Algorithmes de tri et invariant de boucle

Objectifs :

- Découvrir la notion d'invariant de boucle et preuve de correction d'un algorithme ;
- Poursuivre le travail sur la vérification des fonctions à l'aide d'écritures de tests ;
- Écrire un algorithme de tri : par insertion, par sélection ;
- Une première approche de la complexité en temps d'un algorithme ;

Matériel et logiciel nécessaire :

- Un serveur Jupyter ;

Durée estimée : 4 h

Tous les exercices seront réalisés dans un Notebook sur serveur Jupyter, y compris les questions pour lesquelles une explication est attendue.

I - Invariant se boucle

Nous avons déjà rencontré l'importance de la documentation des programmes pour ne pas perdre la trace des idées qui ont motivées leur écriture. Quand les programmes contiennent des boucles, notamment, il est particulièrement important de se persuader de la logique sous-jacente du programme. Nous devons toujours nous poser les questions suivantes :

- les variables du programme sont-elles correctement initialisées ?
- le nombre de tours de la boucle est-il le bon ?
- l'indice de la boucle est-il le bon ?
- les valeurs obtenues au final sont-elles bonnes ?

Toutes ces questions peuvent être abordées avec la notion d'invariant de boucle : il s'agit d'une propriété attachée à une boucle, qui :

- est vraie avant de commencer la boucle ;
- est vraie après chaque itération de la boucle ;
- est vraie à la sortie de la boucle.

Exemple 1 -

Considérons la fonction suivante qui calcule le quotient et le reste de la division euclidienne de a par b (a et b sont des entiers naturels) par la méthode des soustractions successives :

```
def division_euclidienne(a : int, b : int) -> tuple:
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return q, r
```

On suppose $a \geqslant 0$ et $b > 0$ et on cherche à vérifier que cette fonction renvoie bien une paire d'entiers q, r telle que :

$$a = q \times b + r \text{ et } 0 \leqslant r < b$$

ce qui est la définition d'une division euclidienne.

Le programme initialise la variable r avec la valeur contenue dans la variable a, puis lui retranche la valeur de b tant que $r \geqslant b$. En particulier, on aura bien $r < b$ une fois sorti de la boucle.

Mais il faut également vérifier l'autre inégalité, à savoir $0 \leqslant r$: on est parti d'une valeur de $a \geqslant 0$ à laquelle on a ensuite retranché la valeur de b uniquement lorsque $r \geqslant b$, ainsi on a toujours $r \geqslant 0$: voici un **invariant de cette boucle**.

Il reste à établir l'égalité de la division euclidienne. Elle est vraie avant la boucle car $a \geqslant 0$ et $r = a$. Ensuite, chaque tour de boucle, ajoute 1 à q et retranche b à r, ce qui maintient l'égalité. On peut s'en convaincre avec l'égalité suivante :

$$a = b \times q + r = (q+1) \times b + (r-b)$$

On peut maintenant commenter notre fonction avec ces deux invariants :

```
while r >= b:
    # invariant : 0 <= r
    # invariant : a = q*b+r
```

Si on a le moindre doute sur ces invariants, ou si le programme contient une erreur que l'on est en train de chercher à identifier, on peut faire vérifier ces deux invariants de boucle par Python en les écrivant sous forme d'`assert` plutôt que de commentaires :

```
while r >= b:
    assert 0 <= r
    assert a == q*b+r
```

Ces invariants seront systématiquement vérifiés, une fois la mise au point effectuée, on peut revenir vers les commentaires pour rendre le programme plus efficace.

Un invariant de boucle peut également être attaché à une boucle `for`. Dans ce cas, la propriété invariante peut faire référence à l'indice de boucle.

Exemple 2 -

Prenons l'exemple d'une fonction calculant la somme des entiers de 1 à n :

```
def somme_premiers_entiers(n : int) -> int:
    s = 1
    for i in range(n-1):
        # invariant : s = 1 + 2 + ... + (i+1)
        s = s + (i+2)
    return s
```

Au **début de la boucle**, on a $i = 0$, où i est l'indice de la boucle qui prend toutes les valeurs de 0 à $n-2$, ainsi on a bien $s = 1$.

Lorsque l'**on effectue une itération** de la boucle, on ajoute $i+2$ à la somme s, ce qui préserve l'invariant car la variable i est incrémentée de 1 par la boucle `for`.

