Sujet n°3

Algorithmique et Structures de Données
Piles, files

`Temps de réalisation: 1h30`

Le TP n°3 a pour objectif de :
- implémenter des types abstraits à partir de différentes structures de données

Échelle de progression :

● « débutant·e » ● « confirmé·e » ● « avancé·e » ● « expert·e »

Ex.	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Pour acquérir un niveau de compétence donné, il faut achever **tous** les exercices de ce niveau et des niveaux inférieurs. L'objectif est d'atteindre le niveau *confirmé·e* (●) à l'issue du TP.

Table des matières

1 Les piles

La pile est un type abstrait dont l'interface est régie par l'acronyme **LIFO** (*Last-In-First-Out*), « Dernier arrivé, premier servi ! ».

Pour implémenter le type abstrait *pile*, il existe différentes options, comme vu dans le cours.

Pour rappel, les primitives de gestion des piles sont les suivantes :

— *Push* : ajouter un élément en fin de pile
— *Pop* : retirer le dernier élément de la pile et renvoyer sa valeur
— *Top* : renvoier la valeur du dernier élément de la pile
— *Size* : renvoier le nombre d'éléments
— *isEmpy* : indiquer si la pile est vide

Exercice 1: La pile à partir d'une list

Une façon très simple de procéder en Python est de remarquer que les `list`, ces « super tableaux », disposent déjà de l'interface appropriée pour les piles :

— *Pop* est fourni par `l.pop()` : attention toutefois à la valeur de retour ;
— *Push* devient `l.append(.)` ;
— *Top* revient à consulter `l[-1]` ;
— la condition *isEmpty* est simplifiée en `if l` ;
— et *Size* est donné par `len(l)`.

QUESTION 1. Mettre en place une pile à partir d'une `list` et tester l'ensemble de ses opérations sur un exemple réel.

Exercice 2: La pile à partir d'une classe

L'utilisation d'une `list` pour implémenter une pile a des avantages car permet de bénéficier des performances de la structure de données `list`. Par contre, cette implémentation ne nous permet pas d'assurer l'accès aux éléments imposé par la structure d'une pile ; en effet, on peut acccéder, par exemple, à l'élément sur la position `1` de la liste.

Pour résoudre ce problème, nous pouvons mettre en place une classe `Pile` qui contient un attribut d'instance `list` et qui permet, à travers ses méthodes, de réaliser les opérations sur la pile.

QUESTION 1. Définir la classe `Pile` contenant un attribut d'instance `list` et implémenter le constructeur.

QUESTION 2. Implémenter les primitives de gestion des piles - chanque primitive dans une méthode séparée.

QUESTION 3. Tester votre classe à travers plusieurs instances et opérations.

Exercice 3: Notation polonaise inverse (à l'aide d'une pile à partir d'une classe)

La notation polonaise inverse, en référence au mathématicien polonais ŁUKASIEWICZ, est une forme d'écriture des expressions arithmétiques. Plutôt que d'adopter une notation infixe, dans laquelle l'opérateur est entre les deux opérandes, comme dans $10 + 2$, on utilise une écriture postfixe (ou postfixée) :

```
10 2 +
```

Cette notation est fondée car l'arité de chaque opérateur est connue (opérateurs binaires uniquement, parmi $\{+, -, \times, \div\}$).

Les opérandes peuvent être eux-mêmes des expressions. Par exemple, l'expression $(10+2)/3$ s'écrit en notation polonaise inverse :

```
10 2 + 3 /
```

Notez l'abandon salvateur des parenthèses...

QUESTION 1. Quelle est l'expression infixe correspondant à l'écriture postfixée

```
4 3 - 2 * 5 2 / +
```

Remarque: *Les opérateurs commutatifs (`+` et `*`) mènent à plusieurs expressions équivalentes :*

```
4 3 - 2 * 5 2 / +  :=  2 4 3 - * 5 2 / +
                   :=  5 2 / 2 4 3 - * +
                   :=  5 2 / 4 3 - 2 * +
```

QUESTION 2. Écrire sous forme postfixée l'expression suivante

$$((3-4)/(6-12)) * ((7+2)/6)$$

Nous allons implémenter une méthode permettant d'évaluer la valeur d'une expression sur des nombres entiers [1] en notation polonaise inverse, c'est-à-dire de calculer son résultat. Pour cela, nous utiliserons une pile, appelée *pile de résultats*.

