Corrigé

Tableaux

Exercice 1 Le principe consiste à itérer deux suites (m_k) et (s_k) telles que m_k et s_k représentent pour tout $k \geqslant 1$ respectivement le plus grand et le deuxième plus grand élément du tableau $(t_0, \ldots, t_k)$.

```
def secondMaximum(t):
    if t[0] < t[1]:
        m, s = t[1], t[0]
    else:
        m, s = t[0], t[1]
    for i in range(2, len(t)):
        if t[i] > m:
            m, s = t[i], m
        elif t[i] > s:
            s = t[i]
    return s
```

Exercice 2 Plusieurs solutions sont possibles. On peut par exemple chercher le premier élément, puis le second comme dans le code ci-dessous.

```
def chercheSecond(prop, t):
    for i in range(len(t)):
        if prop(t[i]):
            for j in range(i+1, len(t)):
                if prop(t[j]):
                    return t[j]
            break
```

Exercice 3

a)

```
def egal(i, m, s):
    if i + len(m) > len(s):
        return False
    for j in range(len(m)):
        if m[j] != s[i+j]:
            return False
    return True
```

b)

```
def sousMot(m, s):
    for i in range(len(s)-len(m)+1):
        if egal(i, m, s):
            return True
    return False
```

c) La fonction `egal` réalise dans le pire des cas k comparaisons, donc la complexité temporelle est un $O(k)$. La fonction `sousMot` fait appel $n-k+1$ fois à la fonction `egal` donc sa complexité temporelle est un $O(k(n-k))$. Si on suppose $k \leqslant n$ la quantité $k(n-k)$ est maximale pour $k = n/2$ donc la complexité dans le pire des cas est un $O(n^2)$.

Exercice 4

a) On crée un nouveau tableau de même taille avant de le remplir.

```
def rotation(t):
    r = [None for i in range(len(t))]
    for i in range(1, len(t)):
        r[i] = t[i-1]
    r[0] = t[len(t)-1]
    return r
```

b) Ici par de `return` : on procède à une mutation de l'argument.

```
def rotation2(t):
    a = t[len(t)-1]
    for i in reversed(range(1, len(t))):
        t[i] = t[i-1]
    t[0] = a
```

c) Utiliser k fois la fonction `rotation` serait maladroit. Il vaut mieux calculer l'emplacement où doivent se trouver chacune des valeurs contenues dans t : l'élément de rang i doit être déplacé dans la case de rang $(i+k) \bmod n$.

```
def rotation3(k, t):
    r = [None for i in range(len(t))]
    for i in range(len(t)):
        r[(i+k) % len(t)] = t[i]
    return r
```

Exercice 5 Le plus petit entier manquant dans un tableau de longueur n est inférieur ou égal à n ; il suffit donc de créer un tableau de longueur $n+1$ chargé de marquer parmi les entiers de $[\![0, n]\!]$ ceux qui sont présents dans t, puis d'en prendre le plus petit.

```
def entierManquant(t):
    n = len(t)
    r = [False for i in range(n+1)]
    for i in t:
        if i <= n:
            r[i] = True
    for i in range(n+1):
        if not r[i]:
            return i
```

Exercice 6

a) On définit successivement :

```
def delta(k, t):
    s = 0
    for i in range(k+1):
        s += t[i]
    for i in range(k+1, len(t)):
        s -= t[i]
    return s

def equilibre(t):
    m = delta(0, t)
    for k in range(1, len(t)-1):
        m = max(m, delta(k, t))
    return m
```

La fonction `delta` réalise n additions ou soustractions donc la complexité temporelle de cette fonction est en $O(n)$. La fonction `equilibre` fait appel $n-1$ fois à la fonction `delta` et $n-1$ fois à la fonction `max` donc sa complexité temporelle est en $nO(n) + O(n) = O(n^2)$.

b) On utilise la relation $\Delta_k = \Delta_{k-1} + 2t[k]$ pour calculer les différentes valeurs de Δ_k une fois Δ_0 calculé :

```
def equilibre2(t):
    d = delta(0, t)
    m = d
    for k in range(1, len(t)-1):
        d = d + 2 * t[k]
        m = max(m, d)
    return m
```

Cette fonction fait appel une fois à la fonction `delta` puis réalise $n-1$ additions, $n-1$ multiplications et $n-1$ appels à la fonction `max` donc sa complexité temporelle est en $O(n) + O(n) = O(n)$.

Exercice 7 On utilise un tableau de x cases modélisant les feuilles tombées sur la rivière, et on stoppe l'algorithme lorsque ce tableau est entièrement rempli.

```
def grenouille(x, t):
    r = [False for i in range(x+1)]  # au départ il n'y a aucune feuille
    s = 0                            # s est le nombre de cases couvertes par une feuille
    for k in range(len(t)):
        if not r[t[k]]:              # une feuille tombe sur une case vide
            r[t[k]] = True
            s += 1
        if s == x:
            return k
```

La création du tableau `r` a une complexité temporelle et spatiale en $O(x)$; la suite de l'algorithme a une complexité temporelle en $O(n)$ donc la complexité temporelle de cet algorithme est un $O(n+x)$ et sa complexité spatiale un $O(x)$.

Exercice 8

a) On prouve l'invariant suivant : « s'il existe un couple (k,ℓ) tel que $x = t[k,\ell]$ alors $k \leqslant i$ et $\ell \geqslant j$ ».

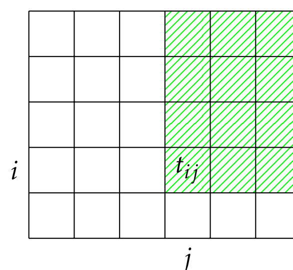

FIGURE 1 – La zone dans laquelle x peut se trouver.

L'invariant est évidemment vérifié pour les valeurs initiales $i = m-1$, $j = 0$.
Montrons maintenant qu'il reste vérifié durant tout l'algorithme. La figure 1 représente la zone dans laquelle x peut se trouver par hypothèse. Si $x < t[i,j]$ alors $x < t[i,k]$ pour tout $k \geqslant j$ donc on peut supprimer la dernière ligne de la zone de recherche, ce qui correspond à remplacer i par $i-1$. Et si $x > t[i,j]$ alors $x > t[l,j]$ pour tout $l \leqslant i$ donc on peut supprimer la première colonne de la zone de recherche, ce qui correspond à remplacer j par $j+1$.
L'invariant est donc bien maintenu durant tout l'algorithme, ce qui prouve la validité de ce dernier dans les deux cas de terminaison : lorsque x est trouvé ou lorsque la zone de recherche devient vide (lorsque $i < 0$ ou $j \geqslant n$).

b) Notons $C(n,p)$ la complexité temporelle de cette fonction. Lorsque x n'est pas présent dans le tableau on supprime à chaque étape une ligne ou une colonne. Les comparaisons entre x et $t[i,j]$ étant de complexité constante on a $C(n,p) = C(n-1,p) + O(1)$ ou $C(n,p) = C(n,p-1) + O(1)$.
Il existe donc une constante M telle que $C(n,p) \leqslant C(n-1,p) + M$ ou $C(n,p) \leqslant C(n,p-1) + M$, et dès lors il est facile de prouver par récurrence sur $n+p$ que $C(n,p) \leqslant (n+p)M$, soit $C(n,p) = O(n+p)$.