

---

# Structure de données et complexité

## GUIDE DE L'ÉTUDIANTE ET DE L'ÉTUDIANT S2 Génie Informatique – APP5

*Hiver 2024 – Semaines 9 et 10*

---

## Historique des modifications

Date	Responsables	Description
Février 2018	Frédéric Mailhot	Version 1.0
Janvier 2019	Frédéric Mailhot	Version 1.1
Janvier 2020	Frédéric Mailhot	Version 1.2
Janvier 2021	Frédéric Mailhot	Version 1.3
Janvier 2022	Frédéric Mailhot	Version 1.4
Janvier 2023	Frédéric Mailhot	Version 1.5
Février 2023	Frédéric Mailhot	Version 1.6
Avril 2023	Frédéric Mailhot	Version 1.7
Novembre 2023	Frédéric Mailhot	Version 2.0

Auteur : Frédéric Mailhot et Aref Meddeb  
Version : 2.1 (Février 2024)

Ce document est réalisé avec l'aide de  $\text{\LaTeX}$  et de la classe `gegi-app-guide`.

©2024 Tous droits réservés. Département de génie électrique et de génie informatique, Université de Sherbrooke.

# TABLE DES MATIÈRES

1	ACTIVITÉS PÉDAGOGIQUES ET COMPÉTENCES	1
2	SYNTHÈSE DE L'ÉVALUATION	2
3	QUALITÉS DE L'INGÉNIEUR	3
4	ÉNONCÉ DE LA PROBLÉMATIQUE	4
5	CONNAISSANCES NOUVELLES	6
6	GUIDE DE LECTURE	8
7	LOGICIELS ET MATÉRIEL	10
8	SANTÉ ET SÉCURITÉ	11
9	SOMMAIRE DES ACTIVITÉS	12
10	PRODUCTIONS À REMETTRE	13
11	ÉVALUATIONS	18
12	POLITIQUES ET RÈGLEMENTS	19
13	INTÉGRITÉ, PLAGIAT ET AUTRES DÉLITS	20
14	PRATIQUE PROCÉDURALE 1	21
15	PRATIQUE EN LABORATOIRE	30
16	PRATIQUE PROCÉDURALE 2	59
17	VALIDATION AU LABORATOIRE	65

## LISTE DES FIGURES

14.1 Arbres AVL et rotation à droite . . . . .	23
14.2 Arbres AVL et rotation à gauche . . . . .	23
16.1 Graphe à analyser . . . . .	60
16.2 Fonction de transition de la machine de Turing à analyser . . . . .	63

## LISTE DES TABLEAUX

2.1	Synthèse de l'évaluation de l'unité . . . . .	2
2.2	Calcul d'une cote et d'un niveau d'atteinte d'une qualité . . . . .	2

# 1 ACTIVITÉS PÉDAGOGIQUES ET COMPÉTENCES

1. Sélectionner et utiliser les structures de données appropriées pour solutionner un problème donné.
2. Analyser la complexité des algorithmes applicables à un problème donné.

Description officielle : <https://www.usherbrooke.ca/admission/fiches-cours/GIF270>

## 2 SYNTHÈSE DE L'ÉVALUATION

Évaluation	GIF270-1	GIF270-2	Qualités	
Rapport d'APP	15	15	Q1	Q02
Validation	30	30	Q1	Q05
Évaluation sommative	120	120	Q1	Q02
Évaluation finale	135	135	Q1	Q02
Total	300	300		

TABLEAU 2.1 Synthèse de l'évaluation de l'unité

La grille de correspondances des notes avec les cotes et le niveau d'atteinte global d'une qualité est donnée plus bas.

TABLEAU 2.2 Calcul d'une cote et d'un niveau d'atteinte d'une qualité

Note(%)	<50	50	53	57	60	64	68	71	75	78	81	85
Cote	E	D	D+	C-	C	C+	B-	B	B+	A-	A	A+
Niveau	N0	N1	N1	N1	N2	N2	N2	N3	N3	N3	N4	N4
Libellé	Insuffisant	Passable (seuil)			Bien			Très bien (cible)			Excellent	

### 3 QUALITÉS DE L'INGÉNIEUR

Les qualités de l'ingénieur visées et évaluées par cette unité d'APP sont données dans le tableau un peu plus bas. D'autres qualités peuvent être présentes sans être visées ou évaluées dans cette unité. Pour une description détaillée des qualités et leur provenance, consultez le lien suivant : [qualités et BCAPG](#)

Qualité	Libellé	Touchée	Évaluée
Q01	Connaissances en génie	✓	✓
Q02	Analyse de problèmes	✓	✓
Q03	Investigation		
Q04	Conception		
Q05	Utilisation d'outils d'ingénierie	✓	✓
Q06	Travail individuel et en équipe		
Q07	Communication		
Q08	Professionnalisme		
Q09	Impact du génie sur la société et l'environnement		
Q10	Déontologie et équité		
Q11	Économie et gestion de projets		
Q12	Apprentissage continu		



## 4 ÉNONCÉ DE LA PROBLÉMATIQUE

### Comment reconnaître quelqu'un par ses écrits ?

Chaque jour ou presque, nous utilisons des sites et applications en ligne pour nous renseigner, nous divertir, contacter des amis, etc. Ces différentes activités, nous le savons, produisent des signatures numériques permettant à des tiers de connaître nos intérêts, nos besoins et, en partie, de nous reconnaître. Cette situation, qu'on peut apprécier ou non, peut paraître relativement nouvelle dans l'histoire de l'humanité. En fait, c'est une extension d'un problème beaucoup plus ancien : depuis l'invention de l'écriture, la question de la reconnaissance de l'auteur d'un texte s'est posée à maintes reprises. Dans cette unité d'APP, nous explorerons des structures de données et des algorithmes permettant de calculer la fréquence des mots dans un ensemble de textes, pour ensuite évaluer la possibilité qu'une certaine personne ait écrit un certain texte <sup>1</sup>. Il est à noter que l'analyse des fréquences de mots dans un texte repose sur des concepts similaires à ceux utilisés par les entreprises web pour connaître nos intérêts et cibler la publicité qui nous est présentée. C'est en quelque sorte une version rudimentaire et simplifiée de *ChatGPT* ou autre système analogue. Ici, pour analyser les textes, nous utiliserons des unigrammes, des bigrammes et, en général des n-grammes de mots, un n-gramme représentant une séquence de "n" mots. Comme les textes analysés comprendront de grandes quantités de mots (il s'agit de textes d'auteurs francophones célèbres), l'efficacité des algorithmes sera importante et il faudra en évaluer la complexité.

Pour calculer la fréquence de mots dans un texte, il pourrait s'avérer utile d'utiliser des listes chaînées (simplement ou doublement), des arbres, des tableaux de hachage ou des graphes. Une combinaison de ces structures de données est aussi possible. Chacun des mots (ou séquences de mots) d'un texte étudié sera lu et comparé à l'ensemble des mots (ou séquences de mots) lus auparavant. Il faudra alors faire le compte des mots (ou séquences de mots) identiques qui apparaissent dans les textes analysés.

Chacune des structures de données étudiées implique des considérations spécifiques. Par exemple :

- l'utilisation de listes exige la comparaison et l'insertion de nouveaux éléments ;
- l'utilisation d'arbres implique l'insertion et le rebalancement ;
- l'utilisation des tableaux de hachage vient avec la définition de la fonction de hachage, de la fonction de comparaison et de la taille du tableau (entre  $2^8$  et  $2^{16}$  serait proba-

---

1. Par exemple : Voir les [Federalist Papers](#) et la [découverte de leurs auteurs à l'aide de techniques informatiques statistiques](#).

blement d'intérêt ici, en utilisant la fonction modulo pour assurer une valeur d'index acceptable), ainsi que la gestion des collisions ;

- l'utilisation de graphes requiert la détection et la gestion des cycles lors d'ajout de noeuds .

Pour faire l'analyse de mots dans des textes, il est important de considérer la complexité associée à la méthode utilisée pour découvrir si un élément a déjà été observé et pour faire le décompte des mots observés. Selon la complexité de la méthode choisie, le système exigera au maximum un temps de recherche constant, logarithmique, linéaire, quadratique ou même exponentiel par rapport au nombre de mots à traiter. On parle alors de complexité  $\mathcal{O}$  (on prononce big-O) d'un algorithme, dont on dira, par exemple, qu'il est d'ordre  $\mathcal{O}(1)$ ,  $\mathcal{O}(\log n)$ ,  $\mathcal{O}(n)$ ,  $\mathcal{O}(n \log n)$ ,  $\mathcal{O}(n^2)$  ou  $\mathcal{O}(e^n)$ .

Il existe aussi les analyses de complexité  $\Omega$  et  $\Theta$ <sup>2</sup>. D'un point de vue formel, on effectue les analyses de complexité en supposant l'utilisation d'un système de calcul théorique nommé *machine de Turing*. Ceci permet d'évaluer la complexité d'un algorithme de façon indépendante du processeur spécifique utilisé pour réaliser les calculs. Le problème de l'arrêt (halting problem en anglais) a été démontré par Turing en utilisant cette machine.

À partir de la fréquence de mots calculée pour un ensemble de textes, il est possible de générer un texte "aléatoire", selon un processus qu'on appelle une chaîne de Markov, en utilisant un générateur de nombres pseudo-aléatoires. Pour ce faire, il faut trier les mots observés, du plus au moins fréquent. Plusieurs méthodes de tri existent, tels le tri fusion (*merge sort* en anglais), le tri à bulle (*bubble sort* en anglais) et le tri rapide (*quicksort* en anglais), chacune des méthodes ayant une complexité distincte. Un texte aléatoire généré à partir des fréquences observées pour un certain auteur permet de valider le fonctionnement de l'ensemble du système.

---

2.  $\mathcal{O}()$ ,  $\Omega()$  et  $\Theta()$  font partie de ce qu'on appelle la notation de Landau

## 5 CONNAISSANCES NOUVELLES

### Connaissances déclaratives (quoi)

- Structures de données
  - Listes chaînées, listes doublement chaînées
  - Graphes, cycles
  - Arbres
    - Profondeur
    - Binaires
    - AVL
      - Facteur d'équilibrage
      - Équilibrage et rotations
  - Fonctions et tableaux de hachage
- Algorithmes
  - Tri
    - à bulle
    - fusion
    - rapide (quicksort)
  - Recherche binaire
  - Largeur d'abord, profondeur d'abord
- Machine de Turing
- Complexité
  - logarithmique
  - linéaire
  - polynomiale
  - exponentielle
  - P, NP
  - Analyse  $\mathcal{O}$ ,  $\Omega$ ,  $\Theta$
- Language Python

### Connaissances procédurales (comment)

- Mettre en oeuvre les structures de données appropriées, en définissant tous les éléments et paramètres nécessaires
- Concevoir les algorithmes associés à certains types de structures de données, tels la fonction de hachage, le tri, l'insertion, l'extraction

- Créer un arbre, insérer, retrancher des éléments ; rebalancer
- Établir la complexité d'un algorithme
- Effectuer un tri efficace
- Utiliser le langage Python pour produire un système d'analyse de texte

### **Connaissances conditionnelles (quand)**

- Choisir et combiner les structures de données appropriées pour un problème spécifique
- Choisir et combiner les algorithmes appropriés selon leur complexité pour un problème spécifique
- Déterminer quelles structures et bibliothèques en Python sont nécessaires pour réaliser un logiciel de complexité moyenne

## 6 GUIDE DE LECTURE

hyperref

### 6.1 Lectures en lien avec les structures de données, la complexité et les algorithmes

- **Référence principale** - [Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures Using Python](#) :
  - Complexité : Sections 2.1 à 2.7
  - Structures de données de base : Sections 3.1, 3.2
    - Listes : Sections 3.19 à 3.23
  - Hachage : Section 5.5
  - Tri : Sections 5.6 à 5.12
  - Arbres : Sections 6.1 à 6.6, 6.11 à 6.17
  - Graphes : Sections 7.7 à 7.9, 7.19 à 7.21
- **Tableaux de hachage** -
  - [Comprendre le hachage en Python](#)
- **Complexité** - [Explications au sujet de la notation de Landau sur wikipedia](#) :
  - Comparaison des valeurs usuelles : Voir : "2.2 Échelle de comparaison"
  - Notation de Landau : Voir "3. La famille de notations de Landau  $\mathcal{O}$ ,  $o$ ,  $\Omega$ ,  $\omega$ ,  $\Theta$ "
- **Complexité** - [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein, Introduction to Algorithms](#) :
  - Chapitre 3 (Introduction à la complexité - 'Growth of Functions')
- **Machine de Turing** - [Explications en français](#) :
  - [La machine de Turing](#)
  - [Autre description de la machine de Turing \(lecture facultative\)](#)
  - [Explication du problème de l'arrêt - Voir la preuve "classique"](#)

### 6.2 Lectures en lien avec la problématique

- **Analyse et génération de texte (chaînes de Markov)** - [Brian Kernighan et Rob Pike, The Practice of Programming](#) :
  - Chapitre 3 (Calcul de fréquence de mots)

### 6.3 Lectures en lien avec le langage Python (au besoin)

- **Python** - [Gérard Swinnen, Apprendre à Programmer en Python 3](#) :
  - Introduction à Python : chapitres 1 à 7

## 6.4 Lectures facultatives

- Tableaux de hachage - [Type et fonctionnement](#)
- Tableaux de hachage - [Utilisation de \*dict\* en Python](#)
- Visualisations :
  - **IDEA**
    - [Explication arbre binaire](#)
    - [Explication arbre AVL](#)
    - [Explication tri rapide \(quicksort\)](#) [Explication tri fusion \(merge sort\)](#)
    - [Explication recherche binaire](#)
  - **Animations**
    - [15 algorithmes de tri en 6 minutes \(amusant\)](#)
    - [Arbre AVL](#)
    - [Arbre binaire](#)
    - [Méthodes de tri](#)
    - [Graphes](#)
    - [Tableau de hachage : adressage ouvert](#)
    - [Tableau de hachage : gestion par chaînage](#)
- **Pour en savoir plus (références à se procurer:)**
  - **Algorithmes**
    - [Art of Computer Programming \(Donald Knuth\)](#) - LA référence
    - [Introduction to Algorithms \(Thomas Cormen et al.\)](#) - Une autre excellente référence sur les algorithmes
    - [Algorithms \(Sanjoy Dasgupta et al.\)](#) Excellente présentation, concise mais accessible

## 7 LOGICIELS ET MATÉRIEL

- L’environnement de développement intégré (IDE) PyCharm sera utilisé pendant cette unité d’APP. Il est recommandé de l’installer sur votre ordinateur avant la période de laboratoire.
- Les fichiers Python nécessaires pour le laboratoire et la problématique seront disponibles sur le site de l’unité d’APP.
- Pour obtenir une license gratuite d’utilisation des outils JetBrains, ainsi que la version la plus récente de l’IDE PyCharm, consultez les sites suivants :
  - <https://www.jetbrains.com/community/education/#students>
  - <https://www.jetbrains.com/pycharm/>
- Il est aussi recommandé d’installer la version la plus récente de Python. Pour ce faire, vous pouvez consulter le site suivant :
  - <https://www.python.org/downloads/>

## 8 SANTÉ ET SÉCURITÉ

### 8.1 Dispositions générales

Dans le cadre de la présente activité, vous êtes réputés avoir pris connaissance des politiques et directives concernant la santé et la sécurité. Ces documents sont disponibles sur les sites web de l'Université de Sherbrooke, de la Faculté de génie et du département. Les principaux sont mentionnés ici et sont disponibles dans la section *Santé et sécurité* du [site web du département](#).