L'algorithme accepte un à un les nombres et les opérateurs de l'expression postfixée. Lorsqu'il rencontre un nombre, il l'empile. Si c'est un opérateur, il dépile deux nombres et réalise l'opération, puis il empile le résultat.

QUESTION 3. Dérouler l'algorithme à la main et observer son comportement sur l'exemple de la question 1. Si l'expression est complète, la pile ne contient qu'un seul élément à la fin du calcul : c'est le résultat définitif de l'opération. Constater que l'algorithme est correct.

QUESTION 4. Implémenter une fonction d'évaluation d'une expression arithmétique postfixe, saisie et traitée élément par élément au clavier. Une saisie vide indique la fin du traitement. Il est alors attendu l'affichage de l'état de la pile. Par exemple :

```
: 10
: -2
: +
: 3
: /
:
[2]
```

Exercice 4: La pile à partir du tableau

Nous allons proposer une implémentation des piles d'entiers à l'aide de « vrais tableaux », disponibles grâce au type personnalisé `ArrayList`. Pour faire cet exercice, vous devez d'abord réaliser l'Exercice 3.

QUESTION 1. Définir la structure de données du type personnalisé `Stack` de pile d'entiers, en réutilisant par composition le type `ArrayList`. Un point de vigilance concerne la quantité limitée de places dans la pile, imposée par la taille fixe du tableau. Il s'agit donc d'une version du type abstrait *pile* contrainte par une capacité maximale.

QUESTION 2. Implémenter les primitives de gestion des piles exclusivement à partir des opérations disponibles sur les tableaux (`ArrayList`) :

1. L'opérateur de division se traduit donc par une division euclidienne , de symbole "`//`" en Python.

```python
def s_new(n: int = 10) -> Stack
def s_size(s: Stack) -> int
def s_is_empty(s: Stack) -> bool
def s_str(s: Stack) -> str
def s_push(s: Stack, item: int) -> Stack
def s_pop(s: Stack) -> Stack
def s_top(s: Stack) -> int | None
```

Exercice 5: Notation polonaise inverse (à l'aide d'une pile à partir d'un tableau)

QUESTION 1. Résoudre le problème et répondre aux questions en utilisant une pile à partir d'un tableau.

2 Les files d'attente

Les files sont le pendant des piles, et suivent un principe **FIFO** (*First-In-First-Out*). Elles reprennent l'adage « Premier arrivé, premier servi ! », propre à caractériser toute file d'attente, comme la queue d'accès au R.U.

Pour implémenter une file d'attente, il existe différentes oprtions, comme vu dans le cours.

Pour rappel, les primitives de gestion des files d'attente sont les suivantes :

— *Enqueue* : insèrer un élément en fin de file
— *Dequeue* : retirer le premier élément de la file et renvoyer sa valeur
— *Front* : renvoyer la valeur de l'élément en début de file
— *Rear* : renvoyer la valeur de l'élément en fin de file
— *Size* : renvoyer le nombre d'éléments
— *isEmply* : indiquer si la file est vide

Exercice 6: La file d'attente à partir d'une liste

Une façon très simple de procéder en Python est de remarquer que les `list`, ces « super tableaux », disposent déjà de l'interface appropriée pour les files, à l'aide des méthodes `.pop()` et `.append(.)`, de la fonction `len()` et de l'opérateur `[]`.

QUESTION 1. Mettre en place une file d'attente à partir d'une `list` et tester l'ensemble de ses opérations sur un exemple réel.

Exercice 7: La file d'attente à partir d'une classe

L'utilisation d'une `list` pour implémenter une file a des avantages car permet de bénéficier des performances de la structure de données `list`. Par contre, cette implémentation ne nous permet pas d'assurer l'accès aux éléments imposé par la structure d'une file ; en effet, on peut acccéder, par exemple, à l'élément sur la position `1` de la liste.

Pour résoudre ce problème, nous pouvons mettre en place une classe `File` qui contient un attribut d'instance `list` et qui permet, à travers ses méthodes, de réaliser les opérations sur la file d'attente.

QUESTION 1. Définir la classe `File` contenant un attribut d'instance `list` et implémenter le constructeur.

QUESTION 2. Implémenter les primitives de gestion des files - chanque primitive dans une méthode séparée.

