

# Sujet n°0

## Algorithmique & Programmation Découverte de Python et PyCharm

Temps de réalisation: 1.5h

L'enjeu principal du cours est d'apprendre les fondements du langage Python, de l'algorithmique et des structures de données. Nous utiliserons le langage **Python**, version **3.8+**, comme support d'implémentation et le logiciel **PyCharm** comme environnement de programmation.

Le TP n°0 a donc pour objectif de :

- ▶ se familiariser avec les rudiments du langage Python ;
- ▶ prendre en main l'IDE PyCharm ;
- ▶ écrire et interpréter un programme ;
- ▶ appréhender les *fonctions*, *modules* et *paquets*, leur création et leur utilisation ;
- ▶ mettre en œuvre les **docstring** ;
- ▶ explorer un outil de test.

L'échelle de progression est la suivante :

	● « débutant·e »				● « confirmé·e »				● « avancé·e »				● « expert·e »	
Ex.	01	02	03	04	05	06	07	08	09	10	11	12	13	
	●	●	●	●	●	●	●	●	●	●	●	●	●	

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

Votre meilleure amie pour la réalisation des TPs est la documentation officielle de Python :

<https://docs.python.org/3/>

Python » French » 3.10.5 » 3.10.5 Documentation »

Recherche rapide Go | modules | index

Choisissez la langue et la version de votre interpréteur, puis utilisez le champ de recherche.

## Table des matières

<b>1</b>	<b>Environnement de développement</b>	<b>2</b>
1.1	L'interpréteur Python	3
	Exercice 1 : Interpréter des instructions	3
1.2	Variables et affectations	4
	Exercice 2 : Types et variables	4
1.3	Écrire des programmes	4
	Exercice 3 : Hello World!	4
1.4	Débogage avec PyCharm	5
	Exercice 4 : Pas-à-pas	6

<b>2</b>	<b>Premières fonctions, premiers problèmes</b>	<b>6</b>
	Exercice 5 : Un peu d'arithmétique . . . . .	6
	2.1 Les fonctions d'entrée / sortie . . . . .	7
	Exercice 6 : Attraper la saisie! . . . . .	7
	Exercice 7 : Inventaire à la Prévert . . . . .	7
<b>3</b>	<b>Les modules et les paquets</b>	<b>8</b>
	Exercice 8 : Importer des modules . . . . .	8
<b>4</b>	<b>Les collections Python : list, tuple, set, dict</b>	<b>9</b>
	Exercice 9 : Lancer de rayon! . . . . .	9
	Exercice 10 : Analyse fréquentielle . . . . .	10
<b>5</b>	<b>Pour aller plus loin. . .</b>	<b>11</b>
	Exercice 11 : Les docstrings . . . . .	11
	Exercice 12 : Les tests unitaires . . . . .	11
	5.1 Vers l'infini et au-delà. . . . .	12
	Exercice 13 : Quelques outils supplémentaires . . . . .	12

## 1 Environnement de développement

PyCharm est un environnement de développement (*Integrated Development Environment* - IDE) pour Python, V2 et V3, disponible gratuitement en version *Community*<sup>1</sup>. C'est un IDE professionnel, dédié à la programmation en Python. Il vous permettra rapidement de disposer d'un environnement de développement complet, notamment sur vos machines personnelles.

Bien que PyCharm soit très riche, son utilisation est plutôt intuitive. Outre la fenêtre principale d'édition de fichier, avec coloration syntaxique, complétion, indentation automatique, etc., l'interface se divise en plusieurs fenêtres, accessibles via le menu [View/Tool Windows], parmi lesquelles :

- la fenêtre **Project** offre un aperçu des contenus –répertoires, fichiers, bibliothèques externes, etc.– du projet en cours ;
- la fenêtre **Python Console** permet d'exécuter des instructions Python ligne à ligne,
- La fenêtre **Run**, proposée lorsqu'un programme est exécuté, diffuse la sortie standard d'un programme ;
- la fenêtre **Debug** permet de paramétrer et contrôler l'exécution du débogueur.

La documentation officielle (guides, tutoriels, tips, etc.) pour Pycharm se trouve ici :

<https://www.jetbrains.com/fr-fr/pycharm/learn/>.

### Mon premier projet avec PyCharm

Pour débiter avec PyCharm, il est nécessaire de créer un projet. Nous vous invitons à créer un **unique projet pour toute la série des TPs**, à la manière d'un *cahier d'exercices*. Ce sera ensuite bien plus facile pour naviguer dans vos sources et réexploiter des portions de code d'un TP à l'autre.

#### 1. Ouvrir PyCharm :

- MS-Windows : menu Démarrer > PyCharm
- Linux : exécuter le script `pycharm-community` dans le répertoire `/snap/pycharm-community/current`

#### 2. Créer un nouveau projet dans la fenêtre *Welcome to PyCharm* avec le bouton **New Project**.

---

1. En tant qu'étudiant-e, vous avez accès à une licence éducative pour tous les produits JetBrains! Profitez-en : <https://www.jetbrains.com/fr-fr/community/education/#students>

- (a) Sélectionner l'emplacement souhaité pour le projet et nommer le répertoire racine (`algo` ou `exercices` par exemple)
  - MS-Windows : créer le projet dans votre dossier personnel partagé ( `Z:` )
  - Linux : créer le projet dans le répertoire partagé `/media/EXXX`
- (b) Sélectionner l'interpréteur Python *pré-configuré*, soit Python 3.10 (MS-Windows), soit Python 3.9 (Linux)
- (c) Cliquer sur `Create`

Vous êtes fin prêt pour travailler en Python avec PyCharm.

## 1.1 L'interpréteur Python

La fenêtre de l'interpréteur peut être invoquée à l'aide du menu `View > Tool Windows > Python Console`. Vous pouvez y saisir des commandes interactives.

Cette fenêtre contient une ligne de commande symbolisée par 3 chevrons :

```
>>>
```

Dans cette ligne de commande, vous pouvez écrire un ordre –aussi appelé instruction–, puis envoyer cet ordre à l'ordinateur qui se charge de l'interpréter et de produire le résultat attendu. Par exemple, tapez la commande suivante :

```
>>> 1 + 1
```

Validez l'ordre avec la touche `Entrée` et observez le résultat que vous obtenez. Essayez maintenant d'exécuter les ordres suivants :

```
>>> print("Bonjour !")
>>> 23.2 * (2 + 6.3)
```

### Exercice 1: Interpréter des instructions

QUESTION 1. Prévoir le résultat que l'ordinateur livrerait s'il devait interpréter les ordres suivants :

```
>>> 1 > 2
>>> 5.0 / 2
>>> 5 / 2
>>> 5 // 2
>>> 5. // 2
>>> 5 % 2
>>> print("bonjour", 5*2)
>>> True and False
>>> True or False
>>> print( non_defini )
>>> True and non_defini
>>> False and non_defini
>>> non_defini and False
```

QUESTION 2. Confirmer vos prévisions en observant le résultat produit par ces commandes dans l'interpréteur.

## 1.2 Variables et affectations

### Exercice 2: Types et variables

QUESTION 1. Examiner la série de commandes ci-dessous. Prévoir chaque résultat puis le confirmer à l'aide de l'interpréteur Python de PyCharm.

```
>>> i = 2
>>> i = i + 4
>>> i
>>> print(i)
>>> j = 5
>>> i > j
>>> b = i == 9
>>> not b
>>> i != 9
>>> x, y = 1, 2.0
>>> x = x+y
>>> y = x**y
>>> x, y
```

QUESTION 2. Tester la séquence d'ordres suivante :

```
>>> z, t, u = 2, 'True', True or False
>>> type(z), type(t), type(u)
>>> z += 1.0
>>> z, type(z)
>>> z += t
>>> u, type(u)
>>> u -= z
>>> u, type(u)
```

et commenter les résultats.

Nous reviendrons sur la notion de *typage* dans la section suivante.

## 1.3 Écrire des programmes

### Exercice 3: Hello World!

Voici sur un exemple, la procédure pour créer et jouer un programme Python :

1. Sélectionner le répertoire racine du projet dans la fenêtre **Project**
2. Créer un nouveau fichier, nommé `hello.py`, avec **File > New ... > Python File**
3. Écrire ou recopier les 6 lignes suivantes dans le fichier `hello.py` :

```
text = 'Hello'
word = 'le monde'
if text == 'Bonjour':
    word = 0o7 # chiffre 0 (zéro), lettre 'o' et chiffre 7 (sept)
text += ' ' + word
print(text)
```

4. Enregistrer le fichier via le menu **File > Save All**

5. Exécuter le programme `hello.py` à l'aide de la commande Run `'hello'` du menu Run.

QUESTION 1. Reproduire la séquence précédente pour exécuter le programme `hello.py`. Si tout s'est bien déroulé, une fenêtre d'exécution doit indiquer la fin normale du programme (*exit code 0*) et afficher le résultat de l'instruction `print`.

QUESTION 2. Ajouter au programme `hello.py` les lignes suivantes :

```
print(f'Dissection de la chaîne de caractères {repr(text)} :')
for i, c in enumerate(text):
    print(i, c)
```

Enregistrer les modifications dans le fichier `hello.py` via le menu File > Save All. Vous pouvez à nouveau exécuter votre programme en suivant le menu Run > Run `'hello'` ou en cliquant sur le triangle vert (*play*). Observer et décrire ce qu'il se passe.

**Remarque:** Pour en savoir plus sur les chaînes de caractères formatées littérales (ou *f-strings*), veuillez vous reporter à la [documentation officielle](#).

QUESTION 3. Modifier à nouveau le programme `hello.py` en changeant la valeur de la variable `text` en `text = 'Bonjour'`. Exécuter le programme et proposer une explication à ce nouveau comportement. En particulier, pourquoi l'erreur observée n'a pas été détectée *avant* l'exécution ?

Pour poursuivre l'exemple, modifier la déclaration des variables `text` et `word` comme suit :

```
text: str = 'Bonjour'
word: str = 'le monde'
```

QUESTION 4. Passer la souris sur la valeur surlignée `0o7` et analyser l'indication fournie par PyCharm. Corriger en conséquence –ajouter des guillemets, par exemple–. Il est important de noter que cette correction a lieu *avant* toute exécution.

**Remarque:** Notez que ces annotations (`<var>: <type>`) n'ont pas valeur d'obligation et sont purement ignorées par l'interpréteur ! La plupart des IDE génériques (VS Code, Sublime Text, TextMate, Atom, Emacs, vi, etc.) ne proposent pas –encore– de vérification de type, mais disposent de mécanismes d'extension qui peuvent parfois y remédier. À défaut, la vérification de type peut-être réalisée à l'aide de la commande `mypy` du package du même nom.

## 1.4 Débogage avec PyCharm

Pour activer le débogueur, il faut poser préalablement un point d'arrêt (*Breakpoint*) sur une ligne de code du programme. En cliquant à droite du numéro de ligne dans la marge, vous devriez voir apparaître un point rouge qui matérialise ce *Breakpoint*. Ensuite, sélectionnez [Run/Debug `'xxx'`]. Le programme est exécuté jusqu'au premier point d'arrêt. La fenêtre du débogueur livre alors des informations utiles.

L'exécution se poursuit grâce à quelques actions usuelles à partir d'un point d'arrêt :

- **Resume Program** pour reprendre l'exécution, jusqu'au prochain point d'arrêt ou la fin du programme ;
- **Step Into** pour exécuter l'instruction suivante. Si l'instruction courante est un appel de fonction, le débogueur entre dans la fonction ;
- **Step Over** pour exécuter jusqu'à la ligne suivante du programme ;
- **Step Out** pour exécuter jusqu'à la fin de la fonction courante (très utile pour sortir d'une fonction si on y est entré par erreur).

Le panneau **Frames** affiche la *pile d'appels*, mais dans le cas d'un programme simple il n'y a qu'une seule ligne indiquant « l'appel » au programme principal.

Le panneau **Variables** indique à tout instant l'état des variables déclarées dans le programme. C'est une source d'information très intéressante pour corriger les anomalies.

## Exercice 4: Pas-à-pas

QUESTION 1. Copier dans un fichier `debug.py`, puis exécuter le programme suivant :

```
# suite de Collatz
u: int = 27
while u >= 1:
    if u % 2:          # si u est impair:
        u = 3 * u + 1  # alors u est multiplié par 3, + 1
    else:              # si u est pair:
        u //= 2        # alors u est divisé par 2
print('La conjecture de Syracuse est vérifiée !')
```

Quel est le comportement observé ?

QUESTION 2. Relever les dix premières valeurs successives de  $u$  à l'aide de la commande *Resume Program* du débogueur, une fois posé un point d'arrêt sur l'instruction `while`.

QUESTION 3. Rendez-vous directement à l'itération de la boucle `while` pour laquelle  $u = 1$ , grâce à un point d'arrêt conditionnel<sup>2</sup>. Ensuite, une exécution pas à pas (`F8`) permet de relever les valeurs suivantes de  $u$  et détecter l'anomalie.

QUESTION 4. Corriger le programme, en prenant des informations sur la conjecture de SYRACUSE si nécessaire.

## 2 Premières fonctions, premiers problèmes

### Exercice 5: Un peu d'arithmétique

QUESTION 1. Copier dans un fichier `arithmetique.py` et corriger, à l'aide du vérificateur de type de PyCharm, le programme suivant :

```
def somme(n: int) -> int:
    return n*(n+1)/2

print(somme(3.))
```

QUESTION 2. Soit  $f(x) = 2x^2 - x + 1$  ; écrire la fonction  $f$  et un programme qui affiche le résultat de  $f(1)$ ,  $f(2)$  et  $f(3)$  dans un fichier `polynome.py`.

QUESTION 3. Dans un fichier `moyenne.py`, écrire une fonction `moyenne` qui prend en paramètres deux entiers et qui calcule la moyenne de ces deux entiers. Écrire un programme qui affiche la moyenne de 11 et 14, de 18 et 15, de 20 et 15.

QUESTION 4. La suite de l'exercice se déroule dans le fichier `arithmetique.py`.

Écrire une fonction `est_divisible_par` qui prend en paramètres deux entiers  $n$  et  $k$  et qui renvoie *vrai* si  $n$  est divisible par  $k$ , *faux* sinon. Écrire un programme qui affiche la divisibilité de 5 par 3, de 6 par 2 et de 9 par 3.

---

2. Clic droit sur le point d'arrêt pour définir une condition d'arrêt.

QUESTION 5. Écrire une fonction `est_pair` qui prend en paramètre un entier et qui renvoie *vrai* si l'entier est pair, *faux* sinon. La fonction `est_divisible_par` doit servir dans la définition de `est_pair`. Étendre le programme `arithmetique` pour afficher la parité des entiers 2, 4, 3 et 7.

QUESTION 6. Écrire une fonction `est_compris_dans` qui prend en paramètres trois entiers,  $a$ ,  $b$  et  $c$  et qui renvoie *vrai* si  $a$  est compris entre  $b$  et  $c$  (inclus). Ajouter au programme 5 tests qui couvrent les cas où  $a < b$ ,  $a = b$ ,  $b < a < c$ ,  $a = c$ ,  $a > c$ . Qu'arrive-t-il si  $b = c$ ?  $b > c$ ?  $a$  n'est pas un entier? ...

## 2.1 Les fonctions d'entrée / sortie

### Exercice 6: Attraper la saisie !

QUESTION 1. Dans un fichier `util.py`, écrire une fonction de collecte d'un nombre entier à partir du clavier, de signature :

```
def saisir_entier() -> int
```

À la suite de la fonction, écrire un programme qui affiche la valeur saisie au clavier.

