TP 2 – SPÉCIFICATION DE FONCTION

Info1.Algo1 - 2022-2023 Semestre Impair

Exercice 1 - 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 **ex01_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 2 - Indices dans liste

Dans cet exercice, les fonctions `indice_stricte_croissance` et `indice_minimum_local` à compléter dans le fichier **ex02_indices.py** prennent en paramètre une liste d'entier `liste` et retournent un indice `i` où un comportement particulier de la liste peut être observé. Dans chacun de ces deux cas, il n'y a pas nécessairement unicité de la solution : il peut exister plusieurs indices valides, l'essentiel étant d'en retourner un.

On appelle `n` la longueur de `liste`.

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

1) La fonction `indice_stricte_croissance` retourne un indice `i` où a lieu une croissance stricte :

- **Pré-condition :** `n>=2 and liste[0]<liste[n-1]`
- **Post-condition :** `1<=i<n and liste[i-1]<liste[i]`

2) La fonction `indice_minimum_local` retourne un indice `i` où a lieu un minimum local :

- **Pré-condition :** `n>=3 and liste[0]>=liste[1] and liste[n-2]<=liste[n-1]`
- **Post-condition :** `1<=i<n-1 and liste[i-1]>=liste[i] and liste[i]<=liste[i+1]`

Exercice 3 - Distance totale minimale

Une version romancée (et en anglais) de ce problème :

https://adventofcode.com/2021/day/7

Fonction fournie : Dans le fichier **ex03_minimum_somme_distances.py** est donnée la fonction `somme_distances` qui accepte en paramètres :

- une liste d'entiers `liste` (on appelle `n` sa longueur).
- une valeur `v`.

La fonction `somme_distances` retourne la somme des distances de chacun des éléments de `liste` à la valeur `v` c'est à dire :

$$|l_0 - v| + |l_1 - v| + ... + |l_{n-1} - v| = \sum_{i=0}^{n-1} |l_i - v|$$

où l_i est l'éléments d'indice i de la liste.

Travail à effectuer : compléter la fonction `minimiser_somme_distances` qui accepte en paramètre une liste d'entiers (de longueur `n`) et retourne une valeur `v`

telle que `somme_distances(liste,v)` admettent un **minimum local** en `v`, ce qui entraîne :

- **Pré-condition :** `n>0`
- **Post-condition :**

```
somme_distances(liste,v)<=somme_distances(liste,v-1)
and somme_distances(liste,v)<=somme_distances(liste,v+1)
```

Indication : On admettra que la valeur à chercher est nécessairement entre la plus petite et la plus grande valeur présente dans `liste`.

Pour aller plus loin : On pourrait prouver mathématiquement que le **minimum local** atteint par `somme_distances(liste,v)` est en fait un **minimum global**, ce qui signifie que la valeur de `somme_distances(liste,v)` est alors la plus petite possible, mais ce n'est pas l'objet de cet exercice.

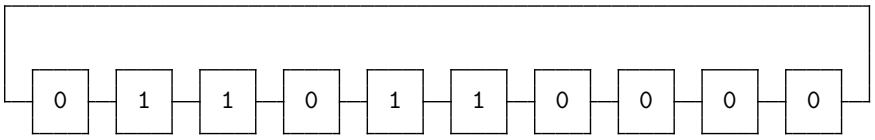
Exercice 4 - Une affaire de collier

Deux cambrioleurs viennent de s'emparer d'un précieux butin : un collier composé de deux types de perles très rares. Les perles de de chaque type sont en nombre pair, et nos deux cambrioleurs se disent qu'ils vont pouvoir se partager ce collier de manière équitable : autant de perles de chaque type pour chacun des deux acolytes.

Ils souhaitent aussi ne pas couper le collier en plus d'endroits que nécessaire lors du partage... Couper le collier en deux endroits seulement serait l'idéal!