À la **sortie de la boucle**, l'invariant est vérifié pour la valeur finale de i, c'est-à-dire $n-2$:

$$s_i = s_{i-1} + (n-2+2) = \underbrace{1+2+\ldots+((n-2)+1)}_{s_{i-1}} + n = 1+2+\ldots+(n-1)+n$$

Exercice 1

1. Donner un invariant de boucle pour la fonction suivante qui calcule la valeur de x à la puissance n.
2. À l'aide de cet invariant, vérifier la correction de cette fonction.
3. Ajouter la documentation et la spécification à cette fonction, puis réaliser quelques tests à l'aide de la commande `assert`.

```
def puissance(x, n):
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Exercice 2

Donner un meilleur nom à la fonction suivante, sa documentation, un invariant de boucle, sa correction ainsi que des tests.

```
def f(t):
    s = 0
    for i in range(len(t)):
        s = s + t[i]
    return s
```

II - Tri par selection

On considère un **tableau** dont les éléments peuvent-être **comparés**. Le tri par sélection parcourt le tableau de la gauche vers la droite, en maintenant sur la gauche une partie déjà triée et sa place définitive :

Déjà trié	Pas encore trié

À **chaque étape** :

- on cherche le plus petit élément dans la partie droite non triée ;
- on l'échange avec l'élément le plus à gauche de la partie non triée.

La **première étape** consiste à :

- déterminer le plus petit élément du tableau ;
- le placer tout à gauche du tableau.

Par construction, **la partie gauche** déjà triée ne contient que des **éléments inférieurs ou égaux** à ceux de la **partie droite** restant à trier.

Exercice 3

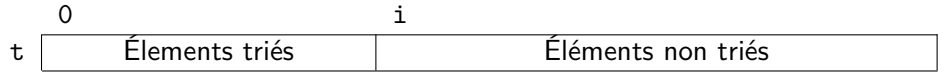
Dans cet exercice et ceux qui suivront, vous n'oublierez pas d'écrire la documentation, la spécification et quelques tests pour chacune des fonctions réalisées.

Écrire une fonction `echange` qui échange deux éléments d'un tableau t situés aux indices i et j.

Pour réaliser le tri par sélection, nous allons utiliser une boucle parcourant toutes les cases du tableau, de la gauche vers la droite :

```
for i in range(len(t)):
```

La partie déjà examinée du tableau étant déjà triée, sur la représentation ci-dessous, on matérialise bien le fait que l'élément d'indice i n'est pas encore examiné et ne fait pas partie des éléments déjà triés. En particulier, à la première itération de la boucle, $i = 0$ et la partie gauche est donc vide.



L'**invariant de boucle** sera : « la partie gauche du tableau déjà triée contient des éléments tous plus petits que ceux de la partie droite ».

Il faut maintenant chercher le plus petit élément dans la partie droite, c'est-à-dire la partie du tableau allant de i à `len(t)-1` inclus. Pour cela, on va écrire une boucle pour parcourir cette partie du tableau et on va se servir d'une variable pour retenir l'indice de la plus petite valeur rencontrée :

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

Exercice 4

Écrire la fonction `tri_par_selection` qui trie un tableau par ordre croissant. Vous ajouterez l'invariant de boucle et vous en servirez pour montrer la correction de votre algorithme.

III - Tri par insertion

Un autre algorithme de tri, souvent utilisé par les joueurs de cartes est le **tri par insertion**. Il suit le même principe que le tri par sélection, en parcourant le tableau de gauche à droite et en maintenant une partie déjà triée sur la gauche :

Éléments triés	Éléments non triés

Mais plutôt que de chercher la plus petite valeur dans la partie non encore triée, le tri par insertion va insérer la première valeur non encore triée dans la partie gauche déjà triée. On décalera d'une case à droite tous les éléments déjà triés qui sont plus grand que la valeur à insérer, puis déposer cette dernière valeur dans la case ainsi libérée.

Commençons par écrire la fonction **`insere`** qui réalise l'insertion d'une valeur `v` dans la partie du tableau `t[0...i]`, en supposant que ces valeurs déjà triées :

	0		i	
t	Éléments triés		?	Éléments non triés

Nous allons ensuite décaler un par un les éléments vers la droite jusqu'à trouver la bonne position de j de la valeur `v` pour se retrouver au final avec la situation :

	0		j		i	
t	Éléments triés inférieurs à v		v	Éléments triés supérieurs à v		Éléments non triés

Dans les cas extrêmes :

- la valeur `v` est plus grande que toutes les autres et on aura $j = i$;
- la valeur `v` est plus petites que toutes les autres et on aura $j = 0$.

Exercice 5

1. Écrire la fonction **`insere`** à l'aide d'une boucle **`while`**.
2. Écrire la fonction **`tri_par_insertion`** qui insère successivement toutes les valeurs du tableau avec la fonction `insere`, en procédant de la gauche vers la droite.
3. En utilisant l'invariant pour la boucle **`for`** principale : « la partie du tableau `t[0...i]` est triée », faire la preuve de la correction de l'algorithme de tri par insertion.

IV - Efficacité et notion informelle de complexité

Il est légitime de se demander si c'est une façon efficace de trier un tableau. On peut se poser la question en ces termes : « quel temps faut-il à notre tri pour trier un tableau de mille éléments, d'un million d'éléments, ou plus encore ? »

Exercice 6

1. La fonction **`shuffle`** de la bibliothèque **`random`** mélange un tableau de manière pseudo-aléatoire. À l'aide de celle-ci, écrire la fonction **`tableau_melange`** retournant un tableau d'entiers mélangés, de taille n. (on pourra commencer par générer par compréhension un tableau contenant les entiers allant de 1 à n).
2. La fonction **`time`** de la bibliothèque **`time`** donne, en secondes, le temps écoulé depuis le 1^{er} janvier 1970. En l'appelant et en conservant la valeur retournée à deux instants d'un programme, elle permet de mesurer le temps écoulé durant ces deux instants :

