EA4 – Éléments d'algorithmique II
Examen de 1re session – 16 mai 2017
Durée : 3 heures

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

Le sujet est trop long (même s'il l'est moins qu'il n'en a l'air). 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.

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

Exercice 1 : afficher les 10 meilleurs résultats *(30 min)*

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

> *Étant donné une liste de n éléments comparables (par exemple, un million de réponses à une requête d'un moteur de recherche), déterminer les k plus grands (par exemple, les 10 réponses les plus pertinentes), dans l'ordre.*

Les solutions étudiées ici sont des applications directes des algorithmes vus en cours. Il vous est demandé de les utiliser, ***pas de les réécrire*** (sauf à la question **??**). Vous pouvez énoncer sans démonstration les résultats de complexité vus en cours – en mentionnant précisément les ordres de grandeur dans le pire cas, le meilleur, et en moyenne.
Dans la suite, k désigne un entier et `T` un tableau de taille n contenant des éléments comparables.

1. *Solution 1 : sélection répétée du maximum*
   a. Écrire une fonction `selectionMax(T, lg)` qui réordonne les éléments de `T[:lg]` de manière à placer le plus grand dans la dernière case.
   b. En déduire un algorithme `selectionMeilleurs(T, k)` qui retourne la liste ordonnée des k plus grands éléments de `T`. Quelle est sa complexité (en fonction de n et k) ?

2. *Solution 2 : par un tri préalable*
Quelle est la meilleure complexité possible d'un algorithme `triMeilleurs(T, k)` basé sur le tri préalable du tableau ? À l'aide de quel algorithme de tri ?

3. *Solution 3 : avec `quickSelect`*
Décrire un algorithme `quickMeilleurs(T, k)` utilisant un appel à `quickSelect` pour calculer la liste ordonnée des k plus grands éléments de `T`. Quelle est sa complexité ?

4. *Solution 4 : avec un arbre binaire de recherche*
   a. Quelle est la complexité de la création d'un ABR contenant les mêmes éléments que `T` ?
   b. Comment en extraire ensuite les k plus grands éléments ?
   c. Quelle est la complexité totale de cette méthode ?

5. *Solution 5 : avec un tas*
   a. Quelle est la complexité de la transformation du tableau `T` en tas ? Rappeler le principe de l'algorithme en le déroulant sur un exemple bien choisi.
   b. Comment en extraire ensuite les k plus grands éléments ?
   c. Quelle est la complexité totale de cette méthode ?

6. (∗) Discuter de la meilleure solution, en fonction de n, k, et des constantes cachées dans les Θ.

Exercice 2 : programmation dynamique *(20 min)*

On considère l'algorithme suivant :

```
def algoIdiot(n) :
   if n < 3 : return 1
   else : return 4 * algoIdiot(n-1) + 2 * algoIdiot(n-3)
```

Soit v_n la valeur calculée par `algoIdiot(`n`)`.

1. Montrer que pour tout $n \geqslant 2$, $v_n \geqslant 4^{n-2}$.

2. Soit $A(n)$ le nombre d'opérations arithmétiques sur des entiers effectuées lors de l'exécution de `algoIdiot(`n`)`. Montrer que $A(n)$ est croissante, puis que $A(n) \in \Omega(2^{n/3})$, et conclure.

3. Proposer un algorithme `lineaire(n)` calculant v_n en effectuant seulement un nombre linéaire d'opérations arithmétiques.

4. La complexité en temps de `lineaire(n)` est-elle réellement linéaire ? Justifier.

5. $(*)$ Proposer un algorithme `meilleur(n)` calculant v_n avec une complexité en temps inférieure à celle de `lineaire(n)`.

Exercice 3 : B-arbres *(40 min)*

Les *B-arbres d'ordre p* constituent une variante des arbres binaires de recherche, utilisée notamment pour les systèmes de gestion de fichiers. Les différences majeures sont :

- chaque nœud ou feuille contient au plus $2p$ clés ;
- chaque nœud ou feuille (***sauf la racine***) contient au moins p 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, tous les éléments de A_i sont supérieurs à c_{i-1} (si $i > 0$) et inférieurs à c_i (si $i < k$).

