ARITHMÉTIQUE

Ce T.P. a pour but l'implémentation en langage Python, à l'aide de boucles `for` et `while`, des principaux algorithmes rencontrés en arithmétique.

1 Algorithmes de division euclidienne

Lancez IDLE et ouvrez un nouveau fichier texte. Enregistrez-le sous un nom intelligible (par exemple `division_euclidienne.py`) dans un sous-répertoire "Arithmétique" de votre répertoire de travail.

1.1 Rappels mathématiques

On rappelle que la **division euclidienne** d'un entier naturel a par un entier naturel non nul b est l'unique décomposition de a sous la forme

$$a = bq + r \text{ où } (q, r) \in \mathbb{N}^2 \text{ et } r < b.$$

On dit que l'entier q est le **quotient**, et l'entier r le **reste**, de la division euclidienne de a par b.

Les deux algorithmes usuels permettant de déterminer le quotient et le reste de la division euclidienne d'un entier naturel a par un entier naturel non nul b sont donnés ci-dessous. Le premier suit une approche "naïve", et le second est celui qui est enseigné à l'école (si si !).

Algorithme naïf

entrées : a et b entiers naturels, avec $b \neq 0$
initialiser une variable q à 0
initialiser une variable r à a
tant que r $\geqslant b$:
    remplacer q par q+1
    remplacer r par r$-b$
sorties : les valeurs finales de q et r

Algorithme scolaire

entrées : a et b entiers naturels, avec $b \neq 0$
initialiser une variable q à 0
initialiser une variable r à a
tant que r $\geqslant b$:
    trouver le plus grand entier n tq $10^n b \leqslant$ r
    trouver le plus grand chiffre c tq $10^n bc \leqslant$ r
    remplacer q par q+$10^n c$
    remplacer r par r$-10^n bc$
sorties : les valeurs finales de q et r

On peut vérifier qu'à chaque étape des deux algorithmes, la relation $a = b$q + r est vraie (c'est un *invariant de boucle*), et que les valeurs stockées dans r forment une suite d'entiers naturels strictement décroissante. Les deux algorithmes s'arrêtent donc, i.e. la condition r $\geqslant b$ devient fausse, et les valeurs finales de q et r sont alors bien le quotient et le reste de la division euclidienne de a par b.

1.2 Implémentation des algorithmes de division euclidienne en Python

1. a. Écrire une fonction Python `div1(a,b)` implémentant l'algorithme naïf décrit ci-dessus.

b. Tester cette fonction sur quelques exemples. En particulier, on doit obtenir :

```
>>> div1(9876543,21)
(470311, 12)
```

2. a. À l'aide d'une boucle `while`, trouver le plus grand entier n tel que $10^n \times 21 \leqslant 9\,876\,543$.
Adapter cette boucle pour trouver le plus grand chiffre c tel que $10^n \times 21 \times c \leqslant 9\,876\,543$.
Rq. Ayez un regard critique sur les résultats obtenus : à la main, quelles sont les valeurs de n et c attendues ? Corrigez votre code si besoin.

   b. Écrire une fonction Python `div2(a,b)` implémentant l'algorithme scolaire décrit ci-dessus.
   c. Tester cette fonction sur les exemples précédents.

3. **a.** Écrire une fonction Python `div3(a,b)` analogue aux deux fonctions précédentes, mais utilisant cette fois les commandes Python prédéfinies `/` (ou `//`) et `%`.
   b. Tester cette fonction sur les exemples précédents.

1.3 Comparaison des temps de calcul

4. Importer la commande `time` de la librairie `time` (dans le fichier `.py`) :

```
from time import time
```

Consulter l'aide sur cette commande. Que renvoie la commande `time()` ?
On voit que le temps d'exécution d'une instruction peut s'obtenir par le code suivant :

```
t=time()      # on enregistre le temps machine du moment dans la variable t
instruction   # on exécute l'instruction voulue
t=time()-t    # on calcule son temps d'exécution, et on l'enregistre dans t
```

5. **a.** Trouver (par essais successifs) un entier n pour lequel le temps de calcul de l'instruction `div1(10**n,17)` – par exemple – est de l'ordre de quelques secondes.
   b. Comparer alors aux temps de calcul correspondants avec les fonctions `div2` et `div3`.
   Interpréter les résultats obtenus.

6. **a.** Même chose qu'en **5a** avec la fonction `div2`.
   b. Comparer alors au temps de calcul correspondant avec la fonction `div3`.
   Interpréter les résultats obtenus.

