TP (S2) 1 Bonnes pratiques de programmation

Objectifs

- Savoir définir les spécifications (signature, docstring).
- Savoir annoter un bloc d'instructions (précondition, postcondition, invariant).
- Savoir mettre au point un jeu de tests.

Fichier externe ?

OUI fichier `TP_BonPratProg.py` *présent dans le dossier partagé de la classe*

1 SPÉCIFICATIONS

Exercice 1 **Conjecture de GOLDBACH et spécification** [Sol 1] La conjecture de GOLDBACH est la suivante : « tout nombre pair strictement plus grand que 2 peut être écrit comme la somme de deux nombres premiers ».

1. On désire écrire une fonction `isPrime(n:int)->bool` qui respecte la spécification suivante :

```
    """
    Détermine si l'entier 'n' est premier

    Parameters
    ----------
    'n' : int, entier dont on veut déterminer la primalité

    Returns
    -------
    bool, True si 'n' est premier, False sinon. Par convention \
    ↪ 1 n'est pas premier.
    """
```

Pour cela, on pourra tester la divisibilté de l'entier `n` par des entiers `k` vérifiant $k \geq 2$ et $k < n$, à l'aide d'une boucle `while`.

2. Écrire la spécification de la fonction `isGoldbach(n:int)->bool` qui renvoie `True` si et seulement si la conjecture est vérifiée pour l'entier n (si n est impair ou est inférieur ou égal à 2 la fonction renvoie `True`, cette convention peut paraitre étonnante mais justifiée par la prochaine question). Écrire ensuite le code de la fonction `isGoldbach` (on utilisera la fonction `isPrime` précédemment définie).

3. Écrire une fonction `goldbach(p:int)->bool` qui implémente la spécification suivante (on utilisera la fonction `isGoldbach` précédemment définie) :

```
    """
    Vérifie la conjecture de Goldbach jusqu'à un certain rang

    Parameters
    ----------
    p : int, entier maximal pour lequel on veut tester la \
    ↪ conjecture de Goldbach.

    Returns
    -------
    bool, True si et seulement si tout entier inférieur ou \
    ↪ égal à p vérifie la conjecture de Goldbach, False sinon
    """
```

Exercice 2 **Fonctions sans spécifications ni commentaires** [Sol 2] Sur l'espace « données » du réseau pédagogique, téléchargez le fichier `TP_BonPratProg.py` et copiez le sur votre espace personnel. Ce programme contient deux fonctions `f` et `g` dont les spécifications et les commentaires sont absents, ce qui rend la compréhension du rôle de ces fonctions assez difficile.

1. La variable `n` est ici un entier naturel. Faire apparaitre cette contrainte en commençant à écrire les spécifications de chacune des fonctions.

2. On cherche à déterminer le rôle de la fonction `f` (indice : `f` est liée à une suite (u_n) déjà rencontrée dans le cours d'informatique, et de mathématique). En testant différentes valeurs de `n`, pouvez-vous émettre une hypothèse sur la nature du résultat renvoyé par `f` ? Une fois cette identification réalisée, rappeler la définition « usuelle » de la suite calculée par `f`. Le programme fourni est-il compréhensible à l'aide de cette seule définition ?

   On donne une propriété de la suite (u_n) considérée ici :

$$\forall p \in \mathbf{N}, \quad \begin{cases} u_{2p} & = u_p(2u_{p+1} - u_p) \\ u_{2p+1} & = u_p^2 + u_{p+1}^2. \end{cases}$$

3. Détailler le fonctionnement du programme lorsqu'on exécute `f(2)`, `f(6)`. On précisera en particulier les appels successifs à la fonction `g`.

4. Compléter les fonctions (spécifications et commentaires) dans le but de les rendre plus directement compréhensibles. On pourra également renommer les fonctions de manière plus explicite.

5. Proposer le script d'une fonction `f2(n:int)->int` qui calcule renvoie u_n, en appliquant un algorithme itératif. Déterminer le nombre d'additions et affectations nécessaires au calcul de u_n par `f2`.

6. En prenant $n = 2^p, p \in \mathbf{N}$, montrer que la fonction `f` permet de calculer u_n avec moins d'opérations élémentaires que `f2`, pour n « grand » .