1. a. Quelle est la seule forme possible pour un B-arbre d'ordre p contenant au plus $2p$ clés ? Et exactement $2p+1$ clés ?

b. Quelles sont les deux formes possibles pour un B-arbre d'ordre 1 et de hauteur 1 ? Combien chacune d'elles peut-elle contenir de clés ?

c. Dessiner les quatre B-arbres d'ordre 1 contenant les clés 1, 2, 3, 4, 5, 6, 7.

2. Donner une minoration du nombre de clés à profondeur k, pour $k > 0$ (en fonction de p). En déduire que la hauteur d'un B-arbre 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 p et $2p$ indiquant le nombre de clés qu'il contient,
- un tableau `cles` de longueur $2p$, trié, contenant les clés [1],
- un tableau `fils` de longueur $2p+1$ contenant les fils[1], dans l'ordre,

pour lesquels on dispose des accesseurs `get...(...)` et modificateurs `set...(...)` nécessaires.

1. et `None` dans les cases inutilisées

3. Écrire un algorithme `appartient(c, sommet)` *le plus efficace possible* retournant
   - `True` si `sommet` contient la clé `c`,
   - `False` si `sommet` est une feuille ne contenant pas `c`,
   - et l'unique sous-arbre de `sommet` susceptible de contenir `c` sinon.

   Quelle est sa complexité (en fonction de p, qui a vocation à être grand) ?

4. En déduire un algorithme `cherche(c, racine)` *le plus efficace possible* retournant le nœud du B-arbre de racine `racine` contenant `c`, s'il en existe, et `False` sinon.

5. Quelle est la complexité de `cherche(c, racine)`, en fonction de p et du nombre n de clés stockées dans le B-arbre de racine `racine` ?

L'ajout de nouveaux éléments est plus complexe, du fait de la contrainte sur la profondeur des feuilles : comme dans un ABR, on cherche à ajouter l'élément dans une feuille, mais s'il est nécessaire d'en créer une nouvelle, elle doit être au même niveau que les précédentes – ce qui peut avoir des répercussions sur son père, voire toute sa lignée ancestrale. S'il faut finalement augmenter la hauteur de l'arbre, cela devra se faire au niveau de la racine.

6. Comment créer un B-arbre d'ordre 2 en ajoutant successivement les clés 1, 2, 3, 4, 5 ? Et un B-arbre d'ordre 1 ? Poursuivre dans chacun des deux cas avec l'insertion de 6, 7, 8.

7. ($*$) Proposer un algorithme pour le cas général. Quelle est sa complexité ?

Exercice 4 : améliorations du tri fusion *(1h30)*

On considère l'algorithme suivant, où `fusion(T1, T2)` retourne une copie triée de `T1+T2` si `T1` et `T2` sont eux-mêmes triés :

```
def triIteratifNaif(T) :
   pile = [ [elt] for elt in T ]
   while len(pile) > 1 :
      pile.append(fusion(pile.pop(), pile.pop()))
      # rappel : append ajoute (et pop supprime) un élément en fin de liste
   return pile.pop()
```

1. a. Dérouler `triIteratifNaif([1, 3, 6, 2, 4, 8, 5, 7])`.

b. Démontrer que `triIteratifNaif(T)` retourne une copie triée de `T` pour tout tableau `T`.

c. Déterminer sa complexité en temps ainsi que sa complexité en espace.

2. Expliquer comment obtenir une complexité en espace linéaire en changeant la représentation des tableaux dans la `pile` (ainsi que le prototype de `fusion`).
Indication : penser aux indices.

3. a. Appliquer le tri fusion sur le même tableau `[1, 3, 6, 2, 4, 8, 5, 7]`.

b. Dans quel(s) ordre(s) les fusions peuvent-elles être effectuées ?

c. Que modifier dans `triIteratifNaif(T)` pour obtenir une version itérative du tri fusion ? *Pour simplifier, ne considérer que des tableaux ayant comme longueur une puissance de 2.*

Le *tri fusion naturel*, proposé par D. Knuth, procède de manière analogue, mais en cherchant à tirer parti de l'existence de portions déjà triées dans `T`, qu'on appelle des *monotonies* de `T`. Une *décomposition en monotonies* de `T` est ainsi une suite de sous-tableaux $\texttt{T}[i_0:i_1]$, $\texttt{T}[i_1:i_2]$, ..., $\texttt{T}[i_{k-1}:i_k]$, tous triés, et dont la concaténation est `T` (donc $i_0 = 0$ et $i_k = \texttt{len(T)}$).
Par exemple, `[1, 3, 6, 2, 4, 8, 5, 7]` se décompose en `([1,3,6], [2,4,8], [5,7])`. Il se décompose aussi en `([1], [3,6], [2,4], [8], [5,7])`, mais c'est moins intéressant.

