Corrigé

Algorithmes de tris

Exercice 1

```
def minimum(t, j):
    i, m = j, t[j]
    for k in range(j+1, len(t)):
        if t[k] < m:
            i, m = k, t[k]
    return i

def selectionSort(t):
    for j in range(len(t)-1):
        i = minimum(t, j)
        if i != j:
            t[i], t[j] = t[j], t[i]
```

Exercice 2

```
def insere(t, j):
    k, a = j, t[j]
    while k > 0 and a < t[k-1]:
        t[k] = t[k-1]
        k -= 1
    t[k] = a

def insertionSort(t):
    for j in range(1, len(t)):
        insere(t, j)
```

Exercice 3

```
def fusion(t1, t2):
    s = []
    i, j = 0, 0
    while i < len(t1) or j < len(t2):
        if i == len(t1) or (j < len(t2) and t2[j] < t1[i]):
            s.append(t2[j])
            j += 1
        else:
            s.append(t1[i])
            i += 1
    return s

def mergeSort(t):
    n = len(t)
    if n < 2:
        return t.copy()
    t1 = mergeSort(t[:n//2])
    t2 = mergeSort(t[n//2:])
    return fusion(t1, t2)
```

Exercice 4

```
def segmente(t, i, j):
    p = t[j-1]
    a = i
    for b in range(i, j-1):
        if t[b] < p:
            t[a], t[b] = t[b], t[a]
            a += 1
    t[a], t[j-1] = t[j-1], t[a]
    return a

def quickSort(t, i, j):
    if i + 1 < j:
        a = segmente(t, i, j)
        quickSort(t, i, a)
        quickSort(t, a + 1, j)
```

Exercice 5

a) On rédige une fonction qui détermine si l'élément d'indice i du tableau est présent ailleurs dans ce dernier :

```
def estPresent(i, t):
    for j in range(len(t)):
        if j != i and t[j] == t[i]:
            return True
    return False
```

puis la fonction qui recherche d'éventuels doublons :

```
def doublon(t):
    for i in range(len(t)):
        if estPresent(i, t):
            return True
    return False
```

Si t est un tableau de taille n, la fonction `estPresent` est de complexité linéaire $O(n)$ donc la fonction `doublon` est de complexité quadratique $O(n^2)$.

b) Si le tableau est trié, les doublons éventuels sont placés consécutivement dans le tableau, d'où la version :

```
def doublonTrie(t):
    for i in range(1, len(t)):
        if t[i-1] == t[i]:
            return True
    return False
```

Cette fonction est de complexité linéaire $O(n)$ donc il est intéressant de commencer par trier le tableau, à condition d'utiliser pour ce faire un algorithme de tri de complexité optimale $O(n \log n)$.

Exercice 6 Lorsqu'on parcourt le tableau (de la gauche vers la droite) en permutant deux éléments consécutifs à chaque fois que l'élément le plus petit se trouve à droite du plus grand, on est assuré en fin de parcours d'avoir placé le plus grand élément du tableau à sa place définitive, en ayant effectué $n-1$ comparaisons et au plus $n-1$ permutations.
Il reste à réitérer le procédé sur le tableau privé de sa dernière case pour obtenir l'algorithme de tri bulle, ainsi nommé car les éléments les plus grands du tableau se dirigent vers leur place définitive à l'image des bulles d'air qui remontent à la surface d'un liquide.
On rédige une fonction de « remontée » des bulles (jusqu'au niveau k) :

```
def monte(t, j):
    for k in range(j-1):
        if t[k] > t[k+1]:
            t[k], t[k+1] = t[k+1], t[k]
```

puis la fonction de tri proprement dit :

```python
def bubbleSort(t):
    for j in range(len(t), 0, -1):
        monte(t, j)
```

Notons c_n le nombre de comparaisons effectuées, et p_n le nombre de permutations de deux éléments. Nous avons $c_n = n-1+c_{n-1}$ et $p_{n-1} \leqslant p_n \leqslant p_{n-1}+n-1$, donc $c_n = \frac{n(n-1)}{2}$ et $0 \leqslant p_n \leqslant \frac{n(n-1)}{2}$.
Il s'agit donc d'un algorithme de coût quadratique, le pire des cas ayant lieu lorsque le tableau est trié à l'envers, car alors $c_n = p_n = \frac{n(n-1)}{2}$.

