T` ?

Exercice 3 : les bleus dominent-ils les rouges ?

On considère des vecteurs dans l'espace de dimension $d \geqslant 1$. Étant donné deux tels vecteurs `u` et `v` dans $\mathbb{R}^d$, on dit que `u` *domine* `v`, et on note $\mathtt{u} \succcurlyeq \mathtt{v}$, si $\mathtt{u}[i] \geqslant \mathtt{v}[i]$ pour tout $i < d$.

Le problème étudié dans cet exercice est le suivant : étant donné deux ensembles `B` (bleu) et `R` (rouge) de vecteurs de dimension d, compter le nombre de couples $(\mathtt{b}, \mathtt{r})$ formés d'un vecteur bleu `b` et d'un vecteur rouge `r` tels que $\mathtt{b} \succcurlyeq \mathtt{r}$. On note ce nombre $\Delta(\mathtt{B}, \mathtt{R})$.

On note enfin n_b et n_r les cardinaux respectifs de `B` et `R`, et $n = n_b + n_r$.

Par souci de simplification, on supposera que, dans chaque dimension, les coordonnées des vecteurs sont toutes distinctes.

1. Écrire un algorithme `domine(u, v)` qui teste si `u` $\succcurlyeq$ `v` (supposés de même dimension).

2. Donner un algorithme `delta_naif(B, R)` résolvant le problème de la manière la plus naïve possible. Exprimer sa complexité en fonction de n_b, n_r et d.

3. On suppose ici $d = 1$. Comment résoudre le problème le plus efficacement possible si `B` et `R` sont supposés triés (en ordre croissant) ? En déduire un algorithme `delta_dim1(B, R)` pour `B` et `R` quelconques. Quelle est sa complexité ? La comparer à celle de l'algorithme `delta_naif(B, R)`, en supposant que n_b et n_r sont du même ordre de grandeur ($\Theta(n)$, donc).

4. On suppose ici $d = 2$.

a. Supposons `B` et `R` triés par abscisse croissante. Comment couper `B` en `B1 + B2` et `R` en `R1 + R2` de telle sorte que `len(B1) + len(R1) = len(B2) + len(R2)` (+1 si n est impair), et que tous les vecteurs dans `B1` ou `R1` aient une abscisse inférieure à tous ceux de `B2` ou `R2` ?

b. Étant donné un tel découpage, que peut-on dire de Δ(`B1`, `R2`) ?

c. Expliquer comment Δ(`B2`, `R1`) peut être calculé efficacement si on suppose que ces tableaux sont triés par *ordonnée* croissante.

d. En déduire un algorithme `delta_dim2_aux(B, R)` qui, étant donné `B` et `R` triés par *abscisse* croissante, renvoie Δ(`B`,`R`) ainsi que `B` et `R` triés par *ordonnée* croissante.

e. En déduire un algorithme `delta_dim2(B, R)` efficace pour calculer Δ(`B`,`R`) et préciser sa complexité en fonction de n.

Exercice 4 : arbres 2-3-4

Les arbres 2-3-4 sont des cas particuliers de B-arbres, dont les nœuds ont arité 2, 3 ou 4 (ce ne sont donc pas exactement des B-arbres d'ordre p comme étudiés lors du TD n° 12, mais bien sûr les propriétés sont proches) :

- chaque nœud ou feuille d'un arbre 2-3-4 contient entre 1 et 3 clés ;
- un nœud d'arité $k+1$ contient exactement k clés ;
- ***toutes les feuilles ont la même profondeur.***

La propriété d'ordre des ABR s'étend quant à elle simplement aux nœuds d'arité $k+1$: si un nœud contient les clés $c_0 < c_1 < \cdots < c_{k-1}$ et possède les sous-arbres A_0, A_1... A_k, toutes les clés de A_i sont supérieures à c_{i-1} (si $i > 0$) et inférieures à c_i (si $i < k$).

1. a. Décrire toutes les formes possibles pour un arbre 2-3-4 de hauteur 1. Combien de clés un tel arbre peut-il contenir ?

b. Quelles sont les hauteurs possibles pour un arbre 2-3-4 contenant 15 clés ? Donner un exemple pour chacune (avec comme clés les entiers de 1 à 15).