2 INVARIANTS

Exercice 3 **Recherche dichotomique** [Sol 3] On reprend ici le script de recherche dichotomique dans une liste triée (par ordre croissant), sans spécification ni signature :

```
def present_dicho(t:_____, v:_____)->_____:
    d = 0
    f = len(t)
    trouve = False
    # Précondition : ____________________________
    while not trouve and d < f:
        m = (d + f) // 2
        if t[m] == v:
            trouve = True
        elif t[m] < v:
            d = m + 1
        else:
            f = m
    # Postcondition : ____________________________
    return trouve
```

1. Cette fonction à trous est dans le fichier fourni. Compléter la en précisant sa signature, une docstring, ainsi que précondition et postcondition en commentaire.

2. Montrer que : $\mathscr{P}(i)$ «$(v \notin t)$ **ou** $(v \in t[d_i : f_i])$» est un invariant de la boucle `while`.

Exercice 4 **Tester l'ordre croissant** [Sol 4] Écrire une fonction qui prend en entrée une liste de nombres et qui renvoie `True` si ces nombres sont dans l'ordre croissant, `False` sinon. Cette fonction devra être correctement documentée et commentée, un invariant de boucle sera proposé et justifié.

3 TESTS

Exercice 5 **Algorithme de HÖRNER itératif (retour)** [Sol 5] On considère un polynôme $P = \sum_{i=0}^{n-1} a_i X^i$, et on souhaite évaluer ce polynôme sur des réels x. Pour cela on choisit de représenter ce polynôme avec la suite de ses coefficients, à savoir : $[a_0, \dots, a_{n-1}]$, coefficient constant à gauche de la liste. On conviendra que le polynôme nul est codé par la liste vide.

1. **[Méthode naïve]** Un étudiant propose la fonction (non documentée!) suivante :

```
def eval_poly1(P: list, x: float) -> float:
    S = P[0]
    for k in range(1, len(P)):
        S += P[k]*(x**k)
    return S
```

1.1) Écrire et exécuter dans la console un test qui permet de montrer que la fonction ne renvoie pas toujours le résultat attendu. Apporter la correction nécessaire à la fonction.

1.2) Proposer un invariant pour la fonction modifiée.

1.3) En remarquant que $x^k = 1 \times x \times \cdots \times x$ et nécessite ainsi k multiplications, déterminer combien exactement il y a de multiplications effectuées lors de l'évaluation d'un polynôme par cette fonction.

2. **[On peut faire mieux]** L'idée de l'algorithme de HÖRNER s'appuie sur l'idée suivante, supposons pour simplifier que l'on cherche à évaluer $ax^3 + bx^2 + cx + d$, alors on peut écrire :

$$ax^3 + bx^2 + cx + d = ((a \times x + b) \times x + c) \times x + d$$

il y a seulement trois multiplications. Proposer une troisième version de la fonction précédente, nommée `eval_poly2` qui utilise l'algorithme de HÖRNER, le nombre de multiplications ne doit pas dépasser n. Documenter votre fonction.

3. **[Tests de performance]** Nous allons comparer les performances en temps d'exécution de ces deux fonctions en prenant le polynôme $\mathrm{P} = \sum_{k=0}^{n-1} kx^k$ avec $n = 1000$ et $x = -20$. Pour cela on importe la fonction `time()` du module `time`, on exécute un certain nombre de fois (par exemple 1000 fois) chacune de ces fonctions en mesurant le temps écoulé pour chacune. Par exemple pour la première cela donnerait :

```
t1 = time()  # on relève l'heure initiale
for _ in range(1000):
    r1 = eval_poly1(P,x)
t2 = time()  # on relève l'heure de fin
print("eval_poly1: ",t2-t1)  # on affiche le temps écoulé
```

Comparer ainsi le temps d'exécution des deux fonctions, commenter les résultats.

Remarque : si on est dans un notebook, alors on peut plus simplement utiliser l'instruction `timeit eval_poly1(P,x)` qui va mesurer automatiquement le temps d'exécution de la fonction.

Exercice 6 Conception d'un jeu de tests [Sol 6]

Dans le fichier `TP_BonPratProg.py`, on a écrit quatre fonctions différentes (`max1`,`max2`,`max3` et `max4`) dont l'objectif est de renvoyer le maximum d'une liste d'entiers passée en argument. On désire tester ces fonctions, uniquement en regardant si elles renvoient le résultat attendu lorsqu'on les applique à différentes listes (supposées non vide). Pour chaque test, on a donc un couple `(L,v)` où `L` est une liste d'entiers et `v` la valeur du maximum attendu.