QUESTION 3. Tester votre classe à travers plusieurs instances et opérations.

Exercice 8: E l i f

Écrire, à l'aide des seules opérations de la file (implémentée à partir d'une classe), une fonction récursive qui inverse tous les éléments d'une file d'entiers.

Astuce: *L'idée est de sortir le premier élément, puis d'inverser récursivement le reste de la file, et enfin de le réinsérer.*

Exercice 9: Problème de Joséphus

Résoudre le problème de Joséphus, qui s'énonce ainsi : soient n condamnés, identifiés de 1 à n et réunis en cercle ; le bourreau décide d'exécuter un condamné tous les k condamnés, en suivant la ronde à partir du condamné n°1. D'humeur badine, le bourreau promet la vie sauve au dernier condamné survivant ! Le problème consiste alors à trouver au plus vite la place qu'il faut prendre pour rester en vie à la fin de ce jeu macabre.

Utilisez des files d'attentes implémentées à partir d'une classe.

Astuce: *Une résolution du problème passe par l'utilisation d'une file de condamnés encore en vie...*

Une variante consiste à supposer que le bourreau est magnanime, et qu'il propose de sauver non pas 1 condamné mais les m derniers survivants de son « jeu ».

Exercice 10: La file d'attente à partir du tableau

Nous pourrions distraitement considérer que le travail est similaire à celui des piles, pour disposer d'un type personnalisé de file d'entiers implémenté à partir des `ArrayList`. C'est effectivement le cas...à une différence essentielle près : comme la file augmente d'un côté et diminie de l'autre, le tableau sous-jacent se remplit – du début vers la fin – et se vide – du début vers la fin également – au point de créer une situation absurde où il n'y aurait plus qu'un élément dans la file d'attente, se trouvant à la toute fin du tableau et interdisant de ce fait l'insertion de nouveaux éléments !

Une première façon de régler le problème consiste à opérer au besoin un décalage des éléments du tableau pour les ré-indicer à partir du début. Évidemment, cette résolution induit des performances très mauvaises à catastrophiques. Une façon plus élégante et efficace de procéder consiste à « brancher » la fin sur le début du tableau de sorte à créer un tampon circulaire. Dans ce cas, il est nécessaire de retenir en permanence le début et la fin effectives des éléments de la file dans le tableau.

Définir un type personnalisé `Queue` de file circulaire d'entiers et implémenter les primitives de gestion des files, en réutilisant le type `ArrayList` :

```
def q_new(n: int = 10) -> Queue
def q_size(q: Queue) -> int
def q_is_empty(q: Queue) -> bool
def q_str(q: Queue) -> str
def q_enqueue(q: Queue, item: int) -> Queue
def q_dequeue(q: Queue) -> Queue
def q_front(q: Queue) -> int | None
def q_rear(q: Queue) -> int | None
```

Exercice 11: E l i f et Problème de Joséphus à l'aide d'une file à partir d'un tableau

Question 1. Résoudre les problèmes et répondre aux questions en utilisant une file à partir d'un tableau.

3 Les tableaux en Python

Python ne propose pas de structure de données *tableau* native, mais dispose du type `list` –sorte de tableau dynamique– qui se conforme à la spécification du type abstrait *séquence*.

La perspective ici est de construire un type personnalisé pour implémenter le type abstrait *séquence d'entiers* à l'aide d'une structure de données « tableau d'entiers ». Pour ce faire, nous utiliserons la fonction `alloc` suivante qui alloue en mémoire un ensemble de m cases contigues, chacune de la taille d'un entier et par conséquent, qui créé un tableau d'entiers de taille m :

```
from ctypes import Array, c_int

def alloc(m: int) -> Array:
  IntArrayType = c_int * m  # création d'un type "tableau de m entiers"
  return IntArrayType()     # déclaration et initialisation à zéro du tableau
```

Copier la fonction `alloc` ci-dessus dans la bibliothèque `tp0.util`. L'usage de cette fonction est illustré ci-dessous :

```
>>> from tp0.util import alloc
>>> tab: Array = alloc(5)
>>> print(tab)
<__main__.c_int_Array_5 object at 0x1041442c0>
>>> tab[2] = 2
>>> for i in range(len(tab)):   # alt.: for i, val in enumerate(tab):
>>> ... print(i, tab[i])
0 0
1 0
2 2
3 0
4 0
```

Remarque: *Contrairement aux entiers natifs de Python, les `c_int` ont une taille fixe de 32 bits, et donc une plage de valeurs limitée.*

Remarque: *Un tableau `tab` de type `ctypes.Array` dispose des opérateurs `tab[i]`, `tab[i]=` et `len(tab)`.*

Exercice 12: Quelques fonctions utilitaires

À titre exceptionnel, les fonctions de cet exercice seront définies dans la bibliothèque `tp0.util`, qui contient déjà la fonction `saisir_entier`, plutôt que dans le paquet `tp3` [2].

