Activité 03 - Analyse de code

Equipe Pédagogique LU1IN0*1

Consignes : Cette activité se compose d'une première partie guidée, suivie de suggestions. Il est conseillé de traiter en entier la partie guidée avant de choisir une ou plusieurs suggestions à explorer.

L'objectif de cette activité est d'écrire du code (annotations, modifications de programmes, fonctions, tests) qui permet d'étudier la sûreté et l'efficacité d'un programme existant.

Arithmétique.

- Un entier naturel non-nul k **divise** un entier naturel n quand il existe un entier naturel m tel que $n = m.k$; autrement dit, k divise n quand n est un multiple de k.
- Un **diviseur propre** d d'un entier naturel n est un entier différent de n qui divise n.
- Un nombre entier naturel n est **parfait** quand il est égal à la somme de ses diviseurs propres. Par exemple, les diviseurs propres de 6 sont 1, 2 et 3 et $6 = 1 + 2 + 3$, donc 6 est parfait.

On pourra consulter la page Wikipedia des nombres parfaits, si nécessaire :
`https://fr.wikipedia.org/wiki/Nombre_parfait`

1 Partie Guidée : Simulation

Voici une fonction `est_parfait` (et sa fonction auxilliaire `divise`) qui décide si un nombre entier non-nul est parfait :

```python
def divise(k : int, n : int) -> bool :
  """ Pre : k > 0 and n >= 0
  Decide si k divise n """

  return n % k == 0

def est_parfait(n : int) -> bool :
  """ Pre : n >= 1
  Decide si n est un nombre parfait."""

  s : int = 0
  i : int = 1
  while i != n :
      if divise(i, n):
          s = s + i
      i = i + 1
  return n == s
```

Question 1. Donner un jeu de test pour chacune des fonctions ci-dessus.

Question 2. En utilisant l'instruction `print`, écrire une fonction `eqt_parfait_simulee` qui renvoie le même résultat que `est_parfait` et qui **affiche** la valeur des variables `i` et `s` en entrée de boucle, et à la fin de chaque tour de boucle.

L'évaluation de `est_parfait_simulee(6)` devrait ressembler à :

```
=== Evaluation de : 'est_parfait_simulee(6)' ===
debut de boucle, s =  0
debut de boucle, i =  1
====================
fin du tour  1 , s =  1
fin du tour  1 , i =  2
====================
fin du tour  2 , s =  3
fin du tour  2 , i =  3
====================
```

```
fin du tour  3 , s =  6
fin du tour  3 , i =  4
====================
fin du tour  4 , s =  6
fin du tour  4 , i =  5
====================
fin du tour  5 , s =  6
fin du tour  5 , i =  6
====================
sortie de boucle, s =  6  et n =  6
True
==================================================
```

On rappelle (Chapitre 2.1 du cours) que `print(e1 : T1, e2 : T2, ..., en : Tn) -> None` (c'est une primitive *variadique*, c'est-à-dire qu'elle peut prendre un nombre variable de paramètres) convertit (quand c'est possible) la valeur de ses arguments en chaînes de caractères et les affiche sur une même ligne, séparées par des espaces. On pourra consulter le Chapitre 3.2.3 du cours qui décrit plus précisément la démarche abordée dans cette question.

2 Suggestion : Test

On peut être tenté, pour tester une fonction `f`, de lancer le calcul de `f(v)` puis, en fonction du résultat `r`, d'ajouter un `assert f(v) == r` au code. Cette méthode de test est mauvaise pour se convaincre de la sûreté d'une fonction[1], car on teste la fonction "contre elle-même".

Généralement, on ne connaît pas à l'avance le résultat attendu pour la fonction `est_parfait` : sans une recherche mathématique préalable, on ne peut pas deviner si `est_parfait(426)` doit renvoyer `True` ou `False`.

Cependant, on dispose (via *Wikipedia*, ou d'autre sources arithmétiques) de la liste des septs premiers nombres parfaits connus. On peut donc vérifier que le résultat de la fonction `est_parfait` est cohérent avec cette liste.

Question. Ecrire une fonction `test_parfait` qui prend en entrée un entier `n` et décide si le résultat de la fonction `est_parfait` est cohérent avec la liste des sept premiers nombres parfaits pour tous les nombres entre 1 et `n`. C'est-à-dire qu'elle vérifie si `est_parfait(k)` (pour tous les `k` entre 1 et `n`) est vrai **si et seulement si** le nombre `k` apparait dans la liste. On fera attention d'ajouter une précondition qui empêche les tests de `est_parfait` sur des entiers trop grand pour qu'on sache s'ils sont parfaits ou non.

3 Suggestion : Invariant

Remarque : Pour les questions qui attendent une réponse en français, on pourra soit écrire la réponse en commentaire dans le fichier `.py` rendu, soit joindre au rendu un fichier texte ou une photographie d'une feuille de brouillon.

Un bon invariant de boucle pour la fonction `est_parfait` est :

`s` *est égal à la somme de tous les diviseurs de* `n` *inférieurs ou égaux à* `i−1`

Question 1. Exprimer la correction de `est_parfait` est montrer que l'invariant permet de prouver sa correction.

