T.P. 8 : algorithmes en arithmétique

1 Tests naifs de primalité

a) Ecrire une fonction `estPremier` qui prend en argument un entier `n` et renvoie un booléen qui vaut `True` ssi `n` est premier, en testant la divisibilité par tous les nombres entiers jusqu'à $\lfloor\sqrt{\texttt{n}}\rfloor$.

b) *Ce qui suit est un extrait du sujet d'informatique commune du Concours Commun Mines-Ponts 2019, avec une bêtise de l'énoncée corrigée*

Le crible d'Eratosthène est un algorithme qui permet de déterminer la liste des nombres premiers appartenant à l'intervalle $[\![0, N]\!]$. Son pseudo-code s'écrit comme suit :

Données : N, entier supérieur ou égal à 1.
Résultat : `liste_bool`, liste de $N+1$ booléens.
Début : `liste_bool` ← liste de $N+1$ booléens initialisés à Vrai
    Marquer comme Faux les éléments d'indices 0 et 1 de `liste_bool` ;
    pour entier i ← 2 à $\lfloor\sqrt{N}\rfloor$ **faire**
        si i n'est pas marqué comme Faux dans `liste_bool` **alors**
            Marquer comme Faux tous les multiples de i différents de i dans `liste_bool` ;
        fin si
    fin pour
    retourner `liste_bool`
fin

A la fin de l'exécution, si un élément de `liste_bool` vaut Vrai alors le nombre codé par l'indice considéré est premier. Par exemple pour N=4 une implémentation Python du crible renvoie [False False True True False].

Ecrire une fonction `erato0(N)` qui réalise cette implémentation Python et renvoie `list_bool`, puis en déduire une fonction `Erato(N)` qui renvoie la liste des nombres premiers inférieurs ou égaux à N.

2 Le test de Fermat

2.1 Exponentiation modulaire

Pour expérimenter le test de Fermat, nous commençons par modifier l'algorithme d'exponentiation rapide pour le faire travailler dans les anneaux de congruences $(\mathbb{Z}/n\mathbb{Z}, +, \times)$.

Ecrire une fonction `exp_mod(x,N,m)` qui applique l'algorithme d'exponentiaton rapide pour calculer x^N, en réduisant *à chaque étape* le résultat modulo m, autrement dit, en remplaçant le résultat *à chaque étape* par son reste dans la division euclidienne par m.

Remarque : La fonction PYTHON `pow` fait exactement cela si on lui rentre l'argument facultatif `m`. (A condition de ne pas prendre celle du module `math` !)

2.2 Le test de Fermat et son efficacité

Le petit théorème de Fermat dit que si n est un nombre premier alors pour tout $a \in \mathbb{N}$, $a^n \equiv a\,[n]$. Ce théorème fournit donc une **C.N.** pour qu'un nombre soit premier. Si on a un a tel que $a^n \not\equiv a\,[n]$, on est sûr que n n'est pas premier. On dira que n ne *passe pas* le test de Fermat pour la valeur a en question (appelé aussi *témoin de Fermat*) et donc n'est pas premier.

a) Ecrire une fonction `test_Fermat` qui prend en argument un nombre `n` et un *argument facultatif* `a`, qui sinon admet `a=2` comme valeur par défaut, et qui teste si `n` « passe » le test de Fermat pour a, en renvoyant un booléen.

b) A l'aide de la fonction `Erato` et de la fonction précédente :

(i) Faire la liste des nombres inférieurs à 10000 qui ne sont pas premiers mais qui passent le test de Fermat pour $a = 2$.

On en trouve 22*, le premier étant* 341 *et le dernier* 8911.

(ii) Faire la liste des nombres inférieurs à 10000 qui ne sont pas premiers mais qui passent le test de Fermat pour $a = 3$.

On en trouve 30 *le premier étant* 6*, le dernier* 8911.

(iii) Finalement faire la liste des nombres inférieurs à 10000 qui ne sont pas premiers mais qui passent le test de Fermat pour $a = 2$ et pour $a = 3$.

On en trouve 8*.*

c) Comparer la rapidité de la réponse des deux fonctions `test_Fermat` et `estPremier` sur des grands nombres : fabriquer pour cela une liste de grands nombres impairs consécutifs (par exemple des nombres 100 chiffres) et tester la fonction `estPremier` sur ces nombres : lorsqu'elle « rame » sur un nombre, essayez le test de Fermat sur ce nombre.

