TP Info 4 | Arithmétique et chaînes de caractères

Dans cette séance de TP, on met en œuvre quelques algorithmes classiques d'arithmétique (exponentiations naïve et rapide, division euclidienne naïve et améliorée, algorithme d'Euclide et algorithme d'Euclide étendu) en prouvant leur correction et en calculant leur complexité. La séance se termine par deux exercices sur la manipulation des chaînes de caractères.

4 Arithmétique et chaînes de caractères 1
1 Ce qu'il faut savoir 2
1.1 La fonction `time` 2
1.2 Rappels sur les chaînes de carctères 2
2 Exercices 3
2.1 Arithmétique des entiers 3
2.2 Chaînes de caractères 6

1. Ce qu'il faut savoir

1.1. La fonction `time`

Évaluation d'un temps de calcul

Pour chronométrer le temps pris par l'ordinateur pour effectuer un calcul, on pourra utiliser la fonction `time` de la bibliothèque `time`. Un appel à `time.time()` fournit un *temps absolu* en secondes. Par différence, on obtient donc une durée entre deux balises. Dans l'exemple suivant, `t1-t0` est la durée du calcul :

```
t0=time.time()
...
[mon calcul]
...
t1=time.time()
```

1.2. Rappels sur les chaînes de caractères

Le type `str` (string)

▷ Les chaînes peuvent être délimitées par des simples quotes (apostrophes) ou doubles quotes (guillemets) : `c='bou'` ou `c="bou"`.

▷ La commande `len` permet de calculer la longueur d'une chaîne de caractères.

▷ Les chaînes de caractères ne sont pas modifiables, mais seulement accessibles en lecture.

▷ La fonction `in` permet de tester l'appartenance d'un élément à une chaîne.

▷ Les chaînes peuvent être concaténées (accolées) avec l'opérateur + et répétées avec ∗.

▷ <4-> Exemples :

```
>>> mot='Help'+' '+'me'
>>> print(mot)
Help me
>>> print(mot*3)
Help meHelp meHelp me
>>> print((mot+' ')*3)
Help me Help me Help me
```

▷ Les chaînes peuvent être décomposées (indexées), comme en `C`. Le premier caractère d'une chaîne est en position (index) 0. L'utilisation du *slicing* est identique au cas des listes :

```
>>> mot='help me'
>>> print(mot[4])
' '
>>> print(mot[5])
m
>>> mot[0:2]
'he'
>>> mot[2:5]
'lp '
>>> mot[:2]
'he'
>>> mot[2:]
'lp me'
```

▷ En Python, les chaînes de caractères sont des itérables, c'est-à-dire que l'on peut les utiliser dans une boucle `for`.

2. Exercices

Les difficultés sont échelonnées de la manière suivante : aucune, ♪, ♪♪, ♪♪♪ et ♪♪♪♪. Certains énoncés sont tirés des annales des concours (oral et écrit) ; leur provenance est le plus souvent précisée. Les exercices notés ♪♪♪ et ♪♪♪♪ sont particulièrement délicats.

2.1. Arithmétique des entiers

1 . [***Algorithmes de division euclidienne dans*** $\mathbb{N}$ ♪]
On se propose d'exposer, prouver puis comparer deux algorithmes de division euclidienne dans $\mathbb{N}$.

a) ***Algorithme naïf.***

   i) Donner un algorithme naïf de division euclidienne reposant sur des soustractions successives. Prouver la terminaison et la correction de cet algorithme en déterminant un invariant de boucle.

   ii) Écrire une fonction `divisionEuclidienne(a,b)` renvoyant le couple (q,r) en suivant l'algorithme naïf.

   iii) Quelle est la complexité de cet algorithme ?

b) ***Algorithme binaire.***
On propose l'amélioration suivante :

Algorithme 1 : Algorithme binaire de division euclidienne

```
Données : Deux entiers naturels a et b tels que b ≠ 0;
Résultat : Le couple quotient-reste de la division euclidienne de a par b;
Initialisation : q ← 0, w ← b, r ← a;
tant que w ⩽ r faire
    w ← 2×w;
fin
tant que w ≠ b faire
    q ← 2×q;
    w ← quotient de w par 2;
    si w ⩽ r alors
        r ← r-w;
        q ← q+1;
    fin
fin
Renvoyer (q,r);
```

   i) Prouver la terminaison de cet algorithme.

   ii) Établir que la propriété suivante est un invariant de la seconde boucle :

$$\begin{cases} qw + r = a \\ 0 \leqslant r < w \end{cases}$$

   En déduire la correction de l'algorithme.

   iii) Écrire une fonction `divisionEuclidienneBinaire(a,b)` renvoyant le couple (q,r) en suivant cet algorithme.

   iv) Quelle est la complexité de cet algorithme ?