QUESTION 2. Que se passe-t-il lorsque l'utilisateur saisit 'bla' (à tester) plutôt qu'un nombre entier? Grâce au bloc `try/except`, modifier la fonction pour traiter l'erreur `ValueError` et redemander la saisie tant que la valeur n'est pas entière. Un squelette de code est fourni ici :

```
try:
    # placer ici le code qui génère une ValueError
except ValueError:
    # sans erreur, le programme ne passe pas ici
    continue    # intercepte silencieusement l'erreur et poursuit le programme

# suite du programme...
```

QUESTION 3. Modifier la fonction pour rendre paramétrable l'invite de saisie. La signature devient alors :

```
def saisir_entier(invite: str = 'Saisir un nombre entier : ') -> int
```

QUESTION 4. Modifier à nouveau la fonction de telle sorte qu'elle accepte un paramètre supplémentaire –motif d'échappement– optionnel. Le comportement général de la fonction doit demeurer identique :

```
def saisir_entier(invite: str = 'Saisir un nombre entier : ',
                 escape: str | None = None) -> int | None
```

Le type de retour `int | None` indique que l'on attend un entier (`int`) ou `None`. En effet, si la saisie correspond au motif d'échappement (`escape`), alors la fonction termine et retourne `None`, plutôt que de lever une exception et demander une nouvelle saisie. L'usage principal du motif d'échappement est `escape=''` qui permet de sortir silencieusement sur une saisie vide.

### Exercice 7: Inventaire à la Prévert

Outre l'entrée (`input`) et la sortie (`print`) standards, les programmes Python peuvent aisément lire et écrire des fichiers de la mémoire externe :

```

data: str
with open('mes-données.txt', 'r') as infile: # ouverture du fichier (mode r: read)
    data = infile.read()                    # lecture du contenu du fichier

# ici, on réalise un traitement sur les données (data)...

with open('mes-résultats.txt', 'w') as outfile: # ouverture du fichier (w: write)
    outfile.write(data)                     # écriture des données dans le fichier

```

En 1946, Jacques PRÉVERT publie *Paroles*, un recueil de poèmes parmi lesquels figure le fameux *Inventaire*. Nous en avons un exemplaire sous la forme de fichier texte : `data_inventaire_prevert.txt`. Malheureusement, le scribe a totalement brouillé la mise en page !

Les questions de cet exercice seront traitées dans un fichier `prevert.py`.

QUESTION 1. Remettre le poème en forme dans un nouveau fichier `inventaire_prevert.txt`, en numérotant chaque ligne.

La version ainsi obtenue commence par :

```

01. Une pierre
02. deux maisons
03. trois ruines
04. quatre fossoyeurs
05. un jardin
...

```

**Astuce:** Il faut changer tous les caractères ‘;’ en ‘\n’ (caractère de saut de ligne) grâce à la fonction `str.replace()`, puis traiter les chaînes de caractères ligne par ligne `str.split()` pour ajouter les numéros au début de chaque ligne (opérateur de concaténation de chaînes ‘+’).

### 3 Les modules et les paquets

#### Exercice 8: Importer des modules

Reprenons, par exemple, le module `arithmetique` du fichier `arithmetique.py`. Il expose les symboles `somme`, `est_divisible_par`, `est_pair` et `est_compris_dans`. Le module `util` expose quant à lui le symbole `saisir_entier`.

Copier le programme suivant dans un fichier `main.py` :

```

import util
import arithmetique as art

n: int = util.saisir_entier()
if not art.est_pair(n):
    n += 1
print(f"Somme de 0 à {n} = {art.somme(n)}")

```

Le fichier `main.py` représente le programme que vous souhaitez exécuter. Ce programme utilise les fonctionnalités des modules `util` et `arithmetique` –abrégé en `art`–. Vous pouvez maintenant exécuter le programme `main` à l’aide du menu `Run > Run 'main'`.

QUESTION 1. Quel effet indésirable peut-on observer dans le résultat de l'exécution ?

Il existe une façon simple de tester les fonctions d'une bibliothèque que vous développez, avant de l'utiliser dans d'autres programmes. Il suffit d'ajouter une condition particulière en fin de fichier et d'y adjoindre le programme de test de votre choix. À titre d'exemple, déplacez le code de test du module `arithmetique` sous la condition suivante :

```
# section exécutée uniquement si le module *est* le programme principal
if __name__ == '__main__':
    # placer les tests (programme) ici
    # attention à l'indentation
    pass
```

Exécutez successivement les programmes `arithmetique` et `main`. Observez que l'exécution du programme `main` n'est plus altérée que par le sous-programme `util`.

QUESTION 2. Faire la modification nécessaire au programme `util` pour gommer définitivement les effets secondaires dans l'exécution de `main`.

QUESTION 3. Modifier le programme `main` pour qu'il continue d'utiliser les fonctions importées, mais sans le préfixe du nom de module.

Dans PyCharm, la création de paquet passe par le menu `File > New ... > Python Package`.

QUESTION 4. Créer un paquet `tp0` dans lequel vous allez verser tous les fichiers déjà créés lors du TP0.

**Remarque:** La modification du chemin d'accès aux fichiers source requiert de revoir les instructions d'importation car les modules sont recherchés à partir de la racine du projet. Par exemple, le module `util` devient `tp0.util` partout où il est importé.

Tous les programmes suivants seront créés dans le paquet `tp0`.

## 4 Les collections Python : list, tuple, set, dict

Ces quatre types natifs, combinés aux types primitifs `bool`, `int`, `float` et `str`, permettent de construire nombre de types dérivés tels que des ensembles de chaînes (`set[str]`), des listes d'entiers (`list[int]`), etc. Les chaînes de caractères (`str`) sont en fait des listes de caractères et se manipulent comme telles.

### Exercice 9: Lancer de rayon !

L'algorithme de « lancer de rayon » (*ray casting* en anglais) résoud le problème *point-in-polygon* pour déterminer si un point est intérieur à un polygone, convexe ou non.

Le principe est le suivant : à chaque fois qu'on traverse le contour du polygone en se déplaçant, on change systématiquement de côté, on entre ou on sort. En se déplaçant depuis l'infini –forcément à l'extérieur– vers notre point, soit on sort autant de fois qu'on entre –nombre pair d'intersections– soit on entre une fois de plus –nombre impair–, cf. Figure 9. Finalement, puisque l'ordre dans lequel on trouve les intersections ne change pas le résultat, il n'est pas nécessaire de trier les intersections et il suffit de parcourir et tester les arêtes du polygone dans l'ordre dans lequel elles sont données.

QUESTION 1. En utilisant la fonction fournie qui teste l'intersection entre une demi-droite horizontale d'origine donnée  $O$  et un segment  $[AB]$ , écrire un programme qui détermine si un point est à l'intérieur d'un polygone.

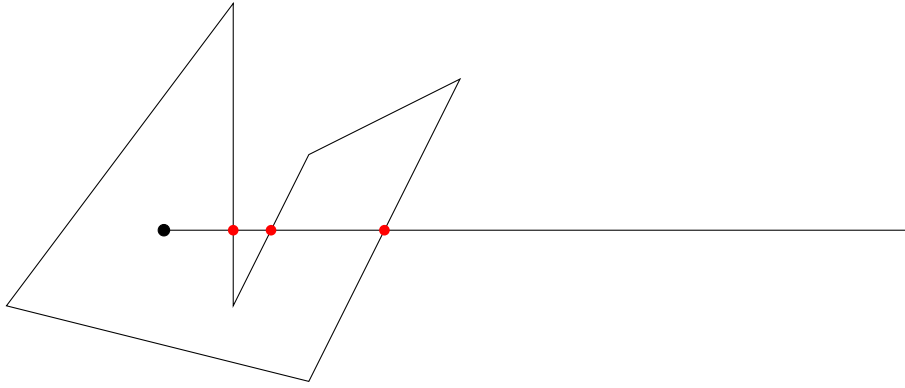


FIGURE 1 – Illustration du principe de l'algorithme de lancer de rayon  
. Le point est intérieur car la demi-droite dont il est issu traverse le polygone un nombre impair de fois.

```
Point = tuple[int, int]      # alias de type

def intersect(O: Point, A: Point, B: Point) -> bool:
    (xO, yO), (xA, yA), (xB, yB) = O, A, B
    return (
        (yO <= yA) == (yO > yB) and          # ordonnée dans l'intervalle
        xO < (xB - xA) * (yO - yA) / (yB - yA) + xA # point du bon côté
    )
```

## Exercice 10: Analyse fréquentielle

QUESTION 1. Créer un programme qui construit l'histogramme de fréquences des caractères apparaissant dans l'*Inventaire* de J. Prévert. Vous prendrez soin de supprimer les caractères de ponctuation et les formes accentuées pour ne retenir que 26 entrées au plus dans un tableau associatif (`dict`).

**Remarque:** La bibliothèque de fonctions dédiées au type natif `str` de Python est riche et peut sans doute offrir quelque'aide pour réaliser ce traitement.

**Astuce:** Pour constituer la liste des caractères à supprimer, il est facile de chercher dans le texte tous les caractères sauf les 26 entrées attendues !

QUESTION 2. Afficher l'histogramme trié selon la fréquence d'apparition des caractères.

QUESTION 3. Normaliser les valeurs produites en pourcentage de la quantité de caractères du texte.

QUESTION 4. Produire l'histogramme normalisé et ordonné de fréquences d'un texte littéraire en français, tiré du projet Gutenberg : <http://gutenberg.org>.

QUESTION 5. Relever les 10 plus grands écarts de valeur entre les deux histogrammes précédents.

## 5 Pour aller plus loin...

### Exercice 11: Les docstrings

Documenter le code Python devient rapidement une nécessité, en particulier pour des projets de développement collaboratifs. Les **docstring** permettent de produire de la documentation à partir d'annotations –à l'instar des commentaires– présentes dans le code source.

QUESTION 1. Écrire les *doctrings* du module `tp0.arithmetique` et de ses fonctions.

**Remarque:** Vous pourrez interroger les docstrings dans l'interpréteur Python à l'aide de la commande `help()`.

QUESTION 2. Produire une documentation complète du module `tp0.arithmetique` au format HTML à l'aide de l'outil **Sphinx**.

### Exercice 12: Les tests unitaires

Pytest est un outil externe à la librairie standard et requiert d'installer le module idoine. Dans la fenêtre de **Terminal**, reproduire la commande suivante :

```
prompt$ pip install -U pytest
```

La suite de tests Pytest est essentiellement fondée sur les *assertions* en Python, qui permettent dans le cas général de déclarer des invariants, des pré- et post-conditions, comme dans l'exemple suivant :

```
def hello_world(name: str) -> str:
    assert name != '' and name[0].isupper() # pré-condition
    return f'Bonjour {name}!'
```

Dans Pytest, un test correspond à l'écriture d'une fonction préfixée par `test_`, et munie d'une ou plusieurs assertion(s) :

```
def inc(n: int) -> int: # fonction à tester
    return n + 1

def test_inc(): # fonction de test
    assert inc(3) == 5 # assertion évaluée par pytest
```

Les fonctions de test peuvent être regroupées dans un module séparé, également préfixé par `test_`.

Pour plus d'information sur les fonctionnalités de Pytest, se reporter à [la documentation officielle](#), et les nombreuses ressources en ligne.

QUESTION 1. Écrire une première suite de tests unitaires dans `test_arithmetique.py` pour la fonction `est_compris_dans` de la bibliothèque `arithmetique`.

QUESTION 2. Introduire une « fixture »<sup>3</sup> à même de produire un ou plusieurs cas de test.

QUESTION 3. Ajouter du paramétrage à l'aide de l'annotation `@pytest.mark.parametrize` pour couvrir tous les cas sans répéter les fonctions de test.

QUESTION 4. Étendre la série de tests à toutes les fonctions du module `arithmetique`.

---

3. Un contexte préparé pour l'exécution du test.

## 5.1 Vers l’infini et au-delà...

Si vous êtes arrivé·e·s jusqu’ici en ayant fait contrôler votre production par un·e chargé·e de TP, vous pouvez poursuivre.

### Exercice 13: Quelques outils supplémentaires

Explorer successivement les thèmes suivants en élaborant de petits cas d’usage à partir du code source des exercices précédents :

1. le guide PEP-8 des bonnes pratiques d’écriture de code et les outils de vérification ( `flake8` ou `pycodestyle` , par exemple) ;
2. la création de paquets Python « prêts à l’emploi » à partir du code source ;
3. la librairie de *profiling* de code `cProfile` .

# Sujet n°1

## Algorithmique & Structures de données Premiers algorithmes itératifs

Temps de réalisation: 3h

Le TP n°1 a pour objectif de :

- résoudre quelques *problèmes simples* en développant des *algorithmes*.

Échelle de progression :

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

Ex.	01	02	03	04	05	06	07	08	09	10	11
	●	●	●	●	●	●	●	●	●	●	●

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

La programmation est une activité délicate et incertaine : pensez dès le début à tester votre code régulièrement, en procédant par petites touches successives, tel un peintre devant sa toile...

Tous les programmes de ce TP seront créés dans le paquet `tp1`.

## Table des matières

<b>1</b>	<b>Cahier d'exercices</b>	<b>2</b>
	Exercice 1 : Disques et cylindres . . . . .	2
	Exercice 2 : Second degré . . . . .	2
	Exercice 3 : Tu devines mon nombre . . . . .	2
	Exercice 4 : Je devine ton nombre . . . . .	2
	Exercice 5 : Calcul d'agrégats . . . . .	2
<b>2</b>	<b>Problèmes</b>	<b>3</b>
	Exercice 6 : Championnat . . . . .	3
	Exercice 7 : Calculatrice . . . . .	3
<b>3</b>	<b>Pour aller plus loin...</b>	<b>4</b>
	Exercice 8 : Suite de Collatz . . . . .	4
	Exercice 9 : Dimanche . . . . .	4
	Exercice 10 : Poker . . . . .	5
	Exercice 11 : Le jeu de Bézout . . . . .	5

# 1 Cahier d'exercices

Dans cette partie, tous les algorithmes qui requièrent une saisie de nombre(s) entier(s) devront réutiliser la fonction `saisir_entier` du module `tp0.util`.

## Exercice 1: Disques et cylindres

QUESTION 1. Écrire un programme qui demande à l'utilisateur le rayon d'un disque et affiche en retour son périmètre et sa surface. Le calcul du périmètre et de la surface doit être fait dans des fonctions idoine.

QUESTION 2. Écrire un autre programme qui demande à l'utilisateur le rayon d'un cylindre ainsi que sa hauteur et affiche sa surface et son volume. Outre la définition de deux nouvelles fonctions pour les calculs de la surface et du volume, vous devrez réutiliser les fonctions créées à la question précédente.

## Exercice 2: Second degré

Écrire une fonction qui résout une équation du second degré, de signature :

```
def eqn_second_degre(a: float, b: float, c: float) -> None
```

Le résultat sera affiché sur la sortie standard. Le déterminant sera calculé dans une fonction dédiée. Le programme de test affichera les racines des équations suivantes :

$$-x^2 + 5x + 14 = 0 \quad 3x^2 + 5x + 7 = 0 \quad x^2 - 6x + 9 = 0 \quad x^2 - \frac{7}{6}x + \frac{1}{3} = 0$$

## Exercice 3: Tu devines mon nombre

Écrire un programme qui génère un nombre au hasard entre 0 et 100 (voir fonction `random.randint`) et le fait deviner à un utilisateur. À chaque réponse de l'utilisateur, le programme indiquera si le nombre proposé est le bon ou s'il est inférieur ou supérieur au nombre à trouver.

Le programme doit itérer tant que la réponse n'a pas été trouvée.