- Politique 2500-004 : Politique de santé et sécurité en milieu de travail et d'études
- Directive 2600-042 : Directive relative à la santé et à la sécurité en milieu de travail et d'études
- Sécurité en laboratoire et atelier au département de génie électrique et de génie informatique

### 8.2 Dispositions particulières

Aucune.



## 9 SOMMAIRE DES ACTIVITÉS

### Semaine 1

- Première rencontre de tutorat
- Étude personnelle et exercices : étude des sujets issus des objectifs d'étude du tutorat
- Formation à la pratique procédurale 1 : *problèmes touchant les structures de données et la complexité des algorithmes*
- Formation à la pratique en laboratoire : *débogage, mise en oeuvre de quelques structures de données de base et évaluation leur impact sur la performance de certains algorithmes*
- Formation à la pratique procédurale 2 : *problèmes touchant les machines de Turing, les algorithmes de tri et leur complexité*

### Semaine 2

- Consultations facultatives
- Étude personnelle et exercices
- Validation théorique de la solution
- Rédaction du rapport d'APP
- Remise du rapport d'APP et du code Python
- Deuxième rencontre de tutorat : validation des connaissances acquises
- Évaluation formative théorique
- Évaluation sommative théorique

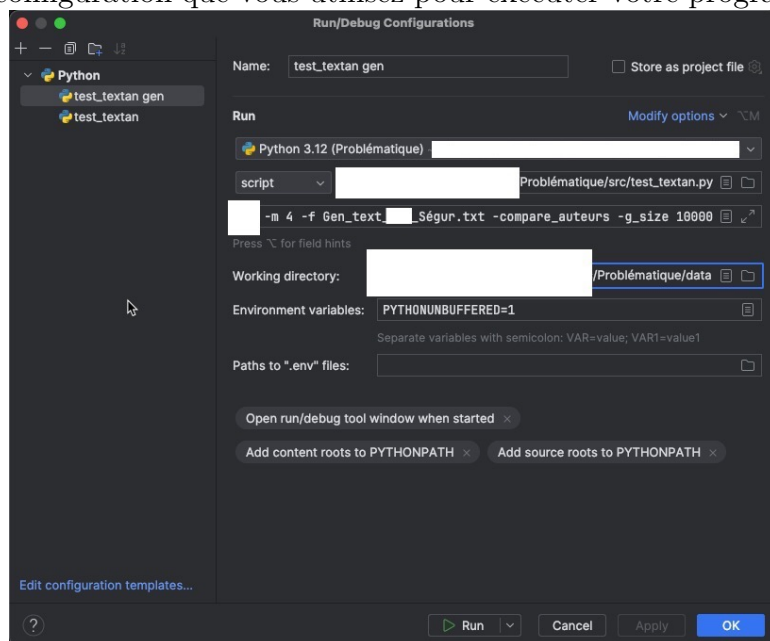
## 10 PRODUCTIONS À REMETTRE

*Application Python à remettre avant 9h00 le matin du tutorat 2*

Il n'y a pas de rapport à remettre dans cette unité d'APP, seul votre fichier python (**textan\_CIP1\_CIP2.py**) doit être remis. Il est préférable d'inclure ce fichier Python dans un fichier compressé zip, certains sites de dépôt refusant d'accepter un fichier ayant une extension **.py**.

L'application Python, nommée **textan\_CIP1\_CIP2.py** doit implémenter les fonctionnalités suivantes :

- Travailler avec des fichiers de texte d'un ensemble d'auteurs. Les calculs doivent être réalisés de façon indépendante pour chacun des auteurs. Les oeuvres de chacun des auteurs sont incluses dans un répertoire (au nom de l'auteur), tous ces répertoires (dont le nom correspond à l'auteur) se trouvant dans un répertoire dont le nom est passé en paramètre. Le tout se trouve dans le sous-répertoire **.../Problématique/data** du fichier **code\_etudiant.zip** que vous avez reçu. Dans l'IDE PyCharm, vous devriez indiquer **.../Problématique/data** comme répertoire dans le champs *Working Directory* de la configuration que vous utilisez pour exécuter votre programme Python :



- Pour le calcul du k-ième n-gramme le plus fréquent, ou pour la production d'un texte aléatoire, utiliser le nom de l'auteur passé en paramètre. Le nom de cet auteur doit correspondre à l'un des auteurs contenus dans le répertoire d'auteurs. Tous les fichiers de texte se trouvant dans le répertoire correspondant à l'auteur choisi doivent être traités : accumuler le nombre d'occurrences des n-grammes de l'auteur indépendem-

ment des différentes oeuvres (ne calculez pas les statistiques par oeuvre, seulement pour l'ensemble des oeuvres, comme s'il s'agissait d'une oeuvre unique pour un auteur donné).

- Faire en sorte qu'il est possible de calculer les fréquences des mots uniques (unigrammes), des séquences de deux mots (bigrammes) ainsi que de séquences de mots de longueur  $n$  arbitraire ( $n$ -grammes).
- Calculer le  $k^e$   $n$ -gramme le plus fréquent dans les fichiers de texte de l'auteur choisi (selon le paramètres de  $n$ -gramme choisi sur la ligne de commande).
- Pour calculer la fréquence des  $n$ -grammes, regrouper les  $n$ -grammes qui ont la même fréquence et retourner la liste de tous ces  $n$ -grammes. Par exemple, si les unigrammes "une", "des" et "ils" apparaissent chacun à 132 reprises, associer la liste de ces trois unigrammes au nombre 132. Si par exemple cette fréquence (132) est la troisième la plus élevée et qu'on demande de retourner le troisième unigramme le plus fréquent, on doit retourner la liste ["une", "des", "ils"].
- Déterminer la proximité d'un texte inconnu avec les statistiques de l'ensemble des auteurs traités et indiquer l'auteur le plus probable.
- [Pour plus de détails, consulter la documentation ReadTheDocs du code fourni pour la problématique : fichier \*index.html\* situé dans le répertoire \*code\\_etudiant/Problématique/build/html\*](#)

Votre code sera appelé par le fichier de test **test\_textan.py**. Votre code, compris dans le fichier **textan\_CIP1\_CIP2.py**, doit offrir les méthodes suivantes dans la classe **TextAn** :

- `analyze` (pour analyser l'ensemble des textes des auteurs fournis)
- `find_author` (pour trouver l'auteur probable d'un texte inconnu, en utilisant les statistiques de l'ensemble des oeuvres de l'ensemble des auteurs traités)
- `gen_text` (pour générer un texte ayant les mêmes statistiques que celles d'un certain auteur, selon l'ensemble de son oeuvre). Valider que votre logiciel peut générer un texte d'au moins **10 000 mots**. **Note : si le temps d'exécution de la génération de 10 000 mots dépasse 2 minutes, votre code a fort probablement une complexité  $\mathcal{O}(n^2)$ . Si c'est le cas, révisez votre façon de faire, car il existe une façon de générer des mots en  $\mathcal{O}(n)$ .**
- `gen_text_all` (pour générer un texte ayant les mêmes statistiques que l'ensemble des auteurs contenus dans le répertoire d'auteurs, comme s'il s'agissait d'un auteur unique). Ici aussi, valider que votre logiciel peut générer un texte d'au moins **10 000 mots**.

- `get_kth_element` (pour retourner le k-ième n-gramme le plus fréquent pour un auteur donné, pour l'ensemble de son oeuvre)
- `dot_product_dict` (pour effectuer le produit scalaire entre les vecteurs correspondant à deux dictionnaires Python)
- `dot_product_aut` (pour effectuer le produit scalaire entre les vecteurs correspondant à deux auteurs)
- `doc_product_dict_aut` (pour effectuer le produit scalaire entre les vecteurs correspondant à un dictionnaire Python et un auteur)

L'application de test `test_textan` accepte les commandes suivantes :

- `-d repertoire` (pour indiquer le répertoire dans lequel se trouvent les sous-répertoires de chacun des auteurs à traiter)
- `-a auteur` (utilisé pour la génération de texte aléatoire ou pour l'extraction du n-ième n-gramme le plus fréquent de cet auteur)
- `-f fichier` (pour indiquer un fichier de texte à comparer avec les fréquences des fichiers de l'ensemble des auteurs)
- `-T` (pour indiquer que les résultats de l'analyse d'un texte inconnu doivent être produits pour l'ensemble des auteurs)
- `-m 1` (faire le calcul avec des unigrammes de mots)
- `-m 2` (faire le calcul avec des bigrammes de mots)
- `-F nombre (n)` (afficher le  $n^e$  élément le plus fréquent (selon l'option `-m 1`, `-m 2`))
- `-G nombre (n)` (produire un texte aléatoire de  $n$  mots, selon le style de l'auteur identifié par l'option `-a auteur`)
- `-g nom du fichier à générer` (pour indiquer le nom du fichier produit, si l'option `-G` est utilisée)
- `-v` (mode verbose)
- `-noPonc` (pour indiquer de retirer toute la ponctuation)

Pour que l'application `test_textan` fonctionne adéquatement, il faut modifier le fichier **etudiants.txt** et y mettre une ligne avec vos CIPs **CIP1\_CIP2**, tels qu'ils apparaissent dans le fichier **textan\_CIP1\_CIP2.py**.

Les commandes suivantes sont des exemples de quelques possibilités :

- `python test_textan.py -d TextesPourEtudiants -f monfichier -m 1` :  
(calcule la fréquence des unigrammes de mots pour toutes les oeuvres de chacun des auteurs dans le répertoire *TextesPourEtudiants*, imprime l'auteur le plus probable du fichier *monfichier*)
- `python test_textan.py -d TextesPourEtudiants -a zola -m 2 -F 3`

(calcule la fréquence des digrammes de mots dans l'ensemble des oeuvres des auteurs compris dans le répertoire *TextesPourEtudiants*) et imprime la 3<sup>e</sup> séquence de 2 mots la plus fréquente dans les textes situés dans le sous-répertoire *zola*)

— `python test_textan.py -d TextesPourEtudiants -f monfichier -m 2`

(calcule la fréquence des digrammes de mots dans les fichiers de tous les auteurs présents dans des sous-répertoire du répertoire *TextesPourEtudiants* et imprime la liste des auteurs et l'évaluation de la proximité du texte *monfichier* pour chacun des auteurs) Dans le cas, l'application produira des résultats de la forme :

balzac 0,0042

ségur 0,0012

zola 0,0033

(en supposant que le répertoire *TextesPourEtudiants* comprend les répertoires balzac, ségur et zola, et que le texte inconnu est à une distance de 0,0042 des textes de balzac, etc)

Note : En Python, [argparse](#) est utilisé dans `test_textan.py` pour lire les arguments donnés en ligne de commande. Notez que Python offre directement des [fonctionnalités pour la génération de nombres pseudo-aléatoire](#).

Quelques recommandations pour assurer un fonctionnement uniforme pour votre application :

- Avant de traiter un mot, assurez-vous que toutes ses lettres sont transformées en minuscules. La méthode "lower" de Python pourrait être utile.
- Conservez (en minuscules) les lettres accentuées ainsi que tous les caractères spécifiques au français (é, è, ê, ë, î, ï, à, â, ù, û, ü, ô, ç, etc.).
- Considérez les chiffres 0-9 comme des lettres minuscules.
- Traitez les caractères qui ne sont pas des lettres ou des chiffres (par exemple : "-", ":", ";", ".", etc .) comme des mots à une lettre.
- Vous pouvez traiter les mots de 2 caractères ou moins comme des espaces, mais vous pouvez aussi les conserver. Voir les options définies dans *argparse*
- Pour simplifier le traitement, considérez que le trait d'union est un mot. Ainsi les mots composés (par exemple, porte-voix) seront considérés comme des séquences de trois mots (pour l'exemple précédent, "porte", "-" et "voix").
- Considérez que les espaces, les tabulations et les changements de ligne sont des séparateurs entre les mots (et ne sont donc pas des mots).

- Considérez que toutes les formes conjuguées d'un verbe ainsi que le pluriel d'un mot représentent des mots distincts (pas de racinisation, ou *stemming* en anglais).
  - Pour comparer l'ensemble des textes d'un auteur  $a$  avec un texte inconnu, faire le calcul suivant lorsque les fréquences (unigrammes ou digrammes) sont établies pour l'auteur  $a$  et pour le texte inconnu :
    - Obtenir le vecteur  $M = [|m_i|]$  pour l'ensemble des mots de l'auteur  $a$  et le vecteur  $T = [|t_i|]$  pour l'ensemble des mots du texte inconnu, où  $|m_i|$  et  $|t_i|$  représentent le nombre de fois que le mot  $m_i, t_i$  (on suppose que  $m_i$  et  $t_i$  représentent le même mot).
    - Calculer la taille des vecteur  $M$  et  $T$  :  $|M| = \sqrt{(\sum_{i=0}^N (|m_i|)^2)}$  et  $|T| = \sqrt{(\sum_{i=0}^N (|t_i|)^2)}$  (on suppose que le nombre total de mots dans les textes est  $N$ )
    - Faire le produit scalaire de  $M$  et de  $T$  :  $(M \cdot T) = (\sum_{i=0}^N (|m_i| \cdot |t_i|))$
    - Normaliser le résultat obtenu en le divisant par le produit des tailles des vecteurs :  
proximité =  $(M \cdot T) / (|M| \cdot |T|)$
    - Plus cette valeur est proche de 1, plus importante est la proximité du texte inconnu avec les écrits de l'auteur sous analyse
  - La méthode `readline()` de Python pourrait vous être utile pour obtenir les lignes d'un fichier de texte.
  - La méthode `split()` de Python pourrait vous être utile pour obtenir les mots d'une ligne.
- $\implies$  Pour la remise : produire et déposer le fichier compressé d'un répertoire nommé *CIP1\_CIP2*, contenant le rapport d'APP (nommé *rapport\_CIP1\_CIP2.pdf*) et le fichier **unique** *textan\_CIP1\_CIP2.py*

## 11 ÉVALUATIONS

Fichier inconnu : evaluations

## 12 POLITIQUES ET RÈGLEMENTS

Dans le cadre de la présente activité, vous êtes réputés avoir pris connaissance des politiques, règlements et normes d'agrément ci-dessous :

### Règlements de l'Université de Sherbrooke

- [Règlement des études](#)

### Règlements facultaires

- [Règlement facultaire d'évaluation des apprentissages / Programmes de baccalauréat](#)
- [Règlement facultaire sur la reconnaissance des acquis](#)

### Normes d'agrément

- [Processus d'agrément et qualités du BCAPG](#)
- [Ingénieurs Canada – À propos de l'agrément](#)

Enfin, si vous êtes en situation de handicap, assurez-vous d'avoir communiqué avec le *Programme d'intégration des étudiantes et étudiants en situation de handicap* à l'adresse : [prog.integration@usherbrooke.ca](mailto:prog.integration@usherbrooke.ca).



## 13 INTÉGRITÉ, PLAGIAT ET AUTRES DÉLITS

Dans le cadre de la présente activité, vous êtes réputés avoir pris connaissance de la page [Intégrité intellectuelle](#) des Services à la vie étudiante.

# 14 PRATIQUE PROCÉDURALE 1

## But de l'activité

Le but de cette activité est de se familiariser avec certains types de structures de données ainsi qu'avec la complexité des algorithmes :

- Listes, arbres, tableaux de hachage
- Complexité des algorithmes (logarithmique, linéaire, quadratique, polynomiale, exponentielle)

NOTE : Il vous est fortement recommandé de lire les documents qui expliquent les structures de données de base et la complexité des algorithmes avant de vous présenter à cette séance.

### P1.E1 Liste chaînée, doublement chaînée

Supposons que nous voulons représenter un ensemble d'éléments ordonnés, par exemple les différents wagons d'un certain train de métro.

