TP 2 – SPÉCIFICATION DE FONCTION

Info1.Algo1 - 2025-2026 Automne

Rappels

La **spécification** d'une fonction doit préciser :

- le **type** des paramètres et valeurs de retour.
- la **pré-condition** : condition devant être vérifiée par les paramètres de la fonction afin que le problème puisse être résolu.
- la **post-condition** : condition devant être vérifiée par la(les) valeur(s) de retour de la fonction.

Les assertions destinées à vérifier la **pré-condition** et la **post-condition** sont placées respectivement en tout **début** et en toute **fin** de la fonction testée :

```
def <nom de la fonction>(<param1>,<param2>):
    assert ..., 'Pre-condition'
    ... # CODE DE LA FONCTION
    assert ..., 'Post-condition'
    return ...
```

Écriture de spécification

Dans cette première partie **(exercice 1 à 3)**, le code de la fonction étudiée vous est donné, et vous devez **écrire la pré-condition et la post-condition** de cette fonction, en vous basant sur les éléments donnés dans l'énoncé. Ces éléments de spécification doivent être écrits dans les fonctions dédiées `verifie_pre_condition` et `verifie_post_condition` selon le schéma général suivant :

```
def verifie_pre_condition(...,...):
    return ... # Pre-condition

def verifie_post_condition(...,...):
    return ... # Post-condition'

def <nom de la fonction>(<param1>,<param2>):
    assert verifie_pre_condition(...,...), 'Pre-condition'
    ... # CODE DE LA FONCTION
    assert verifie_post_condition(...,...), 'Post-condition'
    return ...
```

Lors d'un examen, les fonctions de tests `test_verifie_pre_condition` et `test_verifie_post_condition` ne seront pas nécessairement fournies. Une partie des points pourra être attribuée si une partie de la pré- ou de la post-condition est correcte. **Exemple :** une comparaison large au lieu d'une comparaison stricte (et vice versa), un encadrement incomplet, si la validité d'un indice n'est pas vérifiée avant de lire un élément d'une liste, etc.

Exercice 1 – division euclidienne ★

Contraintes pour cet exercice :

- la fonction `division_euclidienne` ainsi que les fonctions de test ne doivent pas être modifiées,
- les opérateurs `//` et `%` sont interdits.

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 **ex01_division_euclidienne.py** on donne la fonction `division_euclidienne` qui accepte en paramètres deux entiers `a` et `b` positifs (`b` étant non nul). La fonction retourne les entiers `q` et `r` correspondant au **quotient** et au **reste** de la division euclidienne de `a` par `b`.

1) Compléter l'instruction `return` de la fonction `verifie_pre_condition` avec l'expression booléenne de la **pré-condition** de la fonction `division_euclidienne`. La fonction `verifie_pre_condition` accepte en paramètres les entiers `a` et `b`.

2) Compléter l'instruction `return` de la fonction `verifie_post_condition` avec l'expression booléenne de la **post-condition** de la fonction `division_euclidienne`. La fonction `verifie_post_condition` accepte en paramètres les entiers `a`, `b`, ainsi que les valeurs de retour `q` (quotient) et `r` (reste).

Exercice 2 - calcul du minimum ★

> **Contrainte pour cet exercice :** la fonction `calculer_minimum` ainsi que les fonctions de test ne doivent pas être modifiées.

Dans le fichier **ex02_minimum.py** on donne la fonction `calculer_minimum` qui accepte en paramètre une liste non vide d'entiers, et retourne le plus petit des éléments de cette liste.

1) Dans cette première question, on écrit deux **fonctions auxiliaires** qui seront utiles pour l'écriture de la post-condition.

a) Écrire la fonction auxiliaire `est_membre` qui accepte en paramètres une liste d'entiers `liste` et un entier `m`, et retourne le booléen indiquant si `m` est un élément de `liste`.

b) Écrire la fonction auxiliaire `est_minorant` qui accepte en paramètre une liste d'entiers `liste` et un entier `m`, et retourne le booléen indiquant si `m` est inférieur ou égal à tous les éléments de liste.

2) Compléter l'instruction `return` de la fonction `verifie_pre_condition` avec l'expression booléenne de la **pré-condition** de la fonction `calculer_minimum`. La fonction `verifie_pre_condition` accepte en paramètre une liste d'entiers `liste`.