## Exercice 4: Je devine ton nombre

Écrire un programme qui demande à un utilisateur de choisir un nombre qu'il doit garder secret, puis essayer de deviner ce nombre. À chaque proposition du programme l'utilisateur doit indiquer s'il s'agit du bon nombre ou s'il est inférieur ou supérieur au nombre à trouver.

Le programme doit itérer tant que la réponse n'a pas été trouvée, ou qu'une erreur de l'utilisateur a été détectée.

## Exercice 5: Calcul d'agrégats

Demander à l'utilisateur de saisir des nombres entiers, et afficher à la suite de chaque nombre fourni : le minimum, le maximum et la moyenne des nombres déjà saisis. La saisie s'arrête lorsque l'utilisateur entre une ligne vide.

**Remarque:** Pour ce programme, la mémorisation systématique des éléments déjà saisis, par exemple dans une liste, n'est pas autorisée.

## 2 Problèmes

### Exercice 6: Championnat

Écrire un programme élaborant une saison de championnat comportant  $n$  équipes, par exemple de football. Le principe consiste à fixer une équipe « pivot », puis à faire tourner les autres équipes d'une journée à l'autre.

Par exemple, avec 6 équipes, et l'équipe n°6 comme pivot :

Jour 1: 1-6	Jour 2: 2-6	Jour 3: 3-6	Jour 4: 4-6	Jour 5: 5-6
2-5	3-1	4-2	5-3	1-4
3-4	4-5	5-1	1-2	2-3

Lorsque l'on est revenu à la permutation originale, toutes les équipes se sont affrontées une fois. Il est important d'équilibrer les matchs à domicile et à l'extérieur pour chaque équipe, ce qui n'est pas vérifié pour l'équipe pivot dans l'exemple ci-dessus. Enfin, Si le nombre d'équipes est impair, il suffit d'ajouter une « équipe fictive » et de considérer que ses rencontres correspondent à des journées de repos pour l'équipe adverse.

Le résultat escompté est la liste des journées de championnat, avec pour chacune, la liste des rencontres. Vous proposerez un affichage similaire à :

Journée N°2:

équipe 3 reçoit équipe 2  
équipe 1 reçoit équipe 5  
équipe 4 au repos

**Astuce:** Si vous avez du mal à construire l'algorithme par vous-même (essayez d'abord!), voici sa version mathématique.

Soit  $n$  le nombre d'équipes,  $j$  le numéro de la journée et  $i$  le numéro du match de la journée ; soit  $n'$  le nombre d'équipes à prendre en compte pour le calcul. Si  $n$  est pair :  $n' = n$ , sinon  $n' = n + 1$ . On aura alors  $n' - 1$  jours de championnat et  $\frac{n'}{2}$  matchs par jour.

Pour déterminer le  $i^{\text{ème}}$  match d'une journée, pour chaque jour  $j$  :

— L'équipe locale est :

- si  $i = 1$  :  $n'$  si  $n$  est pair et 0 si  $n$  est impair (pas de match).
- sinon ( $i > 1$ ) :  $((j + i - 2) \bmod (n' - 1)) + 1$

— L'équipe des visiteurs est :

- $((j - i + n' - 1) \bmod (n' - 1)) + 1$

**Remarque:** La version ci-dessus n'équilibre pas les déplacements de l'équipe pivot. Trouvez un correctif!

### Exercice 7: Calculatrice

Écrire un programme qui demande la saisie au clavier d'une opération sur des nombres entiers naturels et affiche le résultat du calcul.

Les opérateurs à considérer sont  $\{+, *, \%, /\}$ . Chaque saisie doit se terminer par le signe '='. La calculatrice acceptera par exemple, les suites de caractères suivantes (sans les guillemets) :

```
"344+15="
"22 // 432 ="
"93451* 0="
"12 %3="
```

et refusera les saisies comme celles-ci :

```
"1+2= "  
"1+b_="   
" "  
"3+15"  
"2.4+3="   
"-4//3="   
"4%(-2)="
```

Afficher un message d'erreur le cas échéant.

**Remarque:** La fonction `str.split()`, voire la construction d'une *expression rationnelle*, offrent des moyens pratiques pour analyser la chaîne qui représente l'opération à réaliser. Un examen séquentiel des caractères est également possible.

### 3 Pour aller plus loin...

Les exercices de cette section sont issus des archives du portail [Project Euler](#). Il s'agit des problèmes n°14, 19, 54 et 787.

#### Exercice 8: Suite de Collatz

La suite de COLLATZ, aperçue au TP0, est définie par :

$$C(n) = \begin{cases} n/2 & \text{si } n \text{ est pair,} \\ 3 \cdot n + 1 & \text{sinon.} \end{cases}$$

La séquence obtenue à partir de  $n = 13$  est :

$$13 \hookrightarrow 40 \hookrightarrow 20 \hookrightarrow 10 \hookrightarrow 5 \hookrightarrow 16 \hookrightarrow 8 \hookrightarrow 4 \hookrightarrow 2 \hookrightarrow 1$$

Cette séquence contient 10 termes. Bien que cela n'ait pas été prouvé, la conjecture de SYRACUSE pose le principe d'une convergence à 1, ou plus exactement à la sous-séquence répétée (4, 2, 1), quel que soit le nombre entier  $n$  de départ.

QUESTION 1. Donner le (ou les) nombre(s)  $n \leq 10\,000$  qui génère(nt) la plus longue suite jusqu'à 1.

QUESTION 2. Même question, pour  $n \leq 10^7$ .

#### Exercice 9: Dimanche

La construction du calendrier grégorien est donnée par la comptine suivante, dont il est facile de vérifier la pertinence :

1 Jan 1900 was a Monday.  
Thirty days has September,  
April, June and November.  
All the rest have thirty-one,  
Saving February alone,  
Which has twenty-eight, rain or shine.  
And on leap years, twenty-nine.

Une année bissextile (*leap year*) survient tous les 4 ans, lorsqu'elle est divisible par 4 et à l'exception des centaines, sauf si elles sont elles-mêmes divisibles par 400 !

Combien de dimanches sont tombés un premier jour du mois au vingtième siècle (du 1er janvier 1901 au 31 décembre 2000) ?

## Exercice 10: Poker

Au jeu de poker, une main est composée de cinq cartes, pouvant former l'une des combinaisons suivantes, dans l'ordre croissant de leur puissance :

1. plus forte carte ;
2. une paire : deux cartes de même rang (ou valeur) ;
3. deux paires ;
4. brelan : trois cartes de même rang ;
5. quinte : cinq cartes consécutives ;
6. couleur : cinq cartes de même enseigne (ou couleur) ;
7. full : brelan et paire ;
8. carré : quatre cartes de même rang ;
9. quinte flush : quinte dont les cinq cartes sont de même enseigne ;
10. quinte flush royale : quinte flush incluant un As.

Pour mémoire, les cartes sont, dans l'ordre : 2, 3, 4, 5, 6, 7, 8, 9, 10 (Ten), Valet (Jack), Reine (Queen), Roi (King), et As (Ace). Les enseignes sont Pique (Spade), Coeur (Heart), Carreau (Diamond), Trèfle (Club).

En cas d'égalité de combinaison entre deux joueurs, c'est la plus haute carte complétant la main qui l'emporte, et ainsi de suite. Voici cinq exemples de mains à deux joueurs :

Main	Joueur 1	Joueur 2	Vainqueur
1	5H 5C 6S 7S KD paire de 5	2C 3S 8S 8D TD paire de 8	Joueur 2
2	5D 8C 9S JS AC plus haute carte As	2C 5C 7D 8S QH plus haute carte Reine	Joueur 1
3	2D 9C AS AH AC brelan	3D 6D 7D TD QD couleur à Carreau	Joueur 2
4	4D 6S 9H QH QC paire de Reines plus haute carte 9	3D 6D 7H QD QS paire de Reines plus haute carte 7	Joueur 1
5	2H 2D 4C 4D 4S full par les 4	3C 3D 3S 9S 9D full par les 3	Joueur 1

Le fichier `poker.txt` contient cent mains aléatoires de 2 joueurs. Sur chaque ligne sont représentées 10 cartes, les cinq premières correspondent à la main du joueur 1, les cinq dernières celle du joueur 2. On suppose que toutes les mains sont valides. En outre, les cartes de chaque main ne sont pas triées. Enfin, il y a systématiquement une main meilleure que l'autre.

Combien de mains le joueur 1 aura-t'il gagné à l'issue de la partie ?

## Exercice 11: Le jeu de Bézout

Imaginons un jeu conçu à partir de deux tas de cailloux. Il se joue à deux, au tour par tour. S'il y a  $a$  cailloux dans le premier tas et  $b$  dans le second, un tour consiste à retirer  $c \geq 0$  cailloux du premier tas et  $d \geq 0$  du second, de telle sorte que  $ad - bc = \pm 1$ . Le vainqueur est le premier joueur qui vide l'un des deux tas.

Une condition nécessaire pour jouer est que les nombres  $a$  et  $b$  soient premiers entre eux.

On dit que l'état  $(a, b)$  du jeu est une « position gagnante » si le joueur suivant, considérant qu'il joue de la meilleure des manières, remporte la partie. Les positions  $(a, b)$  et  $(b, a)$  sont distinctes.

On désigne  $H(N)$  comme le nombre de positions gagnantes, avec  $\text{pgcd}(a, b) = 1$ ,  $a > 0$ ,  $b > 0$  et  $a + b \leq N$ .

QUESTION 1. Retrouver  $H(4) = 5$  et  $H(100) = 2043$ .

QUESTION 2. Calculer  $H(10^9)$ .

**En marge:** *L'algorithme d'EUCLIDE étendu, au cœur de cet exercice, admet beaucoup d'applications comme par exemple en géométrie algorithmique (trouver un point sur une droite dont on donne l'équation), ou en cryptographie (déterminer une paire de clés dans le protocole RSA).*

## Sujet n°2

Algorithmique & Programmation  
Problèmes algorithmiques récurrents

Temps de réalisation: 3h

L'objectif du TP n°2 est de :

- identifier les problèmes adaptés à une résolution récursive
- décliner la récurrence mathématique en programme récursif
- définir et implémenter les éléments d'une résolution récursive

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

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

La programmation est une activité délicate et incertaine : pensez dès le début à tester votre code régulièrement, en procédant par petites touches successives, tel un diamantaire taillant sa gemme...

La documentation officielle de Python est toujours accessible ici :

<https://docs.python.org/3/>

Le contenu intégral des exercices du TP2 doit figurer dans un paquet `tp2` de votre projet PyCharm *cahier d'exercices*.

# Table des matières

<b>1</b>	<b>Premiers pas en récursivité</b>	<b>2</b>
	Exercice 1 : Factorielle . . . . .	2
	Exercice 2 : PGCD . . . . .	2
	Exercice 3 : L'incontournable Fibonacci . . . . .	2
	Exercice 4 : Dissection de nombres . . . . .	2
	Exercice 5 : Coefficients binomiaux . . . . .	3
<b>2</b>	<b>Jeux et dessin</b>	<b>4</b>
	Exercice 6 : Le robot cupide . . . . .	4
	Exercice 7 : Tours de Hanoï . . . . .	4
	2.1 Aperçu de Tkinter . . . . .	5
	Exercice 8 : Tracé d'un segment . . . . .	6
	Exercice 9 : Triangle de Sierpinski . . . . .	6

<b>3 Pour aller plus loin...</b>	<b>6</b>
Exercice 10 : Les permutations . . . . .	6
Exercice 11 : La diagonale de Cantor . . . . .	6
Exercice 12 : Le monnayeur . . . . .	7
Exercice 13 : Labyrinthe . . . . .	7
Exercice 14 : Les nombres dominants . . . . .	8

## 1 Premiers pas en récursivité

Tous les problèmes de cet énoncé doivent être résolus par un algorithme récursif, sauf mention contraire.

### Exercice 1: Factorielle

Écrire une fonction récursive qui calcule la factorielle d'un entier naturel  $n$ . On rappelle que

$$n! = n \times (n-1)! \quad \text{si } n \geq 1 \quad \text{et} \quad 0! = 1.$$

### Exercice 2: PGCD

QUESTION 1. Écrire une fonction récursive qui retourne le plus grand commun diviseur (pgcd) de deux nombres entiers positifs par l'algorithme d'EUCLIDE :

$$\text{pgcd}(a, b) = \begin{cases} a & \text{si } b = 0 \\ \text{pgcd}(b, a \bmod b) & \text{sinon} \end{cases}$$

**Remarque:** *L'algorithme d'EUCLIDE est un cas emblématique de récursion terminale. La version itérative se décline donc aisément.*

**En marge:** *L'algorithme d'EUCLIDE et sa version étendue sont utilisés dans le cours de cryptographie (S7), et sont des outils fondamentaux en géométrie algorithmique.*

### Exercice 3: L'incontournable Fibonacci

La suite de FIBONACCI notée  $F_n$  est définie par  $F_0 = 0$ ,  $F_1 = 1$ , et  $F_n = F_{n-1} + F_{n-2}$  pour  $n \geq 2$ .

QUESTION 1. Écrire une fonction récursive qui calcule le  $n$ ième terme de la suite de FIBONACCI.

QUESTION 2. Afficher les 100 premiers termes de la suite. Commenter.

QUESTION 3. Construire l'arbre d'appel de fonctions pour  $n = 6$ . Combien de fois est calculé  $F_3$  pour obtenir le terme  $F_6$  ?

QUESTION 4. Écrire une version itérative du calcul du  $n$ ième terme de la suite de FIBONACCI. Recalculer les 100 premiers termes. Commenter.

### Exercice 4: Dissection de nombres

QUESTION 1. Construire une fonction

```
def nb_digits(n: int) -> int
```

prenant en paramètre un entier naturel  $n$  et retournant le nombre de chiffres de cet entier.

**Remarque:** Le quotient de la division euclidienne d'un entier  $n$  par 10 donne le nombre de dizaines de cet entier. Par exemple, le quotient de la division euclidienne de  $n = 5478$  par 10 vaut 547.

QUESTION 2. Modifier cette fonction en ajoutant un deuxième paramètre **base** permettant de donner le nombre de chiffres de l'entier  $n$  –toujours exprimé en base 10– en base **base**.

QUESTION 3. Créer une fonction récursive

```
def convert(n: int, base: int) -> str
```

affichant un nombre  $n$  exprimé en base 10 en une base **base** fournie en paramètre (pour simplifier l'affichage, on considère **base** < 10).

*Ex. : `convert(10,2)` affiche 1010*

QUESTION 4. Créer une fonction récursive **convert\_mirror** affichant en sens inverse un nombre  $n$  exprimé en base 10 en une base **base** fournie en paramètre (pour simplifier l'affichage, on considère **base** < 10).

*Ex. : `convert_mirror(10,2)` affiche 0101*

## Exercice 5: Coefficients binomiaux

On connaît la définition d'un coefficient binomial :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

que l'on note aussi  $C_n^k$  comme le nombre de parties ou « combinaisons » de  $k$  éléments parmi  $n$ .

La formule de PASCAL définit la série des coefficients binomiaux à l'aide de la relation de récurrence :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{pour } 0 < k < n$$

De plus, on pose :

$$\binom{n}{0} = \binom{n}{n} = 1$$

QUESTION 1. Écrire la fonction de calcul d'un coefficient binomial par la formule de PASCAL :

```
def binom(n: int, k: int) -> int
```

QUESTION 2. Retrouver  $\binom{5}{4} = 5$ ,  $\binom{10}{5} = 252$ ,  $\binom{100}{1} = 1$  et calculer  $\binom{100}{50}$ . Commenter.

QUESTION 3. Proposer une version avec *mémoïsation*<sup>1</sup>.

**Astuce:** Dans le cas des coefficients binomiaux, la *mémoïsation* consiste à conserver en mémoire les termes déjà calculés du triangle de PASCAL, et les piocher dans la mémoire à chaque réutilisation.