- a. Expliquer pourquoi une liste serait une bonne représentation d'un train de wagons.
- b. Supposons que le modèle que nous voulons créer sera utilisé pour représenter le nombre de passagers présents dans chacun des wagons tout au cours d'une journée. Supposons de plus que le train de métro comprend 10 wagons, numérotés 1 à 10, du wagon de tête au wagon de queue :
  1. À un certain moment, 3 passagers montent dans le wagon 5. Comment pourrions-nous accéder à l'élément représentant le wagon 5 ?
  2. À midi, un nouveau wagon (le wagon numéro 42) est ajouté entre les wagons 6 et 7. Quel type de liste nous permettrait de représenter cette situation ? Quelles opérations devraient alors être faites ?
  3. Plus tard, le wagon 7 est retiré. Comment devons-nous modifier la liste ?

### P1.E2 Tableau de hachage

- a. Quels sont les paramètres et éléments importants dans un tableau de hachage ?
- b. Quelle est l'utilité de la fonction de hachage ?
- c. Qu'est-ce qu'une collision ? Est-ce que les collisions sont inévitables ? Expliquer pourquoi.
- d. Quelle est la différence entre la gestion des collisions par chaînage et par adressage ouvert ? Quels sont les avantages et les inconvénients de chacun ?
- e. Supposons que  $N$  éléments sont ajoutés à un tableau de hachage. Combien d'opérations seront nécessaires pour ajouter tous ces éléments à la table (supposons que

les  $N$  éléments sont différents) ? Par la suite, combien d'opérations seront nécessaires pour vérifier si un élément  $x$  se trouve dans la table ou non ?

### P1.E3 Arbres

- a. Qu'est-ce qu'un arbre ? En quoi diffère-t-il d'une liste ?
- b. Qu'est-ce qu'un arbre binaire ?
- c. Supposons qu'on crée un arbre binaire où chaque noeud représente un nombre, et où lors de l'ajout d'un nouveau nombre, une comparaison est faite au niveau de chaque noeud : si le nouveau nombre est plus petit que celui du noeud, il va à gauche, sinon, il va à droite. Simuler le traitement de la séquence de nombres qui arrivent dans l'ordre suivant :  
Cas 1 : 5, 8, 2, 1, 4, 3, 7, 6, 9, 10  
Cas 2 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- d. Comment faire en sorte que le problème identifié plus haut ne se produise pas (on suppose qu'on ne peut modifier l'ordre d'arrivée des nombres) ?

- e. Calculez le nouveau *balance factor* des noeuds *B* et *D* après une rotation droite dans la figure suivante (figure 14.1). La **note** qui suit la figure 14.1 pourrait s'avérer utile. Inspirez-vous du calcul effectué pour la rotation gauche et adaptez le pour obtenir les résultats pour la rotation droite.

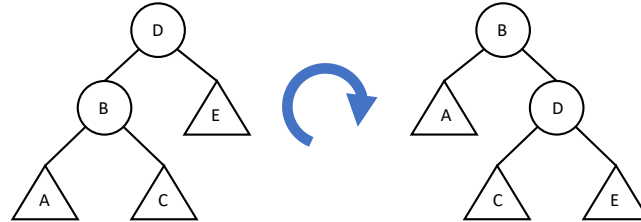


FIGURE 14.1 Arbres AVL et rotation à droite

**Note** : dans le livre (section 6.17), on indique comment calculer le nouveau *balance factor* après une rotation gauche (figure 14.2).

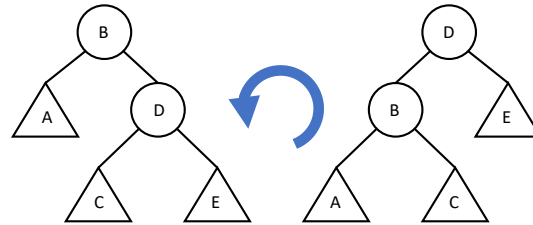


FIGURE 14.2 Arbres AVL et rotation à gauche

Pour le noeud *D* :

$$new\_bal(D) = new\_h_B - h_E \quad (14.1)$$

$$old\_bal(D) = h_C - h_E \quad (14.2)$$

Pour le noeud *B* :

$$new\_bal(B) = h_A - h_C \quad (14.3)$$

$$old\_bal(B) = h_A - old\_h_D \quad (14.4)$$

Cependant :

$$old\_h_D = 1 + \max(h_C, h_E) \quad (14.5)$$

Alors, en remplaçant 14.5 dans 14.4 :

$$old\_bal(B) = h_A - (1 + \max(h_C, h_E)) \quad (14.6)$$

On peut maintenant soustraire l'équation 14.6 de l'équation 14.3. On obtient :

$$\begin{aligned} new\_bal(B) - old\_bal(B) &= h_A - h_C - (h_A - (1 + \max(h_C, h_E))) \\ new\_bal(B) - old\_bal(B) &= 1 + \max(h_C, h_E) - h_C \\ new\_bal(B) - old\_bal(B) &= 1 + \max(h_C - h_C, h_E - h_C) \\ new\_bal(B) &= old\_bal(B) + 1 + \max(0, h_E - h_C) \end{aligned}$$

Mais :

$$h_C - h_E = old\_bal(D)$$

D'où :

$$\begin{aligned} new\_bal(B) &= old\_bal(B) + 1 + \max(0, -old\_bal(D)) \\ new\_bal(B) &= old\_bal(B) + 1 - \min(0, old\_bal(D)) \end{aligned}$$

On peut aussi calculer  $new\_bal(D)$  en soustrayant l'équation 14.2 de l'équation 14.1 :

$$\begin{aligned} new\_bal(D) - old\_bal(D) &= new\_h_B - h_E - (h_C - h_E) \\ new\_bal(D) - old\_bal(D) &= new\_h_B - h_C \\ new\_bal(D) &= old\_bal(D) + new\_h_B - h_C \end{aligned}$$

Mais on connaît  $new\_h_B$  :

$$new\_h_B = 1 + \max(h_A, h_C) \quad (14.7)$$

En remplaçant  $new\_h_B$  de l'équation 14.7 dans l'équation pour  $new\_bal(D)$  on obtient :

$$\begin{aligned} new\_bal(D) &= old\_bal(D) + 1 + \max(h_A, h_C) - h_C \\ new\_bal(D) &= old\_bal(D) + 1 + \max(h_A - h_C, h_C - h_C) \\ new\_bal(D) &= old\_bal(D) + 1 + \max(h_A - h_C, 0) \end{aligned}$$

Mais :

$$h_A - h_C = new\_bal(B)$$

Après rotation gauche, on peut donc obtenir les nouveaux *balance factors* :

$$\begin{aligned} new\_bal(B) &= old\_bal(B) + 1 - \min(0, old\_bal(D)) \\ new\_bal(D) &= old\_bal(D) + 1 + \max(new\_bal(B), 0) \end{aligned}$$

Il est donc possible d'obtenir les nouveaux *balance factors* sans connaître les hauteurs des sous-arbres  $A$ ,  $C$  et  $E$ . Il suffit de connaître les *balance factor* des anciens noeuds  $B$  et  $D$  pour obtenir les *balance factors* des nouveaux noeuds  $B$  et  $D$  (après rotation gauche).

#### P1.E4 Complexité

Dans cette question, nous étudierons certains concepts en lien avec la complexité des algorithmes et nous ferons une brève évaluation de la complexité de quelques méthodes déjà codées.

- Que représentent  $\mathcal{O}$ ,  $\Omega$  et  $\Theta$  ? Donner des exemples avec des courbes.
- Considérez la création d'une liste chaînée, où les éléments doivent être ordonnés du plus petit au plus grand. Faites l'analyse  $\mathcal{O}$  et  $\Omega$  (pour vous aider dans votre ré-

flexion, considérez la création de deux listes (ordonnées) distinctes, l'une avec l'ordre des données suivante :  $(1, 2, 3, \dots, n)$ , l'autre avec l'ordre des données  $(n, n - 1, n - 2, \dots, 3, 2, 1)$

- c. En supposant qu'un algorithme effectue  $5n^2 + 3n + 2$  opérations lorsqu'il traite  $n$  données, quelle est sa complexité  $\mathcal{O}$ ? Expliquer pourquoi. L'utilisation de logarithmes peut être intéressante pour cadrer l'explication.

- d. Compléter le tableau suivant, en indiquant le nombre d'opérations et le temps d'exécution pour les différentes valeurs de  $n$  et les différents niveaux de complexité :

		Complexité					
<i>Pour nombre d'éléments (<math>n</math>)</i>		$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(10^n)$
<i>Quantité de calculs (<math>N</math>) ou Temps d'exécution (<math>T</math>) (<math>10^{-9}</math> sec par op)</i>							
N :	$n = 10^3$						
N :	$n = 10^6$						
N :	$n = 10^9$						
T :	$n = 10^3$						
T :	$n = 10^6$						
T :	$n = 10^9$						

Note : 1 heure représente environ  $10^5$  secondes, 1 an représente environ  $10^8$  secondes (en réalité,  $3 \times 10^7$  secondes), mille ans représentent environ  $10^{11}$  secondes, et l'âge de l'univers est environ de  $10^{18}$  secondes.



e. Déterminer la complexité  $\mathcal{O}$  des extraits de code suivants :

Extrait de code 1 :

```
1      for i in range(0,N):  
2          z += 1  
3      for j in range(0,N-3):  
4          z += 1
```

Extrait de code 2 :

```
1      for i in range(0,N):  
2          z += 1  
3      for j in range(0,N*N):  
4          z += 1
```

Extrait de code 3 :

```
1      for i in range(0,N):  
2          for j in range(0,i):  
3              z += 1
```

Extrait de code 4 :

```
1      for i in range(0,1000):  
2          z += 1
```

Extrait de code 5 :

```
1      def une_fonction(N):  
2          if (N == 0):  
3              return 1  
4              return 1 + une_fonction(N >> 1)
```

Extrait de code 6 :

```
1      def une_autre_fonction(N,t):  
2          if (N == 0):  
3              print t  
4              return  
5              une_autre_fonction(N >> 1, 2 * t)  
6              une_autre_fonction(N >> 1, 1 + 2 * t)  
7              return
```

Extrait de code 7 :

```
1      def une_autre_fonction(N,t):  
2          if (N == 0):  
3              print t  
4              return  
5              une_autre_fonction(N - 1, 2 * t)  
6              une_autre_fonction(N - 1, 1 + 2 * t)  
7              return
```

## 15 PRATIQUE EN LABORATOIRE

Dans cette activité, on collabore par équipe de deux.

⇒ **Note importante :**

- Vous devez préparer le laboratoire **AVANT** le début de l'activité.
- Voir à la page suivante pour une description de ce que vous devez faire pour cette préparation.
- La préparation devrait prendre environ 1 heure.

### But de l'activité

Le but de ce laboratoire est de maîtriser l'utilisation du langage Python, de mettre en œuvre quelques structures de données de base et de comprendre comment les utiliser.

### Description du laboratoire :

1. Nous commencerons ce laboratoire par une activité sur le débogage. Le code Python qui vous est fourni dans ce premier exercice est truffé de bogues que vous devrez identifier et éliminer. L'objectif de cette activité est de vous familiariser avec des techniques de débogage plus efficaces que l'utilisation de *prints*.
2. Après l'activité sur le débogage, l'exercice suivant vous permettra de vous familiariser avec l'utilisation plus contrôlée de tableaux de hachage. En particulier, vous verrez comment redéfinir la fonction de hachage ainsi que la fonction de comparaison des clés. Vous verrez aussi comment faire en sorte qu'il ne soit pas nécessaire de redéfinir ces fonctions dans des situations où il peut sembler que ce soit la seule solution. . .
3. Le troisième exercice implique un arbre AVL et le code Python sous-jacent. Vous aurez à compléter le code d'un arbre AVL en y ajoutant les calculs liés à une rotation droite.
4. Enfin, la quatrième activité est optionnelle et vous permet de pousser plus loin votre capacité à programmer en Python et votre compréhension des structures de données. Cet exercice touche la création d'échelles de mots, telles que proposées par Lewis Carroll, et implique l'utilisation de tableaux de hachage et de graphes, ces derniers devant être parcourus pour y détecter des séquences de mots *adjacents*.

Dans ce laboratoire, nous utiliserons le langage Python. L'utilisation de l'IDE PyCharm (IDE : Integrated Development Environment) est recommandée, les explications seront données spécifiquement en fonction de cette application.

⇒ Note : **AVANT** la période de laboratoire, vous devriez :

0. Installer Pycharm sur votre ordinateur (si ce n'est déjà fait)
1. Préparer votre environnement PyCharm
2. Vous familiariser avec la navigation dans le code avec l'IDE PyCharm
3. Vous familiariser avec la lecture de la documentation ReadTheDocs du code Python des exercices

0. Installation de l'environnement PyCharm :

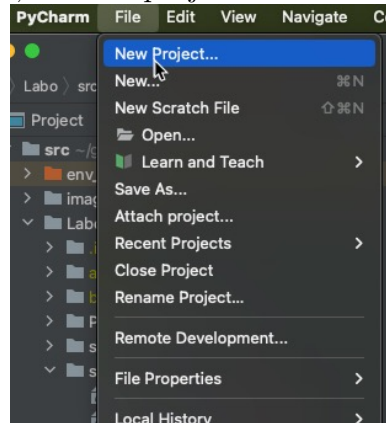
[Demander une licence JetBrains gratuite](#)

[Télécharger PyCharm Professional](#)

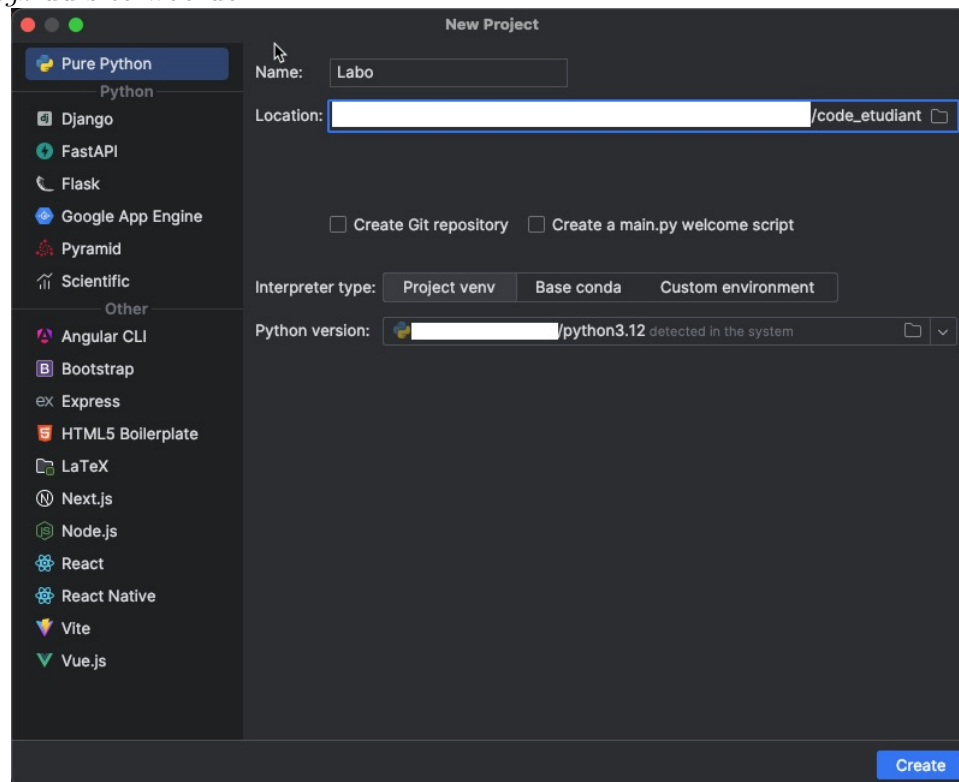
Installer PyCharm sur votre ordinateur

## 1. Préparation de l'environnement PyCharm :

Dans l'application PyCharm, créer le projet *Labo*



Configurer le projet *Labo*. Utiliser le répertoire *code\_etudiant* extrait du fichier *code\_etudiant.tgz* du site web de l'APP.