3) Compléter l'instruction `return` de la fonction `verifie_post_condition` avec l'expression booléenne de la **post-condition** de la fonction `calculer_minimum`. La fonction `verifie_post_condition` accepte en paramètre la liste d'entiers `liste` ainsi que l'entier `m` destiné à être retourné par la fonction `calculer_minimum`.

Exercice 3 - indice de stricte croissance ★

> **Contrainte pour cet exercice :** la fonction `indice_stricte_croissance` ainsi que les fonctions de test ne doivent pas être modifiées.

Dans le fichier **ex03_indice_stricte_croissance.py** on donne la fonction `indice_stricte_croissance` qui accepte en paramètre une liste non vide d'entiers `liste` dont **le dernier élément est strictement supérieur au premier élément**. La fonction retourne un entier `i` tel que l'élément de la liste situé à l'indice `i` est **strictement supérieur à l'élément qui le précède**

1) Compléter l'instruction `return` de la fonction `verifie_pre_condition` avec l'expression booléenne de la **pré-condition** de la fonction `indice_stricte_croissance`. La fonction `verifie_pre_condition` accepte en paramètre la liste d'entiers `liste`.

Indication : Vérifier la validité de la taille de la liste avant d'en comparer les éléments.

2) Compléter l'instruction `return` de la fonction `verifie_post_condition` avec l'expression booléenne de la **post-condition** de la fonction `indice_stricte_croissance`. La fonction `verifie_post_condition` accepte en paramètre la liste d'entiers `liste` ainsi que l'indice `i` recherché.

Indications :

- Vérifier la validité de l'indice `i` avant de comparer des éléments.
- Il peut exister **plusieurs indices valides**.

Implémentation selon une spécification

Dans cette deuxième partie **(exercice 4 à 6)**, la pré-condition et la post-condition vous sont données dans la fonction. Vous devez compléter la fonction afin de respecter cette spécification.

Pour garantir que la post-condition est cohérente avec le problème posé :

- **les paramètres de la fonction ne doivent pas être modifiés**.
- **un seul `return` est autorisé** *(sauf dans les fonctions auxiliaires)*.

Exercice 4 - réciproques entières ★

Dans cet exercice, on souhaite inverser le calcul d'un carré (c'est-à-dire trouver une **racine carrée**) et d'une exponentielle (c'est à dire un **logarithme**). Les valeurs exactes n'étant pas nécessairement des entiers, on cherche la partie entière de la solution réelle.

Indications :

Dans les 3 questions qui suivent :

- Il est **interdit d'utiliser les opérations de calcul flottant** des racines ou logarithmes. Toutes les variables et expressions doivent être de type `int`.
- La recherche peut se faire par une **boucle** en balayant de façon croissante les solutions possibles.
- La fonction de test se contente d'appeler la fonction à tester sans vérifier l'entier retourné. C'est en effet l'**assertion de post-condition** qui effectue cette vérification avant de retourner son résultat.

1) Dans le fichier **ex04_reciproques_entieres.py**, compléter la fonction `racine_entiere` qui accepte en paramètre un entier `n` et retourne un entier `a`

- **Pré-condition :** `n>=0`
- **Post-condition :** `a>=0 and a**2<=n<(a+1)**2`

2) Compléter la fonction `log2_entier` qui accepte en paramètre un entier `n` et retourne un entier `k`

- **Pré-condition :** `n>0`
- **Post-condition :** `k>=0 and 2**k<=n<2**(k+1)`

3) Compléter la fonction `logb_entier` qui accepte en paramètres deux entiers `n` et `b` et retourne un entier `k`

- **Pré-condition :** `n>0 and b>=2`
- **Post-condition :** `k>=0 and b**k<=n<b**(k+1)`

Exercice 5 - médiane ★

Dans cet exercice, on appelle **médiane** d'une liste d'entiers toute valeur de cette liste vérifiant les deux conditions suivantes :

- au moins la moitié des éléments de cette liste lui sont inférieurs ou égaux.
- au moins la moitié des éléments de cette liste lui sont supérieurs ou égaux.