**Remarque:** Par défaut, la *mémoïsation* est implémentée à l'aide d'un dictionnaire (`dict` en Python).

QUESTION 4. Recalculer le terme  $\binom{100}{50}$ . Commenter.

1. Usage d'une mémoire tampon dans laquelle figurent les résultats de calculs coûteux qui devraient sinon être réalisés plusieurs fois.

## 2 Jeux et dessin

### Exercice 6: Le robot cupide

Exercice proposé par O. Bodini de l'Institut Galilée, Univ. Paris-Nord.

Well-A le robot se trouve sur une case quelconque  $(x, y)$  d'un damier rectangulaire de taille  $N \times M$ . La case  $(0, 0)$  se situe au Nord-Ouest. Il doit sortir par la case Sud-Est de coordonnées  $(N - 1, M - 1)$ . Il a le choix à chaque pas (un pas = une case) entre :

- descendre verticalement vers le Sud ;
- ou avancer horizontalement vers l'Est.

Il y a un sac de Bitcoins sur chaque case, dont la valeur est lisible depuis la position initiale de Well-A. Le but de Well-A est de ramasser le plus de Bitcoins possible durant son trajet.

QUESTION 1. On veut donc écrire la fonction :

```
def robot_cupide(damier: list[list[int]], x: int = 0, y: int = 0) -> int
```

qui, étant donné le damier  $N \times M$  renseigné avec les gains potentiels dans chaque case, et les coordonnées  $x$  et  $y$  d'une case de départ, rend la quantité maximum de Bitcoins (gain) que peut ramasser le robot en se déplaçant de  $(x, y)$  jusqu'à la sortie Sud-Est.

QUESTION 2. Faire évoluer la fonction `robot_cupide` de telle sorte qu'elle enregistre le trajet emprunté par Well-A, en plus de la somme récoltée.

### Exercice 7: Tours de Hanoï

Le jeu des Tours de Hanoï est constitué de trois piquets verticaux, notés 1, 2 et 3 et de  $n$  disques superposés de tailles strictement décroissantes avec un trou au centre et enfilés autour du piquet 1 ; ces empilements forment les tours, comme le montre la Figure 1.

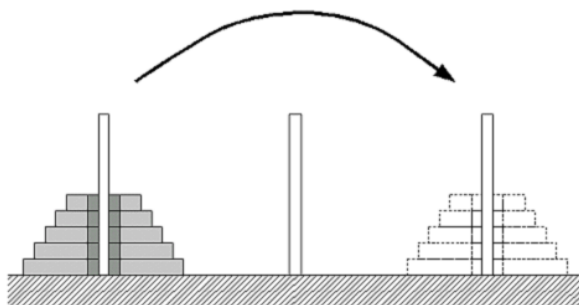


FIGURE 1 – Schéma des Tours de Hanoï.

Le but du jeu consiste à déplacer l'ensemble des disques pour que ceux-ci se retrouvent enfilés autour du piquet 3, en respectant les deux règles suivantes :

1. les disques sont déplacés un par un ;
2. un disque ne doit pas se retrouver au-dessus d'un disque plus petit.

On suppose évidemment que la règle n°2 est également respectée dans la configuration de départ.

Le problème a toujours une solution en  $2^n - 1$  coups (admis). Il se résout de manière récursive. En effet, supposons le problème résolu pour  $n - 1$  disques, c'est-à-dire que l'on sache transférer  $n - 1$  disques depuis le piquet  $i \in \{1, 2, 3\}$  jusqu'au piquet  $j \in \{1, 2, 3\} \setminus \{i\}$  en respectant les règles du jeu. Pour transférer  $n$  disques du piquet  $i$  vers le piquet  $j$ , on procède alors comme suit :

1. on amène les  $n - 1$  disques du haut du piquet  $i$  sur le piquet intermédiaire, qui a le numéro  $6 - i - j$ ;
2. on prend le dernier disque du piquet  $i$  et on le met seul en  $j$ ;
3. on ramène les  $n - 1$  disques de  $6 - i - j$  en  $j$ .

QUESTION 1. Écrire une fonction récursive

```
def hanoi(n: int, i: int, j: int) -> list[tuple[int,int]]
```

qui construit la séquence des mouvements élémentaires à accomplir pour déplacer  $n$  disques du piquet  $i$  au piquet  $j$ . Chaque mouvement est représenté par un couple  $(s, t)$  indiquant que l'on déplace un disque du piquet  $s$  vers le piquet  $t$ . Afficher la séquence une fois construite.

QUESTION 2. Considérant que les disques sont identifiés de 1 à  $n$ , à partir du plus grand, modifier la fonction récursive pour qu'elle ajoute les numéros des disques à chaque mouvement. Un mouvement est alors un triplet  $(d, s, t)$  avec  $d$  le numéro du disque,  $s$  et  $t$  resp. les piquets source et cible.

## 2.1 Aperçu de Tkinter

Le code ci-dessous est un squelette opérationnel de programme `tkinter` :

```
from tkinter import *

# window creation
window = Tk()

# canvas creation
w = Canvas(window, width=400, height=300)
w.pack()

# drawing functions
# put your own drawing below
w.create_line(0, 0, 400, 300)
w.create_line(0, 300, 400, 0, fill="red")
w.create_rectangle(100, 75, 300, 225, fill="green")

# main event loop to allow interaction
mainloop()
```

La liste complète des figures géométriques disponibles n'est pas donnée ici, la documentation de *Tkinter* étant facilement disponible sur Internet.

Curieusement, il n'existe pas de fonction permettant de dessiner un point à l'écran dans ce module. La fonction peut cependant être émulée par le code suivant :

```
# function definition
def draw_point(canvas: Canvas, x: int, y: int, color: str="black") -> None:
    canvas.create_rectangle(x, y, x, y, fill=color, width=0)

# effective point drawing - to insert after pack() statement
draw_point(w, 10, 10)
```

### Exercice 8: Tracé d'un segment

En informatique graphique, tous les points ont des coordonnées entières, exprimées en fonction d'une grille de résolution, ici la taille du canvas. Soient donc deux points  $A$  de coordonnées  $(x_A, y_A)$  et  $B$  de coordonnées  $(x_B, y_B)$ ; proposer un algorithme récursif pour tracer à l'écran le segment  $[A, B]$  uniquement à l'aide de la fonction `draw_point` précédente.

**En marge:** L'algorithme de BRESENHAM résoud ce problème élémentaire de tracé de segment.

### Exercice 9: Triangle de Sierpinski

Le triangle de SIERPIŃSKI est une fractale composée à partir de triangles, et illustrée sur la Figure 2.

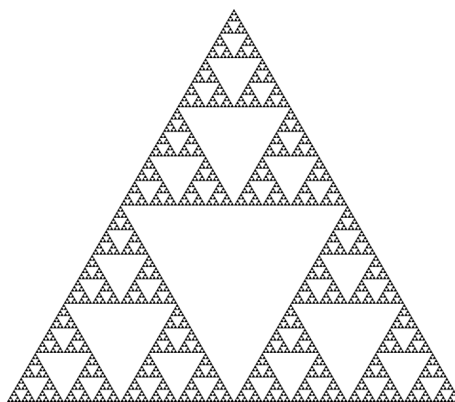


FIGURE 2 – Illustration d'un triangle de SIERPIŃSKI.

L'algorithme permettant d'obtenir cette figure consiste à tracer un triangle puis le triangle qui joint les milieux des côtés et ainsi de suite jusqu'à obtenir des segments plus petits qu'un pixel.

Écrire une fonction récursive permettant de dessiner ce triangle pour une taille de côté donnée.

## 3 Pour aller plus loin...

### Exercice 10: Les permutations

Proposer une fonction récursive qui calcule toutes les permutations des entiers de 1 à  $n$ . Par exemple, les permutations de  $\mathbb{N}_3$  sont  $(1, 2, 3)$ ,  $(1, 3, 2)$ ,  $(2, 1, 3)$ ,  $(2, 3, 1)$ ,  $(3, 1, 2)$  et  $(3, 2, 1)$ .

**Astuce:** La fonction se construit en observant que pour passer de  $\mathbb{N}_2$  à  $\mathbb{N}_3$ , il « suffit » d'insérer 3 à toutes les positions des permutations de  $\mathbb{N}_2$ . En effet, à partir de  $(1, 2)$  on obtient  $(1, 2, \mathbf{3})$ ,  $(1, \mathbf{3}, 2)$  et  $(\mathbf{3}, 1, 2)$  tandis qu'avec  $(2, 1)$  on obtient  $(2, 1, \mathbf{3})$ ,  $(2, \mathbf{3}, 1)$  et  $(\mathbf{3}, 2, 1)$ .

### Exercice 11: La diagonale de Cantor

La fonction de couplage de CANTOR (cf. Fig.3) réalise une bijection de  $\mathbb{N} \times \mathbb{N}$  dans  $\mathbb{N}$ . Par exemple  $\text{cantor}(1, 1) = 4$  et  $\text{cantor}(1, 3) = 13$ .

Définir par récurrence et implémenter la fonction de couplage de CANTOR.

**Astuce:** Élaborer les différents cas de figure pour définir  $\text{cantor}(x, y)$  à partir du point précédent sur le tracé de couleur bleue sur la Figure 3.

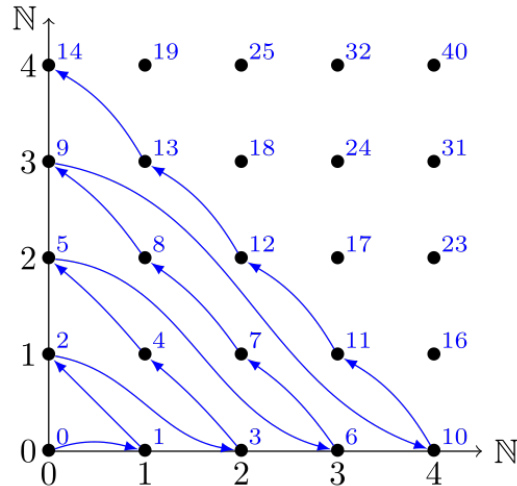


FIGURE 3 – Représentation de la fonction de couplage de CANTOR (src : Florent Demeslay, Wikipédia, 2021 - CC BY-SA 4.0).

## Exercice 12: Le monnayeur

Supposons qu'une caisse enregistreuse dispose d'un stock inépuisable de pièces de 2 cts, 5 cts, 10 cts, 20 cts, 50 cts et 100 cts (1€). Le monnayeur automatique doit rendre une somme  $n$ , exprimée en centimes, avec *le minimum de pièces*.

Par exemple, si  $n = 45$  cts, le monnayeur rend (20, 20, 5) avec 3 pièces. On peut bien sûr envisager (20, 10, 5, 2, 2, 2, 2, 2) ou tout autre arrangement de somme 45, néanmoins ils sont tous plus longs que 3.

QUESTION 1. Écrire la fonction  $NP(n)$  qui fournit le nombre de pièces nécessaires pour rendre  $n$  centimes.

**Remarque:** *Le monnayeur ne peut pas appliquer de méthode gloutonne qui consisterait à épuiser les pièces par ordre décroissant de leur valeur faciale. Si cela semble fonctionner pour  $n = 45$ , ce n'est pas applicable par exemple pour  $n = 11$ . En effet, on obtiendrait d'abord 10 puis rien (!?), or il existe une solution avec (5, 2, 2, 2).*

**Astuce:** *Si le monnayeur sait rendre 72 cts avec un minimum de pièces, alors il sait rendre  $72 - i$  cts pour  $i$  dans  $\{2, 5, 10, 20, 50, 100\} \setminus \{100\}$ . Cette observation esquisse la relation de récurrence.*

QUESTION 2. Retrouver par le calcul  $NP(11) = 4$  et  $NP(53) = 7$ . Calculer  $NP(173)$ .

**En marge:** *Le problème du monnayeur, plus connu en anglais sous l'appellation Change-making Problem, est un cas particulier du célèbre problème du « sac à dos » (Knapsack Problem en anglais).*

## Exercice 13: Labyrinthe

Dans cet exercice, nous vous proposons de programmer un algorithme qui permet de s'orienter dans un labyrinthe. Le labyrinthe est représenté sous la forme d'un tableau à deux dimensions (des listes de listes en Python). La valeur 0 représente un mur, tandis que 1 désigne un espace que l'on peut occuper. La déclaration initiale du tableau ressemble donc à ceci :

```

maze: list[list[int]] = [
    [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0, 1, 0, 1],
    [0, 0, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 0, 0, 1, 0, 1],
    [0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [0, 1, 0, 0, 1, 0, 1, 1, 0, 1],
    [0, 1, 0, 1, 1, 0, 1, 0, 1, 0],
    [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 1]
]

```

Pour compléter le problème, on définit une case départ à l'aide de ses coordonnées dans le tableau, ainsi qu'une case arrivée.

QUESTION 1. Créer un algorithme récursif permettant de trouver un chemin dans le labyrinthe, de la case départ jusqu'à la case arrivée. S'il n'existe aucun chemin, la fonction retourne une liste vide [], sinon elle produit le chemin parcouru.

QUESTION 2. Proposer une version de cet algorithme qui garantisse que le chemin emprunté est le plus court.

## Exercice 14: Les nombres dominants

Un « nombre dominant » est un entier positif dont plus de la moitié de ses chiffres sont égaux.

Par exemple, 2022 est un nombre dominant car il contient trois occurrences du chiffre 2. À l'inverse, 2021 n'est pas un nombre dominant.

Soit  $D(N)$  la quantité de nombres dominants inférieurs à  $10^N$ .

QUESTION 1. Retrouver par le calcul  $D(4) = 603$  et  $D(10) = 21\,893\,256$

QUESTION 2. Calculer  $D(2022)$ . Donner la réponse modulo 1 000 000 007.

**Remarque:** Problème n°788 de [projecteuler.net](https://projecteuler.net).

# 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

<b>1 Les piles</b>	<b>2</b>
Exercice 1 : La pile à partir d'une list . . . . .	2
Exercice 2 : La pile à partir d'une classe . . . . .	2
Exercice 3 : Notation polonaise inverse (à l'aide d'une pile à partir d'une classe) . . . . .	2
Exercice 4 : La pile à partir du tableau . . . . .	3
Exercice 5 : Notation polonaise inverse (à l'aide d'une pile à partir d'un tableau) . . . . .	4
<b>2 Les files d'attente</b>	<b>4</b>
Exercice 6 : La file d'attente à partir d'une liste . . . . .	4
Exercice 7 : La file d'attente à partir d'une classe . . . . .	4
Exercice 8 : E l i f . . . . .	5
Exercice 9 : Problème de Joséphus . . . . .	5
Exercice 10 : La file d'attente à partir du tableau . . . . .	5
Exercice 11 : E l i f et Problème de Joséphus à l'aide d'une file à partir d'un tableau . . . . .	5
<b>3 Les tableaux en Python</b>	<b>6</b>
Exercice 12 : Quelques fonctions utilitaires . . . . .	6
3.1 Petits défis . . . . .	7
Exercice 13 : Inversion . . . . .	7
Exercice 14 : Nombres pairs et impairs . . . . .	7
3.2 Définition des <code>ArrayList</code> . . . . .	7
Exercice 15 : Vers un type de listes d'entiers à base de tableaux statiques . . . . .	7
Exercice 16 : Opérations sur les tableaux d'entiers . . . . .	9
3.3 <code>ArrayList</code> à l'usage . . . . .	11
Exercice 17 : La chasse aux doublons . . . . .	11

# 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* : renvoyer la valeur du dernier élément de la pile
- *Size* : renvoyer le nombre d'éléments
- *isEmpty* : 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 accéder, 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 - chaque 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 LUKASIEWICZ, 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 :

$$\begin{aligned} 4 \ 3 \ - \ 2 \ * \ 5 \ 2 \ / \ + &:= 2 \ 4 \ 3 \ - \ * \ 5 \ 2 \ / \ + \\ &:= 5 \ 2 \ / \ 2 \ 4 \ 3 \ - \ * \ + \\ &:= 5 \ 2 \ / \ 4 \ 3 \ - \ 2 \ * \ + \end{aligned}$$

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<sup>1</sup> 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. Dérouter 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 postfixée, 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.

```

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 options, comme vu dans le cours.

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

- *Enqueue* : insérer 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
- *isEmpty* : 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 accéder, 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 - chaque 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 distraitemment 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 diminue 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 contigües, chacune de la taille d'un entier et par conséquent, qui crée 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`<sup>2</sup>.

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  $\llbracket a, b \rrbracket$ .

**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<sup>3</sup>, 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  $(t_0, \dots, t_{n-1}, \perp, \dots, \perp)$  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.