Question 1. Écrire la fonction

```
def saisir_tableau(n: int) -> Array
```

qui permet de remplir un tableau (de type `ctypes.Array`) avec n nombres entiers saisis un à un au clavier par l'utilisateur.

Question 2. Écrire la fonction

2. à créer et à alimenter avec les programmes du TP3.

```
def remplir_tableau(n: int, a: int = 0, b: int = 100) -> Array
```

qui génère un tableau de n entiers tirés aléatoirement dans $[\![a, b]\!]$.

Remarque: *La fonction* `randint()` *du module* `random` *réalise des tirages aléatoires.*

3.1 Petits défis

Exercice 13: Inversion

Demander à l'utilisateur de saisir un tableau de nombres puis l'afficher. Construire le tableau qui permute tous les nombres –le premier devient le dernier, etc.– et l'afficher également.

Astuce: *La fonction* `tp0.util.saisir_tableau()` *produit le tableau initial.*

Exercice 14: Nombres pairs et impairs

QUESTION 1. Écrire un programme qui, à partir d'un tableau de nombres entiers compris entre 0 et 100, recopie tous les nombres pairs au début d'un nouveau tableau, et les nombres impairs à la fin de ce nouveau tableau.

QUESTION 2. Écrire une seconde version du programme sans utiliser de tableau supplémentaire, i.e., opérer un partitionnement « en place ».

3.2 Définition des `ArrayList`

Remarque: *À partir d'ici et jusqu'à la fin des séances de TP, la démarche pédagogique consiste à découvrir la « machinerie » sous-jacente aux structures de données usuelles, ainsi que leurs avantages comparés. L'enjeu consiste à vous permettre par la suite de faire des choix éclairés lors de la résolution de problèmes complexes, parmi les très nombreuses options d'implémentation existantes.*

Ainsi, les types personnalisés que vous allez créer, et leurs opérations, sont en règle générale déjà disponibles aussi bien en Python que dans la très grande majorité des langages de programmation.

Exercice 15: Vers un type de listes d'entiers à base de tableaux statiques

La structure de *tableau* suggère une taille fixe m spécifiée à l'initialisation. La manière usuelle d'introduire un peu de souplesse dans la gestion des tableaux [3], consiste à définir la taille m comme la *capacité maximale* du tableau. Cela revient à introduire le symbole *indéterminé* ($\perp$) et considérer que l'on peut construire des tableaux $(t_0, \ldots, t_{n-1}, \perp, \ldots, \perp)$ de taille n, avec $n \leq m$.

À propos des `dataclasses`

La construction `dataclass` du langage Python permet de créer des *structures de données* définissant de nouveaux types de variables complexes. Ces structures de données sont construites comme des tuples, avec un ou plusieurs *champs* nommés (aussi appelés *attributs*), chacun de type quelconque.

Munies d'opérations, les structures de données construites à l'aide de `dataclass` définissent de nouveaux types personnalisés. Voici par exemple la définition du type *Point* composé de 2 attributs, ses coordonnées cartésiennes x et y :

3. Autoriser des opérations limitées d'ajout et de suppression d'éléments.

```python
from dataclasses import dataclass

# création du nouveau type Point
@dataclass
class Point:
    x: int
    y: int = 0    # y a une valeur par défaut (0), pas x. C'est idiot mais possible
```

Une fois défini, le nouveau type `Point` peut bien entendu servir dans les programmes, au même titre que tout autre type natif Python. Voici un exemple, à reproduire et à tester :

```
>>> p1 = Point(1)    # création du point (1,0). y prend la valeur par défaut (0)
>>> p2 = Point(3, 4) # création du point (3,4).
>>> print(p1)
Point(x=1, y=0)
>>> print(f'p2=({p2.x}, {p2.y})')
p2=(3, 4)
>>> p1.y = 2
>>> dist: int = (p2.x - p1.x) + (p2.y - p1.y)
>>> print(dist)
4
```