4. Écrire une fonction `monotonies(T)` qui retourne une liste représentant une décomposition en monotonies de `T`. Quelle est sa complexité ?

5. Décrire l'algorithme `triFusionNaturel(T)` obtenu. Quelle est sa complexité dans le pire cas ? dans le meilleur cas ? en moyenne ?

Certains algorithmes de tris très optimisés [2] procèdent plus ou moins de cette manière. Cependant, un inconvénient de l'algorithme précédent est sa complexité en espace. Pour diminuer la hauteur de la structure auxiliaire, on peut envisager de procéder à certaines fusions au fur et à mesure de la recherche de monotonies [3]. Le schéma général de tels algorithmes est le suivant [4] :

```
def triParPileGenerique(T, conditionsDePile, effectueFusionsBienChoisies) :
   pile = []
   for m in monotonies(T) : # où monotonies(T) serait un itérateur et non une liste
      pile.append(m)
      # parfois, effectuer une ou plusieurs fusions
      while not conditionsDePile(pile) : effectueFusionsBienChoisies(pile)
   # une fois toutes les monotonies insérées, terminer les fusions
   while len(pile) > 1 :
      pile.append(fusion(pile.pop(), pile.pop()))
   return pile.pop()
```

Les deux fonctions `conditionsDePile` et `effectueFusionsBienChoisies` permettent de spécifier le comportement exact de l'algorithme de tri.

6. Quelle est la complexité en temps de l'algorithme dans le meilleur cas ?

7. Pourquoi ne faut-il pas *systématiquement* fusionner la monotonie `m` avec le sommet de pile ?

8. Il est fréquemment nécessaire d'utiliser un algorithme de tri *stable*. Quelle condition les fusions réalisées par `effectueFusionsBienChoisies` doivent-elles vérifier pour que l'algorithme obtenu soit stable ?

Un exemple de condition de pile fournissant un algorithme à la fois simple et efficace est la suivante : les deux monotonies de dessus de pile, `m = pile[-2]` et `n = pile[-1]`, vérifient $\texttt{len(m)} \geqslant 2 \cdot \texttt{len(n)}$.

9. Expliquer comment rétablir la condition de pile dans chacun des cas suivants :

a. ajout de `[3]` à `pile = [[2,5,8,9], [1,6]]`

b. ajout de `[2,8]` à `pile = [[1,3,4,6,7,9], [5]]`

c. ajout de `[3]` à `pile = [[2,5,8,9], [1,6], [4]]`

10. Écrire les fonctions `conditionsDePile` et `effectueFusionsBienChoisies` correspondantes.

11. Démontrer qu'à la fin de chaque tour de boucle, pour toutes monotonies `m` et `n` consécutives dans la pile, $\texttt{len(m)} \geqslant 2 \cdot \texttt{len(n)}$.

12. En déduire un minorant de la longueur de la monotonie `pile[-`k`]` (en fonction de k), puis un majorant de la hauteur de `pile` (en fonction de n, longueur de `T`).

13. (∗) On considère un élément `elt` donné. Quelle est la longueur minimale de la monotonie à laquelle `elt` appartient après k fusions le concernant ? En déduire une majoration du nombre de fusions pouvant concerner `elt`.

14. (∗) En déduire la complexité en temps de cet algorithme dans le pire cas.

2. par exemple `TimSort`, écrit pour le `sort` de Python, et maintenant utilisé par de nombreux autres langages.

3. manipuler des données qui viennent d'être traitées plutôt que de plus anciennes constitue d'ailleurs une meilleure stratégie du point de vue des *caches* du système.

4. pour simplifier, on reprend ici la fonction `fusion` du début de l'exercice, mais il faudrait bien sûr procéder comme à la question **??**.