TP 6 - INVARIANT

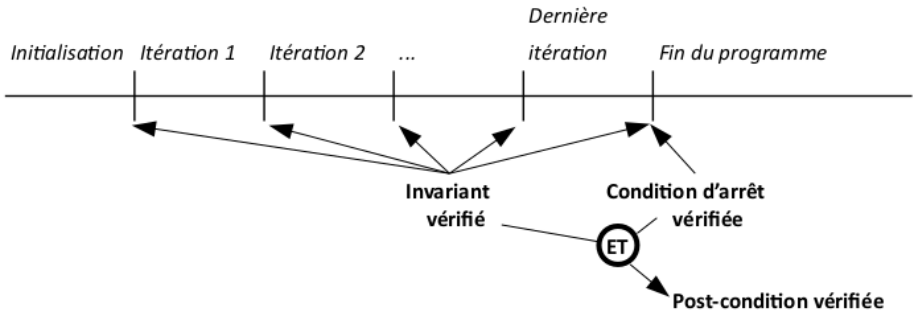
Info1.Algo1 - 2022-2023 Semestre Impair

Invariant

Rappels

On décompose la **post-condition** en 2 propriétés :

- L'**invariant**, vrai à l'initialisation ainsi qu'à chaque itération.
- La **condition d'arrêt** de la boucle.



La vérification expérimentale peut s'effectuer selon le modèle suivant :

```
def <nom de la fonction>(<parametre 1>,...):
  assert <pre-condition>, 'Pre-condition'
  # initialisation des variables
  assert <invariant>, 'Invariant (initialisation)'
  while not <condition arret>:
    # traitement de la boucle et itération
    assert <invariant>, 'Invariant (iteration)'
  assert <invariant> and <condition arret>, 'Post-condition'
  return <valeur de retour>
```

Exercice 1 - Factorielle (analyse d'erreurs) ★

> **Rappel :** La factorielle $n!$ d'un entier naturel $n \in \mathbb{N}$ est définie comme le produit des entiers de 1 à n :
>
> $$n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$$
>
> On a ainsi $0! = 1$ puisque le produit des éléments d'un ensemble vide est 1.

Dans le fichier **ex01_factorielle.py** est définie la fonction `factorielle` qui accepte en paramètre un entier positif `n` et est censée retourner la factorielle `f` de `n`. Pour vérifier la validité de l'algorithme utilisé on le compare avec la fonction de référence `factorial` du module `math`.

On a alors :

- **Post-condition** : `f==math.factorial(n)`
- **Invariant** : `f==math.factorial(k)`

Cependant, le code de cette fonction contient exactement 3 erreurs que l'on souhaite corriger en utilisant pour cela les assertions associées à l'invariant et à la post-condition.

Pour localiser la modification à effectuer, on pourra se servir du tableau suivant :

Nature de l'erreur	Localisation de l'erreur
Invariant (initialisation)	Initialisation
Invariant (itération)	Corps de boucle
Post-condition	Condition d'arrêt

1) Exécuter le code proposé et **analyser la nature de l'erreur observée**. Grâce à la **modification d'une valeur entière** dans le code, faire en sorte que cette erreur ne soit plus observée *(la deuxième erreur doit alors apparaître)*.

2) Analyser la nature de la deuxième erreur observée. Grâce à **un couper-coller** judicieux, faire en sorte que cette erreur ne soit plus observée *(la troisième erreur doit alors apparaître)*.

3) Analyser la nature de la troisième erreur observée. Grâce à l' **effacement d'un seul caractère**, faire en sorte que cette erreur ne soit plus observée.

Pour aller plus loin :

La factorielle $n!$ est plus fréquemment définie par la relation de récurrence :

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \quad \text{si } n \geq 1 \end{cases}$$

Cette définition peut être traduite quasiment mot-à-mot sous la forme d'une fonction récursive :

```
def factorielle_rec(n):
  assert n>=0, 'Pré-condition'
  if n==0:
    return 1
  else:
    return n*factorielle_rec(n-1)
```

La fonction `factorielle_rec` peut alors servir de fonction de référence pour valider l'implémentation itérative de la fonction `factorielle` :

- **Post-condition** : `f==factorielle_rec(n)`
- **Invariant** : `f==factorielle_rec(k)`

Exercice 2 - Logarithme entier (écriture guidée par la spécification) ★

Dans le fichier **ex02__log2__entier.py** est définie la fonction `log2_entier` qui accepte en paramètre un entier strictement positif `n` et retourne le logarithme entier `k` de `n` (en base 2).