2 Algorithmes autour des nombres premiers

Ouvrez un nouveau fichier texte et enregistrez-le sous un nom intelligible (par exemple `nombres_premiers.py`) dans votre répertoire "Arithmétique".

2.1 Test de primalité

On rappelle qu'un entier $n \geqslant 2$ est **premier** s'il n'est divisible par aucun entier d compris, au sens large, entre 2 et $\sqrt{n}$.

7. En utilisant ce principe, écrire une fonction Python `premier(n)` prenant en argument un entier $n \geqslant 2$ et renvoyant `True` ou `False` selon que l'entier n est premier ou non.
[Pour tester la divisibilité de l'entier n par un entier d, on pourra à profit utiliser l'une ou l'autre des commandes utilisées dans la partie précédente.]

8. **Applications.** À l'aide de la fonction `premier` :
   a. Déterminer tous les nombres premiers inférieurs à 1000. Combien y en a-t-il ?
   b. Écrire une fonction Python `premier_suivant(n)` renvoyant le plus petit nombre premier strictement supérieur à un entier n.
   Quelles sont les six prochaines années premières ?
   c. Les **nombres de Mersenne** sont les entiers $M_p = 2^p - 1$, où p est premier [1].
   Étudier la primalité des onze premiers nombres de Mersenne.
   d. **Bonus.** Déterminer la première plage de 10 entiers naturels consécutifs non premiers.

1. Les plus grands nombres premiers connus à ce jour sont tous des nombres de Mersenne, mais tous les nombres de Mersenne ne sont pas premiers (malheureusement) ...

2.2 Décomposition en produit de facteurs premiers

Tout entier naturel non nul se décompose de façon unique, à l'ordre près des facteurs, en un produit de nombres premiers (c'est le **théorème fondamental de l'arithmétique**).

9. a. Adapter la fonction Python `premier(n)` du paragraphe précédent en une fonction Python `plus_petit_diviseur(n)` prenant en argument un entier $n \geqslant 2$ et renvoyant le plus petit diviseur $d \geqslant 2$ de n.

Rq. Ce plus petit diviseur d de n est nécessairement premier. **Bonus.** Pourquoi ?

b. Tester cette fonction sur quelques exemples. En particulier, on doit obtenir :

```
>>> print plus_petit_diviseur(11111)
41
```

10. a. À l'aide de `plus_petit_diviseur`, écrire une fonction Python `decomposition(n)` prenant en argument un entier $n \geqslant 2$ et renvoyant la liste (avec répétitions éventuelles) de ses facteurs premiers.

b. Tester cette fonction sur quelques exemples. En particulier, on doit obtenir :

```
>>> decomposition(12936)
[2, 2, 2, 3, 7, 7, 11]
```

11. Applications. À l'aide des fonctions `plus_petit_diviseur` et `decomposition` :

a. Déterminer les plus petits diviseurs premiers des entiers 11, 111, 1 111, ..., 1 111 111 111.

b. Déterminer la décomposition en produit de facteurs premiers de ces entiers.

2.3 Crible d'Ératosthène

Le **crible d'Ératosthène** est une méthode de détermination de tous les nombres premiers inférieurs (ou égaux) à un entier naturel n fixé, plus efficace que la méthode naïve utilisée en **8a**.

Méthode du Crible d'Ératosthène

> Dans la liste des entiers de 2 à n, en partant de $d = 2$ et tant que $d \leqslant \sqrt{n}$:
> supprimer les multiples stricts de d,
> remplacer d par le premier entier restant dans la liste après d.

Les entiers restants dans la liste après ces suppressions sont les nombres premiers inférieurs ou égaux à n. En effet, les nombres premiers de la liste initiale n'ont (par définition) pas de diviseur entre 2 et $\sqrt{n}$ donc n'ont pas été supprimés, et les nombres non premiers de la liste initiale admettent nécessairement un diviseur premier entre 2 et $\sqrt{n}$ (pourquoi ?) donc ont été supprimés à une certaine étape de l'algorithme.

12. Écrire une fonction Python `crible(n)` renvoyant la liste des nombres premiers inférieurs ou égaux à un entier naturel n, implémentant la méthode du crible d'Ératosthène.

[Pour éviter de trop nombreuses suppressions d'éléments dans une liste (coûteuses en temps), on pourra travailler sur une liste de booléens initialisée à `[True]*(n+1)`*, et simuler la suppression d'un entier par la modification du booléen correspondant en* `False`*.]*

13. Applications.

a. À l'aide de la fonction `crible`, déterminer la liste des nombres premiers inférieurs à 1000.

b. Comparer le résultat et son temps de calcul avec ceux de la méthode naïve de **8a**.