```

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 vocabulaire 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`<sup>4</sup> de Python pour représenter des tableaux<sup>5</sup> 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 crée un nouvel objet de type `ArrayList` à partir d'une capacité maximale  $m$  et d'une liste initiale  $\ell$  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 élémentaires 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  $i \in \llbracket 0, n-1 \rrbracket$ . Cette opération est le pendant de `tab[i]`. Pour un tableau de taille<sup>6</sup>  $n$ , 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  $i \in \llbracket 0, n-1 \rrbracket$  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<sup>7</sup>. 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  $i \in \llbracket 0, n-1 \rrbracket$  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  $i \in \llbracket 0, n \rrbracket$  du tableau `tab`<sup>8</sup>. 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

<sup>8</sup>. 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  $\llbracket 0, n-1 \rrbracket$ .

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<sup>9</sup> ( $\mathcal{O}(n^2)$ ). 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  $|\cdot|$  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.

# Sujet n°4

## Algorithmique et Structure de Données Structures de données linéaires

Temps de réalisation: 1h30mn

L'objectif de ce TP est de :

- se familiariser avec les listes chaînées, dans une version avec double chaînage circulaire
- concevoir une implémentation alternative des piles et des files d'attente
- appréhender les files d'attente à deux extrémités
- évaluer le coût des opérations sur les listes chaînées
- comparer les listes et les tableaux
- résoudre des problèmes en mobilisant des structures de données linéaires et leurs opérations

*ProTip* : il est super-méga-extra-recommandé de se munir d'un papier et d'un crayon pour dessiner des tonnes de cases et de flèches qui représentent les listes chaînées en mémoire et leur évolution en fonction des scénarios étudiés.

Échelle de progression :

● « débutant·e »	● « confirmé·e »				● « avancé·e »			● « expert·e »	
Ex.	01	02	03	04	05	06	07	08	09
	●	●	●	●	●	●	●	●	●

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

<b>1</b>	<b>Les listes chaînées</b>	<b>2</b>
	Exercice 1 : Définition d'une liste doublement chaînée circulaire . . . . .	2
	Exercice 2 : Les opérations élémentaires . . . . .	3
	Exercice 3 : Topswops . . . . .	4
<b>2</b>	<b>Files, piles et dèques</b>	<b>5</b>
	Exercice 4 : Implémentation de dèques à l'aide de listes chaînées . . . . .	5
	Exercice 5 : La valeur maximale d'une fenêtre glissante . . . . .	5
	Exercice 6 : Retour sur les piles et les files . . . . .	5
<b>3</b>	<b>Pour aller plus loin. . .</b>	<b>6</b>
	Exercice 7 : Algorithme de Graham . . . . .	6
	3.1 Les listes à enjambements . . . . .	6
	Exercice 8 : Première implémentation . . . . .	7
	Exercice 9 : La médiane mobile . . . . .	7

# Introduction

Les structures de données linéaires offrent la possibilité de mémoriser des variables qui sont des collections finies totalement ordonnées d'éléments pouvant présenter des répétitions. L'idée simple et fondamentale sous-jacente est que chaque élément de la collection vient avant ou après un autre élément.

Elles permettent donc d'implémenter certains types abstraits « linéaires » très utiles en informatique tels que les séquences et des formes restreintes comme les piles et les files d'attente à une ou deux extrémités. Nous avons d'ailleurs déjà rencontré une structure de données linéaire : le tableau. Il s'agit en fait d'une implémentation de « séquence » présentant des opérations et des performances spécifiques, obtenues grâce à une allocation contigüe qui permet un accès direct aux éléments à partir du rang (ou indice).

Les structures de données linéaires sont également fondamentales pour certaines implémentations de types abstraits non linéaires, tels que les files de priorités, les partitions d'ensembles voire certains arbres et graphes !

Dans la suite de cet énoncé, nous nous intéressons à nouveau au type abstrait *séquence* (ou *liste*). Cependant, nous mettons l'accent sur la propriété d'accès séquentiel aux éléments, plutôt que sur l'accès direct par l'indice. En d'autres termes il s'agit, à partir d'un voisinage donné, de parcourir la liste de proche en proche pour accéder à l'élément visé. En effet, dans certaines situations, il est utile de pouvoir désigner la « place » d'un élément sans pour autant connaître son rang : par exemple, lorsque l'on s'écarte temporairement d'une file d'attente, on souhaite pouvoir s'y insérer à nouveau « au même endroit » même si on ne connaît pas son rang ou si celui-ci évolue. Les repères sont alors les voisins de devant et de derrière.

## 1 Les listes chaînées

Afin de traduire cette idée d'accès séquentiel, il faut disposer d'un moyen pour désigner une « place » dans la séquence. En informatique, les structures de données vont permettre de créer des objets « place » et leurs *références* vont être employées pour réaliser un chaînage entre places consécutives et représenter la séquence : on parle alors de *liste chaînée*.

Il existe en pratique de multiples variantes de listes chaînées, répondant toutes au type abstrait *séquence* et proposant des fonctionnalités supplémentaires et des performances variées :

- les listes simplement chaînées ;
- les listes doublement chaînées ;
- les listes chaînées circulaires ;
- les listes doublement chaînées circulaires ;
- toute variante de chaînage propre à un usage particulier...

### Exercice 1: Définition d'une liste doublement chaînée circulaire

Dans cet exercice, vous allez créer à l'aide d'une classe les structures de données et les opérations nécessaires à la gestion d'une liste de nombres entiers.

**QUESTION 1.** Créer les types `LinkedList` et `Cell` pour représenter une **liste doublement chaînée circulaire** (*Circular Doubly Linked List*) de maillons<sup>1</sup> (*cell*) contenant des nombres entiers.

La définition de la classe `Cell` requiert le symbole `Cell` lui-même : elle est récursive. En Python, à partir de la version 3.7, ce mécanisme est disponible.

**QUESTION 2.** Écrire le constructeur, et le test de liste vide :

---

1. Les maillons représentent les « places » dans la liste.

QUESTION 3. Ajouter 3 « observateurs », i.e., des méthodes d'accès aux informations de la liste : la taille, la tête et la queue. qui donnent respectivement, la taille, la première et la dernière cellule de la liste. Les fonctions `head()` et `tail()` retournent `None` lorsque la liste est vide.

## Exercice 2: Les opérations élémentaires

QUESTION 1. Écrire la méthode `insert`

```
def insert(item: int,
           neighbor: Cell,
           after: bool = True) -> LinkedList
```

qui insère l'élément `item` dans une cellule adjacente à `neighbor`. Si l'indicateur `after` est à *vrai*, alors l'insertion a lieu après `neighbor`, sinon elle a lieu avant.

QUESTION 2. Décliner les méthodes `insert_head` et `insert_tail`, cas particuliers de la fonction d'insertion, qui ajoutent un élément respectivement en tête et en queue de liste.

QUESTION 3. Puis proposer une méthode de sérialisation des listes :

```
def __str__() -> str
```

qui retourne la représentation sérialisée (textuelle) de la liste  $\ell$ , principalement en vue d'un affichage sur la sortie standard (Terminal) :

```
>>> print(liste)
[1, 2, 3, 2, 1]
```

L'affichage de la liste doit reproduire l'affichage des listes natives de Python, e.g. `[1, 2, 3, 2, 1]`.

**Remarque:** La méthode de sérialisation, qui nécessite un parcours de la liste, peut s'écrire de manière itérative ou récursive, au choix.

QUESTION 4. Définir un test de validité :

```
def isChained() -> bool
```

qui vérifie que le chaînage et la taille de la liste sont corrects.

QUESTION 5. Écrire les méthodes de recherche par valeur et par rang :

```
def find(item: int) -> Optional[Cell]
def findAt(i: int) -> Cell
```

La méthode `find()` recherche la première occurrence de l'élément `item` et retourne—une référence vers—le maillon si la recherche aboutit, `None` sinon. La méthode `findAt()` fournit le maillon en position `i`.

QUESTION 6. Écrire les méthodes de lecture et d'écriture suivantes :

```
def getValue(idx: ???) -> int
def setValue(idx: ???, item: int) -> LinkedList
```

respectivement, qui retourne l'élément (la valeur entière) donné à la place `idx` de la liste chaînée  $\ell$ , et qui affecte la valeur `item` à la place `idx` de la liste. Il vous appartient de spécifier le type (???) d'une place.

QUESTION 7. Ajouter la méthode

```
def remove(cell: Cell) -> LinkedList
```

qui supprime la cellule `cell` de la chaîne.

QUESTION 8. Écrire la méthode

```
def extend(l2: LinkedList) -> LinkedList
```

qui concatène deux listes chaînées. Les données de la liste  $\ell_2$  sont ajoutées à la fin de la liste objet.

QUESTION 9.

### Exercice 3: Topswops

Le défi *Topswops* figure dans la liste des *Al Zimmermann's Programming Contests* et consiste à trouver la permutation des entiers de 1 à  $n$  qui maximise le nombre d'itérations nécessaires pour atteindre une permutation commençant par 1, en appliquant une opération unique : inverser la sous-séquence préfixe, de taille égale à la valeur du premier élément. On réitère jusqu'à observer la valeur 1 en première position !

Par exemple, pour  $n = 6$ , la permutation  $(\mathbf{3}, 6, 5, 1, 4, 2)$  requiert une inversion des  $\mathbf{3}$  premiers éléments—la sous-séquence  $(3, 6, 5)$  est transformée en  $(5, 6, 3)$ —dans la première itération. L'intégralité du procédé donne lieu à 10 itérations :

0. [3, 6, 5], 1, 4, 2
1. [5, 6, 3, 1, 4], 2
2. [4, 1, 3, 6], 5, 2
3. [6, 3, 1, 4, 5, 2]
4. [2, 5], 4, 1, 3, 6
5. [5, 2, 4, 1, 3], 6
6. [3, 1, 4], 2, 5, 6
7. [4, 1, 3, 2], 5, 6
8. [2, 3], 1, 4, 5, 6
9. [3, 2, 1], 4, 5, 6
10. 1, 2, 3, 4, 5, 6

La sous-séquence entre crochets indique les éléments à inverser à chaque itération. Après la dixième itération, le chiffre 1 se trouve en première position. L'algorithme s'arrête. Donc pour  $n = 6$ , il existe une permutation,  $(3, 6, 5, 1, 4, 2)$ , qui génère 10 itérations du mécanisme proposé. Le défi est de trouver, pour  $n$  donné, la permutation qui génère le maximum d'itérations !

QUESTION 1. Pour commencer, il faut être capable d'inverser les éléments dans une liste. Ajouter à la bibliothèque d'opérations sur les listes circulaires doublement chaînées, une fonction d'inversion des  $k$  premiers éléments.

**Remarque:** Il est possible soit d'inverser les valeurs des cellules, soit d'inverser les cellules elles-mêmes. C'est cette dernière approche qui est préconisée.

QUESTION 2. Ensuite, nous avons besoin d'une fonction qui énumère une à une, toutes les permutations d'une liste circulaire doublement chaînée. Écrire cette fonction.

**Remarque:** L'énumération des permutations s'écrit aisément de manière récursive : l'étape d'induction considère successivement l'insertion de l'élément de tête à toutes les positions de chaque permutation du reste de la liste.

QUESTION 3. Résoudre le problème *Topswops* par énumération exhaustive des permutations, pour  $n \in \llbracket 4, 9 \rrbracket$ .

**Remarque:** *La solution a été trouvée pour  $n = 97$ . N'essayez pas !*

## 2 Files, piles et dèques

Au cours du TP précédent, nous avons vu comment implémenter les types abstraits *pile* et *file d'attente* de plusieurs manières. En fait, les files et les piles sont des cas particuliers de la *file d'attente à deux extrémités* nommée *dèque*.

Dans cette partie, nous allons donc nous intéresser à une implémentation alternative des piles et des files d'attente, à partir d'un nouveau type *dèque*.

### Exercice 4: Implémentation de dèques à l'aide de listes chaînées

Un dèque est une file d'attente dans laquelle il est possible d'entrer et de sortir aux deux extrémités. En cela, il combine les opérations sur les files d'attente et sur les piles.

Créer une classe `Deque` de file d'attente à deux extrémités pour des nombres entiers, par composition à partir des listes circulaires doublement chaînées `LinkedList`.

Proposer l'interface de programmation pour la classe `Deque` qui regroupe toutes les primitives pour dèques vues en cours sous forme de méthodes d'instance :

### Exercice 5: La valeur maximale d'une fenêtre glissante

Étant donné une séquence d'entiers, on souhaite fournir la valeur maximale de toute sous-séquence de  $k$  entiers consécutifs. Il s'agit en pratique de calculer un agrégat (le max) d'une fenêtre glissante de taille  $k$  sur un flux de données.

La séquence initiale est de type `list[int]` et peut être construite à l'aide de la fonction `remplir_tableau` du module `tp1.util`.

QUESTION 1. Écrire une première version très simple du programme à l'aide d'une double boucle sur les éléments de la séquence et sur ceux de la fenêtre courante.

QUESTION 2. Écrire une version plus astucieuse, à l'aide d'un dèque (type `Deque`) en charge de mémoriser la valeur maximale courante et les éventuelles valeurs maximales suivantes.

**Remarque:** *Le dèque contient toujours la valeur maximale en tête et insère les valeurs suivantes par la queue. Il est à noter que toute valeur du dèque qui serait moins grande que la valeur courante n'aurait aucune chance de figurer dans les réponses (le max) des fenêtres suivantes !*

QUESTION 3. Comparer les deux résolutions, en terme de coût.

### Exercice 6: Retour sur les piles et les files

QUESTION 1. Proposer une implémentation alternative des types `Stack` et `Queue` à partir de `Deque`. Tous les opérateurs sur les files et les piles seront redéfinis.

QUESTION 2. Vérifier que les programmes qui utilisent les types `Stack` et `Queue` fonctionnent parfaitement avec cette implémentation alternative.

### 3 Pour aller plus loin...

#### Exercice 7: Algorithme de Graham

L'enveloppe convexe d'un ensemble de points est la plus petite forme convexe qui les contient tous. Pour un ensemble fini de points, cette enveloppe est un polygone. Il existe de nombreux algorithmes d'extraction d'enveloppe convexe. Celui de GRAHAM repose sur l'idée que les points d'un polygone convexe tournent dans le même sens. Une fois les points triés (ici, dans le sens trigonométrique), il suffit de supprimer ceux qui provoquent des concavités.

