LYCÉE VAUGELAS 2021/2022	
MPSI	Informatique
Corrigé du TD Semaine 13	13 – 14/12/2021

Exercice 1

1. La fonction `tri_fusion` intègre deux fonctions locales : la fonction `fusion` réalise la fusion de deux tableaux triés et la fonction locale récursive `tri` se charge du partage en deux parties d'un tableau, puis s'appelle récursivement sur les deux moitiés, puis appelle la fonction de fusion, tant que le tableau contient au moins deux éléments. La première ligne de la fonction `fusion` sauvegarde la partie gauche de tableau en cours de traitement afin de pouvoir copier dans le tableau d'origine `t` le résultat du tri. La fonction `tri` est initialement appelée sur l'ensemble du tableau `t`, des indices 0 à $n-1$, où n est la taille de `t`. Le tableau est trié en place et la fonction ne retourne donc rien (`None` par défaut). Le tableau auxiliaire `tmp` est déclaré de même taille que `t` et initialisé avec `None` pour rappeler qu'une fonction de tri est polymorphe (on ne préjuge pas du type des éléments triés) ; on pourrait aussi réaliser une copie du tableau à trier. On tire partie du fait que la portée (visibilité) des tableaux `t` et `tmp` s'étend aux fonctions locales pour ne pas retransmettre les références de ces tableaux en argument. Les seuls arguments transmis sont les indices (entiers) des limites des parties traitées. **La récursivité n'est pas terminale.**
2. Algorithme de fusion de deux tableaux triés :
   - S'il reste des éléments dans les deux tableaux à fusionner, on extrait le plus petit des premiers éléments qui n'ont pas encore été extraits.
   - Si l'un des tableaux est épuisé, on finit de recopier l'autre

   On ne réalise la comparaison que si l'on n'a pas atteint la fin d'un tableau.

Fusion de deux tableaux triés

```
Entrée : indices g, m, d indiquant les tranches de t à fusionner
         de g à m d'une part et de m+1 à d d'autre part
Sortie : rien, le tableau t est mis à jour en place entre g et d
fonction fusion(g, m, d)
    Copie de la tranche de g à d de t dans tmp
    i <- g
    j <- m + 1
    k <- g
    Tant que l'on n'a pas atteint la fin d'un tableau
        Si tmp[i] <= tmp[j]
            t[k] <- tmp[i]
            incrémenter i
        Sinon
            t[k] <- tmp[j]
            incrémenter j
        FinSi
        k <- k + 1
    FinTantque
    Insérer les éléments restant de la tranche non épuisée dans t
```

Remarque 1 : on peut facilement programmer la fonction `fusion` de manière récursive puisque l'on prend toujours le premier élément restant de chaque liste. Il n'y a cependant aucun intérêt à le faire si l'on tri un tableau, qui est une structure de données de nature impérative. En revanche, pour trier une liste chaînée et dans un contexte fonctionnel, l'idée serait à retenir.

Remarque 2 : il est relativement efficace de programmer la fonction `fusion` à l'aide de trois boucles `while` successives, en s'appuyant sur la logique suivante :

— Tant qu'aucun des tableaux n'est épuisé, prendre le plus petit élément
— Tant que le premier tableau n'est pas épuisé, piocher dans ce tableau
— Tant que le second tableau n'est pas épuisé, piocher dans ce tableau

On supprime ainsi toute comparaison d'éléments du tableau (qui peut être l'opération la plus couteuse pour des données complexes) dès qu'un tableau est épuisé. On retient une autre approche dans la suite pour privilégier la brièveté du code, sans nuire à la complexité.

3. On complète le code de l'énoncé de manière à placer dans `t` les valeurs triées. L'algorithme est astucieusement formulé avec une boucle `for` en s'appuyant sur l'évaluation paresseuse des opérateurs booléens et leur préséance (conjonction logique prioritaire sur disjonction) :

```
def fusion(g, m, d):
    tmp[g : m + 1] = t[g : m + 1]
    i, j = g, m + 1
    for k in range(g, d + 1):
        if j > d or i <= m and tmp[i] <= t[j]:
            t[k] = tmp[i]
            i = i + 1
        else:
            t[k] = t[j]
            j = j + 1
```

Remarque 1 : on pourrait davantage optimiser la fonction `fusion` comme indiqué dans la remarque 2 de la question précédente.