1. Imaginer un ensemble de couples de type `(L,v)` et stocker ces couples dans une liste `jeu_test` (cette liste est donc du type `[(list,int)]`).

2. La fonction `test_max(f:callable,jeu_test:[(list,int)])->None` fournie permet de tester une fonction `f` (à choisir parmi les quatre fonctions `max` proposées) sur les tests contenus dans la liste `jeu_test`. Appliquer votre jeu de tests à chacune des fonctions et déterminer quelle est (sont) la (les) fonctions correctes.

Exercice 7 Division euclidienne dans $\mathbb{Z}$ [Sol 7]

Pour cet exercice, on reprend l'exemple du cours `division(a, b)`, qui était la division euclidienne dans $\mathbb{N}$, pour l'étendre aux entiers relatifs. La nouvelle version devra envoyer un tuple d'entiers (q, r) tels que : $a = bq + r$ avec $0 \leqslant r < |b|$.

1. Pour cette nouvelle version, préciser signature, pré et post condition. Contrairement à l'exemple du cours, on conviendra que la fonction doit provoquer une erreur lorsque la pré-condition n'est pas remplie.

2. Écrire une `docstring` pour cette nouvelle version incluant un jeu de tests.

3. En reprenant l'idée du code de l'exemple du cours, proposer une adaptation de celui-ci tout en conservant la relation $a = bq + r$ comme invariant. On sera amené à distinguer $a \geqslant 0$ et $a < 0$.

4. Saisir et tester la nouvelle version.

SOLUTIONS DES EXERCICES

Solution 1

1.

```
def isPrime(n):
    """
    Détermine si 'n' est premier

    Parameters
    ----------
    n : int, supposé > 1, entier dont on veut déterminer la primalité

    Returns
    -------
    bool, True si 'n' est premier, False sinon
    """
    k = 2
    while k < n and n%k != 0:
        k += 1
    return k == n
```

```
>>> isPrime(3)
True
>>> isPrime(4)
False
>>> isPrime(1)
False
```

2.

```
def isGoldbach(n):
    """
    Teste la conjecture de Goldbach sur l'entier "n"

    Parameters
    ----------
    n : int
        entier pour lequel on veut tester la conjecture de Goldbach.

    Returns
    -------
    bool
        False si "n" est pair et que la conjecture de Goldbach n'est pas vérifiée pour "n", True sinon.
        True si "n" est impair
    """
    # cas où la conjecture est à vérifier
    if n%2 == 0 and n > 2:
        gold = False
        # on cherche si on peut écrire n = k + q avec k et q premiers
        k = 2
        while k <= n//2 and not gold:
            if isPrime(k) and isPrime(n-k):
                gold = True
            k += 1
        return gold
    else:
        # autres cas (convention)
        return True
```

```
>>> isGoldbach(13)
True
>>> isGoldbach(14)
True
```

3.

```
def golbach(p:int)->bool:
    """
    Vérifie la conjecture de Goldbach jusqu'à un certain rang

    Parameters
    ----------
    p : int
        entier maximal pour lequel on veut tester la conjecture de Goldbach.

    Returns
    -------
    bool
        True si et seulement si tout entier inferieur ou égal à p vérifie la conjecture de Goldbach, False sinon

    """
```

```
    for n in range(p+1):
        if not isGoldbach(n):
            return False
    return True
>>> golbach(30)
True
```

Solution 2

1. On peut commencer ainsi :

```
def f(n):
    """
    Parameters
    ----------
    n : int
    Returns
    -------
    int
    """

    assert n >=0, "la variable n n'est pas un entier naturel"
    return g(n)[0]

def g(n):
    """
    Parameters
    ----------
    n : int
    Returns
    -------
    tuple

    """
    if n == 0:
        return (0, 1)
    a, b = g(n//2)
    c = a*(2*b-a)
    d = a**2+b**2
    if n%2 == 0:
        return (c, d)
    else:
        return (d, c+d)
```

2. On obtient les résultats suivants :

```
>>> f(0),f(1),f(2),f(3),f(4),f(5)
(0, 1, 1, 2, 3, 5)
```

Cela semble correspondre aux premiers termes de la suite de FIBONNACCI, définie par :

$$u_0 = 0, \quad u_1 = 1, \quad \forall n \geq 0, u_{n+2} = u_{n+1} + u_n.$$

Le programme fourni ne semble pas s'appuyer directement sur cette définition.