Question 2. Ecrire une fonction `invariant` qui prend en entrée trois nombres `i`, `n` et `s` et qui décide si l'invariant est vrai.

Question 3. Ecrire une fonction `est_parfait_invariant` qui renvoie le même résultat que `est_parfait` et qui affiche si l'invariant est vrai ou faux en début de boucle et à la fin de chaque tour de boucle.

```
=== Evaluation de : 'est_parfait_invariant(6)' ===
Invariant vrai
Invariant vrai
Invariant vrai
Invariant vrai
Invariant vrai
```

[1]sauf dans des cas très particuliers, comme les tests de non-régression

```
Invariant vrai
True
=====================================================
```

Question 4. Prouver à la main que l'invariant est vrai (par récurrence sur les tours de boucle).

4 Suggestion : Tableau de simulation

Plutôt que d'afficher des indications de simulation sur la sortie standard avec `print`, on veut maintenant écrire ces indications dans un **fichier**, stocké sur le disque de l'ordinateur. Cela permet, entre autres, d'accéder à la simulation même apres avoir fermé MrPython.
En Python l'instruction :

```
with open(nom, 'w') as fichier :
  <corps>
<suite>
```

permet d'ouvrir **en écriture** (c'est ce qu'indique le `'w'`) un fichier dont le nom est `nom` (une chaîne de caractères), et de le désigner par `fichier` dans le <`corps`> du `with`.
Dans le corps du `with`, l'instructin `fichier.write(s)` écrit la chaîne de caractère `s` dans le fichier.
On peut passer à la ligne en écrivant le caractère de retour chariot `fichier.write("\n")`.
Le fichier est créé s'il n'existe pas déjà sur le disque (et il est remplacé s'il existe déjà). Il est accessible depuis un gestionnaire de fichier.
L'énoncé de l'exercice 7.3 du cahier d'exercices donne plus de précisions à ce sujet.

Question 1. Ecrire une fonction `est_parfait_fichier` qui écrit dans un fichier les mêmes indications de simulation que celles affichées par la fonction `est_parfait_simulee`.

Question 2. Ecrire une fonction `est_parfait_tableau` qui écrit dans un fichier un tableau de simulation pour `est_parfait`, en faisant en sorte que les cases du tableau soit ”alignées” (il faut prendre en compte la taille des entiers affichés pour compléter avec le nombre d'espaces correspondants).

```
==================================================
|          Simulation est_parfait(100)           |
==================================================
|         tour         |     i     |      s      |
==================================================
| entree               |         1 |           0 |
==================================================
| tour 1               |         2 |           1 |
==================================================
| tour 2               |         3 |           3 |
==================================================
| tour 3               |         4 |           3 |
==================================================
| tour 4               |         5 |           7 |
==================================================
| tour 5               |         6 |          12 |
==================================================
                     (...)
==================================================
| tour 97              |        98 |         117 |
==================================================
| tour 98              |        99 |         117 |
==================================================
| tour 99   (sortie)   |       100 |         117 |
==================================================
```

5 Suggestion : Comparaison de complexité

Question 1. On décide d'étudier la complexité de `est_parfait` vis-à-vis du nombre d'appels à la fonction `divise`. Ecrire une fonction `est_parfait_appels` qui renvoie le même résultat que la fonction `est_parfait` et qui affiche le nombre de fois où la fonction `divise` a été appelé.

```
=== Evaluation de : 'est_parfait_appels(100)' ===
99 appels a la fonction divise.
False
==================================================
```

La fonction `est_parfait` teste tous les nombres entre 1 et `n−1` pour calculer la somme des diviseurs propres de `n`. Or, on sait que si `k` divise `n`, alors `n // k` divise `n` aussi. On en déduit une nouvelle version de la fonction :

```python
def est_parfait_opti(n : int) -> bool :
    """ Pre : n >= 1
    Decide si n est un nombre parfait """
    s : int = 1
    i : int = 2
    if i == 1:
      return False
    while i != int(math.sqrt(n)) + 1 :
        if divise(i, n):
            if i != math.sqrt(n) :
                s = s + i + (n // i)
            else :
                s = s + i
        i = i + 1
    return n == s
```

Question 2. Ecrire une fonction `est_parfait_opti_appels` qui compte le nombre d'appel à `divise` que fait la fonction `est_parfait_opti`.

Question 3. Estimer combien d'appels à `divise` sont faits lors de l'évaluation de `est_parfait(n)` et de `est_parfait_opti(n)` en fonction de `n`.

Question 4. Produire une image qui représente les courbes du nombre d'appels à `divise` des deux fonctions expressions (`est_parfait(n)` et `est_parfait_opti(n)`), en fonction de `n`:

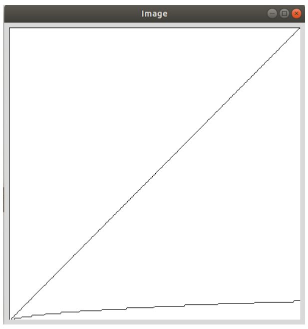

Figure 1: Comparaison de la complexité des deux fonctions pour les valeurs inférieures à 200

6 Suggestion : D'autres fonctions

On pourra appliquer les raisonnements présentés dans cette activité à d'autres fonctions, vues en cours ou en TDs, ou à des fonctions inédites.