Par exemple en commençant à `N=10**(100)+1` *j'ai trouvé* `N+36` *qui résistait à* `estPremier` *mais pas à Fermat avec le témoin* 2*..*

d) Un nombre de Carmichael est un entier $n \geq 2$ *non premier* qui vérifie la propriété que pour tout entier $a \in \mathbb{Z}$: $a^n \equiv a\,[n]$.

Ecrire un programme qui donne la liste de tous les nombres de Carmichael inférieurs à 10000.

Réponse : on trouve seulement : `[561, 1105, 1729, 2465, 2821, 6601, 8911]`.

3 Autour de l'algorithme d'Euclide

3.1 L'algorithme d'Euclide usuel :

Pour travailler le cours !

Ecrire une fonction `Euclide` qui reçoit deux arguments `a` et `b` supposés de type *integer* et renvoie le pgcd de `a` et `b` calculé par l'algorithme d'Euclide.

La solution est dans le cours, essayez de la retrouver, puis regardez le cours !

3.2 L'algorithme d'Euclide étendu

On considère deux entiers positifs a et b avec $a > b > 0$ et la suite (r_k) définie par $r_0 = a$, $r_1 = b$ et tant que $r_k \neq 0$, r_{k+1} est le reste de la division euclidienne de r_{k-1} par r_k. On notera $r_{k-1} = q_k r_k + r_{k+1}$ cette égalité de division euclidienne.

On note N l'indice du dernier reste non nul, $r_N = \text{pgcd}(a, b)$.

a) **Maths (peut être sauté en TP d'info)** Vérifier qu'à chaque étape : $r_k = u_k a + v_k b$ où les suites (u_k) et (v_k) vérifient la relation de récurrence d'ordre deux suivante : $u_{k+1} = u_{k-1} - u_k q_k$, et $v_{k+1} = v_{k-1} - v_k q_k$.

b) **Info :** écrire une fonction PYTHON qui prend deux arguments `a,b` (supposés entiers non nuls) et qui renvoie `d,u,v` où `d` est le pgcd de `a` et `b` et `u,v` sont tels que `au +bv=d`.

Pour cela on calculera les suites (r_k), (q_k), (u_k) et (v_k) récurrentes d'ordre deux, (sauf q_k) comme suit :

- **Initialisation (double sauf pour q_k) :** $\begin{cases} r_0 = a,\ q_0 = 0,\ u_0 = 1,\ v_0 = 0, \\ r_1 = b, u_1 = 0, v_1 = 1. \end{cases}$
- **Formule de récurrence :** pour tout $k \geq 1$, *tant que* $r_k \neq 0$

  i) (q_k, r_{k+1}) est le couple quotient-reste de la division euclidienne de r_{k-1} par r_k,

  ii) $u_{k+1} = u_{k-1} - u_k q_k$, et $v_{k+1} = v_{k-1} - v_k q_k$.

Indication : une illustration concrète.

$$
\begin{aligned}
97 &= 5\times 18 + 7, & 7 &= 1\times 97 - 5\times 18,\\
18 &= 2\times 7 + 4, & 4 &= -2\times 97 + 11\times 18,\\
7 &= 1\times 4 + 3, & 3 &= 3\times 97 - 16\times 18,\\
4 &= 1\times 3 + 1, & 1 &= -5\times 97 + 27\times 18.\\
3 &= 3\times 1 + 0.
\end{aligned}
$$

4 Bonus pour les plus intéressés, retour aux tests de primalités : algorithme de Miller Rabin

On veut savoir si un nombre n, impair, est premier. On considère le nombre pair $n-1$ qu'on écrit sous la forme $n-1 = 2^k.m$ avec m impair.

On considère le témoin de Fermat $a = 2$, (on pourra ensuite changer de témoin) et la suite des puissances de a de la forme a^m, a^{2m}, a^{4m} jusqu'à $a^{2^k.m}$ (notons qu'on passe d'un terme de la suite à son suivant par élévation au carré) qu'on va considérer modulo n.

Remarque : La suite ne peut pas être réduite à un seul terme car $k \neq 0$ sinon $n-1$ serait impair.