3. Lors de l'appel $f(2)$, on appelle $g(2)$, et l'écriture récursive de g nécessite l'appel de $g(1)$, qui elle-même nécessite $g(0)$ (cas terminal). On a ainsi $g(0)$ qui renvoie (u_0, u_1), $g(1)$ utilise la propriété décrite avec $p = 0$ (ce qui ne nécessite que u_0, u_1) et renvoie (u_1, u_2), enfin $g(2)$ utilise la propriété décrite avec $p = 1$ (ce qui ne nécessite que u_1, u_2) et renvoie (u_2, u_3). Ainsi $f(2)$ renvoie u_2.

De même, pour $f(6)$, on appelle successivement $g(6), g(3), g(1)$ et $g(0)$. Comme précédemment, $g(0)$ renvoie (u_0, u_1), $g(1)$ renvoie (u_1, u_2). L'appel à $g(3)$ utilise la propriété avec $p = 1$ (ce qui ne nécessite que u_1, u_2) et renvoie (u_3, u_4), l'appel à $g(6)$ utilise la propriété avec $p = 3$ (ce qui ne nécessite que u_3, u_4) et renvoie (u_6, u_7). Finalement $f(6)$ renvoie u_6.

4. Afin de rendre les fonctions plus compréhensibles, on renomme `f` en `fibo` et `g` en `fibo_aux`. On précise les entrées et surtout les sorties de chaque fonction dans la spécification, et on fournit en commentaire la propriété de la suite (u_n) sur laquelle s'appuie le code. On obtient pour la réécriture de `f` :

```
def fibo(n):
    """ renvoie le nième terme de la suite de Fibonnacci

    Parameters
    ----------
    n : int
        rang de la suite recherché

    Returns
    -------
    int
        nième terme de la suite de Fibonnacci

    Examples
```

```
    --------
    >>> fibo(0)
    0
    >>> fibo(1)
    1
    >>> fibo(6)
    8

    """
    assert n >=0, "la variable n n'est pas un entier naturel"
    # on utilise une fonction auxiliaire fibo_aux(n) qui \
    ↪ renvoie (u_n,u_{n+1})
    return fibo_aux(n)[0]
```

et pour celle de g :

```
def fibo_aux(n):
    """
    Renvoie le couple de termes (u_n,u_{n+1}) de la suite de \
    ↪ Fibonnacci

    Parameters
    ----------
    n : int

    Returns
    -------
    tuple
        (u_n,u_{n+1})

    Examples
    --------
    >>> fibo_aux(0)
    (0,1)
    >>> fibo_aux(1)
    (1,1)
    >>> fibo_aux(6)
    (8,13)
    """
    # cas de base
    if n == 0:
        return (0,1)
    else:
        # On utilise les propriétés suivantes :
        #   u_{2p} = u_p*(2*u_{p+1}-u_p)
        #   u_{2p+1} = u_p^2+u_{p+1}^2
        a, b = fibo_aux(n//2) # a,b contiennent \
        ↪ (u_(n//2),u_{n//2+1))
        c = a*(2*b - a)
        d = a**2 + b**2
        # si n = 2p, c contient le terme de rang \
        ↪ 2*(n//2)=2p=n,et d le terme suivant
        # sinon n = 2p+1, c contient le terme de rang  \
        ↪ 2*(n//2)=2p=n-1, et d le terme de rang n : on \
        ↪ utilise alors la relation u_{n+1} = u_n+u_{n-1} pour \
        ↪ calculer le terme de rang n+1
        if n%2 == 0:
            return (c, d)
        else:
            return (d, c+d)
```

5. La fonction `fibo2` :

```
def fibo_2(n):
    """
    Renvoie le terme u_n de la suite de Fibonnacci

    Parameters
    ----------
    n : int
        rang de la suite recherché

    Returns
    -------
    int
        nième terme de la suite de Fibonnacci
    """
    # u et v contiennent u_n et u_{n+1} à chaque itération
    u, v = 0, 1
    # on utilise ici u_{n+2} = u_n+u_{n+1}
    for k in range(n):
        u, v = v, u+v
    return u
```

À chaque itération, on réalise une somme et deux affectation. En comptant les deux premières affectations (indépendantes de n), le nombre $C(n)$ d'opérations est $C(n) = 2 + 3n$.