2 . [***Deux algorithmes d'exponentiation*** ♪♪]
On expose dans cet exercice deux algorithmes classiques de calcul de a^n où a et n sont deux entiers naturels non nuls.

a) ***Algorithme naïf.***

i) Écrire un algorithme itératif renvoyant a^n. Prouver la terminaison puis la correction de cet algorithme.

ii) Quelle est la complexité de cet algorithme ?

iii) Écrire une fonction `e(a,n)` renvoyant a^n par cet algorithme.

iv) En utilisant la fonction `time`, comparer la vitesse d'exécution sur machine de `e(23,n)` et de `23**n` pour les valeurs suivantes $n \in \{10, 100, 1000, 10000, 100000\}$.

b) ***Algorithme d'exponentiation rapide.***
On considère l'algorithme suivant :

Algorithme 2 : Exponentiation rapide

Données : Deux entiers a et n de $\mathbb{N}^*$;
Résultat : a^n;
Initialisation : R ← 1, A ← a, N ← n;
tant que $N > 0$ **faire**
  si *N est impair* **alors**
    R ← R × A;
  A ← A × A;
  N ← quotient dans la division euclidienne de N par 2;
Renvoyer R

i) Montrer la terminaison de l'algorithme.

ii) On note $n = (c_{m-1} \cdots c_0)$. En examinant les valeurs successives des variables A, R et N, déterminer un invariant de boucle.

iii) En déduire la correction de cet algorithme.

iv) Calculer la complexité de cet algorithme.

v) Écrire une fonction `er(a,n)` effectuant cet algorithme.

vi) En utilisant la fonction `time`, comparer la vitesse d'exécution sur machine de `er(23,n)` et de `23**n` pour les valeurs suivantes $n \in \{10, 100, 1000, 10000, 100000\}$.

3 . [***Algorithme d'Euclide du calcul du PGCD, application au PPCM*** ♪♪]

a) ***Terminaison et correction de l'algorithme d'Euclide.***

i) Rappeler l'algorithme d'Euclide du calcul du PGCD deux entiers naturels a et b vérifiant la condition $(a, b) \neq (0, 0)$. On donnera un pseudo-code. La terminaison et la correction de cet algorithme ont été prouvées dans le cours d'arithmétique.

ii) Écrire une fonction `pgcd(a,b)` renvoyant le PGCD de deux entiers naturels a et b selon l'algorithme d'Euclide.

iii) Écrire une fonction `ppcm(a,b)` renvoyant le PPCM de deux entiers naturels a et b.

iv) Déterminer le PGCD et le PPCM des entiers $a = 1113245$ et $b = 5478221$ au moyen de cette fonction.

b) ***Nombre d'étapes de l'algorithme d'Euclide.***
Soient a et b deux entiers naturels tels que $a \geqslant b > 0$. On note (f_n) la suite de Fibonacci :

$$\begin{cases} f_0 = 0, \ f_1 = 1 \\ \forall n \in \mathbb{N}, \ f_{n+2} = f_{n+1} + f_n \end{cases}$$

i) Rappeler l'expression de f_n en fonction du nombre d'or $\phi = \dfrac{\sqrt{5}+1}{2}$.

ii) Établir le *théorème de Lamé* : si le calcul de $d = a \wedge b$ avec $a \geqslant b > 0$ par l'algorithme d'Euclide demande n étapes, alors :

$$a \geqslant d\mathrm{F}_{n+2} \quad \text{et} \quad b \geqslant d\mathrm{F}_{n+1}$$

iii) Vérifier qu'en particulier, si $a = \mathrm{F}_{n+2}$ et $b = \mathrm{F}_{n+1}$, il y a exactement n étapes de calcul.

iv) En déduire que le nombre d'étapes de l'algorithme d'Euclide est logarithmique en b.

4 . [*Crible d'Eratosthène* ♪♪]
Au III$^{\text{e}}$ siècle avant Jesus-Christ, Ératosthène, mathématicien, astronome et philosophe invente une méthode efficace pour énumérer tous les nombres premiers inférieurs à un entier donné n. Cette méthode est connue sous le nom de crible d'Ératosthène, que nous allons décrire dans un langage moderne :

- ▷ On initialise un tableau `t` de n booléens à `True`. En k-ième case, le `True` s'interprète comme : *jusqu'ici, et jusqu'à preuve du contraire, k semble être premier.*
- ▷ Tous les multiples stricts de 2 ne sont pas premiers : on passe donc à False tous les `t[2i]` pour $4 \leqslant 2i \leqslant n$.
- ▷ Tous les multiples stricts de 3 ne sont pas premiers : on passe donc à `False` tous les `t[3i]` pour $9 \leqslant 3i \leqslant n$.
- ▷ On constate que 4 n'est pas premier (`t[4]` a été mis à `False` lors du premier passage) : inutile de *rayer* les multiples de 4, qui l'ont déjà été.
- ▷ Tous les multiples stricts de 5 ne sont pas premier : on passe donc à `False` tous les `t[5i]` pour $25 \leqslant 5i \leqslant n$.
- ▷ Ainsi de suite.