Remarque 2 : on peut remplacer la condition booléenne du `if` par

```
i <= m and (j > d or tmp[i] <= t[j])
```

Une astuce consiste à copier dans le tableau auxiliaire `tmp` les valeurs du second tableau à fusionner en ordre inverse. Ainsi lorsqu'un tableau est épuisé, l'indice `i`, ou l'indice `j`, peut continuer à évoluer à l'intérieur de l'autre partie (faire un schéma pour bien comprendre). Dans la fonction suivante, l'indice `j` progresse en décroissant et on accepte d'allouer de nouveau le tableau temporaire :

```
def tri_fusion(t):
    def fusion(g, m, d):
        tmp = t[g : m + 1] + t[d : m : -1]
        i, j = 0, d - g
        for k in range(g, d + 1):
            if tmp[i] <= tmp[j]:
                t[k] = tmp[i] ; i = i + 1
            else:
                t[k] = tmp[j] ; j = j - 1
    def tri(g, d):
        if g < d:
            m = (g + d) // 2
            tri(g, m) ; tri(m + 1, d)
            fusion(g, m, d)
    tri(0, len(t) - 1)
```

L'astuce permet de ne jamais tester les valeurs des indices `i` et `j`, car on ne tente jamais de lire une case hors des parties à fusionner. On réduit ainsi le travail dans la partie de l'algorithme la plus couteuse en temps ; cependant il est inutile de continuer à comparer les éléments lorsque le croisement s'est produit et **si la comparaison des éléments du tableau est couteuse, c'est à la remarque 2 de la question 2 qu'il faut revenir pour optimiser l'implémentation** plutôt que de chercher à supprimer les tests sur les indices entiers, qui sont peu coûteux.

4. Le tri fusion est **stable**, c'est l'une de ses nombreuses qualités. Il suffit de remarquer que l'on extrait en priorité la valeur de la partie la plus à gauche du tableau d'origine en cas de valeurs équivalentes pour la comparaison.

5. **Invariant de fin de boucle** `for` : « `t[g:k+1]` contient, en ordre croissant, les $k-g+1$ plus petits éléments des deux tableaux à fusionner ». En sortie, cela démontre que `t[g:d+1]` contient les deux parties fusionnées dans l'ordre, c'est ce que l'on demande à la fonction `fusion`. La démonstration de l'invariant est immédiate : il est initialement vérifié avant l'entrée dans la boucle et chaque itération prélève le plus petit élément restant dans les deux tableaux à fusionner, évidemment supérieur à ceux que l'on a déjà prélevés, sinon il serait déjà parti !

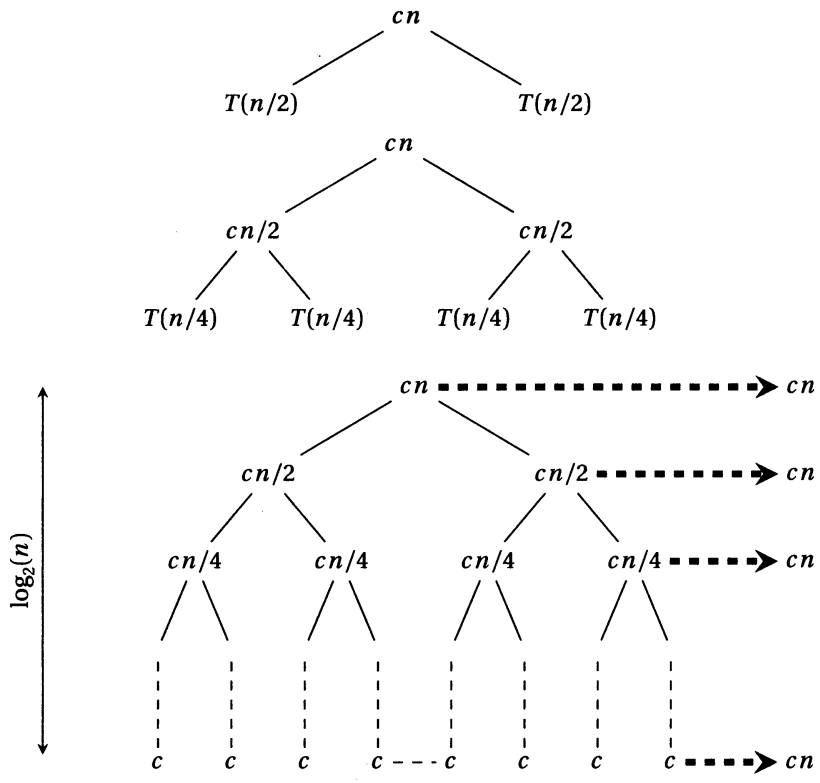
   Notons que la correction de la fonction `tri` se démontre simplement en remarquant que l'on réalise un nombre fini d'appels sur des parties du tableau dont les longueurs décroissent strictement.