Compléter le code de la fonction afin que les assertions soient validées.

Contraintes :

- Aucun `assert` ne doit être modifié.
- Chaque portion manquante (indiquée par une *ellipsis* `...`) doit être remplacée par **une seule** instruction (initialisation et corps de boucle) ou **une seule** expression (condition de boucle).
- La condition de boucle ne doit comporter **qu'une seule comparaison** que l'on peut déterminer en se basant sur la nature de la pré-condition et de l'invariant.

Exercice 3 - Division euclidienne (instrumentation de l'invariant) ★

> **Rappel :** Étant donnés deux entiers positifs a et b (b étant non nul), le quotient q et le reste r de la division euclidienne de a par b sont les uniques entiers tels que $a = q \cdot b + r$, avec $r \in [0; b[$

Dans le fichier **ex03_division_euclidienne.py** est définie la fonction `division_euclidienne` qui accepte en paramètre deux entiers `a` et `b` et retournant les entiers `q` et `r`. L'objectif de cet exercice est d'instrumenter le code fourni avec les `assert` nécessaires à la vérification de :

- la **pré-condition**.
- la **post-condition**.
- l'**invariant**

1) Écrire dans le corps de la fonction `division_euclidienne` les assertions de **pré-condition** et de **post-condition** correspondant à la définition rappelée ci-dessus.

2) Écrire dans le corps de la fonction `division_euclidienne` les 2 assertions pour la vérification de l'**invariant** *(on pourra s'aider de la formulation de la post-condition et de la condition de boucle afin d'écrire un invariant pertinent).*

Faire valider le code final obtenu par votre encadrant.e de TP.

Exercice 4 - Racine entière par dichotomie (analyse d'erreurs) ★

Dans le fichier **ex04_racine_entiere.py** est définie la fonction `racine_entiere` qui accepte en paramètre un entier positif `n` et est censée retourner la partie entière de la racine carrée de `n`, c'est-à-dire l'entier positif `a` tel que `a**2<=n<(a+1)**2`. Pour obtenir le résultat recherché, un algorithme de dichotomie est mis en oeuvre.

Cependant, le code de cette fonction contient exactement 3 erreurs que l'on souhaite corriger en utilisant pour cela les assertions associées à l'invariant et à la post-condition.

Exécuter le code proposé et corriger au fur et à mesure qu'elles se présentent les 3 erreurs d'assertion. Pour chaque erreur :

- **Analyser la nature de l'erreur observée**.
- Grâce à la **modification d'un seul caractère** dans le code, faire en sorte que cette erreur ne soit plus observée.

Pour localiser la modification à effectuer, on pourra se servir du tableau suivant :

Nature de l'erreur	Localisation de l'erreur
Invariant (initialisation)	Initialisation
Invariant (itération)	Corps de boucle
Post-condition	Condition d'arrêt

Exercice 5 - Recherches dans un tableau ★

On considère un tableau d'entiers `tab` trié par ordre croissant dont le premier élément est négatif ou nul et le dernier est strictement positif. On souhaite écrire une fonction qui détermine l'indice `i` du **dernier élément négatif ou nul** de ce tableau.

On se propose de résoudre ce problème selon deux algorithmes différents qui seront écrits dans les fonctions `indice_dernier_negatif_1` et `indice_dernier_negatif_2` du fichier **ex05_indice_dernier_negatif.py**.

1) a) Compléter le corps de la fonction auxiliaire `est_croissant` qui :

- accepte en paramètre un tableau d'entiers `tab`
- vérifie que ce tableau est trié par ordre croissant (au sens large)
- retourne le booléen correspondant.

b) Traduire par un booléen la **pré-condition** du problème et compléter les assertions correspondantes dans les deux fonctions.

2) Le modèle de solution de la fonction `indice_dernier_negatif_1` est un parcours gauche-droite pour lequel on donne l'invariant suivant : `0<=i<n-1 and tab[i]<=0`.

a) Écrire les 2 assertions de vérification de l'invariant.

b) La post-condition et l'invariant étant donnés, déterminer la **condition d'arrêt** ainsi que la **condition de boucle** pour cette fonction.

c) Compléter le code de la fonction de façon à ce que les assertions soient vérifiées.