Le butin

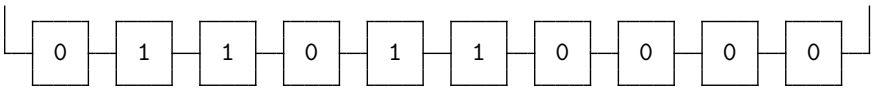
Un collier possible est le suivant :

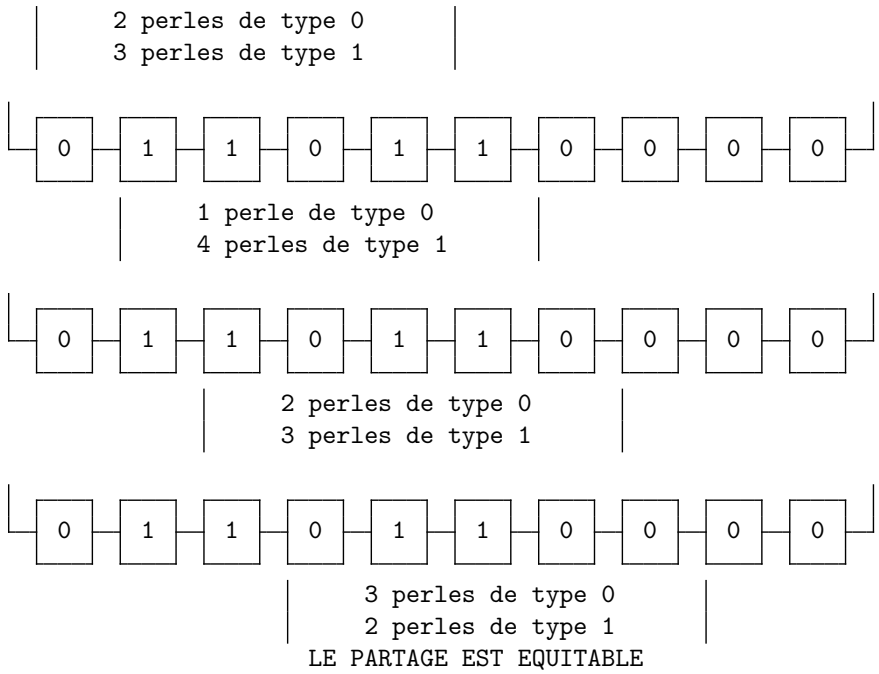


Il est composé de `10` perles, dont `6` de type `0` et `4` de type `1`. Il faudrait donc découper le collier en deux endroits seulement de façon à ce que chacun des cambrioleurs reparte avec `5` perles, dont `3` de type `0` et `2` de type `1`.

Principe

Le principe consiste à tester toutes les séquences de `5` perles successives (de la gauche vers la droite) et de s'arrêter lorsque le partage est équitable :





Une première solution

1) Dans le fichier **ex04_collier.py** compléter la fonction auxiliaire `est_collier_valide` qui accepte en entrée une liste d'entiers `collier` et retourne le booléen indiquant s'il s'agit ou non d'un collier valide, c'est-à-dire **ne contenant que des `0` (en nombre pair) et des `1` (en nombre pair)**.

2) Compléter la fonction `partager_collier` qui accepte en paramètre une liste `collier` représentant un collier valide *(pré-condition)* et retournant l'entier `i` représentant l'indice gauche du partage *(l'indice droit étant automatiquement obtenu puisque l'on doit couper en deux le collier)*

Exemple : dans le cas où `collier=[0,1,1,0,1,1,0,0,0,0]` la fonction `partager_collier` retourne `3` ce qui correspond au partage :

```
[0,1,1,0,1,1,0,0,0,0]
      |<------->|
```

Contraintes :

- Ne pas modifier la liste `collier` et les variables `n` et `s1`.
- Ne pas utiliser d'autre liste que la liste `collier` donnée en paramètre.
- Ne pas utiliser de fonctions élaborée de Python pour déterminer la somme (`sum`, ...).

Exercice 5 - 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.

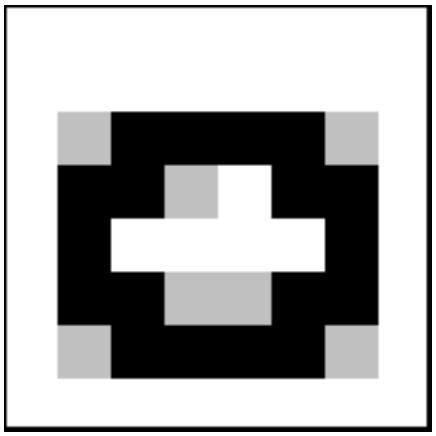


Figure 1: image à représenter

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

```
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 **ex05_rle.py**, compléter le code de la fonction `decompresser_liste`, qui accepte en paramètre une liste compressée et retourne la liste originale.

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` quelconque en entrée et renvoie la liste compressée `liste_compressee`.

- 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 6 - Les quatre carrés de Lagrange (pour aller plus loin)

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 **ex06_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`