6. **La démonstration demandée dans cette question est un grand classique de calcul de complexité d'une fonction récursive. La méthode, ici rappelée dans l'énoncé, est à bien connaître.**

   • Dans le cas où $n = 2^k$ avec $k \in \mathbb{N}$, il vient $T(2^k) = 2\,T(2^{k-1}) + c\,2^k$. On pose $u_k = \frac{T(2^k)}{2^k}$, par conséquent $u_k = u_{k-1} + c$. Cette suite arithmétique se résout en $u_k = u(0) + kc = T(1) + kc$. Donc $T(2^k) = 2^k(T(1) + kc)$ et comme $k = \log_2 n$:

$$T(n) = n(T(1) + c\,\log_2 n) = \Theta(n\,\ln n) \tag{1}$$

   On peut appréhender cette démonstration de manière visuelle pour mieux la retenir en représentant l'arbre des appels récursifs. On représente les deux premiers niveaux puis l'arbre complet :



   On constate donc que si $n = 2^k$, l'arbre possède $1 + \log_2 n$ niveaux et tous ajoutent cn, donc $T(n) = (1 + \log_2 n)cn$, qui est bien le résultat précédent avec $T(1) = c$.

   • On traite le cas d'un n quelconque en posant $k = \lceil \log_2 n \rceil$. On a alors $k - 1 < \log_2 n \leqslant k$, donc $2^{k-1} < n \leqslant 2^k$. Comme la fonction $T(n)$ est croissante, on en déduit

$$T(2^{k-1}) < T(n) \leqslant T(2^k) \tag{2}$$

   D'après le point précédent $T(2^k) = 2^k(T(1)+kc)$. En outre $\log_2 n \leqslant k < \log_2 n+1$ et $n \leqslant 2^k < 2n$. Donc

$$\frac{n}{2}(T(1) + (\log_2 n - 1)c) < T(n) < 2n(T(1) + (\log_2 n + 1)c) \tag{3}$$

De plus $T(1) = \Theta(1)$. Il existe donc des constantes positives c_1 et c_2 et $N \in \mathbb{N}$ telles que, si $n > N$:

$$c_1\, n \log_2 n < T(n) < c_2\, n \log_2 n \tag{4}$$

Par exemple $c_1 = c/3$ et $c_2 = 3c$. Donc $T(n) = \Theta(n \ln n)$, **par définition de l'ordre de grandeur asymptotique.**

7. On applique directement le résultat de la question précédente au nombre de comparaisons, même si, en toute rigueur il n'y en a que $n-1$, avec $c = 1$. On trouve donc un nombre de comparaisons en $\Theta(n \ln n)$. La constante est difficile à calculer pour n quelconque et dépend de l'implémentation : la remarque 2 de la question 2 divise le nombre de comparaisons par 2 dans le meilleur des cas (tableau déjà trié). Notons que la variante de la remarque 2 de la question 3 peut réduire le nombre de tests également pour un tableau déjà trié puisque la condition `i <= m` n'est plus satisfaite dès que la première partie est épuisée.
8. Le nombre d'écritures suit la même récurrence, il est donc en $\Theta(n \ln n)$. On peut même suspecter un équivalent $T(n) \sim 2\, n \log_2 n$ dans tous les cas (du meilleur au pire) puisque chaque élément est systématiquement copié deux fois : une fois dans le tableau auxiliaire et une fois recopié dans le tableau `t`.
9. D'après les deux questions précédentes, la complexité temporelle du tri fusion est en $\Theta(n \ln n)$.
10. La complexité spatiale est en $\Theta(n)$, d'une part à cause du tableau auxiliaire mais aussi pour la gestion de la pile des appels récursifs. En effet le coût spatial de la pile vérifie une récurrence du type