3) Le modèle de solution de la fonction `indice_dernier_negatif_2` est un algorithme de dichotomie pour lequel on donne l'invariant suivant :

`0<=i<j<n and tab[i]<=0<tab[j]`

où `i` et `j` délimitent donc la zone de recherche `tab[i:j]`.

a) Écrire les 2 assertions de vérification de l'invariant.

b) La post-condition et l'invariant étant donnés, déterminer la **condition d'arrêt** ainsi que la **condition de boucle** pour cette fonction.

c) Compléter le code de la fonction de façon à ce que les assertions soient vérifiées.

★★ **4)** On souhaite analyser ce qui se passe lorsque la condition de croissance n'est pas présente dans la pré-condition. On considère donc un tableau d'entiers `tab` dont le premier élément est négatif ou nul et le dernier est strictement positif.

a) Commenter (temporairement!) la pré-condition des fonctions `indice_dernier_negatif_1` et `indice_dernier_negatif_2` et appeler chacune des deux fonctions sur le

tableau `[-7,-5,-2,4,-1,7,6,-2,9,15]` (qui n'est pas trié par ordre croissant). Chacune des deux fonctions renvoie-t-elle un résultat? Si oui, sont-ils les mêmes? Expliquer.

b) Peut-on trouver un tableau `tab` dont le premier élément est négatif ou nul et le dernier est strictement positif tel que l'une des deux fonctions (ou les deux) soi(en)t incapable(s) de retourner un résultat? Si oui lequel? Si non pourquoi?

Exercice 6 - Plus long palindrome centré ★★

Rappel : Une séquence est appelée **palindrome** lorsque l'on obtient la même séquence de valeurs si on la lit de gauche à droite ou de droite à gauche. **Exemple :** `[4,3,1,3,4]` et `[5,2,6,6,2,5]` sont des palindromes.

On souhaite écrire la fonction `plus_long_palindrome_centre` qui accepte en paramètre un tableau d'entiers `tab` de longueur `n` et retourne les entiers `d` et `f` tels que la tranche `tab[d:f]` soit le **plus long palindrome centré** extrait de `tab`.

Modèle de solution : On démarre d'une sous-tableau de longueur `0` ou `1` au centre du tableau passé en paramètre, puis on l'agrandit à gauche et à droite simultanément et étape par étape jusqu'à obtenir le tableau en entier ou rencontrer deux éléments différents.

Pour effectuer les différents tests :

- On dispose d'une fonction `est_tranche_palindrome` qui vérifie si la tranche `tab[d:f]` d'un tableau `tab` passé en paramètre est un palindrome et qui retourne le booléen correspondant.
- Pour vérifier que le palindrome est centré, il suffit de vérifier que `d+f==n`.

L'**invariant** est alors :

```
0<=d<=f<=n and d+f==n and est_tranche_palindrome(tab,d,f)
```

La **post-condition** est obtenue en ajoutant à l'invariant le fait qu'on englobe la totalité du tableau ou que les prochains éléments sont différents :

```
(d==0 or tab[d-1]!=tab[f])
```

1) Dans le fichier **ex06_palindrome_centre.py**, compléter les éléments méthodologiques indiqués en commentaire sans les justifier.

2) Écrire les assertions de **pré-condition**, de **post-condition** et d'**invariant** dans le corps de la fonction `plus_long_palindrome_centre`.

3) Compléter le code de la fonction de façon à ce que les assertions soient vérifiées.

Exercice 7 - Recherche par dichotomie ★★

On souhaite écrire la fonction `recherche_par_dichotomie` qui accepte en paramètre un tableau d'entiers `tab` non-vide et trié par ordre croissant, ainsi qu'un entier `valeur`. On impose de plus la condition suivante : `valeur>=tab[0]`. La fonction doit retourner l'indice `i` du tableau tel que :

```
tab[i]<=valeur and (i+1==len(tab) or valeur<tab[i+1])
```

Le modèle de solution étant une **recherche par dichotomie**, on a l'invariant suivant :

```
tab[i]<=valeur and (j==len(tab) or valeur<tab[j])
```

1) a) Dans le fichier **ex07__recherche__par__dichotomie.py**, compléter le corps de la fonction auxiliaire `est_croissant` qui :

- accepte en paramètre un tableau d'entiers `tab`
- vérifie que ce tableau est trié par ordre croissant (au sens large)
- retourne le booléen correspondant.

b) Compléter les éléments méthodologiques indiqués en commentaire sans les justifier.

2) Écrire les assertions de **pré-condition**, de **post-condition** et d'**invariant** dans le corps de la fonction `recherche_par_dichotomie`.

3) Compléter le code de la fonction de façon à ce que les assertions soient vérifiées.