```
from time import time

debut = time()
...
fin = time()
duree = fin - debut
```

Écrire une fonction **`mesure_temps`** prenant pour paramètres la fonction de tri **`f`**, la taille du **`n`** tableau à trier et renvoyant le temps de calcul.

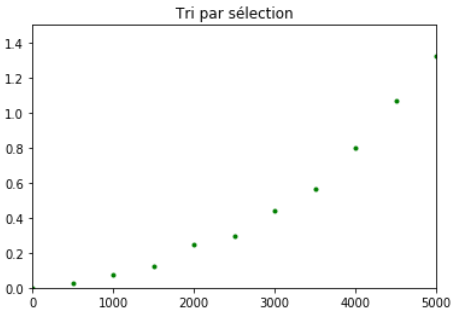
3. Compléter le tableau ci-dessous à l'aide de cette fonction, en indiquant le temps de calcul relevé pour ces différentes valeurs de `n`.

Taille `n`	100	500	1 000	2 000	5 000	10 000
`tri_par_selection`						
`tri_par_insertion`						

4. Le module `pyplot` de la bibliothèque **`matplotlib`** permet générer des graphiques à partir de tableaux de valeurs. Les ligne suivantes vous permettent d'observer l'évolution du temps de calcul de la fonction **`tri_par_selection`** en fonction de la taille des données :

```
import numpy
from matplotlib.pyplot import plot, xlim, ylim, show, title

nombre_tests = 10
taille_max = 5000
x = [i*taille_max/nombre_tests for i in range(nombre_tests + 1)]
y = [mesure_temps(tri_par_selection, int(i)) for i in x]
plot(x, y, linestyle = 'none', marker = '.', color = 'green')
xlim(0, taille_max)
ylim(0, y[nombre_tests])
title("Tri par sélection")
show()
```



Recopier ce programme et l'adapter pour afficher également le graphique du tri par insertion.

Il semble que ces deux méthodes de tri soient équivalentes du point de vue de la vitesse d'exécution et que le temps de calcul n'est pas proportionnel à la taille du tableau mais plutôt polynomial en la taille du tableau (la courbe ressemble à une parabole). Essayons de l'expliquer :

Dans la première étape du tri par sélection, on parcourt tout le tableau à la recherche de la plus petite valeur, il y a $n-1$ comparaisons (si n est la taille du tableau), dans la deuxième étape on effectue $n-2$ comparaisons, puis $n-3$, ... jusqu'à ce que le tableau à trier soit de taille 2 pour lequel il n'y aura qu'une comparaison.

Au total, on fait :

$$(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2}$$

calculs élémentaires (cette formule sera démontré lors du cours de mathématiques de première, c'est-à-dire de l'« **ordre** » de n^2 : on parle de **complexité en temps** de l'algorithme. Voici la définition de Wikipédia de celle-ci :

« En algorithmique, la complexité en temps est une mesure du temps utilisé par un algorithme, exprimé comme fonction de la taille de l'entrée. Le temps compte le nombre d'étapes de calcul avant d'arriver à un résultat. »

Lorsque le nombre de calculs élémentaires (comme pour le tri par sélection) est dans le pire des cas un polynôme du second degré, on parle de complexité **quadratique**. Ici, le temps de calcul est $\frac{1}{2}n^2-\frac{1}{2}n$ et l'**ordre de grandeur** de ce temps de calcul est $O(n^2)$.

Exercice 7

Déterminer de la même manière le nombre de calculs élémentaires nécessaires pour le tri par insertion.