6. L'appel récursif de $g(n)$ se fait sur des arguments qui décroissent selon les valeurs successives suivantes : $2^p (= n), 2^{p-1} (= n/2), 2^{p-2}, 2^0 (= 1)$ et finalement 0 (cas terminal qui n'effectue aucune opération). On a donc $p+1$ itérations, avec à chaque fois un nombre faible d'opérations, que l'on peut majorer par $K > 0$. On a donc $C'(n) < Kp = K\log_2(n)$. On a bien $C'(n) < C(n)$ à partir d'un certain rang. L'algorithme proposé dans les fonctions f et g nécessite moins d'opérations pour les grandes valeurs de n. En revanche, il est plus délicat à comprendre (et nécessite donc une spécification et des commentaires détaillés).

Solution 3

1. En rajoutant les éléments demandés, on peut proposer pour la partie demandée (le code étant inchangé) :

```
def present_dicho(t:[int], v:int)->bool:
    """
    Determines if value 'v' belongs to 't' by dichotomic method.

    Parameters
    ----------
    t : list of n int, sorted in ascending order
    v : int, value to be tested

    Returns
    -------
    bool, True if value 'v' is in 't', False in the other case

    Examples
    --------
    >>> dicho([1,3,5],5)
    True
    >>> dicho([1,3,5],4)
    False
    >>> dicho([],1)
    False
    """
    d = 0
    f = len(t)
    trouve = False
    # précondition (P) : t est une liste triée par ordre \
    ↪ croissant
    while not trouve and d < f:
        m = (d + f) // 2
        if t[m] == v:
            trouve = True
        elif t[m] < v:
            d = m + 1
        else:
            f = m
    # postcondition (Q) : trouve vaut True si 'v' appartient à \
    ↪ 't', False sinon.
    return trouve
```

2. Distinguons deux cas :

- Si `t` ne contient pas `v`, alors l'invariant est bien sûr vrai.
- Si `t` contient `v` (`t` n'est donc pas vide), montrons que $v \in t[d_i : f_i]$ est vrai pour tout $i \in \{0, \dots, n\}$ où n désigne ici le nombre d'itérations de la boucle `while`.

  Initialisation. Pour $i = 0$: avant la boucle on a $d_0 = 0$ et $f_0 =$ `len(t)` et ainsi $t[d_0 : f_0] = t[0 : \text{len}(t)] = t$, l'invariant est bien donc vérifié.

  Hérédité. Supposons qu'il soit vrai en un rang $i \in [\![0, n-1]\!]$ et qu'il y ait une itération $i+1$. Alors montrons que $v \in t[d_{i+1} : f_{i+1}]$. On a $m_{i+1} = (d_i + f_i)//2$ et faisons alors plusieurs cas.
  - ◇ **[Cas 1]** $t[m_{i+1}] = v$. On a $d_{i+1} = d_i$ et $f_{i+1} = f_i$ donc l'invariant reste vrai.
  - ◇ **[Cas 2]** $t[m_{i+1}] < v$. Alors puisque la liste est croissante, on a $v \in t[m_{i+1} + 1 :]$. Or, $d_{i+1} = m_{i+1} + 1, f_{i+1} = f_i$ donc l'invariant reste vrai.
  - ◇ **[Cas 3]** $t[m_{i+1}] > v$. Alors puisque la liste est croissante, on a $v \in t[: m_{i+1}]$ *(rappel : l'indice de droite dans un slicing est exclu)* Or, $d_{i+1} = d_i, f_{i+1} = m_{i+1}$ donc l'invariant reste vrai.

  L'invariant est donc vérifié.

