TP 1 : Arithmétique Modulaire (et autres joyeusetés)

Rémy Cassini, Théo Pirkl, Jérôme Chételat
Noria Foukia, Eryk Schiller

Introduction

Lors des prochains travaux pratiques, vous aurez besoin d'utiliser un certain nombre d'outils mathématiques liés à l'arithmétique modulaire et à la théorie des nombres. C'est pourquoi ce TP va se concentrer sur la création d'un corpus de fonctions (qu'on appelle *librairie*) rassemblant tout ce qui vous sera nécessaire par la suite.

Ce TP se découpe en **trois parties**. Vous reviendrez donc dessus à plusieurs moments, pour le compléter. Pensez à bien commenter votre code, vous vous remercierez dans quelques semaines...

Environnement

Vous utiliserez Python ainsi que votre éditeur de texte ou IDE préféré. Nous vous fournissons dans une archive *.zip* deux fichiers nommés : `ssi_lib.py` et `ssi_lib_tests.py`.

Le premier, `ssi_lib.py`, contient la définition de toutes les fonctions que vous aurez à coder dans ce TP. Ne changez pas le nom des fonctions ! Vous n'avez qu'à supprimer les `pass` et écrire votre code à la place.

Le second, `ssi_lib_tests.py`, est une batterie de tests à utiliser pour vérifier que votre code fonctionne correctement. Ne modifiez **rien** dedans !

La commande pour lancer les tests sera différente pour chacune des trois parties de ce TP et sera fournie au début de chaque partie.

Évaluation

Ce TP durera **une séance** *(pour la partie 1, car vous reviendrez compléter sur le temps des TP2 et TP3)*. Il est **évalué** et devra donc figurer dans votre journal de laboratoire. Le code compte pour **2/3** et le journal pour **1/3**.

Votre code doit être un minimum commenté et vous devez être capable de l'expliquer à l'oral si nous avons des doutes sur son origine.

Nous vous rappelons qu'il est **formellement interdit** d'utiliser une IA générative (ChatGPT et autres) pour générer votre code.

Exercices : partie 1

Pour tous les exercices qui suivent, vous n'avez **pas le droit** d'utiliser les opérateurs modulo (`%`), xor (`^`) et shift (`<<` et `>>`) qui existent déjà en Python. Le but est de les coder vous-mêmes !

La commande de test pour cette partie est :
`python3 -m unittest ssi_lib_tests.SSILibPartOneTestCase --verbose`

Exercice 1 Modulo

Créez la fonction modulo (`mod(a, n)`). Elle retourne le résultat de l'opération a mod n. Elle ne prend que des nombres a et n **entiers naturels** ($\mathbb{N}^*$) et doit avoir le même comportement que l'opérateur `%` de Python.

Exercice 2 Addition modulaire

Créez la fonction addition modulaire (`add_mod(a, b, n)`). Elle retourne le résultat de l'opération (a + b) mod n. Vous devrez donc vous appuyer sur votre fonction modulo.

Exercice 3 Multiplication modulaire

Créez la fonction multiplication modulaire (`mul_mod(a, b, n)`). Elle retourne le résultat de l'opération (a * b) mod n. Vous vous appuyerez aussi sur votre fonction modulo.

Exercice 4 Shift cyclique à gauche

Créez la fonction bitshift qui effectuera un shift **cyclique** vers la **gauche** (`shift(a, n, length)`). Elle prend uniquement des nombres a, n et length **entiers naturels** ($\mathbb{N}^*$) et doit avoir un comportement similaire à l'opérateur `<<` de Python (excepté que votre fonction est cyclique).

Par exemple (en écrivant `a` et `b` en binaire) :
`shift(0b1100, 1, 4)` doit retourner `0b1001`.
`shift(0b0001, 2, 4)` doit retourner `0b0100`.

Exercice 5 XOR

Créez la fonction XOR (`xor(a, b)`). Elle retourne le résultat de a XOR b (bit à bit). Elle prend uniquement des nombres a et b **entiers naturels** ($\mathbb{N}^*$) et doit avoir le même comportement que l'opérateur `^` de Python.

Par exemple (en écrivant `a` en binaire) :
`xor(0b1100, 0b1010)` doit retourner `0b0110`.

Exercice 6 Chiffre de César

Pour cet exercice et le suivant, vous disposez d'une série de tests particulière. La commande pour la lancer est : `python3 -m unittest cesar_tests.CesarTestCase --verbose`

Afin de faire tout de même un peu de chiffrement dans ce TP, créez un nouveau fichier Python nommé `cesar.py`, dans le même répertoire que votre librairie, qui implémentera la fonction `cesar(message)`.

