EA4 – Éléments d'algorithmique
TD n° 7 : sélection rapide et tri fusion amélioré
(Correction)

Exercice 1 : déroulement de `QuickSelect`

1. On considère le tableau `T` suivant :

153	159	53	135	106	75	36	73

Décrire le déroulement de la sélection rapide sur le tableau `T` pour déterminer l'élément de rang 5 en prenant le premier élément comme pivot. Combien de comparaisons sont effectuées ?

2. On considère maintenant le tableau suivant :

106	153	159	53	135	75	36	73

Combien de comparaisons sont effectuées pour déterminer l'élément de rang 5 ? et de rang 4 ?

3. Déterminer la complexité de l'algorithme dans le meilleur cas et le pire cas.

4. On considère que pour trouver l'élément de rang recherché, l'algorithme divise à chaque tour le tableau en deux sous-tableaux *gauche* et *droite* de même taille et qu'il s'arrête lorsque *gauche* et *droite* sont de taille 0 ou 1. Déterminer dans ce cas la complexité de l'algorithme.

Exercice 2 : améliorations du tri fusion *(examen 2017)*

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 [1] 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 [2]. Le schéma général de tels algorithmes est le suivant [3] :

```
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.

1. par exemple `TimSort`, écrit pour le `sort` de Python, et maintenant utilisé par de nombreux autres langages.
2. 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.
3. pour simplifier, on reprend ici la fonction `fusion` du début de l'exercice, mais il faudrait bien sûr procéder comme à la question 2.