Notez la spécificité de l'initialisation des variables p_1 et p_2, qui nécessite l'usage du nom du type `Point` suivi d'une séquence de paramètres correspondant aux valeurs de chaque attribut, *dans l'ordre de leur déclaration*. L'analogie avec un appel de fonction se justifie par le fait que l'initialisation d'un objet complexe invoque systématiquement une fonction particulière appelée *constructeur* dans le vocable des langages à objets.

L'affectation et l'accès individuel aux champs d'un objet—une variable—complexe sont rendus possibles par la « notation pointée » ('·'), comme dans l'exemple précédent avec `p1.y=` ou encore `p2.x`.

QUESTION 1. Proposer un type personnalisé `ArrayList` à l'aide de la construction `@dataclass`[4] de Python pour représenter des tableaux[5] d'entiers de capacité `max_size` fixe. La taille effective du tableau, i.e., le nombre d'éléments, sera également enregistrée dans la structure.

Remarque: *Si vous êtes déjà à l'aise avec la programmation par objets en Python, vous pouvez poursuivre la série de TPs avec de véritables classes, plutôt que des dataclasses. Les changements à apporter aux énoncés sont les suivants :*

1. *déclarer et initialiser les variables d'instance dans le constructeur* `__init__()` *;*
2. *déclarer tous les opérateurs en tant que méthodes (et supprimer le préfixe, par exemple* `al_`*), en utilisant systématiquement* `self` *(par convention) comme nom de variable pour le premier argument.*

Partout où c'est pertinent et si vous savez ce que vous faites (!), vous êtes autorisés à surcharger des opérateurs existants. Par exemple `__len__()` *pour* `al_len()` *ou encore* `__getitem__()` *pour* `al_get()`*, etc.*

4. Consulter la documentation officielle à propos des `dataclasses`.
5. Par commodité, tableau désigne ici le type abstrait ou la structure de données de manière indifférenciée.

Question 2. Écrire la fonction « constructeur »

```
def al_new(m: int = 10, l: list[int] = None) -> ArrayList
```

qui créé un nouvel objet de type `ArrayList` à partir d'une capacité maximale m et d'une liste initiale ℓ d'entiers fournies en paramètres.

Question 3. Écrire les fonctions

```
def al_len(tab: ArrayList) -> int
def al_is_empty(tab: ArrayList) -> bool
```

qui resp. retourne la taille effective –équivalent à `len(tab)`– du tableau, i.e., le nombre d'éléments, et qui indique si le tableau est vide.

Question 4. Écrire la fonction

```
def al_str(tab: ArrayList) -> str
```

qui propose une représentation du tableau d'entiers sous forme de chaîne. La notation « naturelle » des tableaux est de rigueur : `"[0, 1, 2, 3]"`.

Exercice 16: Opérations sur les tableaux d'entiers

On s'intéresse dans cet exercice à la définition des opérations élementaires sur les tableaux d'entiers, de sorte à compléter la définition du type `ArrayList`.

Question 1. Écrire la fonction

```
def al_get(tab: ArrayList, i: int) -> int
```

qui retourne l'élément à la position $i \in [\![0, n-1]\!]$. Cette opération est le pendant de `tab[i]`. Pour un tableau de taille[6] n, combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

Question 2. Écrire la fonction

```
def al_set(tab: ArrayList, i: int, item: int) -> ArrayList
```

qui affecte la nouvelle valeur `item` à l'élément en position $i \in [\![0, n-1]\!]$ du tableau `tab` (équivalent à `tab[i]=item`). Pour un tableau de taille n, combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

Remarque: *Il est souvent pratique de retourner la liste elle-même à la fin d'une fonction avec effet secondaire*[7]*. Cela permet notamment d'imbriquer les appels, comme dans l'exemple suivant :*

```
>>> tab: ArrayList = al_new(10, [1,2,3])       # tableau [1,2,3] de capacité 10
>>> al_set(al_set(al_set(tab, 0, 0), 1, 1), 2, 2)
[0,1,2]
```