$$S(n) = 2\, S\left(\frac{n}{2}\right) + \Theta(1) \tag{5}$$

Si $n = 2^k$, on pose $u_k = S(2^k)$, d'où $u_k = 2\, u_{k-1} + \Theta(1)$, donc $u_k = \Theta(2^k)$ et $S(n) = \Theta(n)$.

11. Voici une **implémentation impérative du tri fusion** à laquelle on pourra réfléchir.

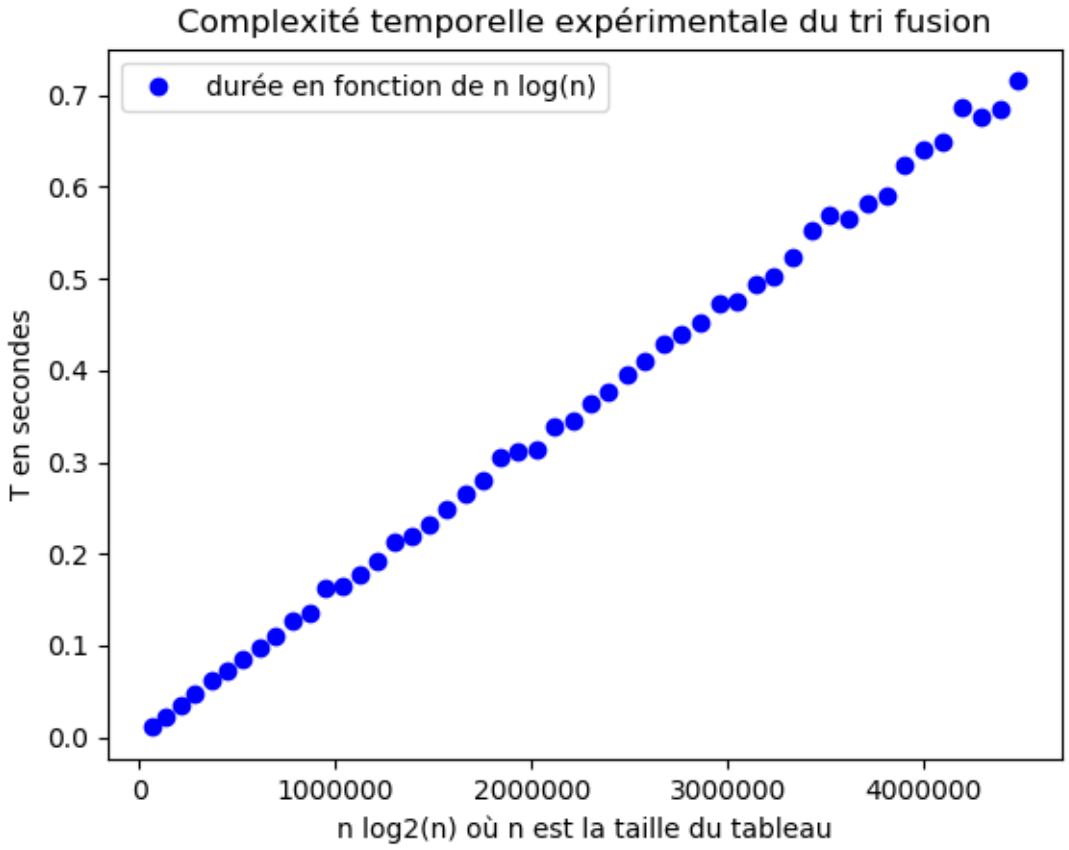
```
def tri_fusion(t):
    n, h = len(t), 1
    tmp = [None] * n
    while h < n:
        for g in range(0, n - 1, h + h):
            m = g + h
            d = min(m + h, n)
            tmp[g : d] = t[g : d]
            i, j = g, m
            for k in range(g, d):
                if j >= d or i < m and tmp[i] <= tmp[j]:
                    t[k] = tmp[i] ; i += 1
                else:
                    t[k] = tmp[j] ; j += 1
        h += h
```

Dans ce programme, h représente la largeur des deux parties à fusionner (de g à m, puis de $m+1$ à $d-1$). On commence par trier les sous-tableaux de longueur 2, puis de longueur 4, etc.