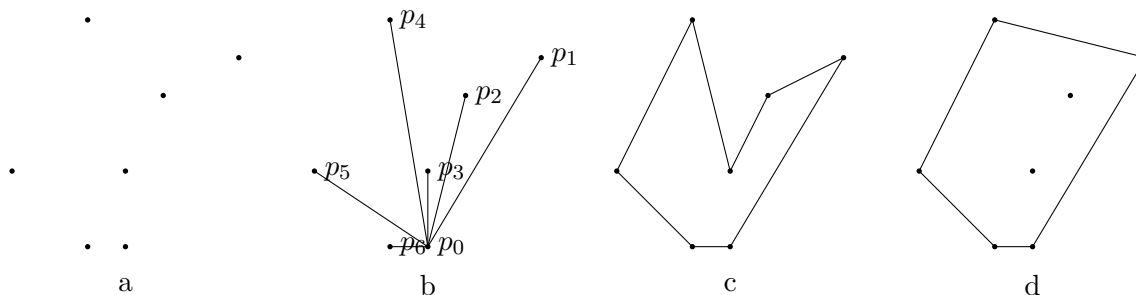


FIGURE 1 – Illustration du principe de l'algorithme de GRAHAM. (a) : un ensemble de points. (b) : choix du point pivot  $p_0$  et tri angulaire des autres points. (c) : polygone non convexe formé par les points ordonnés. (d) : enveloppe convexe obtenue après suppression des concavités.

Cet algorithme met en jeu la recherche dans une liste, le tri d'une liste et la structure de pile.

QUESTION 1. On dispose de  $n$  points représentés par une liste de tuples  $(x, y)$ . Déterminer le point pivot  $p_0$ , situé en bas à droite de tous les autres (celui qui a la plus petite ordonnée ou, à ordonnée minimale, celui qui a la plus grande abscisse).

QUESTION 2. Trier les autres points selon le sens trigonométrique.  $p_i$  et  $p_j$  sont dans l'ordre trigonométrique si  $p_j$  est à gauche de  $\overrightarrow{p_0 p_i}$ , autrement dit si le déterminant  $\det(\overrightarrow{p_0 p_i}, \overrightarrow{p_0 p_j}) = (x_i - x_0)(y_j - y_0) - (y_i - y_0)(x_j - x_0)$  est positif. Écrire une fonction intermédiaire **determinant**, qui servira également à la question suivante.

QUESTION 3. Supprimer les concavités selon l'algorithme suivant. Les points  $p_0$  et  $p_1$  sont placés dans une pile qui contiendra finalement les points de l'enveloppe convexe. Par construction,  $p_0$  et  $p_1$  appartiennent nécessairement à l'enveloppe convexe. On considère le point  $p_k$ ,  $k$  étant initialisé à 2. Tant que  $k$  est inférieur à  $n$  ( $n = 7$  sur la Figure 1), on considère à chaque étape les deux points  $p_i$  et  $p_j$  en haut de la pile. Ils forment la dernière arête du polygone en construction. Si  $p_k$  est à gauche de  $\overrightarrow{p_i p_j}$  alors on empile  $p_k$  et on passe au point suivant (on incrémente  $k$ ), sinon, on enlève le point qui crée une concavité (on dépile  $p_j$  sans changer  $k$ ). On arrête quand on a empilé le dernier point (à cette étape,  $k = n - 1$ ).

#### 3.1 Les listes à enjambements

Les listes à enjambements (ou *skip list* en anglais) sont des listes chaînées triées multi-couches. La première couche consiste en une liste chaînée usuelle dont les éléments ont été ordonnés, tandis que chaque couche supérieure offre un « raccourci » pour la liste immédiatement inférieure :

```
couche 3:  A
couche 2:  A---C-----G
couche 1:  A---C-D-----G---I
couche 0:  A-B-C-D-E-F-G-H-I-J
```

En pratique, les couches supérieures ne sont constituées que de références vers les cellules de la première couche. La recherche dans une liste à enjambements s'opère de la couche la plus haute vers la plus basse, en se déplaçant au plus près de la valeur recherchée dans chaque couche. Par exemple, le chemin de recherche de la valeur H est :

```
A[3]
-> A[2] -> C[2] -> G[2]
      -> G[1]
      -> G[0] -> H[0]
```

L'insertion procède de la même manière pour établir le rang du nouvel élément au sein de la liste ordonnée. On insère toujours l'élément dans la première couche. La décision de faire figurer un élément dans la couche supérieure est totalement aléatoire! Les *skip list* sont en effet des structures probabilistes. Le paramètre clé d'une structure de *skip list* est donc la probabilité  $p$  de figurer dans la couche supérieure. En pratique, on utilise souvent la valeur  $p = 0,5$  qui suggère que la moitié des éléments d'une couche est supposée constituer la couche supérieure. Le nombre de couches est alors variable et déterminé par les tirages aléatoires successifs lors de l'insertion de nouveaux éléments. Il est toutefois d'usage de fixer une borne supérieure  $\ell$  au nombre de couches.

Pour une présentation détaillée des *skip list*, vous pouvez consulter [ce diaporama](#), ou commencer simplement par l'[entrée Wikipédia](#). Pour les plus hardis, il est possible de consulter l'[article original de W. Pugh \(1990\)](#) qui présente les *skip list*.

## Exercice 8: Première implémentation

QUESTION 1. Implémenter un type `SkipList` de nombres entiers, muni de ses opérations élémentaires : `new`, `is_empty`, `insert`, `delete`, `lookup`.

QUESTION 2. Établir les performances—complexités—des opérations sur les *skip list*, en tenant compte du paramètre  $p$ .

## Exercice 9: La médiane mobile

Il s'agit de calculer la valeur médiane pour chaque fenêtre glissante de taille  $w$  sur un tableau de valeurs entières de taille  $n$ .

Avec une liste à enjambements pour stocker les éléments de la fenêtre courante de manière ordonnée, le procédé requiert une recherche par index (recherche de l'élément au rang  $w//2$ ). Or, cette recherche est a priori en  $O(n)$ , sauf si l'on ajoute une information de « rang relatif » avec toute référence vers une cellule.

L'exemple précédent devient :

```
couche 3:  A/0
couche 2:  A/0-----C/2-----G/4
couche 1:  A/0-----C/2-D/1-----G/3-----I/2
couche 0:  A/0-B/1-C/1-D/1-E/1-F/1-G/1-H/1-I/1-J/1
```

Il s'interprète comme suit : le parcours de la couche 2 nous renseigne sur les rangs de A, C et G. Le rang de A est obtenu par un saut de 0 à partir du début de la liste et vaut donc 0! Le rang de C est alors  $\text{rang}(A) + 2$  soit 2. Le rang de G est quant à lui  $\text{rang}(C) + 4 = 6$ . Ce procédé fonctionne également le long de chemins descendants. Par exemple, chercher l'élément de rang 7 consiste à parcourir le chemin :

```
A[3] (0)
-> A[2] (0) -> C[2] (0+2=2) -> G[2] (2+4=6)
      -> G[1] (6)
      -> G[0] (6) -> H[0] (6+1=7!)
```

QUESTION 1. Modifier le type `SkipList` pour y intégrer l'information de rang relatif.

QUESTION 2. Proposer une résolution pour le calcul de la médiane mobile.

# Sujet n°5

## Algorithmique et Structure de Données Algorithmes de tri

Temps de réalisation: 1h30mn

L'objectif de ce TP est de mettre en pratique les algorithmes de tri.

Échelle de progression :

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

Ex.	01	02	03	04	05	06	07	08	09	10
	●	●	●	●	●	●	●	●	●	●

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

<b>1</b>	<b>Mise en place</b>	<b>1</b>
	Exercice 1 : Génération de 6 listes . . . . .	2
	Exercice 2 : Calcul complexité en temps . . . . .	2
	Exercice 3 : Tableau comparatif . . . . .	2
<b>2</b>	<b>Les algorithmes de tri en <math>O(n^2)</math></b>	<b>2</b>
	Exercice 8 : Implémentation . . . . .	2
	Exercice 9 : Calcul du temps de réponse . . . . .	2
<b>3</b>	<b>Les algorithmes de tri en <math>O(n \log_2 n)</math></b>	<b>2</b>
	Exercice 8 : Implémentation . . . . .	2
	Exercice 9 : Calcul du temps de réponse . . . . .	2
<b>4</b>	<b>Les algorithmes de tri en <math>O(n)</math></b>	<b>3</b>
	Exercice 8 : Implémentation . . . . .	3
	Exercice 9 : Calcul du temps de réponse . . . . .	3
<b>5</b>	<b>Interpretation des résultats</b>	<b>3</b>
	Exercice 10 : Interpretation des résultats . . . . .	3

## 1 Mise en place

Dans ce TP, vous allez tester tous les algorithmes de tri que nous avons vu en cours.

Nous souhaitons dans ce TP, en plus de tester les algorithmes de tri sur certaines données, de comparer les résultats obtenus sur les mêmes données.

Ainsi, vous devez dans la suite réaliser les étapes présentées ci-dessous.

### Exercice 1: Génération de 6 listes

Générer 6 listes de 10, 100, 1000, 10000, 100000 et 1000000 éléments entiers. Les éléments seront générés de manière aléatoire en utilisant la bibliothèque *random*.

```
import random

n = random.randint(0, 100000)
print(n)
```

### Exercice 2: Calcul complexité en temps

Mettez en place une architecture pour pouvoir calculer la complexité en temps de chacun des algorithmes que vous allez développer dans la suite. Vous allez exécuter ces algorithmes sur les 6 listes que vous avez générées auparavant.

Le code ci-dessous permet de calculer le temps d'exécution de l'algorithme ALGO exécuté sur la liste LISTE.

```
import time

start_time = time.time()
result = ALGO_TRI(LISTE)
end_time = time.time()
print(f"Complexité en temps d'exécution: {end_time - start_time} seconds")
```

### Exercice 3: Tableau comparatif

Mettez en place un tableau comparatif qui prend sur les lignes les 6 listes et sur les colonnes les différents algorithmes de tri. Les valeurs de ce tableau correspondent au temps d'exécution de chaque algorithme sur chaque liste.

## 2 Les algorithmes de tri en $O(n^2)$

#### Exercice 4: Implémentation

Implémentez les algorithmes de tri en  $O(n^2)$  vus en cours.

#### Exercice 5: Calcul du temps de réponse

Calculer le temps de réponse des algorithmes implémentés et ajouter les valeurs dans le tableau créé auparavant.

## 3 Les algorithmes de tri en $O(n \log_2 n)$

#### Exercice 6: Implémentation

Implémentez les algorithmes de tri en  $O(n \log_2 n)$  vus en cours.

#### Exercice 7: Calcul du temps de réponse

Calculer le temps de réponse des algorithmes implémentés et ajouter les valeurs dans le tableau créé auparavant.

## 4 Les algorithmes de tri en $O(n)$

### Exercice 8: Implémentation

Implémentez les algorithmes de tri en  $O(n)$  vus en cours.

### Exercice 9: Calcul du temps de réponse

Calculer le temps de réponse des algorithmes implémentés et ajouter les valeurs dans le tableau créé auparavant.

## 5 Interpretation des résultats

### Exercice 10: Interpretation des résultats

A partir du tableau des temps de réponse, donnez votre interpretation par rapoort aux algorithmes de tri implémentés.

# Sujet n°6

## Algorithmique et Structure de Données

### Les arbres

Temps de réalisation: 3h00mn

Les objectifs du TP sont :

- ▶ définir et manipuler des structures d'arbres
- ▶ exploiter des arbres binaires de recherche
- ▶ mobiliser ces structures pour la résolution de problèmes

Échelle de progression :

● « débutant·e »

● « confirmé·e »

● « avancé·e »

● « expert·e »

Ex.	01	02	03	04	05	06
	●	●	●	●	●	●

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

Exercice 1 : Les arbres binaires . . . . .	1
Exercice 2 : Fonctions avancées sur les arbres binaires . . . . .	3
Exercice 3 : Les expressions arithmétiques . . . . .	4
Exercice 4 : Arbres binaires de recherche . . . . .	5
Exercice 5 : Conversion d'un ABR vers une liste doublement chaînée . . . . .	6
<b>1 Pour aller plus loin...</b>	<b>6</b>
Exercice 6 : Codage de Huffman . . . . .	6

### Exercice 1: Les arbres binaires

QUESTION 1. Proposer une structure de données `BinaryTree` pour représenter par « liaison explicite » un type abstrait *arbre binaire* dont les clés sont des nombres entiers. Les nœuds de l'arbre sont de type `Node`.

**Remarque:** L'usage d'un nœud fictif que l'on nomme *terminal*, vers lequel pointe chaque feuille de l'arbre, présente des avantages pour les algorithmes de parcours d'arbre. Il remplace alors la valeur `None`. Attention toutefois, ce nœud dit *terminal* est commun à tous les arbres de la classe `BinaryTree` et se déclare soit de manière globale, soit dans la classe `BinaryTree` à l'aide de l'annotation de type `ClassVar` du module `typing`<sup>1</sup>.

1. Dans le langage de la programmation par objets, il s'agit d'une variable de classe.

**Astuce:** Pour faciliter l'appréhension de la notion de nœud terminal, il est recommandé de la représenter sur l'arbre binaire de la Figure 1.

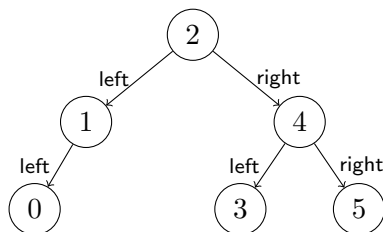


FIGURE 1 – Exemple d'arbre binaire construit à partir de la séquence (2, 1, 4, 0,  $\perp$ , 3, 5).

QUESTION 2. Écrire les premières fonctions :

```
def isEmpty() -> bool
def root() -> Node
def terminal() -> Node
```

Un arbre vide est un arbre sans racine. La fonction `terminal` retourne le nœud terminal de l'arbre ou à défaut, `None`.

QUESTION 3. Proposer un constructeur d'arbres binaires de signature :

```
def __init__(nodes: list[int | None] | None = None) -> BinaryTree
```

Si `nodes` vaut `None` ou `[]`, alors le constructeur fournit un arbre binaire vide.

Le paramètre `nodes` est un tableau de valeurs entières qui encode l'arbre dans son intégralité. Les nœuds sont énumérés par niveau, de gauche à droite, de telle sorte que pour tout nœud d'indice  $i$ , ses fils gauche et droit se trouvent resp. aux indices  $2i + 1$  et  $2i + 2$  dans le tableau. Par exemple, l'arbre binaire de la Figure 1 est encodé par le tableau :

```
>>> t = [2, 1, 4, 0, None, 3, 5]
```

On observe que les fils gauche et droit de  $t[1]=1$  par exemple, sont respectivement  $t[3]=0$  et  $t[4]=\text{None}$ , suivant le motif des indices  $2 \times 1 + 1$  et  $2 \times 1 + 2$ . De la même manière, les fils de  $t[2]=4$  sont  $t[5]=3$  et  $t[6]=5$ . Notez que les fils de  $t[3]=0$  sont théoriquement  $t[7]$  et  $t[8]$  qui n'existent pas : 0 est donc une feuille. De façon similaire, tous les fils de `None` sont soit hors tableau, soit `None` eux-mêmes.

**Remarque:** L'encodage sous la forme du tableau `nodes` est en fait une autre implémentation possible pour les arbres binaires (sic!).

**Astuce:** La conversion du tableau en une structure hiérarchique de `Node` s'opère de façon simple par récursivité sur les nœuds. Il s'agit, pour tout indice  $i$  du tableau, d'observer que le sous-arbre gauche est construit récursivement à l'indice  $2i + 1$ , et le sous-arbre droit de la même manière à l'indice  $2i + 2$ . Ensuite, ne reste plus qu'à créer le nœud courant à l'aide de la valeur à l'indice  $i$  et des deux sous-arbres.