2. Donner une minoration du nombre de clés à profondeur k, pour $k > 0$. En déduire que la hauteur d'un arbre 2-3-4 contenant n clés est en $\Theta(\log n)$ dans tous les cas.

Par souci de simplification, on suppose toutes les clés distinctes, et on considère que chaque `sommet` contient :

- un champ booléen `feuille` indiquant s'il s'agit d'une feuille ou non,

– un champ entier `taille` compris entre 1 et 3 indiquant le nombre de clés qu'il contient,

– un tableau `cles` de longueur 3, trié, contenant les clés[1],

– un tableau `fils` de longueur 4 contenant les fils[1], dans l'ordre,

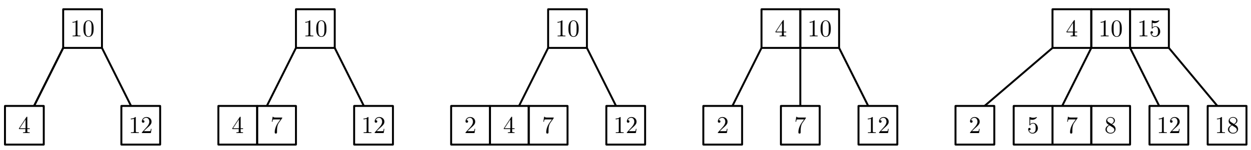
pour lesquels on dispose de tous les accesseurs nécessaires – par exemple, `getTaille(sommet)`, `getCles(sommet)`, `getCle(i, sommet)`...

3. Décrire un algorithme `minimum234(racine)` qui renvoie la plus petite clé d'un arbre 2-3-4 dont `racine` est la racine. Quelle est sa complexité ?

4. Décrire un algorithme `cherche234(c, racine)` *le plus efficace possible* renvoyant le nœud de l'arbre 2-3-4 de racine `racine` contenant la clé `c`, s'il en existe, et `False` sinon. Quelle est sa complexité ?

L'ajout de nouvelles clés est plus complexe, du fait de la contrainte sur la profondeur des feuilles : comme dans un ABR, on cherche à ajouter la clé dans une feuille, mais si celle-ci est déjà *saturée*, c'est-à-dire si elle contient déjà 3 clés, il est nécessaire de l'*éclater* : créer une feuille supplémentaire, nécessairement au même niveau, et faire remonter une clé dans son père – ce qui peut avoir des répercussions sur celui-ci, voire sur toute sa lignée ancestrale ; s'il faut augmenter la hauteur de l'arbre, cela ne peut se faire qu'au niveau de la racine.

5. Insérer la clé 6 dans chacun des arbres ci-dessous.



Vous avez dû constater qu'un éclatement au niveau d'une feuille peut se répercuter sur toute la branche jusqu'à la racine. Il est en fait plus simple de réaliser, lors de la descente, un « éclatement préventif » de chaque nœud saturé rencontré pour s'assurer que de telles répercussions ne se produiront pas.

6. Décrire l'opération d'éclatement d'un nœud, selon qu'il s'agit de la racine ou d'un autre nœud (dont le père a lui-même déjà subi un éclatement si nécessaire).

7. (∗) Écrire un algorithme `insertion234(c, racine)` permettant d'insérer la clé `c` dans l'arbre 2-3-4 de racine `racine`. Quelle est sa complexité ?

Pour la suppression, le problème inverse se pose : celui des nœuds ne contenant qu'une seule clé, qui doivent être *étoffés* en y ajoutant une clé (prise d'un nœud proche). À nouveau, il est plus simple de traiter le problème à la descente plutôt qu'à la remontée, en déplaçant des clés vers les nœuds ayant une seule clé rencontrés lors de la descente.

8. Décrire l'opération d'étoffage d'un nœud, selon qu'il s'agit de la racine ou d'un autre nœud (dont le père a lui-même été étoffé si nécessaire).

9. (∗) Écrire un algorithme `suppression234(c, racine)` permettant de supprimer la clé `c` de l'arbre 2-3-4 de racine `racine`.

1. et `None` dans les cases inutilisées