À la fin de l'algorithme, la k-ème case de `t` contient `True` si et seulement si k est premier.

a) Proposer un algorithme sous la forme d'un pseudo-code pour cette méthode de crible. On s'attachera à respecter l'esprit de cette technique, et en particulier à éviter divisions et racines carrées.

b) Écrire la fonction `cribleEratosthene(n)` qui renvoie la liste des nombres premiers compris entre 1 et n.

5 . [*Algorithme d'Euclide étendu* ♪♪]
Une variante de l'algorithme d'Euclide permet, au-delà du calcul du pgcd d de a et b, d'obtenir les coefficients (u, v) dans $\mathbb{Z}$ d'une relation de Bezout entre a et b :

$$ua + vb = d$$

On suppose $a \neq 0$ et $b \neq 0$. On note $r_0 = a$, $r_1 = b$, ..., $r_{n_0} = a \wedge b$ (resp. $q_0, \dots, q_{n_0}$) les restes (resp. quotients) successifs dans l'algorithme d'Euclide.

a) Prouver que, pour tout $0 \leqslant n \leqslant n_0$, il existe $(u_n, v_n) \in \mathbb{Z}^2$ tel que

$$r_n = u_n a + v_n b$$

b) En déduire, sous la forme d'un pseudo-code, une variante de l'algorithme d'Euclide (*algorithme d'Euclide étendu*) renvoyant $a \wedge b$ et un couple d'entiers relatifs (u, v) tel que $a \wedge b = ua + vb$.

c) Écrire sous Python une fonction `euclideEtendu(a,b)` renvoyant la liste `[a∧b,u,v]` selon l'algorithme d'Euclide étendu.

2.2. Chaînes de caractères

6. [*Bégaiement*]
Écrire une fonction `begaie(L)` qui prend en argument une liste de chaînes et qui affiche chaque caractère de chaque chaîne deux fois de suite. Par exemple, `begaie(['tout','heure'])` doit renvoyer `['ttoouutt','hheeuurree']`.

7. [*Le code de César*]
Le code de César est le plus rudimentaire que l'on puisse imaginer. Il a été utilisé par Jules César pour certaines de ses correspondances. Le principe est de décaler *circulairement* les lettres de l'alphabet vers la droite d'un certain nombre d de lettres appartenant à $[\![0,25]\!]$, appelé clé du code. Par exemple, en choisissant $d = 1$, le caractère A se transforme en B, le B en C, ..., et le Z en A. Le message "LOUIS LE GRAND" est donc codé par "MPVJT MF HSBOE". Afin d'alléger le traitement des messages, ceux-ci, qu'il soient codés ou décodés, sont enregistrés sous la forme de chaînes de caractères ne comportant que des majuscules, aucun caractère accentué, aucune apostrophe, ni aucun signe de ponctuation; de simples espaces séparent les mots du message.

a) *Codage et décodage d'un texte connaissant la clé.*

   i) Ecrire une procédure `codageCesar(M,d)` prenant en argument une chaîne de caractères `M` et un décalage d, et qui retourne le message `M` décalé de d lettres. On pourra utiliser la méthode `index` de la classe `str` : si `c` est une chaîne de caractères et `lettre` une lettre de l'alphabet, `c.index('l')` renvoie la position de la première apparition de la lettre `lettre` dans `c` (attention, la numérotation commence à 0).

   ii) Que donne le codage de "ONLY GOD FORGIVES" pour un décalage de 12 lettres ?

   iii) Ecrire une procédure `decodageCesarAvecCle(M,d)` prenant les mêmes arguments et mais qui réalise le décodage d'un message `M`.

b) *« Casser » un code de César.* Pour décoder un message, il faut connaître la *clé d* employée. L'approche la plus couramment employée est de déterminer la fréquence d'apparition de chaque lettre de l'alphabet dans le message crypté. La lettre **E** étant la plus fréquente dans un texte « *suffisamment long* » en français, cette approche est statistiquement efficace.

   i) Ecrire une procédure `frequences(M)` qui prend en argument un message codé sous la forme d'une chaîne de caractères `M` et qui retourne la liste de taille 26 dont le terme d'indice $0 \leqslant i \leqslant 25$ contient le nombre d'occurrences de la $(i+1)$-ième lettre de l'alphabet dans `M`.

   ii) Ecrire une procédure `cle(M)` qui prend en argument un message codé `M` et qui retourne la valeur de la clé utilisée.

   iii) Ecrire une procédure `decodageCesar(M)` qui prend en argument un message codé `M` et qui retourne le message décodé.

   iv) Décoder le message suivant :

SL YVTHU ZWSLUKLBYZ LA TPZLYLZ KLZ
JVBYAPZHULZ LZA SH ZBPAL KLZ PSSBZPVUZ WLYKBLZ