QUESTION 4. Écrire les fonctions suivantes, par récursivité sur les nœuds :

```
def height() -> int    # calcule la hauteur
def size() -> int      # calcule le nombre de noeuds
```

La hauteur d'un arbre est la taille du plus long chemin de la racine vers une feuille. La hauteur d'un arbre vide est, par convention égale à  $-1$ .

## Exercice 2: Fonctions avancées sur les arbres binaires

QUESTION 1. Écrire un test d'égalité de 2 arbres binaires :

```
def isEqual(bt2: BinaryTree) -> bool
```

Le test retourne *vrai* si les arbres  $bt_1$  et  $bt_2$  ont la même structure et les mêmes valeurs dans chaque nœud.

**Remarque:** Le test d'égalité ci-dessus réalise une comparaison des 2 objets « en profondeur », par opposition aux tests d'égalité superficielle  $bt1 \text{ is } bt2$  et par défaut  $bt1 == bt2$ .

QUESTION 2. Écrire la fonction de test suivante :

```
def isHeap() -> bool
```

qui vérifie que pour tout nœud de l'arbre, la valeur est supérieure ou égale à la valeur de chacun de ses fils non vides. On a alors à faire à une structure de tas (*heap* en anglais). La fonction renvoie *vrai* si cette propriété est satisfaite et *faux* sinon.

QUESTION 3. Écrire la fonction

```
def lca(a: int, b: int) -> int
```

qui calcule le plus petit ancêtre commun<sup>2</sup> (*Lowest Common Ancestor* ou LCA en anglais) des clés  $a$  et  $b$  présentes dans l'arbre  $bt$ . L'hypothèse supplémentaire est que chaque nœud porte une valeur entière unique. Cela garantit que  $a$  et  $b$  désignent précisément deux nœuds de l'arbre.

**Astuce:** Une manière élégante de mener ce calcul de LCA passe par la recherche de  $a$  et de  $b$  dans chacun des sous-arbres d'un nœud  $x$  donné, en partant de la racine ;  $x$  est alors le LCA de  $a$  et  $b$  ssi les deux clés ne se trouvent pas dans le même sous-arbre !

**Remarque:** Une manière plus évidente mais moins efficace consiste à retenir les chemins de la racine à  $a$  et à  $b$ , puis éliminer les préfixes communs.

QUESTION 4. Écrire la fonction de représentation sous forme de chaîne de caractères d'un arbre binaire :

```
def str() -> str
```

de telle sorte qu'elle réalise un « joli affichage » (*pretty print*) d'un arbre binaire, proche de l'exemple suivant :

```
>>> print(tree.str())
  __ (2) ____
 _ (1)      _ (4) _
(0)        (3)   (5)
```

---

2. Célèbre problème qui fut énoncé la première fois par Aho, Hopcroft et Ullman en 1973.

**Astuce:** La fonction doit être capable de parcourir l'arbre en largeur d'abord (breadth-first search), autrement dit niveau par niveau. Une fois que ce parcours est en place, il reste à transmettre certains paramètres de positionnement d'un appel récursif à l'autre pour formater chaque ligne—représentant un niveau—de l'arbre en fonction de la situation « d'en dessous ».

### Exercice 3: Les expressions arithmétiques

On considère des arbres qui représentent des expressions arithmétiques. On appelle cette structure un *arbre de syntaxe abstraite* (ASA ou *Abstract Syntax Tree*—AST—en anglais). Les valeurs des nœuds intermédiaires sont des opérateurs parmi  $\{+, -, \times, \div\}$ , et celles des feuilles sont des nombres. Par exemple, l'expression arithmétique préfixée

$(+ (- 4 2) (* 3 5))$  ou sans parenthèses :  $+ - 4 2 * 3 5$

sera représentée par l'arbre binaire de la Figure 2.

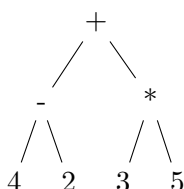


FIGURE 2 – ASA de l'expression arithmétique  $(4 - 2) + (3 \times 5)$ .

Nous allons représenter un ASA d'expressions arithmétiques à l'aide du type `BinaryTree`. Ne disposant que d'arbres binaires de nombres entiers, il est possible d'encoder les valeurs des nœuds intermédiaires, par exemple selon la correspondance arbitraire  $\{+/0, -/1, \times/2, \div/3\}$ .

Dans la suite de l'exercice, nous utilisons l'alias de type `ASTree` pour désigner ce type d'arbres. Déclarer un alias de type est trivial :

```
ASTree = BinaryTree
```

QUESTION 1. Écrire la fonction

```
def eval_ast(ast: ASTree) -> float
```

qui, étant donné un ASA, calcule le résultat numérique de l'expression correspondante. On supposera que l'expression est bien formée, et par conséquent, que l'ASA est strict<sup>3</sup>.

QUESTION 2. Ajouter la fonction « constructeur »

```
def exp_to_ast(tokens: list[str]) -> ASTree
```

qui considère l'expression arithmétique sous forme de liste de termes (`tokens`) et qui construit l'ASA correspondant. Si la syntaxe est incorrecte, l'ASA est vide.

QUESTION 3. Enfin, combiner les briques précédentes pour définir la fonction

```
def eval(expr: str) -> float | None
```

qui prend en paramètre une chaîne de caractères `expr` représentant une expression arithmétique préfixée—sans parenthèses—et qui renvoie le résultat de l'évaluation de l'expression correspondante si elle est bien formée, `None` sinon. La fonction doit être capable de traiter des expressions en nombres entiers.

3. Chaque nœud possède soit 0 fils soit 2 fils.

**Astuce:** Pour l'examen de la chaîne de caractères, il est recommandé d'utiliser la fonction `str.split`.

#### Exercice 4: Arbres binaires de recherche

Dans cet exercice, nous nous intéressons aux *arbres binaires de recherche* (ABR ou *Binary Search Tree*—BST—en anglais), une structure de données permettant de représenter une collection d'éléments ordonnés par leurs clés—des valeurs quelconques munies d'une relation d'ordre. La collection peut être un ensemble, un tableau associatif, parfois même un multi-ensemble.

Pour mémoire, un ABR, aussi appelé *arbre binaire ordonné*, est un arbre binaire tel que pour tout nœud  $x$  :

- toutes les clés du sous-arbre gauche de  $x$  sont inférieures—ou égales—à la clé de  $x$  ;
- toutes les clés du sous-arbre droit de  $x$  sont supérieures à la clé de  $x$ .

Étant donné que les clés sont *a priori* triées, les ABR offrent l'avantage d'une procédure efficace de recherche de clé dans un ensemble, à condition que l'arbre soit bien équilibré. De même, les ABR permettent de trouver aisément le minimum et le maximum d'un ensemble, ainsi que les majorants et minorants d'une clé.

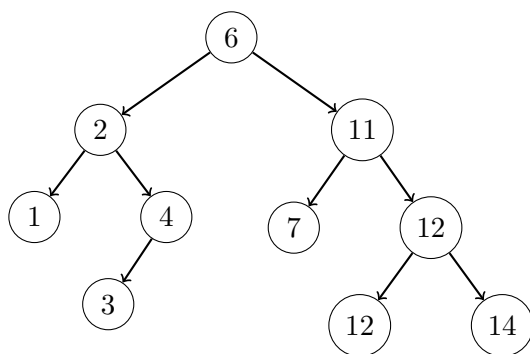


FIGURE 3 – Exemple d'ABR qui représente un multi-ensemble de nombres entiers.

Pour l'exercice, nous définissons un alias de type :

```
BSTree = BinaryTree    # ABR
```

QUESTION 1. Écrire la fonction :

```
def isBst() -> bool
```

qui teste si un arbre binaire est un ABR.

**Astuce:** À chaque étape d'un parcours infixe, il faut tester la clé du nœud vis-à-vis de bornes inférieure et supérieure, établies pendant le parcours.

QUESTION 2. Écrire successivement les fonctions :

```
def lookup_rec(key: int) -> Node
def lookup_it(key: int) -> Node
```

qui recherchent la clé `key` dans un ABR, par une procédure récursive (`_rec`) puis itérative (`_it`). Quelle est la complexité de ces algorithmes ?

QUESTION 3. Écrire la fonction :

```
def insert(key: int) -> BSTree
```

qui ajoute la clé `key` à l'ensemble (sans doublons) représenté par l'ABR `bst`. Quelle est la complexité de cet algorithme ?

QUESTION 4. Proposer une fonction de suppression de clé dans un ABR :

```
def remove(key: int) -> BSTree
```

## Exercice 5: Conversion d'un ABR vers une liste doublement chaînée

Selon Nick Parlante<sup>4</sup>, il s'agit de « *one of the neatest recursive pointer problems ever devised* », autrement dit, un problème compliqué de manipulation de références !

L'idée est pourtant simple : un ABR représente un ensemble totalement ordonné d'éléments, donc il devrait être possible de le convertir en une liste qui respecte cet ordre. Pour ce faire, nous disposons du type `LinkedList` de listes circulaires doublement chaînées. Or, il est aisé d'observer l'analogie entre un maillon de la liste qui maintient deux références vers son prédécesseur et son successeur, et un nœud de l'arbre binaire de recherche qui maintient également deux références vers son fils gauche et son fils droit !

Le problème peut s'énoncer ainsi : étant donné un ABR, proposer une fonction *récursive*

```
def toList() -> LinkedList
```

de conversion en une liste doublement chaînée qui présente les éléments dans l'ordre. Chaque référence vers le sous-arbre gauche, resp. droit, doit correspondre à une référence vers le maillon précédent, resp. suivant.

**Remarque:** *Le cœur du problème réside dans l'assemblage des appels récursifs pour réorganiser les références de chaque sous-arbre lors de la phase d'induction, plutôt que dans la compréhension fine de ce qui se produit lors des appels récursifs*<sup>5</sup>.

**Astuce:** *La phase d'induction crée deux sous-listes, une par sous-arbre, et concatène ces sous-listes autour du nœud courant.*

## 1 Pour aller plus loin...

### Exercice 6: Codage de Huffman

Le codage de Huffman est une technique de compression de données sans perte. C'est une technique fréquentielle –statistique– au même titre que le morse, qui met en œuvre un principe simple : plus le symbole est fréquent, plus son code sera court. L'encodage d'un message fonctionne en 3 étapes :

1. compter le nombre d'occurrences de chaque caractère ;
2. construire l'arbre de codage ;
3. suivre dans l'arbre le chemin gauche-droite (0-1) de chaque caractère pour produire le code du message.

---

4. <http://cslibrary.stanford.edu/109/TreeListRecursion.html>

5. C'est trop compliqué ! Et ici repose la magie de la récursivité...

Le décodage fonctionne de la même manière, à partir du message codé et de l'arbre de codage : pour décoder un caractère, il suffit de suivre dans l'arbre la séquence de bits du message codé, de la racine à une feuille.

La construction de l'arbre d'encodage suit le procédé ci-dessous :

1. initialiser un ensemble d'arbres enracinés par chaque caractère, chacun étant associé à un poids correspondant au nombre d'occurrences du caractère dans le message ;
2. fusionner deux à deux les arbres en prenant systématiquement les arbres de poids minimum. Les deux arbres deviennent les sous-arbres gauche et droite, le plus léger à gauche ; en cas d'égalité, on convient d'utiliser l'ordre lexicographique. Le poids de l'arbre résultat est la somme des poids des deux arbres.

Le dernier arbre est l'arbre de codage. Par exemple, le processus complet en 4 étapes, à partir du message  $m = \text{aabbcccccddddddddd}$ , est détaillé ci-dessous :

1. initialisation : (a), 2 ; (b), 3 ; (c), 4 ; (d), 11
2. après la fusion de 2 et 3 : (c), 4 ; (. a b), 5 ; (d), 11
3. après la fusion de 4 et 5 : (. (c) (. a b)), 9 ; (d), 11
4. après la fusion de 9 et 11 : (. (. (c) (. a b)) (d)), 20

L'arbre de codage résultat de ce procédé est dessiné à la Figure 4.

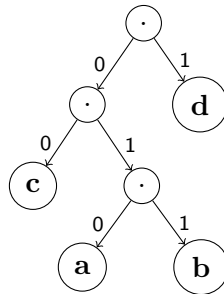


FIGURE 4 – Arbre de codage de Huffman pour  $(a, 2); (b, 3); (c, 4); (d, 11)$ .

Ainsi, le code de Huffman  $H_m(.)$  de chaque caractère du message  $m$  est :

- $H_m(a) = 010$
- $H_m(b) = 011$
- $H_m(c) = 00$
- $H_m(d) = 1$

Notez qu'aucun code n'est le préfixe d'un autre ! Cette propriété importante du codage de Huffman permet de décoder un message de manière totalement déterministe.

QUESTION 1. Écrire la fonction d'encodage qui produit un message codé à partir d'un message clair et de son arbre de codage. Il pourra être utile de construire la table d'encodage comme structure intermédiaire, qui donne pour tout caractère  $c$ , son code  $H_m(c)$ .

QUESTION 2. Écrire la fonction de décodage qui produit un message clair à partir d'un message codé et de son arbre de codage.

QUESTION 3. Écrire la fonction de construction de l'arbre de codage à partir d'un message clair.

QUESTION 4. Appliquer le codage/décodage de Huffman aux messages suivants :

- "intelligence artificielle"
- "quarante crackers croient que croquer des carottes crues crée des crampes"

Calculer à chaque fois le taux de compression, en considérant que les messages sont encodés en ASCII, soit 7 bits par caractère.

**En marge:** cf. *le codage entropique vu en théorie de l'information*.

# Sujet n°7

## Algorithmique & Structures de données

### Les graphes<sup>1</sup>

Temps de réalisation: 3h00mn

L'objectif du TP est de :

- définir des structures de données pour représenter un graphe
- implémenter les opérations élémentaires sur les graphes
- résoudre quelques problèmes classiques de graphes

Échelle de progression :

● « débutant·e »	■	● « confirmé·e »	■	● « avancé·e »	■	● « expert·e »	■
Ex.	01	02	03	04	05	06	07
	●	●	●	●	●	●	●

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*(●)àl'issueduTP.

## Table des matières

<b>1</b>	<b>Vers un type Graph</b>	<b>2</b>
	Exercice 1 : Les représentations d'un graphe . . . . .	2
	Exercice 2 : Vers une bibliothèque de manipulation de graphes . . . . .	4
	Exercice 3 : Génération de graphes . . . . .	5
<b>2</b>	<b>Algorithmique de graphe</b>	<b>5</b>
	Exercice 4 : Parcours de graphe . . . . .	5
	Exercice 5 : Connexité . . . . .	6
<b>3</b>	<b>Pour aller plus loin...</b>	<b>7</b>
	Exercice 6 : Arbre couvrant de poids minimum . . . . .	7
	Exercice 7 : Plus court chemin . . . . .	7

## Introduction

Un graphe non orienté  $G$  est la donnée d'un ensemble de sommets  $V$  (pour *Vertices*) et d'un ensemble d'arêtes  $E \subseteq \{\{x, y\} : x, y \in V\}$  (pour *Edges*) liant deux sommets entre eux. Deux sommets

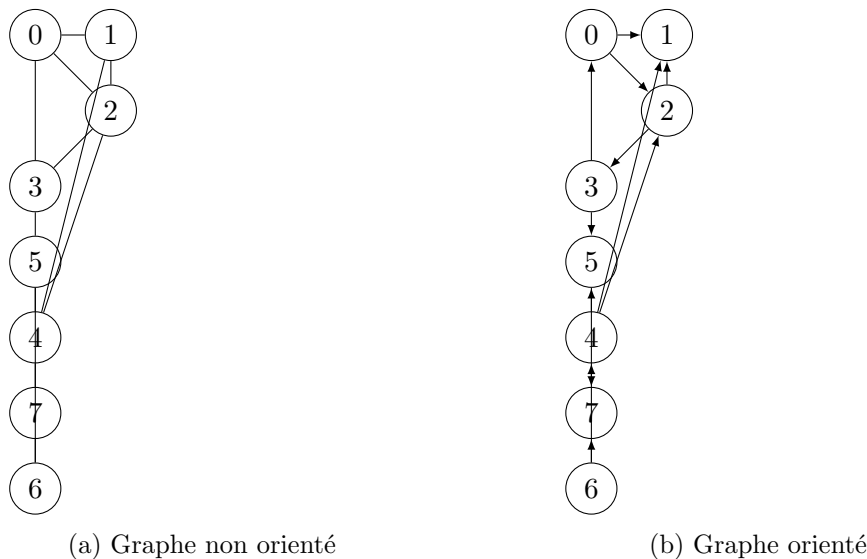


FIGURE 1 – Exemples de graphes simples.

liés par une arête sont dits *adjacents*—ou voisins. Un exemple de graphe simple<sup>2</sup> non orienté  $G$  est donné sur la Figure 1a avec  $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$  et  $E = \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}, \{4, 6\}, \{5, 7\}, \{6, 7\}\}$ .