Rq. Si les temps de calculs sont trop petits, remplacer 1000 par 10^4, ou 10^5, etc.

c. Déterminer la proportion de nombres premiers parmi les 10^n premiers entiers naturels non nuls, pour n variant de 1 à 7.

Interpréter les résultats obtenus.

3 Bonus - Algorithmes de calcul du PGCD de deux entiers

Ouvrez un nouveau fichier texte et enregistrez-le sous un nom intelligible (par exemple `pgcd.py`) dans votre répertoire “Arithmétique”.

3.1 Rappels mathématiques

Si a et b sont deux entiers naturels non nuls, on note $\text{pgcd}(a,b)$ le **Plus Grand Commun Diviseur** de a et b. Par exemple, $\text{pgcd}(9,15)=3$ puisque les diviseurs (positifs) de 9 sont 1, 3 et 9 alors que ceux de 15 sont 1, 3, 5 et 15.

On peut déterminer le PGCD de deux entiers naturels non nuls a et b si l'on connaît leur décomposition en produit de facteurs premiers, ou par l'**algorithme d'Euclide**. Pour rappel (voir votre cours de mathématiques pour des justifications) :

- **Calcul du PGCD par les décompositions de a et b en produit de facteurs premiers.**
  Si les décompositions de a et b en produit de facteurs premiers sont respectivement :
  $$a=\prod_{p\in\mathbb{P}} p^{\alpha_p}=2^{\alpha_2}\times 3^{\alpha_3}\times 5^{\alpha_5}\times\cdots \text{ et } b=\prod_{p\in\mathbb{P}} p^{\beta_p}=2^{\beta_2}\times 3^{\beta_3}\times 5^{\beta_5}\times\cdots$$
  alors $\text{pgcd}(a,b)=\prod_{p\in\mathbb{P}} p^{\min\{\alpha_p,\beta_p\}}=2^{\min\{\alpha_2,\beta_2\}}\times 3^{\min\{\alpha_3,\beta_3\}}\times 5^{\min\{\alpha_5,\beta_5\}}\times\cdots$
- **Algorithme d'Euclide pour le calcul du PGCD de a et b.**
  On initialise deux variables x et y à a et à b, et tant que y est non nul :
  ⋆ on effectue la division euclidienne de x par y,
  ⋆ on remplace x par y et y par le reste obtenu.
  L'algorithme s'arrête et le dernier reste non nul calculé est le PGCD des entiers a et b.

3.2 Implémentation des algorithmes de calcul du PGCD en Python

14. a. Écrire une fonction Python `pgcd1(a,b)` renvoyant le PGCD de deux entiers naturels non nuls a et b, calculé par l'algorithme d'Euclide.

b. Tester cette fonction sur quelques exemples.

15. a. Écrire une fonction Python `pgcd2(a,b)` renvoyant le PGCD de deux entiers naturels non nuls a et b, calculé par les décompositions en produit de facteurs premiers.
[On pourra utiliser les fonctions `decomposition` *de* **10a**, `set` *qui transforme les listes en ensembles (et supprime ainsi les répétitions),* `min`, *et la méthode de listes* `count`.*]*

b. Tester cette fonction sur les exemples précédents.

16. Comparer les temps de calcul nécessaires aux fonctions `pgcd1` et `pgcd2` pour calculer, par exemple, le PGCD des entiers j et k pour j et k variant dans $[\![1;500]\!]$.
Interpréter les résultats obtenus.

17. Application. On rappelle que deux entiers naturels non nuls sont dits **premiers entre eux** lorsque leur PGCD est égal à 1.

a. Implémenter en Python l'**indicatrice d'Euler**, i.e. la fonction φ qui a un entier $n\in\mathbb{N}^*$ associe le nombre d'entiers $d\in[\![1;n]\!]$ tels que d et n sont premiers entre eux.

b. Tester cette fonction sur quelques exemples. En particulier, on doit obtenir :

```
>>> phi(98), phi(99), phi(100)
42, 60, 40
```

c. Vérifier sur tous les entiers a et $b\in[\![1;100]\!]$ qu'on a la propriété suivante (on dit que l'indicatrice d'Euler φ est **multiplicative**) :

$$a \text{ et } b \text{ premiers entre eux} \Longrightarrow \varphi(ab)=\varphi(a)\varphi(b).$$