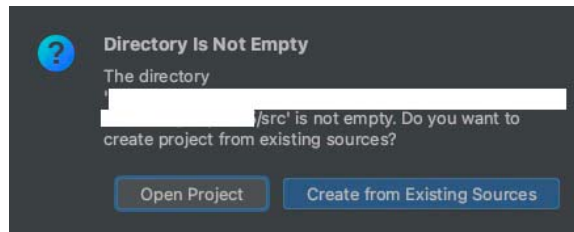


Choisir l'interpréteur (champ *Python version*). L'utilisation de **Python3.12** est recommandée.

- Si Python3.12 n'est pas disponible sur votre ordinateur, installez-le en suivant les instructions sur [le site officiel de Python](#)

Cliquez sur le bouton *Create* :

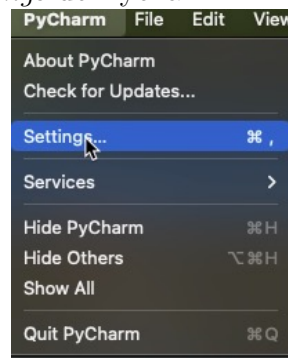
PyCharm indiquera que le répertoire n'est pas vide. Accepter en cliquant sur *Create from existing sources*.



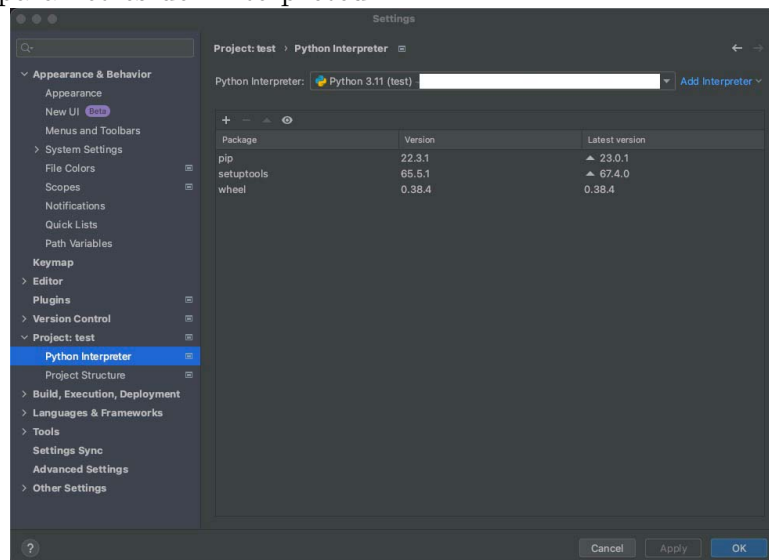
Vous devriez ajouter les *packages* Python suivants dans Pycharm :

— **objprint**, **pip**, **pythonds3**, **tabulate**, **watchpoints**, **wheel**.

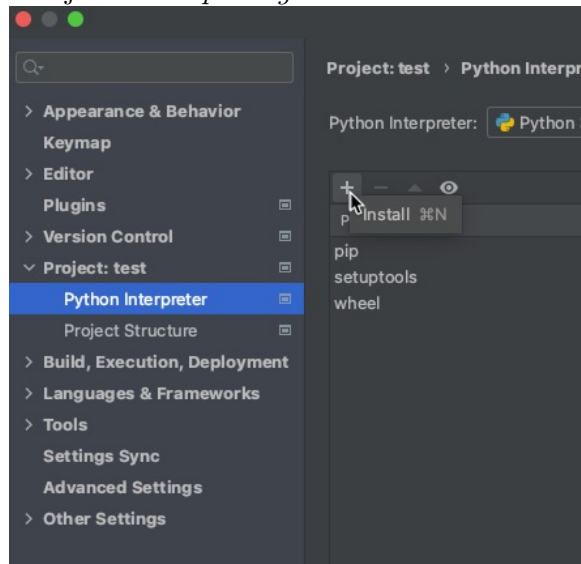
Pour ce faire, accédez aux *settings* de Pycharm :



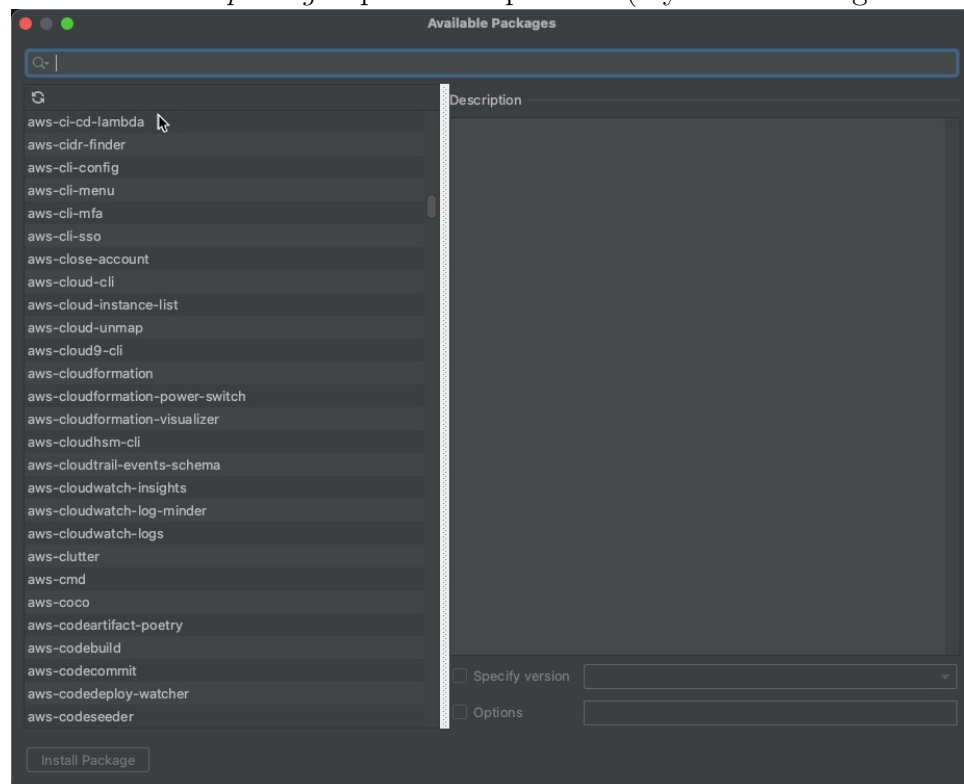
Accédez aux paramètres de l'interpréteur :



Cliquez sur le "+" pour ajouter un *package* :

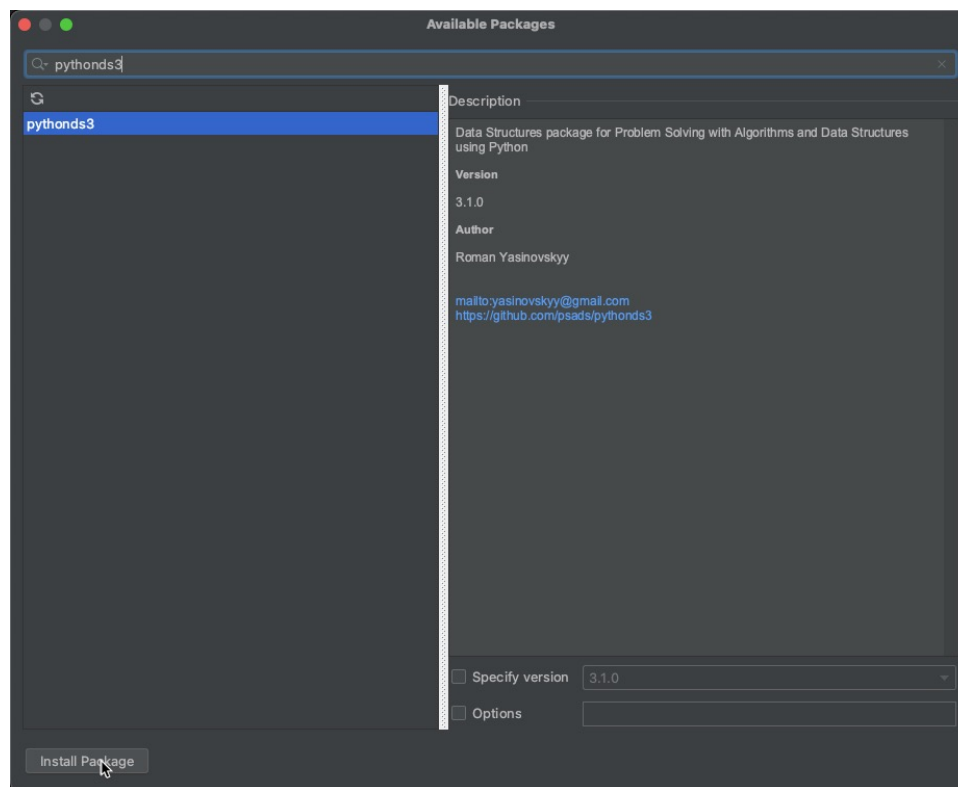


Vous verrez la liste des *packages* qui sont disponibles (il y en a un très grand nombre) :





Et vous pourrez, par exemple, installer `pythonds3` (en tapant le nom dans l'espace de recherche et en cliquant sur le bouton *Install Package* qui se trouve en bas de la page) :

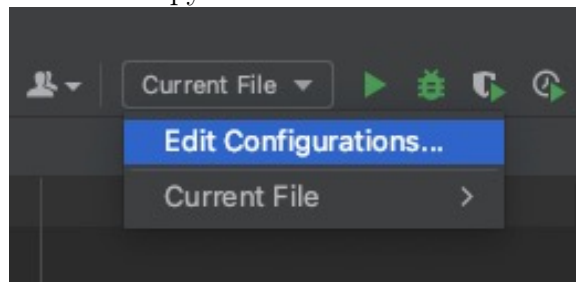


Note : après avoir installé tous les *packages* indiqués plus haut, il est possible qu'il y en ait d'autres qui aient été installés : c'est normal, le système installe automatiquement les *packages* requis par ceux que vous demandez.

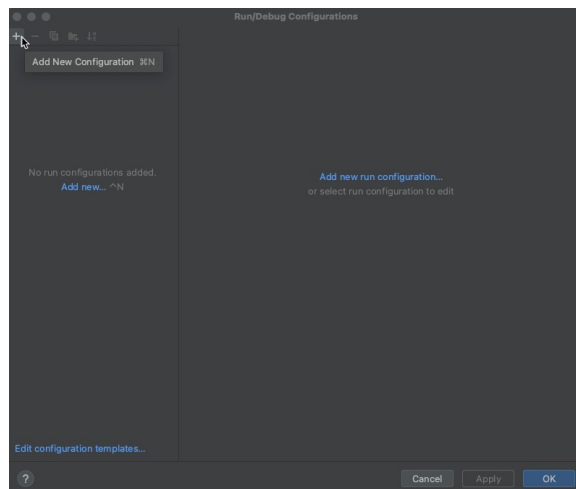
Ouvrir le fichier du premier problème (*labo\_prob1.py*) (cliquez sur ce fichier dans le menu de gauche, dans *Labo/src*) :



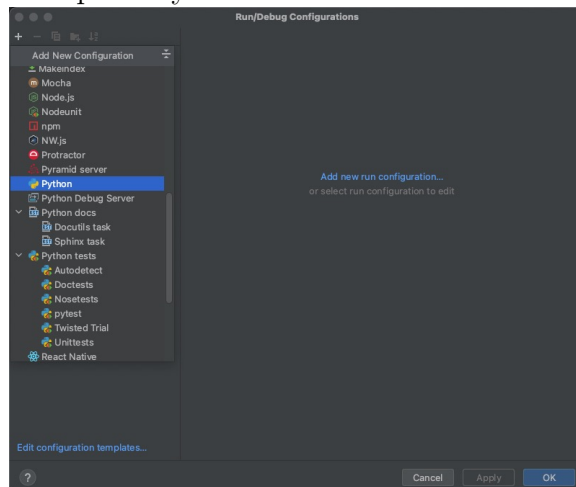
Configurer l'exécution du fichier python :



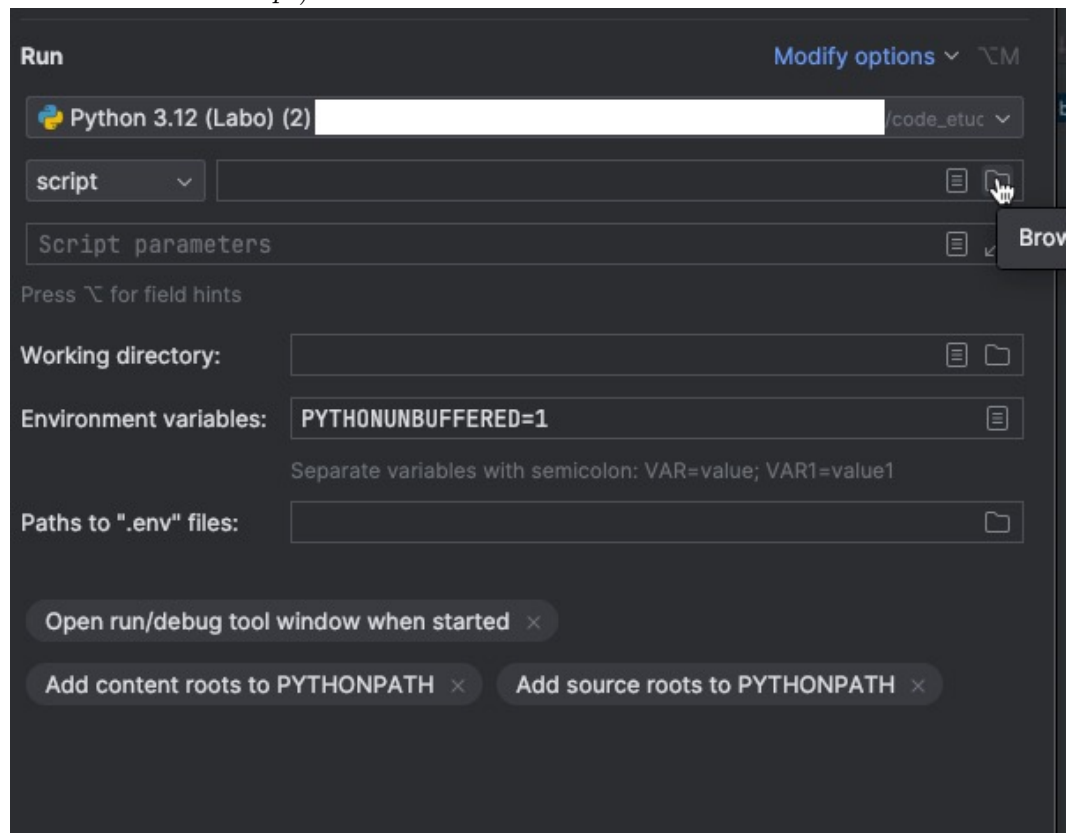
Poursuivre la configuration :