Un graphe orienté  $\vec{G}$  est la donnée d'un ensemble de sommets  $V$  et d'un ensemble d'arcs  $E \subseteq V \times V$  liant un sommet à un autre. Si  $(u, v)$  est un arc,  $u$  est un *prédécesseur* de  $v$ , et  $v$  est un *successeur* de  $u$ . La Figure 1b montre un exemple de graphe orienté  $\vec{G}$ .

## 1 Vers un type Graph

### Exercice 1: Les représentations d'un graphe

On décrit la structure d'un graphe principalement à l'aide de ses **listes d'adjacence**, ou d'une **matrice d'adjacence**<sup>3</sup>.

On rappelle que la matrice d'adjacence  $\text{Adj}$  d'un graphe orienté  $G = (V, E)$  est une matrice carrée de taille  $|V| \times |V|$ , telle que :

$$\text{Adj}_{i,j} = \begin{cases} 1 & \text{si } (i, j) \in E \\ 0 & \text{sinon} \end{cases}$$

On définit de la même manière la matrice d'adjacence d'un graphe non orienté. Notez que celle-ci est alors symétrique.

Les listes d'adjacence d'un graphe non orienté—resp. orienté— $G = (V, E)$  correspondent aux  $|V|$  listes des voisins—resp. successeurs—de chaque sommet.

Parmi les représentations alternatives, on trouve la matrice d'incidence, le *line graph*, ou encore l'ensemble des cliques maximales. En pratique, la matrice d'adjacence et les listes d'adjacence sont les représentations les plus usitées car elles offrent de bonnes performances dans la plupart des résolutions de problèmes sur les graphes. Le choix des listes d'adjacence ou de la matrice d'adjacence est fonction de la *densité* du graphe : un graphe creux sera avantageusement représenté par des listes d'adjacence, tandis qu'un graphe dense se prête mieux à une représentation sous forme matricielle.

1. d'après un sujet de Guillaume Raschia

2. graphe sans arête réflexive ni arête dédoublée.

3. dont une extension est la matrice Laplacienne, obtenue par conjonction avec la matrice des degrés du graphe.

QUESTION 1. Donner la complexité en espace de la représentation d'un graphe par :

1. matrice d'adjacence ;
2. listes d'adjacences.

QUESTION 2. Donner, pour chacune des 2 représentations—listes ou matrice d'adjacence—, la complexité en temps de :

1. la recherche des voisins ;
2. l'ajout d'arête ;
3. la suppression d'arête ;
4. le test d'adjacence entre 2 sommets.

QUESTION 3. Écrire la fonction

```
def am_is_undirected(adj_m: AdjMatrix) -> bool
```

qui accepte une matrice d'adjacence `adj_m` et décide s'il peut ou non s'agir d'un graphe non orienté. On supposera que la matrice d'adjacence est bien formée, ie. elle est carrée et n'admet que des valeurs parmi  $\{0, 1\}$ .

En outre, nous utilisons l'alias `AdjMatrix` pour désigner simplement le type des listes de listes de nombres entiers, là où elles sont employées comme des matrices d'adjacence :

```
AdjMatrix = List[List[int]]
```

QUESTION 4. Proposer, de manière similaire, l'alias de type `AdjLists` pour désigner le type des listes d'adjacence d'un graphe. Nous considérons des graphes dont les sommets sont repérés par—et limités à—leurs clés, ie. des chaînes de caractères.

Un graphe pondéré—orienté ou non—, aussi appelé graphe valué, est la donnée d'un triplet  $G = (V, E, \nu)$  dans lequel  $\nu$  est une fonction de valuation qui affecte un poids à chaque arc—ou arête—du graphe. Sur la Figure 2, l'arête  $\{4, 6\}$  du graphe pondéré a la valeur 5 :  $\nu(\{4, 6\}) = 5$ . On supposera dans la suite que les poids sont des valeurs entières strictement positives.

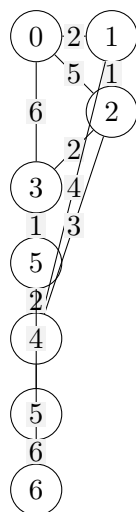


FIGURE 2 – Exemple de graphe valué.

QUESTION 1. Quel est l'impact de la valuation d'arcs/arêtes sur les types `AdjMatrix`, `AdjLists` et sur la fonction `am_is_undirected`? Modifier le code en conséquence.

Désormais toutes les structures de données et les algorithmes proposés doivent accepter des graphes simples, orientés ou non, pondérés ou non.

QUESTION 2. Écrire la fonction

```
def al_is_undirected(adj_l: AdjLists) -> bool
```

qui accepte les listes d'adjacence d'un graphe et décide s'il peut ou non s'agir d'un graphe non orienté.

QUESTION 3. Écrire la fonction

```
def al_undirect(adj_l: AdjLists) -> None
```

qui complète les listes d'adjacence `adj_l` pour transformer le graphe orienté en graphe non orienté.

QUESTION 4. Écrire les fonctions de conversion suivantes :

```
def am_to_al(adj_m: AdjMatrix, vertices: Sequence[str] = ()) -> AdjLists
def al_to_am(adj_l: AdjLists) -> AdjMatrix
```

qui transforment une matrice d'adjacence en listes d'adjacence et vice versa. Le paramètre `vertices` fournit l'ensemble des sommets, utile à la construction des listes d'adjacence. En l'absence de ce paramètre, les  $n$  sommets sont numérotés de 1 à  $n$ .

## Exercice 2: Vers une bibliothèque de manipulation de graphes

Une fois familiarisés avec les différentes représentations des graphes, il convient de créer une structure de données pour permettre leur instantiation et quelques opérations élémentaires.

QUESTION 1. Dans un module dédié, définir une structure de données **Graph** à même de représenter un graphe simple orienté ou non, pondéré ou non, sous la forme de ses listes d'adjacence. Le type **Graph** ainsi créé retiendra également s'il est orienté ou non, s'il est pondéré ou non, son ordre et sa taille.

QUESTION 2. Dans un second temps, il est nécessaire de se doter d'opérations de création et d'édition du graphe. Proposer les fonctions suivantes :

- ajouter et supprimer un sommet ;
- ajouter/mettre à jour et supprimer un arc/arête pondérée ou non ;
- créer un graphe simple non pondéré, orienté ou non, à partir de la liste de ses arcs/arêtes et éventuellement une liste complémentaire de ses sommets.

Un effet secondaire accepté—souhaité?—de l'ajout d'arc/arête consiste à ajouter les sommets reliés qui ne sont pas déjà dans l'ensemble des sommets du graphe. De façon duale, la suppression d'un sommet entraîne mécaniquement la suppression des arcs/arêtes dont il est une extrémité.

QUESTION 3. Ajouter à la bibliothèque de graphes les fonctions suivantes :

- donner le voisinage d'un sommet ;
- calculer le degré d'un sommet ;
- calculer le degré maximal du graphe ;
- calculer la densité du graphe.

QUESTION 4. Proposer une opération de contraction d'un graphe. La contraction supprime un arc/arête d'un graphe en fusionnant ses deux extrémités. Autrement dit, la contraction  $G/e$  d'un arc  $e = (x, y)$  au sommet  $x$  rend le sommet  $x$  adjacent à tous les voisins précédents de  $y$ . Le sommet  $y$  est ainsi supprimé, au profit du sommet  $x$ . Par convention, s'il existe dans  $G$  deux arcs  $(x, z)$  et  $(y, z)$ , alors le poids de l'arc  $(x, z)$  après contraction  $G/(x, y)$  est le minimum des deux poids originaux.

### Exercice 3: Génération de graphes

Dans cet exercice, nous considérons le type générique `Graph` défini à l'exercice précédent.

QUESTION 1. Proposer une fonction de transformation d'un graphe orienté en un graphe non orienté.

QUESTION 2. Écrire une fonction qui transforme un graphe non orienté en graphe orienté, de façon parfaitement aléatoire : chaque arête  $\{a, b\}$  du graphe est transformée indifféremment en arc  $(a, b)$  ou  $(b, a)$ . Tous les sommets du graphe original sont conservés.

QUESTION 3. Écrire une fonction qui, à partir d'un ensemble de  $n$  sommets et un réel  $p \in [0, 1]$ , renvoie un graphe aléatoire ayant pour toute paire de sommets  $u, v$ , l'arc  $(u, v)$  avec probabilité  $p$ . Quel graphe obtient-on avec  $p = 0$  ?  $p = 1$  ?

## 2 Algorithmique de graphe

### Exercice 4: Parcours de graphe

Un parcours de graphe est une façon de visiter successivement les sommets d'un graphe à partir d'un sommet initial, en progressant de proche en proche dans le graphe. C'est un outil fréquemment employé pour étudier des propriétés globales telles que la connexité, le plus court chemin, l'arbre couvrant, *etc.*

En préambule, on appelle *bordure*  $\Gamma(S)$  d'une partie  $S \subseteq V$  dans un graphe  $G = (V, E)$ , le sous-ensemble des sommets de  $V \setminus S$  qui sont les extrémités d'un arc/arête dont l'origine est dans  $S$ .

Un parcours de  $G$  depuis un sommet  $v$  est alors une séquence  $\ell$  de sommets de  $G$  telle que :

- chaque sommet de  $G$  apparaît une fois et une seule dans  $\ell$  ;
- chaque sommet de  $\ell$ —sauf le premier—appartient à la bordure du sous-ensemble des sommets placés avant lui dans  $\ell$ .

Il existe principalement deux paradigmes pour parcourir un graphe : en largeur et en profondeur. Le parcours en largeur consiste, à partir d'un sommet, à visiter tous les voisins, puis à reprendre le parcours à partir de chaque voisin pour visiter la « deuxième couronne », et ainsi de suite. Le parcours en profondeur opère différemment, en suivant intégralement un chemin jusqu'à un cul-de-sac ou une boucle, puis en empruntant successivement les chemins de plus grand préfixe commun jusqu'à épuiser tous les sommets accessibles du graphe.

Par exemple, à partir du graphe représenté sur la Figure 3 et de son sommet 0, le parcours en largeur donne la séquence (0, 1, 2, 3, 4, 5, 6, 7), tandis que le parcours en profondeur produit la séquence (0, 1, 5, 7, 2, 3, 4, 6).

Dans les questions suivantes, vous utiliserez de nouveau la structure `Graph` précédemment définie.

QUESTION 1. Écrire la fonction

```
def bfs_explore(g: Graph, init: str) -> Iterator[str]
```

qui parcourt le graphe en largeur (*breadth-first search*) à partir d'un sommet initial `init`. Seuls les sommets accessibles à partir de `init` sont visités. À chaque appel, la fonction retourne le sommet suivant dans l'ordre *bfs* : c'est un *générateur*<sup>4</sup>.

**Remarque:** *L'implémentation récursive de l'algorithme est très intuitive. L'implémentation itérative, quant à elle, exploite une file contenant les sommets à traiter.*

QUESTION 2. En suivant une démarche analogue, programmer et appliquer l'algorithme `dfs_explore` de parcours de graphe en profondeur (`dfs` pour *depth-first search*).

---

4. Voir le Sujet n°4 pour un exemple de générateur.

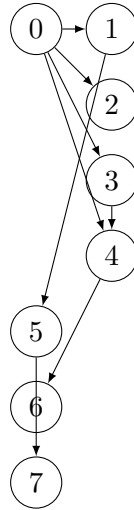


FIGURE 3 – Exemple de graphe à parcourir.

**Remarque:** *L'implémentation itérative de l'algorithme exploite une pile contenant les sommets à traiter.*

### Exercice 5: Connexité

Un graphe non orienté  $G = (V, E)$  est dit *connexe* si toute paire de sommets  $x$  et  $y$  dans  $V$  est reliée par une chaîne. Un sous-graphe induit connexe maximal d'un graphe orienté est appelé une *composante connexe*. Une composante connexe  $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$  d'un graphe  $G = (V, E)$  est donc entièrement décrite par le sous-ensemble  $V_{\mathcal{C}} \subseteq V$  de ses sommets ; ses arêtes  $E_{\mathcal{C}} \subseteq E$  étant toutes les arêtes de  $G$  dont les extrémités sont dans  $V_{\mathcal{C}}$ .

Par exemple, le graphe présenté sur la Figure 4 se compose de 2 composantes connexes :  $\{0, 1, 2, 3, 4, 6\}$  et  $\{5, 7, 8\}$ , qui déterminent de façon univoque les 2 sous-graphes induits visuellement séparés sur la Figure 4.

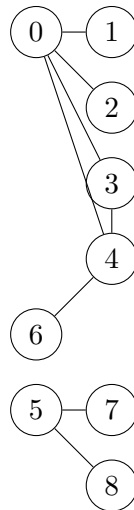


FIGURE 4 – Graphe composé de deux composantes connexes.

**QUESTION 1.** Écrire une fonction qui, étant donné un graphe non orienté, partitionne l'ensemble des sommets suivant ses composantes connexes. Chaque composante connexe sera représentée par « une couleur »  $i \in \mathbb{N}$ .

Par analogie, un graphe orienté  $G = (V, E)$  est *fortement connexe* si, pour toute paire  $x$  et  $y$  de sommets de  $V$ , il existe un chemin  $x \rightarrow_G^* y$  et un chemin  $y \rightarrow_G^* x$ . Une *composante fortement connexe* est alors un sous-graphe induit maximal fortement connexe.

QUESTION 2. Écrire une fonction qui colorie les composantes fortement connexes d'un graphe orienté.

**Remarque:** *Il existe principalement 3 méthodes astucieuses pour résoudre ce problème : l'algorithme de KOSARAJU, celui de TARJAN et un algorithme à base de chemins.*

### 3 Pour aller plus loin...

#### Exercice 6: Arbre couvrant de poids minimum

L'algorithme de KRUSKAL est un algorithme de recherche d'arbre couvrant de poids minimum dans un graphe simple connexe non orienté et valué.

QUESTION 1. Programmer et appliquer la procédure de KRUSKAL pour déterminer un arbre couvrant de poids minimum à partir d'un graphe et étant donnée une racine, ie. le sommet initial.

#### Exercice 7: Plus court chemin

QUESTION 1. En quoi consiste l'algorithme de DIJKSTRA ?

QUESTION 2. Rappeler les conditions d'application de l'algorithme.

QUESTION 3. Programmer et appliquer l'algorithme de DIJKSTRA pour déterminer la distance—longueur du plus court chemin—d'un sommet  $v$  donné à chacun des sommets d'un graphe.