Exemples :

- La médiane de la liste `[5,4,6,2,0]` est `4` puisqu'il y a `3` éléments qui lui sont inférieurs ou égaux (`4`, `2` et `0`) et `3` éléments qui lui sont supérieurs ou égaux (`5`, `4` et `6`).
- Les deux médianes acceptées pour la liste `[5,2,7,4]` sont `4` et `5` puisque dans les deux cas au moins `2` éléments sont inférieurs ou égaux (`2`, `4` et éventuellement `5`) et au moins `2` éléments sont supérieurs ou égaux (`5`, `7` et éventuellement `4`).
- La médiane de la liste `[1,3,3,5,2,3]` est `3` puisqu'il y a `5` éléments qui lui sont inférieurs ou égaux (`1`, `3` , `3` , `2` et `3`) et `4` éléments qui lui sont supérieurs ou égaux (`3` , `3` , `5` et `3`).

Remarque : Nous utilisons ici une définition différente de la définition habituelle de la médiane en mathématiques, car elle permet :

- D'éviter les nombres flottants.
- De chercher la médiane parmi les valeurs de la liste.

1) Dans le fichier **ex05_mediane.py** compléter les fonctions **auxiliaires** `nombre_inferieurs` et `nombre_superieurs` qui acceptent en paramètres :

- Une liste d'entiers `liste`.
- Un entier `valeur`.

Ces deux fonctions retournent respectivement :

- Le nombre d'éléments de `liste` qui sont inférieurs ou égaux à `valeur`.
- Le nombre d'éléments de `liste` qui sont supérieurs ou égaux à `valeur`.

2) Compléter la fonction `calculer_mediane` qui accepte en paramètre une liste **non vide** d'entiers `liste` (de longueur `n`) et retourne la valeur `mediane` qui peut être caractérisée par la **post-condition** suivante :

```
nombre_inferieurs(liste,mediane)>=(n+1)//2
and nombre_superieurs(liste,mediane)>=(n+1)//2
```

Contrainte : Ne pas modifier les variables `liste` et `n`.

Exercice 6 - indice du minimum ★

Dans cet exercice, on souhaite déterminer l'indice dans une liste où se trouve le minimum de cette liste.

1) Dans le fichier **ex06_indice_minimum.py** compléter la fonction **auxiliaire** `est_minorant_plage` qui accepte en paramètres :

- Une liste d'entiers `liste`.
- Deux indices `d` et `f` définissant le début et la fin d'une plage d'éléments de `liste`.
- Un entier `m`.

La fonction `est_minorant_plage` retourne `True` si `m` est inférieur ou égal à tous les éléments de la plage délimitée par `d` *(indice inclus)* et `f` *(indice exclu)*, c'est-à-dire tous les éléments d'indice `i` tel que `d<=i<f`. Elle retourne `False` sinon.

2) Dans cette question on détermine des indices `i_min` où se trouve le minimum de la liste.

Contrainte *(pour les 3 sous-questions)* : Ne pas modifier les variables `liste` et `n`.

a) Compléter la fonction `indice_minimum` qui accepte en paramètre une liste **non vide** d'entiers `liste` (de longueur `n`) et retourne n'importe quel indice `i_min` où se trouve le minimum de la liste.

- **Pré-condition :** `n>0`
- **Post-condition :** `0<=i_min<n and est_minorant_plage(liste,0,n,liste[i_min])`

Il n'y a pas nécessairement unicité de la solution : il peut exister plusieurs indices valides, l'essentiel étant d'en retourner un.

b) Pour caractériser qu'il s'agit du premier indice `i_min` rencontré, on rajoute à la post-condition que tous les éléments d'indices inférieurs à `i_min` sont strictement supérieurs au minimum, ce que l'on peut exprimer par l'expression :

```
est_minorant_plage(liste,0,i_min,liste[i_min]+1)
```

Compléter la fonction `indice_premier_minimum` qui correspond à cette nouvelle post-condition.

c) De même, compléter la fonction `indice_dernier_minimum` qui correspond au cas opposé où l'on cherche le dernier indice `i_min` dans la liste.

La post-condition requiert alors que tous les éléments d'indices supérieurs à `i_min` soient strictement supérieurs au minimum, ce que l'on peut exprimer par l'expression :

```
est_minorant_plage(liste,i_min+1,n,liste[i_min]+1)
```

Exercices d'entraînement

Cette troisième partie mélange des exercices d'**écriture de spécification** avec des exercices d'**implémentation selon une spécification**.

Exercice 7 - indice d'encadrement ★

> **Contrainte pour cet exercice :** la fonction `indice_encadrement` ainsi que les fonctions de test ne doivent pas être modifiées.