Choisir une configuration pour Python



Nommer cette configuration dans le champ *Name* (labo\_prob1 semble approprié) et choisir le chemin du script `code_etudiant/Labo/src/labo_prob1.py` (Cliquer sur le dossier à droite de *script*) :

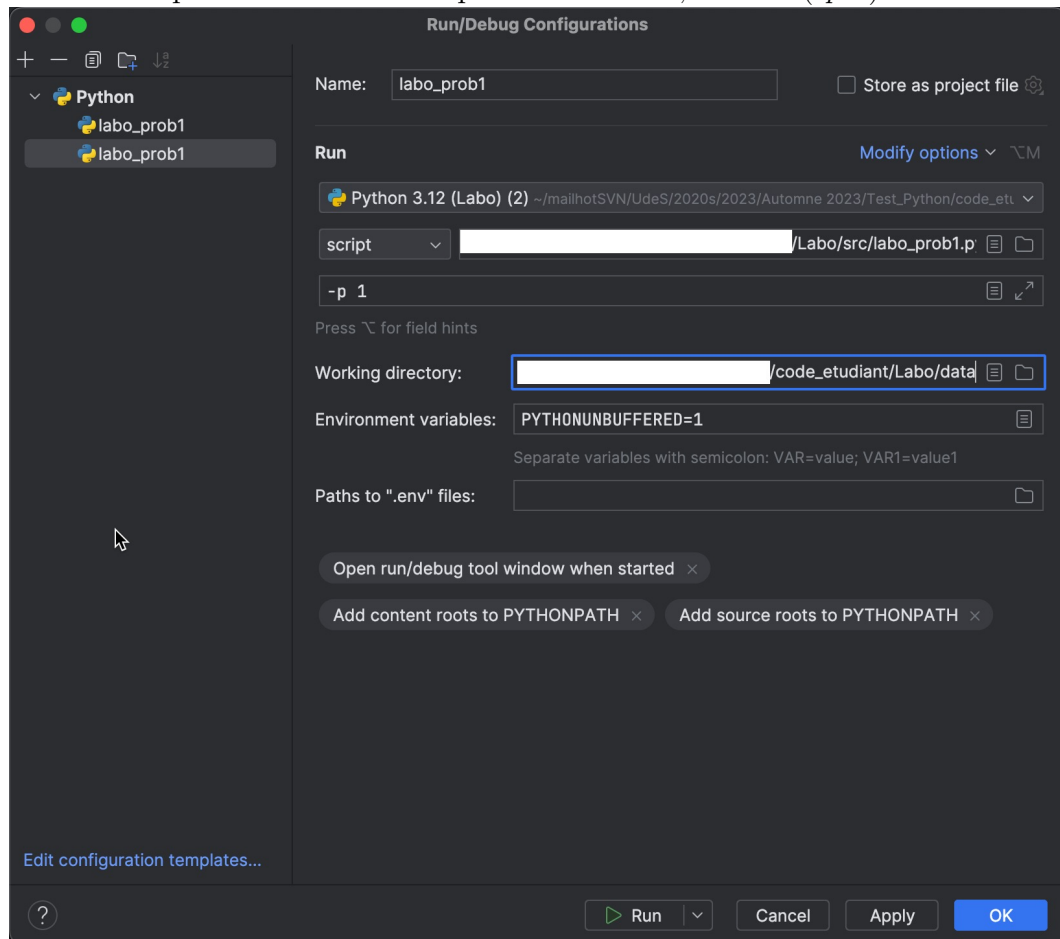


Ajouter le répertoire de travail (*Working directory*), qui contient les fichiers de données que vous utiliserez. Le répertoire de travail devrait être :

**code\_etudiant/Labo/data**

Toujours porter une attention particulière au *Working directory* de vos configurations. C'est dans ce répertoire que le programme Python commencera son exécution. En conséquence, s'il n'y a pas de changements de répertoire effectués à l'intérieur du programme Python, c'est à cet endroit que les fichiers de données seront lus.

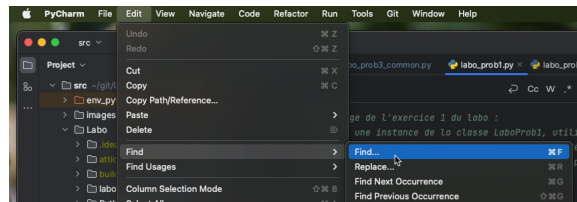
Ajouter aussi des paramètres sur la ligne de commande, dans le champ *Script parameters* Pour la première activité du premier exercice, utiliser (*-p 1*)



## 2. Navigation dans le code à l'aide de PyCharm :

Avant de commencer les activités de débogage et de codage, il est important de savoir comment naviguer dans le code source, pour comprendre la structure du logiciel et bien interpréter ce qui est effectué.

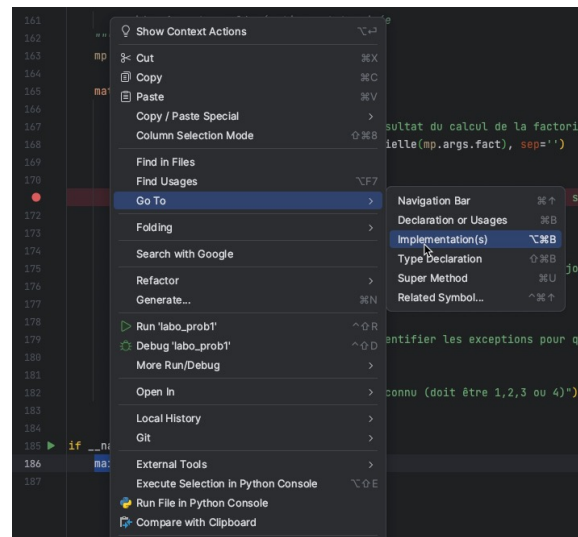
Ouvrir le fichier **labo\_prob1.py**, puis chercher **\_\_main\_\_** :



La commande **Find** déplacera le pointeur de lecture de l'IDE sur la ligne contenant le code suivant :

```
if __name__ == "__main__" :
```

À la ligne suivante, cliquez (à l'aide du bouton droit de la souris) sur **main()** et cliquez sur **GoTo** et **Implementation** :

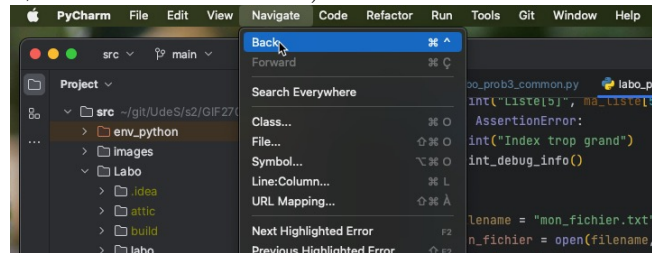


- L'IDE vous positionnera sur la définition de la méthode **main()**.
- Un peu plus bas, vous pouvez voir que l'objet **p1** est créé à l'aide d'un appel à **LaboProb1()** :  

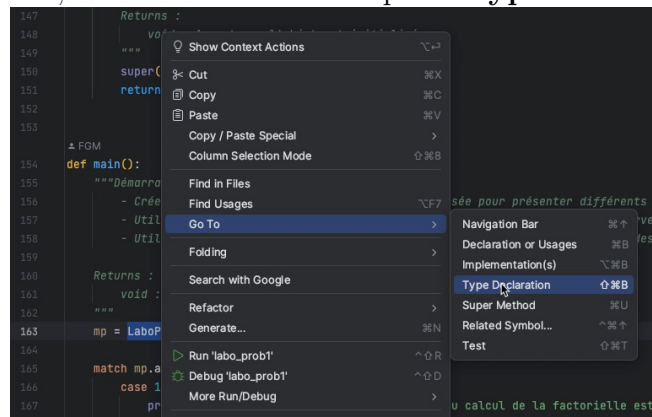
```
p1 = LaboProb1()
```

Cela veut dire que **p1** contiendra une instance de la classe **LaboProb1()**.
- Pour comprendre ce que fait cette classe, utilisez de nouveau **GoTo**.

- Si vous sélectionnez le sous-menu **Implementation**, vous verrez la méthode `__init__(self)` de la classe.
- C'est le constructeur, qui appelle le constructeur de la classe parent de *LaboProb1* (`super().__init__()`).
- À noter que *self*, qui est passé en paramètre au constructeur, indique l'instance (l'objet) qui doit être initialisée et qui sera retournée à la fin (et qui sera assignée à la variable `p1`).
- Pour voir la classe parent, revenez d'abord à la position précédente dans l'IDE (à l'aide de **Navigate, Back** dans le menu) :



- Puis, réutilisez **GoTo**, mais cette fois avec l'option **Type Declaration** :



- L'IDE vous positionnera sur la ligne `class LaboProb1(ParsingClass1)`.
- Ceci indique que la classe *LaboProb1* est dérivée de la classe *ParsingClass1*. C'est d'ailleurs le constructeur de *ParsingClass1* qui est appelé lorsque l'instance de *LaboProb1* est créée et initialisée (avec l'appel `super().__init__()`).
- Pour voir le code de la classe *ParsingClass1*, utilisez **GoTo** et **Type Declaration** sur *ParsingClass1*.
- L'IDE vous positionnera sur la définition de la classe *ParsingClass1*, dans le fichier `labo_prob1_common.py`<sup>1</sup>

1. PyCharm (ainsi que le compilateur Python) savent où trouver *ParsingClass1* grâce à la ligne `from labo_prob1_common import ParsingClass1` en haut du fichier `labo_prob1.py`. Lorsque les commandes `from` ou `import` sont rencontrées, le compilateur recherche d'abord dans le répertoire courant un fichier Python (se terminant par `.py`), ici `./labo_prob1_common.py`.

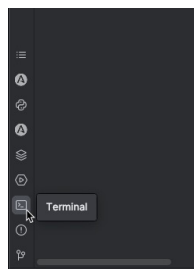
1. Allez voir la méthode `__init__` de la classe *ParsingClass1* (c'est son constructeur). Ici encore, *self*, qui est passé en paramètre, représente l'instance (l'objet) qui est à créer.
2. Vous remarquerez la ligne :  
 $\Rightarrow$  `self.setup_and_parse_cli()`  
 qui représente un appel à une méthode contenue dans la classe *ParsingClass1*.
3. À l'aide de **GoTo** et **Implementation** (sur `setup_and_parse_cli()`), allez lire le code de cette méthode. Vous verrez les lignes suivantes :  
 $\Rightarrow$  `parser = argparse.ArgumentParser(prog='Labo1:Exercice1.py', formatter_class=SmartFormatter)`  
 $\Rightarrow$  `parser.add_argument(...)`  
 $\Rightarrow$  `self.parser = parser`  
 $\Rightarrow$  `self.args = parser.parse_args()`
4. Les deux premières entrées de la liste précédente utilisent le package *argparse*. Pour en savoir plus, vous pouvez consulter la page de documentation qui lui est consacrée sur le [site de Python](#) (vous pouvez aussi chercher **argparse python** avec Google)<sup>2</sup>

---

2. Comment Python sait-il ce que représente *argparse* ?  
 À l'aide de la ligne **import argparse** en haut du fichier.

Si le compilateur ne trouve pas le fichier dans le répertoire courant, il recherchera ensuite dans la bibliothèque locale utilisée par le compilateur, puis dans la bibliothèque globale python de votre ordinateur. Les *packages* que vous avez installés pour préparer ce laboratoire se retrouvent soit dans la bibliothèque locale, soit dans la bibliothèque globale. La bibliothèque locale est définie et utilisée dans le projet *GIF270Labo* que vous avez créé.

Pour connaître où se trouve *argparse*, démarrer un terminal dans PyCharm (menu en bas à gauche) :



Démarrer (taper) *python* dans le terminal, et donner les commandes suivantes :

```
import argparse
print(argparse.__file__)
```

Vous verrez ainsi à quel endroit se trouve *argparse* sur votre ordinateur.

Faites le même exercice pour les modules suivants : *labo\_prob1\_common*, *pythonds3*.

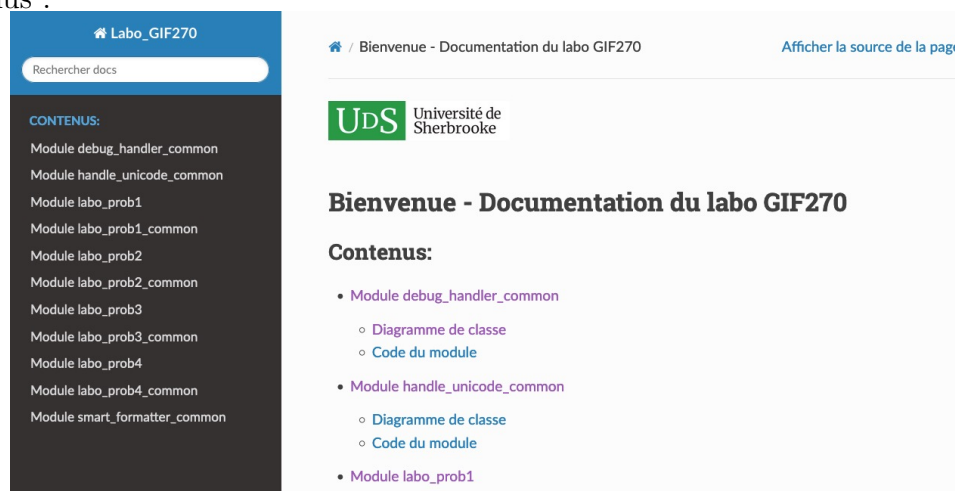
5. L'appel `argparse.ArgumentParser(...)` crée un *parser*, qui permet de lire les paramètres passés sur la ligne de commande de l'application Python.
6. L'appel `parser.add_argument(...)` permet de définir un nouveau paramètre sur la ligne de commande. On peut inclure le type de valeur attendue, une valeur par défaut, ainsi qu'une brève explication qui apparaîtra si on appelle l'application Python avec le paramètre **-h**.
7. La ligne `self.parser = parser` conserve la structure du parser dans l'objet en création (*self*, qui deviendra **p1**).
8. La ligne `self.args = parser.parse_args()` effectue l'opération de parsing et ajoute tous les paramètres qui sont lus sur la ligne de commande dans l'objet *args*, qui est conservé dans l'objet en création (*self*, qui deviendra **p1**). Par la suite, on peut consulter l'objet *args*, dont les champs correspondent aux paramètres passés sur la ligne de commande.



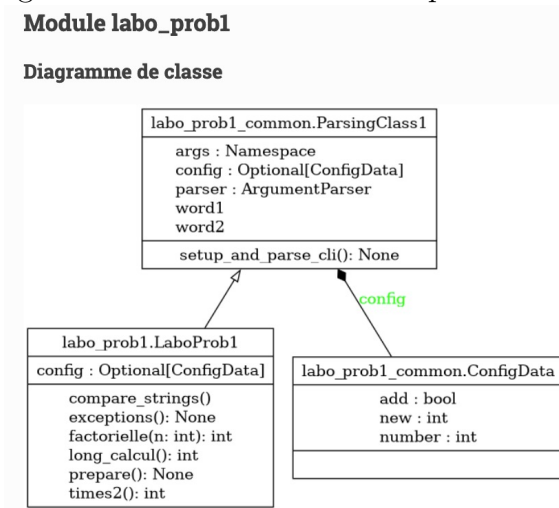
### 3. Lecture de la documentation ReadTheDocs du code Python des exercices :

La documentation de type ReadTheDocs est maintenant le standard utilisé pour le langage Python. Pour en savoir plus, vous pouvez consulter [le site officiel ReadTheDocs](#). Les étapes qui suivent devraient vous aider à profiter pleinement des possibilités de ce type de documentation.