Remarque. Il est possible d'améliorer légèrement cet algorithme en observant que si lors d'une étape de remontée aucune permutation n'est effectuée, c'est que le tableau est trié, et qu'on peut donc s'arrêter là.
Pour exploiter cette remarque on peut par exemple faire en sorte que la fonction `remonte` renvoie une valeur booléenne indiquant si aucune permutation n'a été réalisée. Ceci conduit à cette seconde version :

```python
def monte(t, j):
    b = False
    for k in range(j-1):
        if t[k] > t[k+1]:
            t[k], t[k+1] = t[k+1], t[k]
            b = True
    return b

def bubbleSort(t):
    b, j = True, len(t)
    while b and j > 0:
        b = monte(t, j)
        j -= 1
```

Avec cette nouvelle version, le nombre d'échanges est inchangé mais le nombre de comparaisons n'est plus systématiquement quadratique. Dans le cas d'un tableau déjà trié, par exemple, cet algorithme n'effectue que $n-1$ comparaisons et aucune permutation.

Exercice 7 Pour ce tri, nous avons besoin d'une fonction qui déplace les éléments les plus gros de la gauche vers la droite et d'une fonction qui déplace les éléments les plus petits de la droite vers la gauche. Ces deux fonctions doivent agir sur une coupe $t[i : j+1]$ du tableau :

```python
def gaucheDroite(t, i, j):
    if i < j:
        if t[i+1] < t[i]:
            t[i], t[i+1] = t[i+1], t[i]
        gaucheDroite(t, i+1, j)

def droiteGauche(t, i, j):
    if i < j:
        if t[j-1] > t[j]:
            t[j-1], t[j] = t[j], t[j-1]
        droiteGauche(t, i, j-1)
```

La fonction principale consiste alors à alterner les déplacements de la gauche vers la droite et de la droite vers la gauche :

```python
def cocktailSort(t):
    i, j = 0, len(t)-1
    while i < j:
        gaucheDroite(t, i, j)
        j -= 1
        droiteGauche(t, i, j)
        i += 1
```

Exercice 8

a) On applique le tri par insertion à chacun des sous-tableaux $t[:: h]$:

```
def insert(t, h):
    for j in range(h, len(t)):
        k = j
        while k >= h and t[k] < t[k-h]:
            t[k-h], t[k] = t[k], t[k-h]
            k -= 1
```

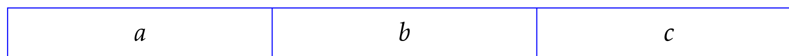
b) Le tri proprement dit consiste essentiellement à calculer la plus grande valeur h_p de la suite qui vérifie $h_p \leqslant n < h_{p+1}$ puis à appliquer la fonction précédente avec $h_p, h_{p-1}, \ldots, h_1 = 1$.

```
def shellSort(t):
    h = 1
    while 2 * h + 1 < len(t):
        h = 2 * h + 1
    while h > 0:
        insertion(t, h)
        h = (h - 1) // 2
```

Exercice 9

a) Montrons par récurrence forte sur $n = j - i \geqslant 2$ que `stoogeSort(t, i, j)` trie correctement le tableau $t[i:j]$.

- Si $n = 2$ l'algorithme réalise au plus une permutation pour trier le tableau à deux cases et ne fait pas d'appel récursif.
- Si $n \geqslant 3$ on suppose le résultat acquis jusqu'au rang $n - 1$.
  L'algorithme scinde la tableau en trois parties : $a = t[i : i + k]$, $b = t[i + k : j - k]$, $c = t[j - k : j]$.

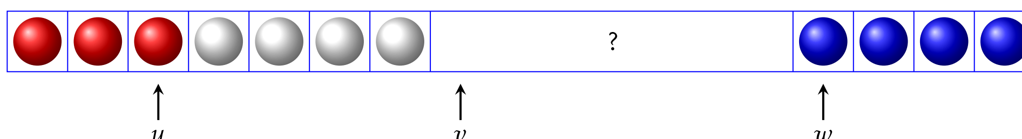
  

  On a $|a| = |c| = k$ et $|b| = j - i - 2k$ avec $k = \left\lfloor \frac{j-i}{3} \right\rfloor$ donc $|b| \geqslant |a| = |c|$.
  Le tableau $a \cup b$ est tout d'abord trié ; ceci assure que dans b se trouvent désormais (au moins) les k plus grands éléments de $a + b$;
  Le tableau $b \cup c$ est ensuite trié ; ceci assure que les k plus grands éléments de t se trouvent maintenant triés dans c.
  Le dernier appel récursif trie enfin les $j - i - k$ derniers éléments dans $a \cup b$, ce qui assure qu'après ces trois appels récursifs le tableau t est trié.

b) Si $\mathrm{C}(n)$ désigne le nombre de comparaison réalisées par cet algorithme lorsque $n = |t|$, on dispose de la relation : $\mathrm{C}(n) = 3\mathrm{C}(n - k) + 1$ avec $k = \lfloor n/3 \rfloor$, soit $\mathrm{C}(n) = 3\mathrm{C}(\lceil 2n/3 \rceil) + 1$.