Dans le fichier **ex07_indice_encadrement.py** on donne la fonction `indice_encadrement` qui accepte en paramètres :

- une liste non vide d'entiers `liste`,
- un entier `valeur` supérieur ou égal au premier élément de `liste` et strictement inférieur au dernier élément de `liste`.

La fonction retourne un entier `i` tel que l'entier `valeur` est supérieur ou égal à `liste[i-1]` et strictement inférieur à `liste[i]`.

1) Compléter l'instruction `return` de la fonction `verifie_pre_condition` avec l'expression booléenne de la **pré-condition** de la fonction `indice_encadrement`. La fonction `verifie_pre_condition` accepte en paramètres la liste d'entiers `liste` et l'entier `valeur`.

Indication : Vérifier la validité de la taille de la liste avant d'en comparer les éléments.

2) Compléter l'instruction `return` de la fonction `verifie_post_condition` avec l'expression booléenne de la **post-condition** de la fonction `indice_encadrement`. La fonction `verifie_post_condition` accepte en paramètres la liste d'entiers `liste`, l'entier `valeur` ainsi que l'indice `i`.

Indications :

- Vérifier la validité de l'indice `i` avant de comparer des éléments.
- Il peut exister **plusieurs indices valides**.

Exercice 8 - compression RLE ★★

> Dans cet exercice la première fonction `decompresser_liste` est vérifiée par des tests unitaires tandis que la seconde fonction `compresser_liste` l'est aussi par un test de post-condition en se servant de la première fonction écrite.

La liste de nombres ci-dessous est la représentation numérique de l'image ci-dessus (image carrée en niveau de gris de 8 pixels de côté). Dans cet encodage, les `0` désignent les pixels noirs et les `255` désignent les pixels blancs. Les valeurs intermédiaires sont utilisées pour les différents niveaux de gris :

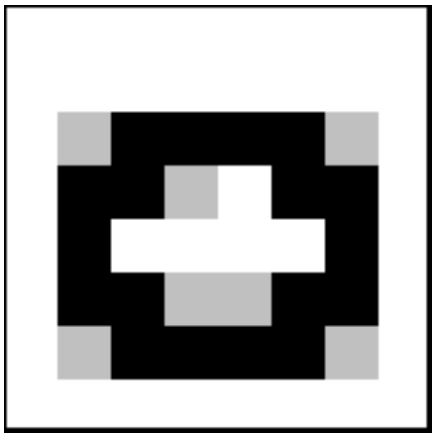

Figure 1: image à représenter

```
255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255
255 127 0   0   0   0   127 255
255 0   0   127 255 0   0   255
255 0   255 255 255 255 0   255
255 0   0   127 127 0   0   255
255 127 0   0   0   0   127 255
255 255 255 255 255 255 255 255
```

On remarque qu'il existe dans cette représentation de nombreuses plages de nombres identiques. Afin que la représentation numérique de cette image occupe moins de place mémoire, on peut utiliser la méthode de compression **RLE** (*Run Length Encoding*) qui est utilisée par de nombreux formats d'images (BMP, PCX, TIFF...).

Le principe de cette méthode est d'encoder une plage de nombres identiques par un couple de nombres :

`[` *la longueur de la plage* `,` *le nombre répété* `]`

Ainsi :

- La plage `[63,63,63,63,63,63]` est encodée par le couple `[6,63]`.
- La liste de nombres `[255,255,0,0,0,0,63,63,63,63,127,127,127]` est encodée par la liste `[2,255,4,0,4,63,3,127]`.

On souhaite écrire les fonctions `decompresser_liste` et `compresser_liste` qui permettent de passer d'un codage à l'autre.

1) Dans le fichier **ex08__rle.py**, compléter le code de la fonction `decompresser_liste`, qui accepte en paramètre une liste compressée et retourne une nouvelle liste représentant le codage original de cette liste. La liste passée en paramètre n'est pas modifiée.

Une liste compressée est une liste d'entiers valide si :

- La liste est de **longueur paire**.
- Les **éléments d'indices pairs** *(c'est-à-dire les longueurs des plages)* sont **strictements positifs**.
- Deux **éléments consécutifs d'indices impairs** *(c'est-à-dire les nombres répétés)* sont **distincts**.

Si la liste compressée n'est pas valide, la fonction doit retourner `None`.

2) Écrire la fonction `compresser_liste` qui accepte une liste `liste` et retourne une nouvelle liste représentant la liste compressée `liste_compressee` de cette liste. La liste passée en paramètre n'est pas modifiée.