- Ouvrir le fichier *index.html* situé dans le répertoire *code\_etudiant/Labo/build/html*
- Cliquer sur *Module labo\_prob1*, soit dans le menu de gauche, soit dans la liste de contenus :



- Vous observerez le diagramme de classe du code du premier exercice de laboratoire :



Dans ce diagramme, on peut voir que la classe *ParsingClass1* est définie dans le module (fichier) *labo\_prob1\_common*. Cette classe comprend trois champs (*args*, *config*, *parser*) et une méthode (*setup\_and\_parse\_cli()*).

De plus, on observe que la classe *LaboProb1* hérite de la classe *ParsingClass1* (la flèche avec une tête blanche entre les deux classes nous l'indique). Ceci veut dire que tous les éléments de la classe *ParsingClass1* (les trois champs et sa méthode unique) sont accessibles par la classe *LaboProb1*.

On observe aussi que la classe *ConfigData* est utilisée pour définir le type du champ *config* (la flèche avec un losange noir nous l'indique, avec le nom du champ en vert, qui annote cette flèche)

- En vous déplaçant sous le diagramme de classe, vous verrez la documentation sur la classe *LaboProb1* et les méthodes qu'elle contient :

### Code du module

Code pour explorer le premier exercice du laboratoire - APP du cours GIF270

L'exercice 1 touche le débogage et les éléments suivants :

- Utilisation de points d'arrêt (breakpoints) pour l'identification de problèmes d'exécution et de logique
- Interruption d'exécution sans fin
- Détection et traitement des exceptions

Copyright 2023, 2024 Frédéric Mailhot et Université de Sherbrooke

`class labo_prob1.LaboProb1` [\[source\]](#)

Bases : `ParsingClass1`

Création d'un ensemble de méthodes qui ont des comportements problématiques - Classe hérite de `ParsingClass1`, qui lit les paramètres de la ligne de commande - Les paramètres sont accessibles dans `mp.args` - Méthodes incluses : `factorielle`, `times2`, `long_calcul`, `exceptions`

Initialisation d'une nouvelle instance de `LaboProb1` :

- Utilise l'initialisation de la classe héritée par `LaboProb1` (`ParsingClass1`)

Returns :

(void) : Au retour, l'objet est initialisé

En cliquant sur le champ [\[source\]](#) (à droite de `labo_prob1.LaboProb1`), vous pourrez regarder directement le code source :

```
[docs]
class LaboProb1(ParsingClass1):
    """Création d'un ensemble de méthodes qui ont des comportements problématiques
    - Classe hérite de ParsingClass1, qui lit les paramètres de la ligne de commande
    - Les paramètres sont accessibles dans mp.args
    - Méthodes incluses : factorielle, times2, long_calcul, exceptions
    """
    [docs]

    def factorielle(self, n: int) -> int:
        """Calcul de factorielle :
        - Appel récursif (sur n, n-1, n-2, ...)
        - La fin de la récursion est déterminée par le champ fact_end

        Args :
            n (int) : Le nombre pour lequel on cherche la factorielle

        Returns :
            (long) : La valeur de (n!)
        """
        if n >= self.args.fact_end:
            fact = n * self.factorielle(n - 1)
            return fact
        return 1
```

⇒ Maintenant que vous savez naviguer dans la documentation ReadTheDocs, nous pouvons commencer les activités du laboratoire à la page suivante, en débutant avec le débogage. Vous pourrez vous référer à la documentation ReadTheDocs en tous temps, lorsque des clarifications sont nécessaires.

⇒ Vous remarquerez que les méthodes qui apparaissent dans le code qui vous est fourni sont annotées avec des types, tant au niveau des paramètres d'entrée (chaque paramètre est suivi de ":", puis de son type) que du type de ce qui est retourné (la fin des parenthèses des méthodes est suivie de "->", puis du type). Ces annotations ne sont pas obligatoires en Python. Vous n'avez donc pas à en mettre dans votre code. Elles sont utilisées dans le code qui vous est fourni ici pour permettre de bien le documenter et de produire automatiquement les diagrammes de classe (à l'aide de l'utilitaire [sphinx](#)).

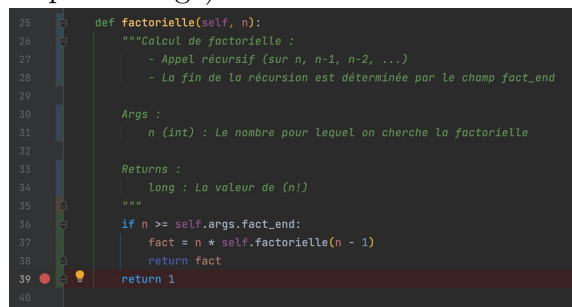
## L.E1 Le débogage

Lorsqu'on fait face à un programme qui n'effectue pas les opérations escomptées, il semble aller de soi de simplement utiliser des *prints*. Cependant, il existe des techniques beaucoup plus efficaces pour identifier rapidement les problèmes, permettant de les résoudre par la suite. Dans cet exercice, vous ferez face à une série de problèmes qui vous feront maîtriser plusieurs de ces techniques d'identification et de solution de problèmes.

Pour cette activité, utiliser les fichiers *labo\_prob1.py* et *labo\_prob1\_common.py*. Avec Py-Charm, utilisez le projet *GIF270Labo* déjà créé. Pour connaître toutes les options, vous pouvez utiliser le paramètre *-h*.

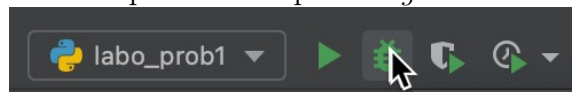
### Exercice 1.1 - Mauvais calcul

Utiliser le paramètre *-p 1*. Est-ce que le résultat du calcul de  $30!$  est le bon ? Tentez de comprendre pourquoi en mettant un point d'arrêt (*breakpoint*) à côté de la dernière ligne de retour de la méthode *factorielle* de la classe *LaboProb1* (cliquez dans la marge pour faire apparaître le point rouge) :



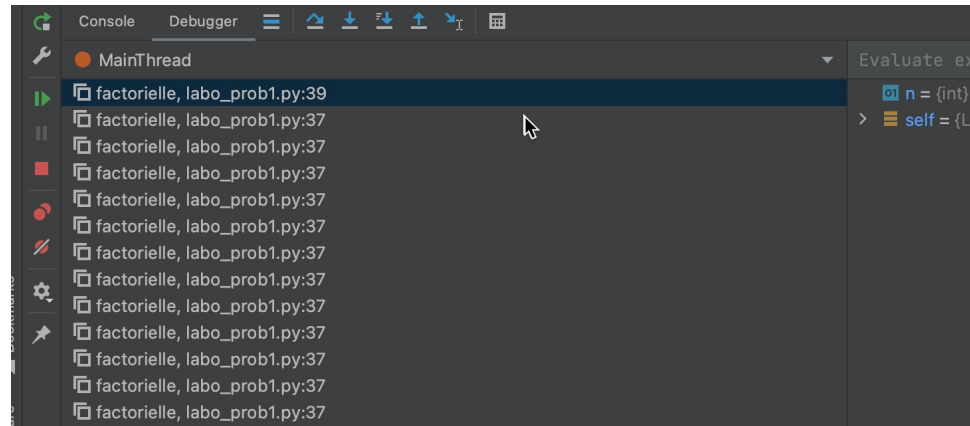
```
25 def factorielle(self, n):
26     """Calcul de factorielle :
27         - Appel récursif (sur n, n-1, n-2, ...)
28         - La fin de la récursion est déterminée par le champ fact_end
29
30     Args :
31         n (int) : Le nombre pour lequel on cherche la factorielle
32
33     Returns :
34         long : La valeur de (n!)
35     """
36     if n >= self.args.fact_end:
37         fact = n * self.factorielle(n - 1)
38         return fact
39     return 1
40
```

Démarrez le débogueur en cliquant sur le petit *bug* vert en haut de la fenêtre :

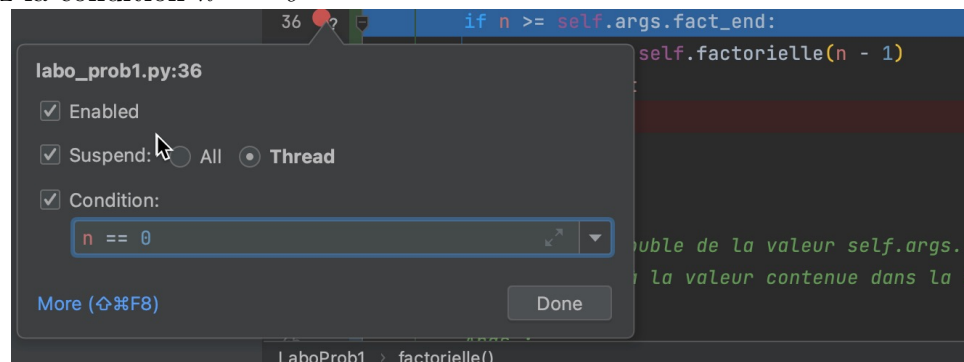


Lorsque le programme s'arrête, observer les valeurs des variables locales, incluant *self*. Est-ce que la valeur de *n* est raisonnable ? Pourquoi ? Dans la colonne de gauche, vous voyez la série d'appels à la méthode *factorielle*, l'entrée tout en haut correspondant à l'appel où le débogueur s'est arrêté, la ligne suivante correspondant à l'appel qui l'a précédé, etc.

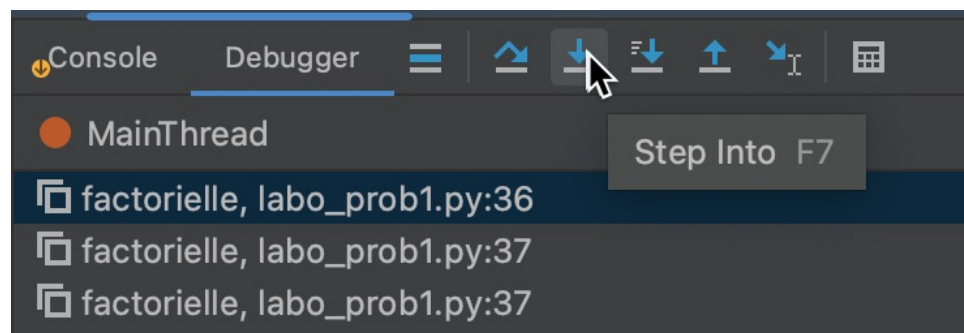
En cliquant sur l'une des entrées, vous pouvez regarder les valeurs des variables locales à ce niveau d'appel.



Mettez un point d'arrêt à la ligne où apparaît : `if n >= self.args.fact_end :`  
En cliquant à l'aide du bouton de droite sur ce point d'arrêt, ajoutez la condition `n == 0`



Redémarrez le débogueur. Lorsqu'il s'arrête, observez toutes les variables locales. Avancez d'une instruction dans le programme, à l'aide du bouton *Step Into*



Pouvez-vous déterminer ce qui ne fonctionne pas ? Faites la correction et exécutez de nouveau, pour vérifier votre travail.

## Exercice 1.2 - Structure manquante

Utiliser le paramètre `-p 2`. Démarrez le programme. Qu'arrive-t-il ?

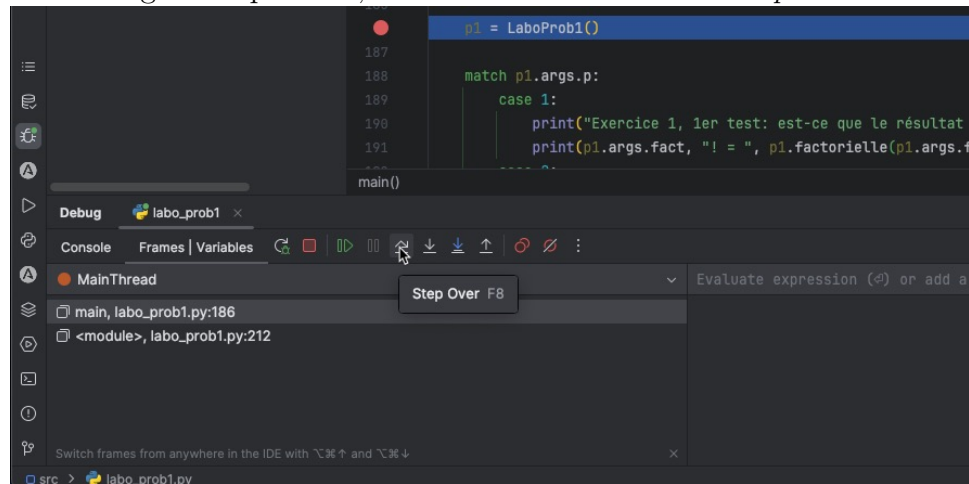
Pour comprendre ce qui se passe, **ajouter des points d'arrêt** aux lignes suivantes :

- Dans la méthode `main`, ajoutez un point d'arrêt à la ligne :  
`p1 = LaboProb1()`
- Dans la méthode `times2` de la classe `LaboProb1`, ajoutez un point d'arrêt à la ligne :  
`if self.config.number != self.args.val :`

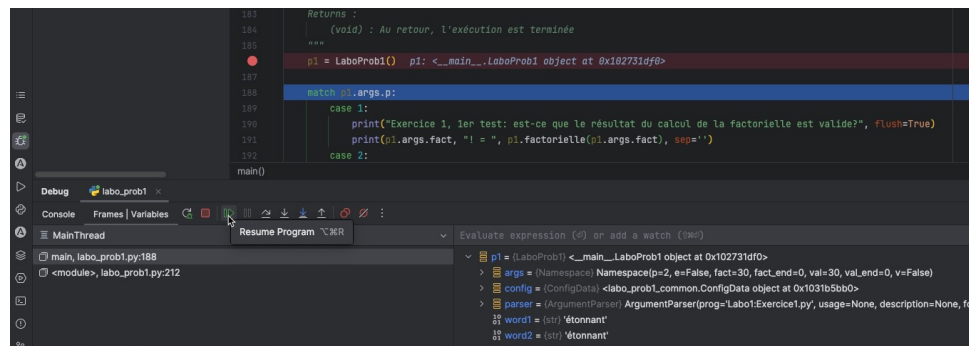
Après l'ajout des points d'arrêt, redémarrez le débogueur.

Le premier arrêt se fera sur la ligne `p1 = LaboProb1()`.

Exécutez cette ligne uniquement, en utilisant la commande *Step Over* :



Observez le contenu de la variable `p1`, puis poursuivre l'exécution du programme :



- Lorsque le débogueur arrête dans la méthode `times2()` de l'objet `p1`, observer de nouveau les champs définis dans l'objet `p1`.
- Est-ce que l'un des champs a changé entre la création de l'objet `p1` et son utilisation dans la méthode `times2` ?

- Il est parfois difficile de déterminer à quel endroit une donnée est modifiée.

Vous allez maintenant utiliser la méthode *watch*,

définie dans le module *watchpoints*

(voir la commande *from watchpoints import watch*) en haut du fichier.

Vous êtes familiers avec les *breakpoints*, qui permettent au débogueur d'arrêter l'exécution à une ligne précise dans un programme.

Un *watchpoint* est un concept similaire, mais pour la mémoire, permettant d'identifier à quel moment une adresse mémoire (une variable) est modifiée. Plusieurs débogueurs populaires (par exemple, *gdb*) permettent l'utilisation directe de *watchpoints*.