Au début de votre code, vous devrez importer votre librairie avec : `from ssi_lib import *`.

Le chiffre de César est un chiffrement symétrique très simple. Vous disposez d'un alphabet de 26 lettres (prenons uniquement les majuscules). Pour chiffrer un caractère, vous l'incrémentez de 3 (par exemple, A -> D, B -> E, W -> Z, X -> A,...). Le déchiffrement se fait en décrémentant de 3, comme vous l'avez déjà deviné. Par exemple : "HELLO" devient "KHOOR".

Appuyez vous sur la ou les fonctions que vous avez déjà codé dans les exercices 1 à 5 et que vous jugerez pertinentes.

Exercice 7 Chiffre de César avancé

Même exercice que précédemment à une différence près. Dans le même fichier `cesar.py`, créez la fonction `advanced_cesar(message, shift)`.

Cette fois-ci, vous pouvez décider de combien vous allez décaler vos caractères. Par exemple : `advanced_cesar("ABC", 10)` doit retourner `"KLM"`.

Exercices : partie 2

À commencer uniquement en même temps que le TP2 !

La commande de test pour cette partie est :
`python3 -m unittest ssi_lib_tests.SSILibPartTwoTestCase --verbose`

Exercice 8 PGCD

Créez la fonction PGCD (`gcd(a, b)`). Elle retourne le plus grand diviseur commun entre les nombres a et b. Elle ne prend que des nombres a et b **entiers naturels** ($\mathbb{N}^*$).

Vous avez déjà dû créer, en cours, une fonction PGCD selon une méthode parmi deux. Vous utiliserez **l'autre méthode** pour cet exercice-ci.

Exercice 9 Euclide étendu

Créez la fonction Euclide étendu (`extended_euclide(a, b)`). Elle retourne les coefficients de Bézout de ces deux nombres a et b. Elle ne prend que des nombres a et b **entiers naturels** ($\mathbb{N}^*$).

Attention, cette fonction ne vous retourne non pas un mais **deux** entiers. Vous pouvez écrire votre return sous la forme : `return x, y`

Exercice 10 Inverse modulaire multiplicatif

Créez la fonction inverse modulaire multiplicatif (`inv_mod(a, n)`). Elle retourne l'inverse modulaire multiplicatif du nombre a ($\in \mathbb{N}^*$) dans le groupe $\mathbb{Z}_n^*$.

Vérifiez qu'un inverse existe avant de le calculer ! Si l'inverse n'existe pas, votre fonction doit retourner **0**.

Exercices : partie 3

À commencer uniquement en même temps que le TP3 !

La commande de test pour cette partie est :
`python3 -m unittest ssi_lib_tests.SSILibPartThreeTestCase --verbose`

Exercice 11 Exponentiation modulaire binaire

Créez la fonction d'exponentiation modulaire binaire (`exp_mod(a, b, n)`) en utilisant la méthode vue en cours. Elle retourne le résultat de l'opération (a * b) mod n et doit avoir le même comportement que la fonction `pow()` de Python lorsqu'elle prend trois arguments.

Elle ne prend que des nombres a, b et n **entiers naturels** ($\mathbb{N}^*$).

Vous n'avez évidemment pas le droit d'utiliser la fonction `pow()` de Python.

Exercice 12 Premier ?

Créez la fonction qui vérifie si un nombre est premier (`is_prime(a)`). Elle retourne `True` si a est un nombre premier, `False` sinon.

Exercice 13 Premier !

Créez la fonction qui génère un nombre premier aléatoire entre 2 et b (`generate_prime(b)`). Elle retourne un nombre premier p tel que $2 \leq p \leq b$, avec $b \in \mathbb{N}^*$.

Si vous essayez avec b trop grand ($b > 10^5$), votre fonction peut prendre un peu de temps avant de vous retourner une réponse.

Exercice 14 Générateur ?

Créez la fonction qui vérifie si un nombre est générateur d'un groupe multiplicatif cyclique (`is_generator(a, n)`). Elle retourne `True` si le nombre a est générateur du groupe $\mathbb{Z}_n^*$, `False` sinon.

Exercice 15 Générateurs !

Créez la fonction qui trouve **tous** les générateurs d'un groupe multiplicatif cyclique (`find_all_generators(n)`). Elle retourne la liste de tous les générateurs du groupe $\mathbb{Z}_n^*$.

Par exemple, `find_all_generators(7)` doit retourner `[3, 5]`.