S'il existe α tel que $\mathrm{C}(n) = \mathrm{O}(n^\alpha)$ alors $n^\alpha \sim 3\left(\frac{2n}{3}\right)^\alpha$ soit $1 = 3\left(\frac{2}{3}\right)^\alpha$, ce qui conduit à $\alpha = \frac{\ln 3}{\ln 3 - \ln 2} \approx 2{,}71$. Cet algorithme a un coût plus que quadratique, il est de complexité plus médiocre que les algorithmes naïfs (tri par insertion ou tri bulle).

Exercice 10 Le principe est semblable au principe de segmentation du tri rapide : on parcours le tableau en ayant pour invariants les entiers u, v, w définis par le shéma ci-dessous :



Tant que $v < w$, on regarde la couleur de $t[v]$:

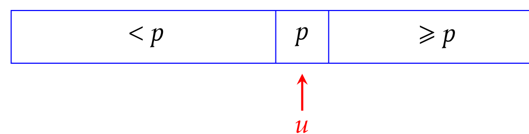
- si celle-ci est rouge, on permute cet élément avec celui d'indice $u + 1$, et on incrémente u et v ;
- si celle-ci est blanche, on incrémente v ;
- si celle-ci est rouge, on permute cet élément avec celui d'indice $w - 1$, et on décrémente w.

Traduit en Python, cela donne :

```python
def drapeau(t):
    u, v, w = -1, 0, len(t)
    while v < w:
        c = t[v].couleur
        if c == 'rouge':
            t[u+1], t[v] = t[v], t[u+1]
            u += 1
            v += 1
        elif c == 'bleu':
            t[v], t[w-1] = t[w-1], t[v]
            w -= 1
        else:
            v += 1
```

La terminaison de cette fonction est assurée par le fait que la valeur de $w-v$ décroit d'une unité à chaque itération.

Exercice 11 Observons la situation après la segmentation par un pivot p :



– Si $k \leqslant u$, le k^e plus petit élément du tableau est à chercher entre les indices 0 et $u-1$;
– si $k = u+1$, le k^e plus petit élément du tableau est l'élément p qui a été pris pour pivot ;
– si $k > u+1$, le k^e plus petit élément du tableau est aussi le $(k-u-1)^e$ élément du sous-tableau délimité par les indices $u+1$ et $n-1$.

Ceci conduit à la fonction suivante (utilisant la fonction segmente du cours) :

```python
def kieme(t, k, *args):
    if len(args) == 0:
        i, j = 0, len(t)
    else:
        i, j = args
    u = segmente(t, i, j)
    if k <= u - i:
        return kieme(t, k, i, u)
    elif k > u - i + 1:
        return kieme(t, k-u+i-1, u+1, j)
    else:
        return t[u]
```

Le calcul de l'élément médian consiste à appliquer la fonction précédente avec $k = \lceil n/2 \rceil$:

```python
def median(t):
    k = (len(t) + 1) // 2
    return kieme(t, k)
```

Après avoir trié un tableau on peut déterminer en temps linéaire l'élément médian de ce dernier ; l'algorithme de tri rapide permet donc d'avoir un algorithme de coût quasi-linéaire en moyenne (et quadratique dans le pire des cas) pour calculer le médian. L'algorithme ci-dessus ne présente donc d'intérêt que s'il permet de faire mieux.
Notons $C(n)$ le coût *en moyenne* du calcul du k^e élément de t. Le coût de la segmentation étant linéaire, on dispose de la relation :

$$C(n) = \frac{1}{n}\sum_{i=1}^{n} \max\Big(C(i-1), C(n-i)\Big) + O(n).$$

Supposons l'existence d'une constante M telle que $C(i) \leqslant Mi$ pour $i < n$. Alors :

$$C(n) \leqslant \frac{M}{n}\sum_{i=1}^{n} \max(i-1, n-i) + O(n) \leqslant (\text{calcul}) \leqslant \frac{3}{4}Mn + O(n).$$

Il suffit donc de choisir M assez grand pour que $C(n) \leqslant Mn$, ce qui prouve que $C(n) = O(n)$. En moyenne, cet algorithme calcule l'élément médian en temps linéaire.

Remarque. En choisissant adroitement le pivot dans cet algorithme, il est possible de modifier cet algorithme pour obtenir un algorithme de coût linéaire dans le pire des cas.