PyCharm n'a pas cette fonctionnalité, mais en utilisant la méthode *watch* du module *watchpoints*, on permet au programme Python d'indiquer à quel endroit s'effectue la modification d'une variable :

- Ajoutez les lignes : *watch(p1)* et *watch(p1.config)* immédiatement après la création de l'objet *p1* (dans la méthode *main*, après la ligne *p1 = LaboProb1()*) et exécutez le code de nouveau, **cette fois sans utiliser le débogueur** (la méthode *watch()* cause parfois des difficultés au débogueur de PyCharm)

Les appels à la méthode *watch()* vont afficher à l'écran à quel moment ces objets sont modifiés. Après l'exécution du programme, regardez tout ce qu'il a imprimé. Vous devriez être en mesure de découvrir où se trouve le problème. Identifiez et retirez (ou commentez) la ligne de code fautive.

- Qu'arrive-t-il lorsque vous enlevez la ligne problématique ? Est-ce que *watch()* est toujours déclenché ? Est-ce que le problème est encore présent ? Lorsque vous avez identifié et corrigé le problème, retirez la ligne *watch(p1)* (comme mentionné plus haut, l'utilisation de cette méthode peut causer des problèmes au débogueur de PyCharm)

### Exercice 1.3 - Calcul interminable

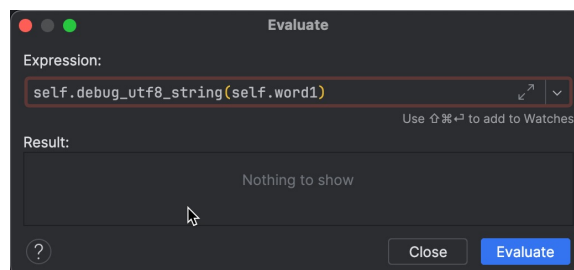
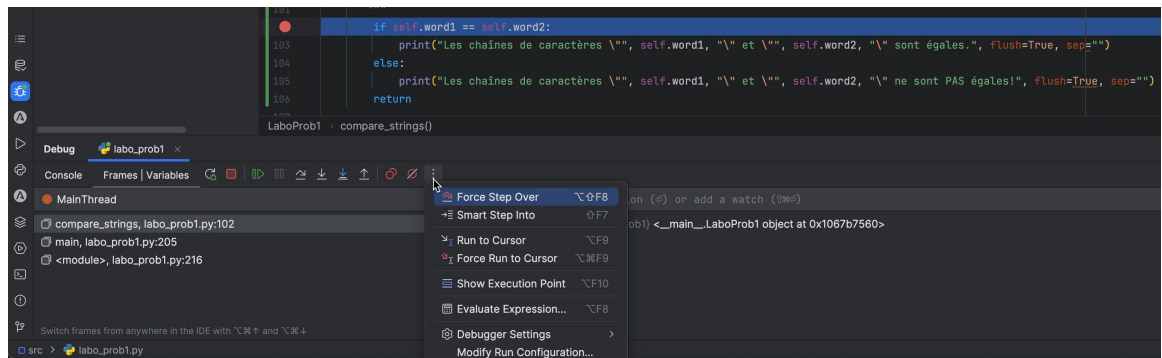
Utiliser le paramètre *-p 3*. Est-ce que le calcul se termine ? Si ce n'est pas le cas, arrêter l'exécution en utilisant le bouton rouge (en haut de la fenêtre). À quelle ligne le programme s'est-il arrêté ? Mettre un point d'arrêt et exécuter avec le débogueur. Pouvez-vous déterminer ce qui se passe ? Est-ce que la variable *res* change ? Qu'en est-il de la variable *a* ? Pour savoir comment corriger la situation, lire le commentaire au tout début de la méthode, qui explique le calcul attendu.

- Quelle est la complexité de ce calcul ? Pourriez-vous le faire en  $\mathcal{O}(1)$  ? Si oui, comment ? Si non, pourquoi ?

## Exercice 1.4 - Chaînes de caractères Unicode (utf-8) : différentes ou égales ?

Utiliser le paramètre `-p 4`. Deux chaînes de caractères sont imprimées à l'écran, et une comparaison est effectuée pour déterminer si elles sont identiques. Il est surprenant de constater que la comparaison indique qu'elles sont différentes, alors que leur impression à l'écran nous porte à penser qu'il s'agit de la même chaîne de caractères.

- Pour déterminer ce qui se passe, ajoutez un *breakpoint* sur le `if` du début de la méthode `compare_strings()` de la classe `LaboProb1` et redémarrez l'exécution.
- Lorsque le débogueur arrête le programme, utilisez le bouton *Evaluate Expression* :

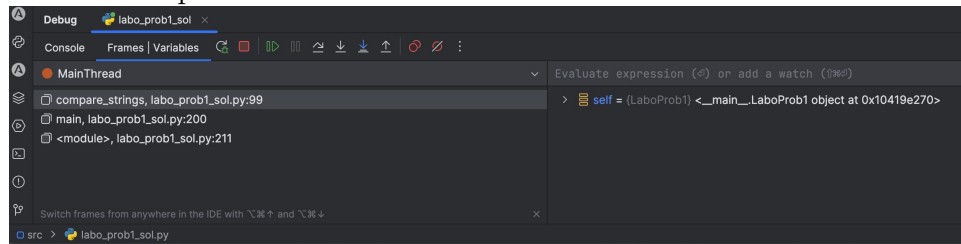


- Vous allez maintenant appeler dynamiquement une méthode du programme à partir du débogueur. On peut toujours appeler ainsi une méthode d'un programme lorsque le débogueur est en pause. Dans le champ *Expression*, appelez la méthode suivante : `self.debug_utf8_string(self.word1)` et observez bien la valeur de retour. Il s'agit des codes hexadécimaux de la chaîne de caractères en format utf8. Faites la même chose avec le champ `self.word2` (au lieu de `self.word1`). Y a-t-il des différences ?
- Nous venons d'utiliser deux fois la commande *Evaluate Expression*, pour comparer les valeurs de deux chaînes de caractères. Cette méthode fonctionne très bien, surtout lorsqu'on veut vérifier une chose unique. Ici, nous avons comparé deux chaînes de caractères, en deux appels distincts, le deuxième appel écrasant le

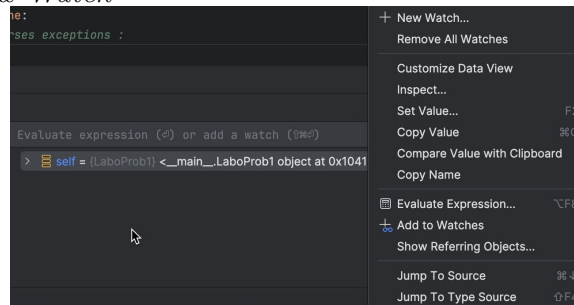


premier, ce qui rend la comparaison difficile. Dans ce cas précis, peut-on faire mieux ?

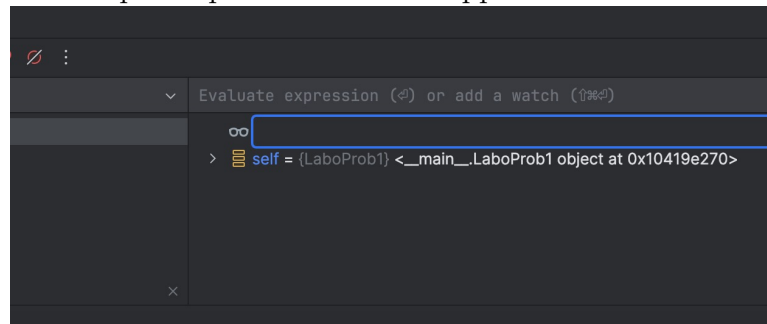
- La réponse est oui : nous allons utiliser une méthode alternative, en ajoutant deux entrées à l'espace *watch*. Cliquez avec le bouton de droite sur le champ *self* qui apparaît dans l'espace en bas à droite.



- Sélectionnez *+ New Watch*



- Un champ avec une petite paire de lunettes apparaîtra.



- Appeler la méthode `self.debug_utf8_string(self.word1)` dans le champ à droite des lunettes, puis recommencer, cette fois pour `self.word2`.
- **Note importante :** La fonctionnalité *watch* du débogueur de PyCharm ne correspond pas à celle que nous avons utilisée dans l'exercice 1.2 : dans le débogueur de PyCharm, il s'agit simplement d'afficher ce qui est demandé (par défaut, seulement les valeurs des variables locales d'une méthode), et non d'arrêter l'exécution du programme lorsque cette variable est modifiée. Il est malheureux que les concepteurs de l'IDE PyCharm aient choisi le même terme (*watch*), mais c'est compréhensible, puisqu'il s'agit de *regarder* une valeur. Il est aussi important de remarquer que lorsqu'on ajoute un champ *watch*, le débogueur tentera d'y afficher

l'information voulue à chaque arrêt. Dans le cas des deux *watch* ajoutés ici, qui utilisent *self*, ces champs n'afficheront rien quand le programme s'arrête dans une méthode où *self* n'est pas disponible.

- Vous pouvez maintenant facilement comparer les deux chaînes de caractères, et vous verrez que les codes hexadécimaux sont différents. C'est pourquoi le *if* donne ce résultat, même si lors de l'impression à l'écran, elles semblent identiques.
- Comment expliquer (et corriger) cette situation ? L'explication (très simplifiée) est que *utf8* est un standard d'encodage de caractères utilisant 8 ou 16 bits, permettant de représenter de façon concise un encodage unicode (qui, lui, utilise toujours 16 bits). Dans certaines situations, il peut exister deux séquences de codes *utf8* qui correspondent au même caractère unicode. C'est le cas pour toutes les lettres accentuées, qui peuvent être représentées soit par un, soit par deux caractères *utf8*. C'est ce qui arrive ici.

Pour corriger la situation, il faut ramener ces chaînes de caractères vers une expression standard, c'est-à-dire la *normaliser*. Pour ce faire, vous pouvez utiliser la méthode *normalize\_string* de la classe *LaboProb1* (héritée de *ParsingClass1*, qui elle-même l'hérite de *HandleUnicodeCommon*). Vous pouvez utiliser cette méthode sur une chaîne de caractères, et **au retour**, on reçoit une chaîne normalisée. Ajoutez deux lignes pour normaliser *self.word1* et *self.word2* au tout début de la méthode *compare\_string*.

**Attention** : que fait la méthode *normalize\_string* ? En particulier, est-ce qu'elle modifie la chaîne de caractères qui lui est passée en paramètre, ou bien est-ce qu'elle produit une nouvelle chaîne corrigée, qui est retournée à la fin de la méthode ? Si vous normalisez adéquatement *self.word1* et *self.word2*, la comparaison devrait maintenant fonctionner.

- Quel est le meilleur endroit pour corriger le problème avec les chaînes de caractères ? Est-ce que la méthode *compare\_string* est vraiment le meilleur endroit pour le faire ? À quel moment les chaînes de caractères *self.word1* et *self.word2* sont-elles créées ? Dans un programme, lorsqu'une méthode utilise des données déjà existantes, il ne devrait pas être nécessaire de s'assurer de la validité de ces données à ce moment-là. Sinon, cela veut dire que toutes les méthodes qui utilisent des données devraient les valider, ce qui serait très fastidieux et sujet à erreur.

### Exercice 1.5 - Traitement des exceptions

Utiliser le paramètre `-p 5`. Vous observerez que l'exécution est arrêtée par des exceptions qui ne sont pas traitées. Vous observerez aussi que l'exécution se termine par le message :

*Process finished with exit code 1.*

Si le code de sortie est différent de 0 (ce qui est le cas ici), cela indique que l'exécution ne s'est pas déroulée normalement.

Utilisez ensuite les paramètres d'entrée `-p 5 -e`, ce qui activera l'utilisation du traitement des exceptions. Malheureusement, les exceptions qui sont "attrapées" (*AssertionError*) ne correspondent pas à celles qui sont déclenchées.

Observez le message de retour du débogueur et remplacez judicieusement le type d'exception utilisé dans la clause *except*. Identifiez chaque type d'exception qui apparaît et modifier le code en conséquence, pour traiter toutes les exceptions qui sont produites dans ce code.

Lorsque toutes les exceptions sont traitées correctement, mettez un point d'arrêt dans l'une des conditions *except*, sur l'un des appels à la méthode *print\_debug\_info*, et entrez (*Step Into*) dans cette méthode. Observez ce que fait cette procédure, et étudiez les structures qu'elle manipule. Il est possible de savoir à quelle ligne de quel fichier l'exception a eu lieu. On peut aussi imprimer le code source (ce qui est fait dans cette méthode statique).

## L.E2 Utilisation de fonctions de hachage et de comparaison personnalisées

Nous avons vu lors du premier procédural que la fonction de hachage ainsi que la fonction de comparaison sont deux éléments essentiels d'un tableau de hachage. Dans cet exercice, nous allons modifier la classe *Bigram*, qui permet de traiter une paire de mots (deux chaînes de caractères). En y redéfinissant les fonctions `__eq__()` et `__hash__()`, cela permettra ensuite d'utiliser des instances de cette classe dans un tableau de hachage standard *dict* de Python.

Vous pourrez utiliser deux fichiers : *paires\_de\_mots.txt* et *paires\_de\_mots2.txt* pour valider le fonctionnement du système. Vous devrez trouver combien de fois chaque bigramme apparaît dans chacun des fichiers, puis identifier les bigrammes qui sont présents dans les deux fichiers. Enfin, vous devrez trouver la longueur des vecteurs correspondant aux fichiers utilisés, les normaliser, et pour terminer, effectuer le produit scalaire de ces vecteurs pour déterminer l'angle qui les sépare.

Le code *labo\_prob2.py* et *labo\_prob2\_common.py* est fourni.

Lisez le code du premier fichier, *labo\_prob2.py* (en particulier, les commentaires qui suivent les *import*). Modifiez le code pour exécuter ce qui est demandé :

- Classe Bigram :
  - `__eq__()`
  - `__hash__()`
- Classe LaboProb2 :
  - `add_bigram()`
  - `vector_size()`
  - `scalar_product()`
  - `cosine()`

**Exercice 2.0 - Test du code modifié** Lorsque vous aurez modifié toutes ces méthodes, appelez le programme avec les paramètres suivants :

```
-f paires_de_mots.txt -t paires_de_mots2.txt
```

L'exécution initiale du code produira une erreur. Quelle est cette erreur ? Qu'est-ce que ça implique ? Vous allez observer que la méthode **`add_bigram_vector`** est incorrecte. Ne la corrigez pas, relancez plutôt l'exécution avec soit :

- `-p 2 -f paires_de_mots.txt -t paires_de_mots2.txt`  
ou
- `-p 3 -f paires_de_mots.txt -t paires_de_mots2.txt`

- Dans le premier cas (**-p 2**), la méthode **add\_bigram\_object** sera appelée (modifiez-la pour qu'elle soit fonctionnelle, en utilisant une instance de la méthode **Bigram**).
- Dans le deuxième cas (**-p 3**), la méthode **add\_bigram\_other** sera appelée (modifiez aussi cette méthode pour qu'elle soit fonctionnelle, cette fois en utilisant un objet Python qui s'apparente à une liste, mais qui est immuable (quel est ce type d'objet ?)).

Lorsque les deux méthodes seront corrigées, vous pourrez alors passer à la suite du laboratoire.

**Exercice 2.1** Y a-t-il des bigrammes communs entre les deux fichiers de texte ?

**Exercice 2.2** Quel est l'angle entre les deux textes ?

**Exercice 2.3** Refaites l'appel, cette fois en utilisant le même fichier pour les paramètres *-f* et *-t*. Quel est maintenant l'angle entre les deux textes ?

**Exercice 2.4** Comment pourriez-vous modifier le code pour supporter des 3-grammes ? Des N-grammes ? Quelles classes et méthodes devriez-vous modifier ?