Le nombre d'itérations de la boucle while est en $\log_2 n$ puisque l'on double h à chaque itération. Contrairement à la version récursive, la fonction de fusion peut être appelée sur deux tableaux de tailles très différentes car la taille du tableau peut-être très éloignée d'une puissance de 2.

Temps d'exécution expérimental du tri fusion

```
from time import perf_counter
from random import randint
import matplotlib.pyplot as plt
from numpy import log2
nb = 50
ni = 5000
N = ni
res = []
for _ in range(nb):
    L = [randint(0,100000) for _ in range(N)]
    t = perf_counter()
    tri_fusion(L)
    t = perf_counter() - t
    res.append(t)
    N += ni
n = [i * ni for i in range(1, nb + 1)]
plt.plot(n*log2(n),res, 'bo', label='durée en fonction de n log(n)')
plt.title('Complexité temporelle expérimentale du tri fusion')
plt.xlabel('n log2(n) où n est la taille du tableau')
plt.ylabel('T en secondes')
plt.legend()
plt.show()
```



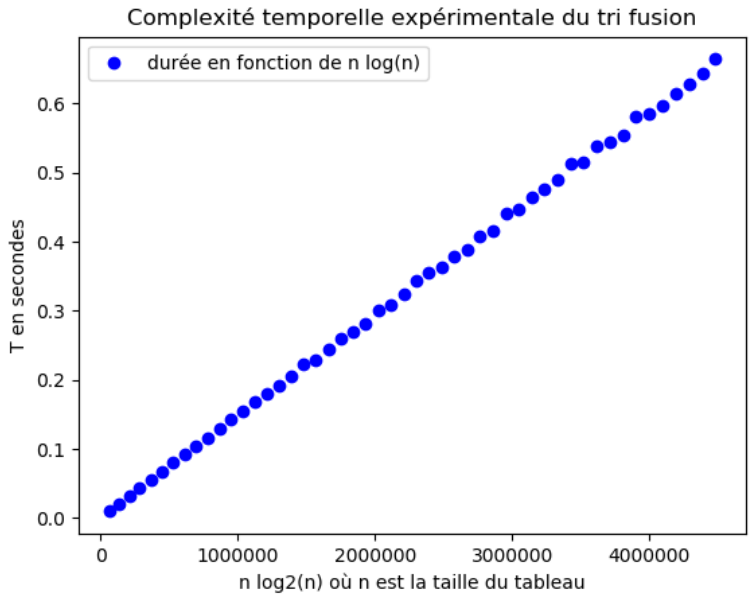
Autres versions du tri fusion

Cette version ne modifie pas le tableau transmis et retourne un nouveau tableau trié

```
def tri(L):
    n = len(L)
    if n == 1: return L
    n1 = n >> 1
    L1, L2 = tri(L[:n1]), tri(L[n1:])
    i, j, n2 = 0, 0, n - n1
    M = [None] * n
    for k in range(n):
        if j >= n2 or i < n1 and L1[i] <= L2[j]:
            M[k] = L1[i] ; i += 1
        else:
            M[k] = L2[j] ; j += 1
    return M
```

Version Python synthétique du tri fusion, rapide et lisible. Le tableau est modifié en place et même retourné en bonus ! La simplicité provient du parcours en sens inverses des deux tableaux à fusionner, qui évite de tester si l'un est épuisé : on attend que les indices se rejoignent, ce que l'on ne teste même pas, grâce à la boucle for qui réalise la fusion.

```
def tri(t):
    n = len(t)
    if n > 1:
        i, j, m = 0, n - 1, n // 2
        z = tri(t[:m]) + tri(t[m:])[::-1]
        for k in range(n):
            if z[i] <= z[j]:
                t[k] = z[i] ; i += 1
            else:
                t[k] = z[j] ; j -= 1
    return t
```



Fin