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

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 : des tris linéaires

Dans cet exercice, nous allons considérer deux situations permettant un tri en temps linéaire.

1. Expliquer en quoi ce n'est pas une contradiction avec la borne $\Omega(n \log n)$ démontrée en cours.
2. Rappeler le fonctionnement du tri par base, pour une liste de n mots de longueur ℓ sur un alphabet de taille k, en précisant comment assurer la stabilité de chaque étape. Quelle est sa complexité en temps et en espace ?
3. Décrire l'exécution du tri par base sur la liste `657`, `819`, `457`, `329`, `416`, `720`, `455`.
4. Pourquoi ne pas exécuter les étapes dans l'ordre inverse ?

Les conditions permettant un tri linéaire ne portent pas nécessairement sur la nature des éléments à trier – les propriétés de la liste peuvent également jouer un rôle. On rappelle (*cf.* TD n° 4) qu'un *anneau coupé* est un tableau `T` de n éléments distincts pour lequel il existe un certain indice `k` (inconnu a priori) tel que `T[k:] + T[:k]` est trié en ordre décroissant (pas forcément strict).

5. Expliquer comment trier en temps linéaire un tel tableau.

Exercice 3 : amélioration de la complexité en espace de la fusion

1. Rappeler l'algorithme de fusion de deux tableaux `T1` et `T2` vers un tableau annexe `T`. Quelle est sa complexité (exacte) en espace ?
2. Supposons que `T1` et `T2` soient deux sous-tableaux consécutifs, et de même longueur ℓ, d'un plus grand tableau `T`. Comment peut-on fusionner ces deux sous-tableaux au sein de `T` avec une complexité en espace de ℓ ?
3. Écrire la fonction `fusion(T, deb1, deb2, fin2)` qui fusionne `T1` et `T2` « en place », où `deb1` est la première case de `T1`, `deb2` la première case de `T2` et `fin2` la première case hors de `T2`.
4. Dans le cas général de deux sous-tableaux consécutifs de longueurs ℓ_1 et ℓ_2 *a priori* différentes, que faudrait-il modifier pour que la complexité en espace auxiliaire reste de $\min(\ell_1, \ell_2)$?
5. Justifier que `fusion(T, deb1, deb2, fin2)` réalise un tri *stable* de `T[deb1:fin2]`.

Exercice 4 : tri fusion itératif et tri fusion naturel

On considère l'algorithme suivant, utilisant la fonction `fusion(T, deb1, deb2, fin2)` de l'exercice précédent, dont on suppose qu'elle retourne le couple `(deb1, fin2)` :

```
def triIteratifNaif(T) :
   pile = [ (i, i+1) for i in range(len(T)) ]
   while len(pile) > 1 :
     deb2, fin2 = pile.pop()
     deb1, fin1 = pile.pop()     # invariant : fin1 == deb2
     pile.append(fusion(T, deb1, deb2, fin2))
     # rappel : append ajoute (et pop supprime) un élément en fin de liste
   return T
```

1. **a.** Dérouler `triIteratifNaif([1, 3, 6, 2, 4, 8, 5, 7])`.
   b. Démontrer que `triIteratifNaif` est un algorithme de tri ; de quel algorithme classique s'agit-il ?
   c. Déterminer sa complexité en temps ainsi que sa complexité en espace.
2. Modifier `triIteratifNaif` pour obtenir un tri fusion itératif.

Le *tri fusion naturel*, proposé par D. Knuth, est une variante du tri fusion qui cherche à tirer parti de l'existence de portions déjà triées dans un tableau `T`, appelées *monotonies* de `T`. Une *décomposition en monotonies* de `T` est ainsi une suite de sous-tableaux `T`$[i_0 : i_1]$, `T`$[i_1 : i_2]$, ..., `T`$[i_{k-1} : i_k]$, tous triés, et dont la concaténation est `T` (donc $i_0 = 0$ et $i_k =$ `len(T)`).
Par exemple, `[1, 3, 6, 2, 4, 8, 5, 7]` se décompose en `[[1,3,6], [2,4,8], [5,7]]`, soit, en termes de bornes des sous-tableaux, `[(0,3), (3,6), (6,8)]` ; il se décompose aussi en `[[1], [3,6], [2,4], [8], [5,7]]`, mais c'est moins intéressant.

3. Écrire une fonction `monotonies(T)` qui retourne une liste représentant une décomposition en monotonies de `T`. Quelle est sa complexité ?
4. Décrire l'algorithme `triFusionNaturel(T)` obtenu. Quelle est sa complexité dans le pire cas ? dans le meilleur cas ? en moyenne ?

Exercice 5 : tri par pile générique et tri (à la) TimSort

Certains algorithmes de tris très optimisés [1] procèdent plus ou moins à la manière du tri fusion naturel. Cependant, un inconvénient de cet algorithme 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 :

```
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 :
```

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.

```
    deb2, fin2 = pile.pop()
    deb1, fin1 = pile.pop()
    pile.append(fusion(T, deb1, deb2, fin2))
return T
```

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

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

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

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, `m1 = pile[-2]` et `m2 = pile[-1]`, vérifient `len(m1)` $\geqslant 2 \cdot$ `len(m2)`.

3. 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]]`

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

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

6. 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`).

7. $(*)$ 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`.

8. $(*)$ En déduire la complexité en temps de cet algorithme dans le pire cas.