**Exercice 2.5** Comment les méthodes `__eq__()` et `__hash__()` sont-elles utilisées dans les *dict* (tableaux de hachage) de Python ?

Note : Le deuxième fichier de code (*labo\_prob2\_common.py* n'a pas à être modifié. C'est là que les paramètres de la ligne de commande sont lus. On y trouve aussi la méthode utilisée pour lire l'ensemble des lignes de chacun des fichiers de texte. Enfin on y trouve les classes de base *BigramCommon* et *LaboProb2Common*, qui fournissent l'ensemble des méthodes nécessaires, certaines (celles identifiées plus haut) devant être redéfinies.

### L.E3 Utilisation d'arbres équilibrés pour ordonner des nombres

Les arbres binaires permettent entre autres choses de représenter des ensembles d'éléments qu'on peut comparer (plus petit, égal, plus grand). Nous avons vu dans le premier procédural qu'il faut faire attention à la structure d'un arbre binaire lors de sa construction. Sinon, dans le pire des cas, il peut devenir l'équivalent d'une liste chaînée et perdre sa capacité d'atteindre un élément dans un temps  $\mathcal{O}(\log n)$ . Pour ce faire, il est possible d'utiliser des arbres qui s'équilibrent automatiquement pendant leur création. Un arbre de type AVL (d'après les noms des deux mathématiciens soviétiques qui l'ont proposé en 1962, Adelson-Velskii et Landis) est un type d'arbres binaires qui a la propriété d'être toujours bien équilibré. En effet, lors de chaque insertion (ou, le cas échéant, de retrait) d'un nœud dans l'arbre, la différence

de profondeur des sous-arbres droit et gauche est calculée, et une opération (rotation droite ou rotation gauche) est appliquée récursivement si cette différence est plus grande que 1. Dans cet exercice, vous utiliserez (en le complétant d'abord) un algorithme qui réalise un arbre AVL pour ordonner une suite de 100 nombres.

Le code disponible à la section 6.13 du livre de référence (sur les arbres binaires) ainsi qu'à la section 6.17, sur les arbres AVL, a été intégré dans le fichier *labo\_prob3\_common.py* et est utilisé par le fichier *labo\_prob3.py*. Vous devrez modifier ce dernier et coder la méthode *AVLTree.rotate\_right()*. Vous observerez que la classe *AVLTree* hérite de la classe *AVLTree\_common*, qui comprend l'ensemble des méthodes nécessaires à la création et la gestion d'un arbre AVL, mais où la méthode *rotate\_right()* est incomplète. L'environnement de test de ce numéro de laboratoire est défini dans la classe *LaboProb3Common*. Vous pouvez parcourir le code de cette classe (ainsi que celui de la classe *ParsingClass3*, qui lit les informations passées en paramètres) pour mieux comprendre comment le tout fonctionne, mais vous pouvez simplement ajouter le code manquant (*rotate\_right()*) et utiliser directement le programme de test.

Les fichiers *nombres\_a\_ordonner.txt* et *nombres\_a\_ordonner2.txt* seront utilisés pour valider votre code.

**Exercice 3.0 - Test du code modifié - Arbre AVL** Lorsque vous aurez modifié la méthode *rotate\_right()*, appelez le programme avec les paramètres suivants :

```
-f nombres_a_ordonner.txt
```

Quelle est la profondeur de l'arbre créé ?

**Exercice 3.1** Refaite le test précédent, cette fois en modifiant ainsi les paramètres :

```
-f nombres_a_ordonner.txt -nrb
```

Le paramètre *-nrb* indique au programme de ne pas faire de rebalancement, qui est alors désactivé. Quelle est la profondeur de l'arbre dans ce cas ?

**Exercice 3.2** Recommencer l'exercice précédent (avec le paramètre *-nrb*), cette fois avec le fichier *nombres\_a\_ordonner2.txt*. Quelle est la profondeur de cet arbre ? Pouvez-vous expliquer ce qui se passe ?

### L.E4 (Facultatif) Le problème de l'échelle de mots - utilisation de graphes

Le problème de l'échelle de mots (*word ladder problem* en anglais) a été proposé le jour de Noël 1877 par Lewis Carroll (l'auteur de *Alice au pays des merveilles*). Il s'agit de trouver une séquence de mots qui ne diffèrent que d'une lettre, reliant deux mots distincts. Par exemple, la séquence " bateau - rateau - rameau " serait acceptable. Pour cet exercice, nous utiliserons

un graphe pour modéliser les mots et leur proximité. Chaque mot sera représenté par un noeud, qui sera relié par des arcs à tous les mots auxquels il peut être associé. Cependant, nous résoudrons une généralisation du problème, où la distance entre deux mots pourra être de 1, 2 ou 3 lettres. Nous permettrons aussi de calculer une distance entre des mots avec des tailles différentes, le mot le plus petit étant considéré comme ayant une ou plusieurs lettre(s) "invisible(s)". Si vous le désirez, vous pourrez utiliser comme point de départ le code disponible dans le livre de référence, [à la section 7.8](#) ([la section 7.7](#) sera aussi intéressante à lire). Vous pourrez utiliser la liste de mots en français disponible sur le site de l'APP comme point de départ.

Lorsque le graphe sera créé, vous pourrez ensuite choisir un mot (par exemple, vous pourriez commencer par "bateau") et chercher s'il existe des séquences de 3, 4 ou 5 mots à partir du mot de départ.

Pour les besoins du laboratoire, nous nous en tiendrons à cette simple recherche, mais il serait possible d'utiliser ce graphe pour, par exemple, déterminer s'il existe un chemin entre un premier mot  $A$  et un deuxième mot  $B$ . Une recherche de type "recherche par parcours en largeur" (*breadth first search* en anglais) serait alors très utile. De plus, en marquant chaque noeud parcouru avec la distance minimale entre ce dernier et le noeud de départ, il serait possible de déterminer le chemin le plus court (s'il existe) entre 2 mots.

## 16 PRATIQUE PROCÉDURALE 2

### **But de l'activité**

Le but de cette activité est d'étudier les graphes, les algorithmes de tri, les arbres AVL, ainsi que la machine de Turing.



## P2.E1 Graphes

Dans ce problème, nous étudierons l'algorithme de Dijkstra, qui permet de trouver la distance entre deux noeuds. Nous utiliserons le graphe qui suit pour l'ensemble des sous-questions.

- Déterminer la distance minimale entre 2 noeuds, en partant du noeud 1. Dans un premier temps, nous supposons que les poids sur les arcs sont tous de 1. Quel est le noeud le plus éloigné du noeud 1 ?
- Refaire le même exercice, mais en utilisant les poids (2, 2 et 3) indiqués sur les arcs entre les noeuds (1, 2), (13, 14) et (18, 21). Quel est le noeud le plus éloigné du noeud 1 ?
- Refaire les calculs de la sous-question b. en démarrant du noeud 17. Quel est le noeud le plus distant du noeud 17 ?
- Qu'arrive-t-il si le graphe contient une boucle dont la somme des poids est négative ?

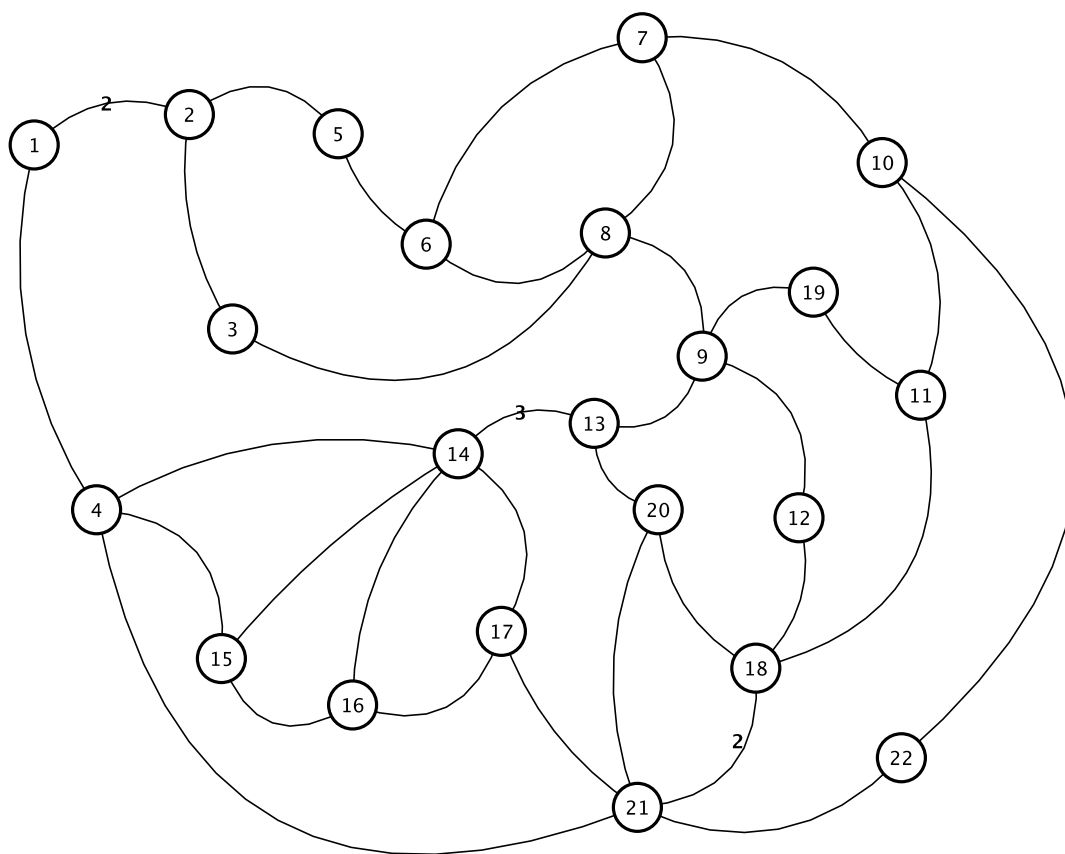


FIGURE 16.1 Graphe à analyser

### P2.E2 Tri

Dans cette question, nous étudierons les trois méthodes de tri suivantes : tri à bulles, tri fusion et tri rapide.

- a. Utiliser le tri à bulles pour ordonner (du plus petit au plus grand) les nombres suivants : [22, 34, 17, 55, 3, 18, 72, 2]
- b. Utiliser le tri fusion pour ordonner (du plus petit au plus grand) les nombres suivants : [33, 12, 4, 28, 1, 31, 75, 5]
- c. Utiliser le tri rapide (quicksort) pour ordonner (du plus petit au plus grand) les nombres suivants : [55, 21, 7, 77, 21, 15, 2, 53]

### P2.E3 Arbres AVL

Les arbres AVL permettent, lors de leur création, de garantir que l'arbre demeurera "balancé". Nous explorerons pourquoi, si la propriété est respectée, le temps de recherche sera toujours proche d'être logarithmique.

- a. À l'aide d'un arbre AVL, créez (et ordonnez) les nombres suivants : [55, 21, 7, 77, 15, 2, 53]

## P2.E4 Machine de Turing

La machine de Turing est un modèle mathématique formel du calcul, et est à la base du fonctionnement (théorique) des ordinateurs classiques modernes.

- a. Comment est définie la machine de Turing, et que représentent les différents éléments de cette définition ?
- b. Supposons la machine suivante, déterminer le calcul qu'elle effectue :
  - La machine utilise deux symboles distincts :  $\{0, 1\}$ . On suppose que le symbole 0 correspond aussi à un espace vide (rien) sur le ruban.
  - Le ruban contient une séquence de deux symboles : 1, 1, représentant le nombre 2 en base **unaire**. Toutes les autres cases du ruban sont vides (on y trouve le symbole 0).
  - La position de départ de la tête de lecture est sur le symbole 1 le plus à gauche, l'état initial étant  $S$ .
  - Les opérations possibles sont :
    - aller à droite,
    - aller à gauche,
    - écrire 0 (effacer le contenu de la case courante du ruban),
    - écrire 1,
    - ou terminer (dans l'état  $F$ ).
  - La fonction de transition se trouve à la page suivante.

— La fonction de transition est représentée par le graphe suivant :

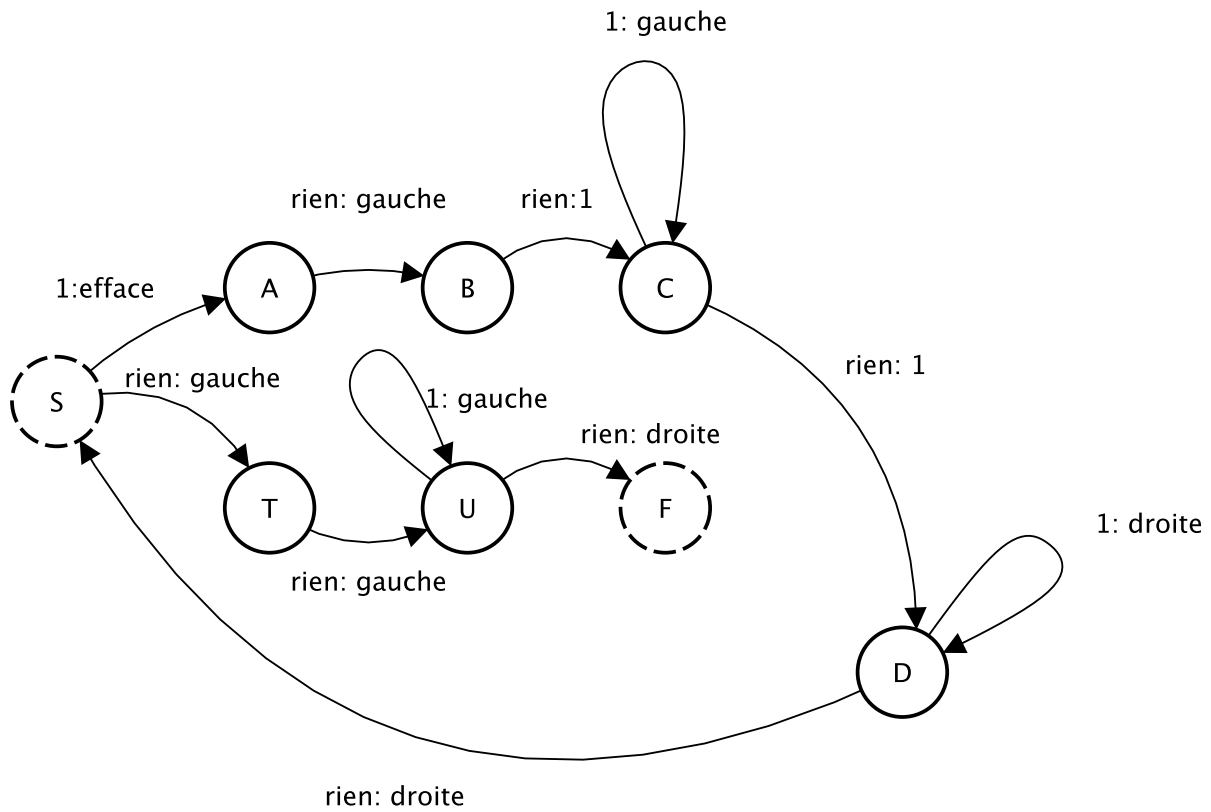


FIGURE 16.2 Fonction de transition de la machine de Turing à analyser

Tableau à compléter pour effectuer la simulation de la machine de Turing :

[illegible]

## 17 VALIDATION AU LABORATOIRE

Le but de cette activité est de valider expérimentalement la solution à la problématique que vous avez développée. Vous devez démontrer que votre application a la capacité d'analyser un ensemble de textes, et expliquer son fonctionnement :

- Structures de données utilisées et explication du choix
- Complexité des éléments importants du système
- Courte démonstration