# Chapitre 2 : Correction des exercices

**Exercice 1** Pour les deux fonctions suivantes écrire la spécification précise, montrer la correction et déterminer la complexité dans le pire des cas.

```python
def fonction1(L:list) -> bool:
'''Renvoie True si la liste est triée par ordre croissante et False sinon
'''
i = 0
while i < len(L)-1 and L[i] <= L[i+1]:
    i = i+1
return i == len(L)-1
```

La quantité `len(L)-i` est un variant de boucle : c'est une quantité positive (et même $> 1$) et elle diminue de 1 à chaque itération. Donc la boucle termine.
La propriété « `L[0:i+1]` est triée »est un invariant de boucle :

- en entrée de boucle : on a bien `L[0:1]` est triée (il n'y a qu'un seul élément) ;
- si la propriété est vraie en début d'itération on a `L[0:i+1]` et si l'itération a lieu, alors `L[i] <= L[i+1]` donc la liste `L[0:i+2]` est triée.

On sort de la boucle à deux conditions :

- soit `L[i] > L[i+1]` pour un certain `i` tel que `i < len(L) - 1` : la liste n'est pas triée et la fonctoin renvoie `False` ;
- soit `i = len(L)-1` et la liste `L[0:len(L)]` est triée, et la fonction renvoie `True`.

Dans le pire des cas, la boucle est répétée `len(L)` fois, donc la complexité est linéaire en la taille de la liste.

```python
def fonction2(s:str, m:str) -> int:
    '''Renvoie le nombre de fois où la chaîne de car m
    apparaît dans s.
    '''
    r = 0
    for i in range(len(s)):
        if s[i:i+len(m)] == m:
            r = r + 1
    return r
```

L'algorithme termine de façon évidente. La propriété « `m` apparaît `r` fois entre les indices `0` et `i` (compris) »est un invariant de boucle :

- en entrée de boucle, `m` n'apparaît pas encore ;
- si la propriété est vraie avant l'itération `i`, alors `r` augmente de 1 si `m` apparaît en position `i`, donc la propriété reste vraie après l'itération `i`.

En sortie de boucle, `r` est donc égal au nombre de fois où `m` apparaît entre les indices `0` et `len(s)-1`.
La boucle est répétée `len(s)` fois, mais dans chaque itération, on a une comparaison entre deux chaînes de caractères des longueur `len(m)`. Donc la complexité est $O($`len(s)*len(m)`$)$. ■

**Exercice 2**

```python
def multiplication(x:int, y:int, c:int) -> int:
    m = 0
    while y != 0:
        m = m + x*(y%c)
        x = x*c
        y = y//c
    return m
```

On suppose que $c \geq 2$ et $y \geq 0$.

1. Montrer que cet algorithme termine.
   La quantité $y$ est un variant de boucle : en effet, $y \geq 0$ avant l'entrée dans la boucle et diminue strictement à chaque itération car $c \geq 2$. Ainsi, la boucle termine.
2. On note `a` et `b` les valeurs d'entrées pour `x` et `y`. Montrer que la propriété : « $ab = m + xy$ »est un invariant de boucle.
   - En entrée de boucle, on a $m = 0$ et $x = a$, $y = b$, donc la propriété est vérifiée.
   - Supposons qu'en entrée d'une itération, on a $ab = m + xy$. On note $m'$, $x'$ et $y'$ les valeurs de $m$, $x$ et $y$ en sortie de l'itération : $y = y'c + (y \mod c)$ et $x' = xc$, donc $ab = m + x(y'c + (y \mod c)) = m + x'y' + x(y \mod c) = m' + x'y'$. La propriété est vraie en sortie de boucle.
3. Montrer la correction de cet algorithme. Que se passe-t-il si $y < 0$ ?
   En sortie de boucle, $y = 0$, donc $ab = m$ et la fonction renvoie bien le produit voulu. Attention, si $y < 0$, la boucle ne termine pas car la division entière `//` donne la partie entière de la division : ainsi, $y$ ne prendra jamais la valeur 0 (car il est toujours $< 0$). .
4. Écrire la spécification de la fonction.
   Renvoie le produit de `x` et `y`. Précondition : `y` doit être positif et `c` plus grand que 2.

■

**Exercice 3**

1. Pour un entier naturel `n` et un flottant `x`, de combien de multiplications a-t-on besoin pour calculer `x**n` ?
   On a $n - 1$ multiplications à faire.
2. On considère un polynôme $P = \sum_{k=0}^{n} a_i X^i$ représenté sous Python par une liste de $n + 1$ coefficients $P[i] = c_i$ pour $i \in [\![0, n]\!]$. On considère la fonction suivante pour évaluer un polynôme :

```python
def evalue(P:list, x:float) -> float:
    val = 0
    for i in range(len(P)):
        val = val + P[i] * x**i
    return val
```

   Quelle est la complexité de cette fonction ?
   À la $i$-ième itération, on fait $(i - 1) + 1 + 1$ opérations, donc $i + 1$ opérations. Donc la boucle a une complexité $\sum_{i=0}^{n}(i + 1) = \frac{(n + 1)(n + 2)}{2} = O(n^2)$ (où $n$ est le degré de $P$).
   Ainsi, la fonction a une complexité quadratique.

3. La méthode de Horner pour évaluer un polynôme consiste à remarquer que :

$$a_nX^n + a_{n-1}X^{n-1} + \cdots + a_1X + a_0 = ((\cdots((a_nX + a_{n-1})X + a_{n-2})X + \cdots)X + a_1)X + a_0$$

Écrire une fonction `horner` qui met en oeuvre la remarque précédente.

```python
def horner(P:list, x:float) -> float:
    '''Renvoie la valeur de P(x) avec la méthode d'Horner
    '''
    v = 0
    for i in range(len(P)-1,-1,-1):
        v = v*x + P[i]
    return v
```

4. Justifier la correction de la fonction.
   La propriété « $v = c_i + c_{i+1}x + \cdots + c_nx^{n-i}$ en fin d'itération $i$ »est un invariant de boucle :
   - à l'entrée dans la boucle, on a $i = n + 1$, donc $v = 0$ ;
   - si la propriété est vérifiée en début d'itération $i$ : $v = c_{i+1} + c_{i+2}x + \cdots + c_nx^{n-i-1}$ (attention, le $i$ diminue). On note $v'$ la valeur en sortie d'itération : $v' = vx + c_i = c_i + c_{i+1}x + \ldots + c_nx^{n-i}$, donc la propriété est bien vérifiée !

   En fin de boucle, on a $i = 0$, donc $v = c_0 + c_1x + \cdots + c_nx^n$.
5. Quelle est la complexité de `horner` ?
   On a une complexité linéaire en le degré de $P$ !