- Il n'y a **pas de pré-condition** pour cette fonction car n'importe quelle liste d'entiers doit pouvoir être traitée.
- La **post-condition** est `decompresser_liste(liste_compressee)==liste`

3) On définit le taux de compression associée une méthode de compression par :

$$T = 1 - \frac{\text{taille de l'image compressée}}{\text{taille de l'image originale}}$$

a) Calculer le taux de compression sur l'image 8x8 pixels *(dernière ligne du fichier python)*

b) Afficher le résultat sous forme de pourcentage.

Exercice 9 - principe des tiroirs ★★

Le **principe des tiroirs** de Dirichlet, affirme que, si n chaussettes sont rangées dans m tiroirs, et que $m < n$, alors il y a un tiroir qui contient au moins deux chaussettes.

Nous nous inspirons dans cet exercice de ce principe, afin d'écrire deux fonctions :

- La fonction `trouver_elements_identiques` accepte en paramètre une liste de longueur `n` d'entiers choisis parmi `n-1` valeurs (de `0` à `n-2`) et a pour objectif de trouver deux indices distincts de la liste où l'on observe la même valeur.
- La fonction `trouver_valeur_absente` accepte en paramètre une liste de longueur `n` d'entiers choisis parmi `n+1` valeurs (de `0` à `n`) et a pour objectif de trouver quel entier entre `0` et `n` est absent de la liste.

1) Dans le fichier **ex09_tiroirs.py**, écrire les deux fonctions **auxiliaires** suivantes :

- La fonction `est_liste_bornee` accepte en paramètres une liste d'entiers `liste` ainsi que deux entiers `m1` et `m2` et retourne le booléen indiquant si tous les éléments `e` de `liste` vérifient l'inégalité `m1<=e<=m2`.
- La fonction `est_valeur_presente` accepte en paramètres une liste d'entiers `liste` ainsi qu'un entier `valeur` et retourne `True` si `valeur` est présente parmi les éléments de `liste` et `False` sinon.

2) Pour décrire les conditions vérifiées par le paramètre `liste` de longueur `n` de la fonction `trouver_elements_identiques`, on précise que sa longueur vérifie :

```
n>=2
```

et que tous ses éléments sont compris entre `0` et `n-2`, ce que l'on peut encore écrire :

```
est_liste_bornee(liste,0,n-2)
```

On doit alors trouver deux indices distincts `i1` et `i2` vérifiant :

```
0<=i1<i2<n and liste[i1]==liste[i2]
```

Exemple : Si `liste=[2,0,3,4,5,3,1]`, on a `n=7`, et on peut vérifier que les éléments sont tous compris entre `0` et `7-2` (c'est-à-dire `5`). On constate qu'on a comme prévu (au moins) deux éléments égaux (ici de valeur `3`) aux indices `i1=2` et `i2=5`.

3) La condition vérifiée par le paramètre `liste` de longueur `n` de la fonction `trouver_valeur_absente` est uniquement :

```
est_liste_bornee(liste,0,n)
```

On doit alors trouver la valeur absente caractérisée par :

```
0<=valeur<=n and not est_valeur_presente(liste,valeur)
```

Exemple : Si `liste=[2,0,3,7,5,6,1]`, on a `n=7`, et on peut vérifier que les éléments sont tous compris entre `0` et `7`. On constate qu'on a comme prévu (au moins) une valeur absente, à savoir la valeur `4`.

Exercice 10 - les quatre carrés de Lagrange ★★★

`https://fr.wikipedia.org/wiki/Théorème_des_quatre_carrés_de_Lagrange`

Le **théorème des quatre carrés de Lagrange**, également connu sous le nom de **conjecture de Bachet**, s'énonce de la façon suivante :

> **Tout entier positif peut s'exprimer comme la somme de quatre carrés.**

Plus formellement, pour tout entier positif ou nul n, il existe des entiers a, b, c et d tels que :

$$n = a^2 + b^2 + c^2 + d^2$$

Dans le fichier **ex10_lagrange.py**, compléter la fonction `quatre_carres` qui accepte en paramètre un entier `n` et retourne quatre entiers `a`, `b`, `c` et `d`.

- **Pré-condition :** `n>=0`
- **Post-condition :** `a**2+b**2+c**2+d**2==n`