Solution 4

```
def ascending(t: list) -> bool:
    """
    Determines whether the elements of t are in ascending order.
```

```
    Parameters
    ----------
    t : list of n values

    Returns
    -------
    bool
        True if the elements of t are in ascending order.

    Examples
    --------
    >>> ascending([1,3,5,7,8])
    True
    >>> ascending([1,5,3,7,8])
    False
    >>> ascending([])
    True
    """
    result = True  # variable pour le résultat
    k = 0  # index pour le parcours
    while result and k < len(t)-1 :  # il faut k+1 <= len(t)-1
    # Invariant: result indique si t[:k+1] est dans l'ordre \
    ↪ croissant
        if t[k] > t[k+1]:  # on compare chaque élément avec le \
        ↪ suivant
            result = False  # 2 éléments ne sont pas dans le bon \
            ↪ ordre (sortie de boucle)
        k += 1  # pour passer au suivant, on a maintenant k <= \
        ↪ len(t)-1
    return result
```

Preuve pour l'invariant :

- Avant la boucle, on a $k = 0$ donc `t[:k+1]` vaut `t[:1]` qui est la liste vide si t est vide, et la liste réduite au premier élément si t n'est pas vide, dans les deux cas, l'invariant est vérifié.
- Supposons l'invariant vérifié à l'issue d'une itération, et qu'il y ait une itération suivante, ce qui signifie que `result` est **`True`** et que `k < len(t)-1`.
  - ◇ Si `t[k] <= t[k+1]` alors grâce à l'invariant on peut affirmer que `t[:k+2]` est dans l'ordre croissant, ce qui donne bien l'invariant en fin d'itération puisque k augmente de 1 et que la valeur de `result` n'a pas changé.
  - ◇ Par contre, si `t[k] > t[k+1]` alors on peut affirmer que `t[:k+2]` n'est pas dans l'ordre croissant, ce qui donne bien l'invariant en fin d'itération puisque k augmente de 1 et que la valeur de `result` est passée à **`False`**.

En sortie de boucle : on a `(not result) or (k >= len(t)-1)` qui est vérifié ainsi que l'invariant. Si `result` a la valeur **`False`** alors l'invariant nous permet d'affirmer que `t[:k+1]` n'est pas dans l'ordre croissant et donc t non plus. Si `result` a la valeur **`True`** alors l'invariant nous permet d'affirmer que `t[:k+1]` est dans l'ordre croissant, mais comme on a également `k >= len(t)-1`, on a en réalité on a `k = len(t)-1`, et la sous-liste `t[:k+1]` est la liste t en entier.

Donc en sortie de boucle, `result` indique bien si la liste est dans l'ordre croissant.

Remarque : le corps de la fonction pourrait être remplacé par le code équivalent suivant :

```
for k in range(len(t)-1):
    if t[k] > t[k+1]:
        return False
return True
```

Solution 5

1. **1.1)** La fonction ne s'exécute pas correctement sur une liste vide : cela provoque une erreur (évaluation de `P[0]` impossible), alors qu'il faudrait que la fonction renvoie 0 (si on convient qu'une liste vide représente le polynôme nul). une version modifiée est :

```
def eval_poly1(P: list, x: float) -> float:
    S = 0
    for k in range(len(P)):
        S += P[k]*(x**k)
    return S
```

1.2) On peut proposer l'invariant : « $S_i = \sum_{k=0}^{i-1} P[k]x^k$ », avant l'entrée dans la boucle on convient que S_0 est nul.

1.3) Il y a $i - 1 + 1 = i$ multiplications à l'itération $i \geqslant 1$, donc puisque `len(P)` $= n$, au total $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ multiplications.

2. Avec l'algorithme de HÖRNER (la `docstring` n'a pas été reproduite) :

```python
def eval_poly2(P: list, x: float) -> float:
    """ ... """
    n = len(P)
    S = 0
    for k in range(1, n+1):
    # Invariant: S_i = sum_{k=1}^i P[n-k]x^{i-k}
        S = S * x + P[n-k]
    return S
>>> eval_poly2([1, 2, 3], -1) # évaluation de P = 1+2X+3X^2 en \
↪ -1
2
```

ou en utilisant un `range` à rebours si on préfère :

```python
def eval_poly2(P: list, x: float) -> float:
    """ ... """
    n = len(P)
    S = 0
    for k in range(n-1, -1, -1):  # on parcourt la liste à \
    ↪ l'envers
    # Invariant: S_i = sum_{k=1}^i P[n-k]x^{i-k}
        S = S * x + P[k]
    return S
>>> eval_poly2([1, 2, 3], -1) # évaluation de P = 1+2X+3X^2 en \
↪ -1
2
```

3. On définit d'abord P, n et x : `n = 1000 ; P = list(range(n)) ; x = -20`. Résultats obtenus :

```
eval_poly1:  2.386896848678589
%eval_poly2:  0.4218263626098633
eval_poly2:  0.24854230880737305
```

Attention, les temps mesurés ne sont pas forcément identiques d'une machine à l'autre, mais les comparaisons restent les mêmes.

