TP Algorithmes de tri corrigé

[4]:
```python
# 1.
def croissant(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True

croissant([1,2,3,4,5]) and not croissant([1,2,4,3,5]) # test
```

[4]: True

[10]:
```python
# 2.
def appartient(e, L, i):
    if i == len(L):
        return False
    return L[i] == e or appartient(e, L, i+1)

appartient(1, [1, 2, 3], 0) and not appartient(1, [1, 2, 3], 1) # test
```

[10]: True

[11]:
```python
# 3.
def doublon(L):
    for i in range(len(L)):
        if appartient(L[i], L, i+1):
            return True
    return False

not doublon([1, 2, 3, 4, 5]) and doublon([1, 2, 3, 4, 4]) # test
```

[11]: True

[7]:
```python
# 4. Voir cours : complexité O(nlog(n))
def fusion(L1, L2, L):
    i1, i2 = 0, 0
    while i1 + i2 < len(L):
        if i1 >= len(L1):
            L[i1 + i2] = L2[i2]
            i2 = i2 + 1
        elif i2 >= len(L2):
            L[i1 + i2] = L1[i1]
            i1 = i1 + 1
        elif L[i1] < L[i2]:
            L[i1 + i2] = L1[i1]
            i1 = i1 + 1
        else:
            L[i1 + i2] = L2[i2]
            i2 = i2 + 1
```

```python
def fusion(L1, L2):
    if len(L1) == 0: return L2
    if len(L2) == 0: return L1
    if L1[-1] > L2[-1]: m = L1.pop()
    else: m = L2.pop()
    L = fusion(L1, L2)
    L.append(m)
    return L

def tri_fusion(L):
    if len(L) <= 1: return L
    L1, L2 = L[: len(L)//2], L[len(L)//2 :]
    return fusion(tri_fusion(L1), tri_fusion(L2))

L = [5, 1, 3, 8, 2, 4, 9, 7, 6]
tri_fusion(L) == [1, 2, 3, 4, 5, 6, 7, 8, 9] # test
```

[7]: True

[3]:

```python
# 5.
def doublon_triee(L):
    for i in range(len(L) - 1):
        if L[i] == L[i+1]:  # O(len(L))
            return True
    return False

not doublon_triee([1, 2, 3, 4, 5]) and doublon_triee([1, 2, 3, 4, 4]) # test
```

[3]: True

[5]:

```python
# 6.
def doublon2(L):
    return doublon_triee(tri_fusion(L))

not doublon2([1, 2, 3, 4, 5]) and doublon2([1, 2, 3, 4, 4]) # test
```

[5]: True

Comparons le temps d'exécution de doublon et doublon2 :

[20]:

```python
%%timeit
import sys
sys.setrecursionlimit(10000)
L = list(range(5000))
doublon(L)
```

2.24 s ± 76.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

[21]:

```python
%%timeit
L = list(range(5000))
doublon2(L)
```

10.2 ms ± 274 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

[2]:
```python
# 7

def frequent(L):
    max_freq = 0
    max_elem = 0 # élément dont la fréquence est max_freq
    for i in range(len(L)):
        freq = 0
        for j in range(len(L)):
            if L[i] == L[j]:
                freq = freq + 1
        if freq > max_freq:
            max_freq = freq
            max_elem = L[i]
    return max_elem
# à cause des deux boucles for imbriquées, frequent(L) est en complexité O(n)*O(n) =␣
↪O(n^2), où n = len(L

frequent([3, 1, 3, 7, 8, 7, 3])
```

[2]: 3

[4]:
```python
# 8

def frequent_triee(L):
    cur_freq = 0
    max_freq = 0
    max_elem = 0
    for i in range(len(L)): # O(n)
        if L[i] == L[i-1]:
            cur_freq = cur_freq + 1
        else:
            cur_freq = 1
        if cur_freq > max_freq:
            max_freq = cur_freq
            max_elem = L[i]
    return max_elem

frequent([1, 3, 3, 3, 7, 7, 8])
```

[4]: 3

[5]:
```python
# 9.
def frequent(L):
    return frequent_triee(tri_fusion(L))
# frequent_triee est en complexité O(n), tri_fusion est en O(nlog(n)), donc frequent␣
↪est en complexité O(nlog(n)) + O(n) = O(nlog(n))
```

[13]:
```python
# 10.
# Pour obtenir une complexité O(nlog k), on peut :
# fusionner les listes 2 par 2 pour obtenir k/2 listes
# fusionner ces listes 2 par 2 pour obtenir k/4 listes
# ...
# jusqu'à avoir qu'une seule liste qui contient tous les éléments
```

```
# Ce processus prend log2(k) étapes (car au bout de log2(k) étapes il reste k/
↪2**log2(k)=1 liste
# Donc la complexité est O(nlog k)

def fusion_all(L):
    while len(L) > 1:
        L_ = []
        for i in range(0, len(L)-1, 2):
            L_.append(fusion(L[i], L[i+1]))
        if len(L) % 2 == 1:
            L_.append(L[-1])
        L = L_
    return L[0]

fusion_all([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

[13]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

I Exercice supplémentaire

[20]:
```
def somme2(L, p):
    i, j = 0, len(L) - 1
    while i < j and L[i] + L[j] != p:
        # Invariant de boucle : si il existe a, b tel que L[a] + L[b] = p, alors i␣
↪<= a <= b <= j
        if L[i] + L[j] < p:
            i = i + 1
        else:
            j = j - 1
    if L[i] + L[j] == p:
        return i, j
    return -1, -1

print(somme2([1, 2, 3, 6], 8))
print(somme2([1, 2, 3, 6], 10))

# À chaque passage dans la boucle while, j - i diminue de 1.
# Donc au bout de len(L) itérations, i = j et la boucle s'arrête
# Donc la complexité est O(len(L))
```

```
(1, 3)
(-1, -1)
```