Les réponses formelles aux questions de coût de cet exercice doivent être consignées dans un tableau (sic) comme celui-ci :

6. Il s'agit ici de la taille effective, et non de la capacité.
7. Fonction qui modifie la liste.

opération	tableau		...	
	meilleur	pire	meilleur	pire
`get`	1			
`set`	1			
`lookup`				
`remove`				
`insert`				
`prepend`				
`append`				
`extend`				

Remarque: *Les colonnes suivantes (...) seront remplies au fur et à mesure des TPs, avec les structures de données rencontrées.*

Question 4. Écrire la fonction

```
def al_lookup(tab: ArrayList, item: int) -> int | None
```

qui recherche la première occurrence de l'élément `item` et qui retourne son indice si la recherche aboutit, `None` sinon. Combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

Question 5. Écrire la fonction

```
def al_remove(tab: ArrayList, i: int) -> ArrayList
```

qui supprime l'élément en position $i \in [\![0, n-1]\!]$ dans le tableau `tab`. Combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

Question 6. Écrire la fonction

```
def al_insert(tab: ArrayList, i: int, item: int) -> ArrayList
```

qui insère, avec décalage des éléments existants, le nouvel élément `item` à la position $i \in [\![0, n]\!]$ du tableau `tab`[8]. Il est important de définir le comportement du tableau lorsque celui-ci est plein et qu'une opération d'insertion est demandée. On pourra, pour cela, lever une exception de type `OverflowError`. L'instruction –à tester dans la console– est la suivante :

```
>>> raise OverflowError("Débordement de capacité")
```

Pour un tableau de taille n, combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

Question 7. Décliner à partir de `tab_insert` les fonctions

```
def al_prepend(tab: ArrayList, item: int) -> ArrayList
def al_append(tab: ArrayList, item: int) -> ArrayList
```

qui ajoutent le nouvel élément `item` resp. au début et à la fin du tableau `tab`.

Avec un tableau de taille n et pour chacune des deux fonctions ci-dessus, combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

Question 8. Écrire la fonction

8. Insérer en position n signifie insérer à la fin du tableau.

```
def al_extend(tab1: ArrayList, tab2: ArrayList) -> ArrayList
```

qui concatène deux tableaux. Les données du tableau `tab2` sont ajoutées à la fin du tableau `tab1`. Combien d'opérations sont nécessaires dans le meilleur des cas ? Dans le pire des cas ?

3.3 `ArrayList` à l'usage

Ici, nous proposons quelques premiers problèmes à traiter à l'aide du type `ArrayList` défini précédemment. Notez que, dans le cas des tableaux d'entiers, nous pourrions tout aussi bien résoudre les problèmes proposés en utilisant le type `list[int]` natif de Python : nous n'avons fait que construire un type personnalisé pour implémenter le type abstrait *séquence d'entiers*, comme alternative au type `list[int]` existant. Mais nous verrons par la suite que la construction de types personnalisés ouvre de très belles perspectives.

Exercice 17: La chasse aux doublons

Il s'agit de trouver les éléments dupliqués dans un tableau d'entiers de taille n, dont les éléments sont tirés dans l'intervalle $[\![0, n-1]\!]$.

Une manière naïve consiste à examiner chaque élément du tableau et à le comparer systématiquement à tous les autres. Le coût de cet algorithme est quadratique [9] ($\mathcal{O}(n^2)$). Il existe néanmoins une version beaucoup plus économe ! Le principe de l'algorithme est alors le suivant :

— Pour chaque indice i du tableau t, trouver l'élément à la position $|t[i]|$ (où $|.|$ désigne la valeur absolue) :
- s'il est positif, inverser sa valeur ($t[k] = -t[k]$, avec $k = |t[i]|$) ;
- s'il est négatif, on peut conclure que l'élément en position i est un doublon !

Question 1. Dérouler l'algorithme à la main sur le tableau $[1, 2, 2, 3, 1]$.

Question 2. Implémenter cet algorithme pour les objets de type `ArrayList`, en prenant soin de *ne pas* écraser le tableau original.

Remarque: *La valeur 0 (zéro) doit être traitée de manière spécifique, pour distinguer le « zéro positif » du « zéro négatif ». Fixer arbitrairement $-0 := -n$ peut être une solution.*

Question 3. Quel est le coût de l'algorithme ?

9. À retrouver par soi-même.