Avec `timeit` :

```
>>> timeit eval_poly1(P,x)
2.56 ms ± 153 µs per loop (mean ± std. dev. of 7 runs, 100 \
↪ loops each)
%>>> timeit eval_poly2(P,x)
%451 µs ± 30.3 µs per loop (mean ± std. dev. of 7 runs, 1000 \
↪ loops each)
```

```
>>> timeit eval_poly2(P,x)
268 µs ± 15.3 µs per loop (mean ± std. dev. of 7 runs, 1000 \
↪ loops each)
```

Solution 6

1. Il faut tester plusieurs cas limites : le max peut être au début, à la fin, au milieu de la liste, et la liste peut contenir des entiers positifs ou bien négatifs (uniquement). On peut proposer par exemple la liste suivante :

$$jeu\_test = [([-3,-1,-2],-1),([5,2,13],13),([9,2,8],9)]$$

2. Les fonctions `max1`, `max2`, `max3` échouent sur les tests proposés et sont donc incorrectes. La fonction `max4` passe tous les tests (ce qui ne signifie pas qu'elle est forcément correcte).

Solution 7

1. Spécifications :
   - Signature : `division(a: int, b: int) -> (int, int)`.
   - Pré-condition : a et b sont des entiers relatifs avec b non nul.
   - Post-condition : le résultat de la fonction est le tuple (q,r) où q et r soxnt respectivement le quotient et le reste de la division euclidienne de a par b (c'est à dire vérifiant $a = bq + r$ avec $0 \leqslant r < |b|$).

2. La `docstring` et un jeu de tests

```python
def division(a: int, b: int) -> (int, int):
    """
    Renvoie le quotient et le reste de la division de a par b

    Paramètres:
    ----------
        a: int
        b: int, entier non nul

    Retour:
    ------
        tuple (q, r) tel que a=bq+r avec 0 <= r < |b|
        ou bien None si b est nul
```

```
    Exemples:
    --------
    >>> division(19,7) == (2,5)
    True
    >>> division(7,19) == (0,7)
    True
    >>> division(0,19) == (0,0)
    True
    >>> division(19,-7) == (-2,5)
    True
    >>> division(-19,7) == (-3,2)
    True
    >>> division(-19,-7) == (3,2)
    True

    """
```

3. Pour conserver l'invariant on initialise encore q à 0 et r à a. Si a est positif le principe est le même que dans le cours mais en prenant $|b|$ à la place de b (on retranche $|b|$ à r et on adapte q en conséquence). Mais si $a < 0$ la même méthode ne marche plus (car $r < 0$), cette fois-ci il faut ajouter $|b|$ à r et adapter q en conséquence, et ceci tant que $r < |b|$. On peut donc proposer ceci (la `docstring` n'a pas été reproduite) :

```
def division(a: int, b: int) -> (int, int):
    """ ... """
    # On comme par tester la pré-condition
    assert type(a) == int, f"division({a}, {b}): {a} n'est pas \
    ↪ entier"
    assert type(b) == int, f"division({a}, {b}): {b} n'est pas \
    ↪ entier"
    assert b != 0, f"division({a}, {b}): division par 0 "
    # La pré-condition est vérifiée
    q, r = 0, a
    r = a
    if b > 0:
        B, sg = b, 1  # on a sg*b = B = |b|
    else:
        B, sg = -b, -1  # on a sg*b = B = |b|
    if a >= 0:
        while r > B:
        # Invariant : a = bq+r et r>=0
            q += sg
            r -= B  # bq+r = b(q+sg)+(r-B)
    else:  # a < 0
        while r < 0:
        # Invariant: a = bq+r et r < B
            q -= sg
            r += B  # bq+r = b(q-sg)+(r+B)
    return(q,r)
```

4. En ajoutant `import doctest` avant la fonction, et `doctest.testmod()` après la fonction, on vérifie que tous les tests proposés sont validés.