- Considérons le premier terme a^m : s'il est congru à 1 ou à -1 modulo n, alors $a^{2^k.m}$ sera congru à 1 modulo n et on dit que n est probablement premier puisqu'il passe le test de Fermat sous la forme $a^{n-1} \equiv 1\ [n]$.
- Sinon, on passe au terme suivant qui est son carré :
  - si $a^{2m} \equiv 1\ [n]$ alors que $a^m \not\equiv 1\,[n]$ et $a^m \not\equiv -1\,[n]$ alors on est sûr que n n'est *pas* premier. En effet, dans ce cas, $x = a^m$ est une solution de l'équation $x^2 \equiv 1\,[n]$ qui n'est ni -1 ni 1 modulo n : or pour un nombre n premier ce sont les deux seules solutions.
    Donc l'algorithme renvoie « n n'est pas premier ».
  - si $a^{2m} \equiv -1\ [n]$ et qu'on n'est pas au dernier terme de la suite on aura encore $a^{4m} \equiv 1\,[n]$ et on déclare n probablement premier.
- sinon on continue avec a^{4m}, etc...
- à l'avant dernier terme de la suite : $a^{2^{k-1}m}$, comme précédemment :
  - si $a^{2^{k-1}m} \equiv 1\ [n]$ on est sûr que n non premier.
  - si $a^{2^{k-1}m} \equiv -1\ [n]$ on déclare n probablement premier.
  - si $a^{2^{k-1}m}$ n'est congru ni à 1 ni à -1 modulo n, on est sûr que n n'est pas premier. En effet :
    • ou bien on aura son carré a^{n-1} qui vérifie $a^{n-1} \not\equiv 1\,[n]$ et donc il ne passe pas le test de Fermat
    • ou bien on aura son carré qui est congru à 1 mais donc 1 aura une racine carrée dans $\mathbb{Z}/n\mathbb{Z}$ différente de 1 et -1 et donc n n'est pas premier dans tous les cas.

Travail à faire : implémenter l'algorithme précédent en une fonction `MillerRabin(n)` qui renvoie `True` si `n` est probablement premier pour $a = 2$ et `False` sinon.

Les faux nombres premiers donnés par Miller-Rabin pour $a = 2$ inférieurs à 10000 sont 2047 3277 4033 4681 8321

En comparaison avec seulement le test de Fermat pour $a = 2$, on a : 341 561 645 1105 1387 1729 1905 2047 2465 2701 2821 3277 4033 4369 4371 4681 5461 6601 7957 8321 8481 8911.

MIEUX : si on teste avec a=2 et a=3, alors *tous* les nombres inférieurs à 10000 donnés premiers par Miller-Rabin sont *vraiment* premiers.

Remarque importante : Miller Rabin ne cherche pas un diviseur de n : s'il répond que n n'est pas premier, reste encore éventuellement à en chercher un diviseur. Ainsi on doit coupler Miller-Rabin avec un autre algorithme.

Complément au TP 8 : DS 2 2018/2019

Entrée chinoise

Ecrire une fonction python `f` qui prend en argument deux entiers m, n supposés premiers entre eux, deux entiers a et b quelconques et renvoie l'unique $x \in [\![0, mn-1]\!]$ vérifiant les deux conditions $\begin{cases} x \equiv a\ [m], \\ x \equiv b\ [n]. \end{cases}$ Cette fonction devra faire un nombre de calculs qui est un $O(\min(m,n))$, ce qui signifie concrètement que le nombre de tours de boucle sera majoré par $\min(m,n)$ (en faisant un nombre fixe d'opérations par tour).

Problème : Algorithmes de factorisation d'entiers

1) L'algorithme des divisions successives :

a) Justifier que si $m \in \mathbb{N}_{\geq 2}$ n'est pas premier alors m admet un diviseur non trivial inférieur ou égal à $\sqrt{m}$.

b) Ecrire une fonction `Testpremier` qui prend en argument un entier naturel m, parcourt tous les entiers de 2 jusqu'à $\lfloor \sqrt{m} \rfloor$ au plus, et renvoie `False` dès que l'un de ces entiers divise m, et `True` si m est premier.

c) Modifier le programme précédent pour fabriquer une fonction `Decompose` qui prend en argument un entier m et renvoie toujours `True` si m est premier et , si m n'est pas premier, renvoie un couple (d_1, d_2) tel que $m = d_1.d_2$ avec $1 < d_1 < m$ et $1 < d_2 < m$ (décomposition non triviale).

d) Nombre de chiffres dans l'écriture décimale d'un nombre :

i) Si $x \in \mathbb{R}^{+*}$ s'écrit avec N chiffres en base 10, exprimer N à l'aide de $\log_{10}(x)$.

ii) Montrer que si $x \in \mathbb{R}^{+}$ alors $\lfloor \frac{x}{2} \rfloor = \lfloor \frac{\lfloor x \rfloor}{2} \rfloor$.

iii) Si m est un nombre ayant N chiffres en base dix, que dire du nombre de chiffres de $\sqrt{m}$?

e) Parmi tous les nombres m ayant N chiffres avec N fixé, pour quels nombres non premiers l'algorithme `Testpremier` sera-t-il le plus long pour répondre ?

2) La méthode de factorisation de Fermat

On considère ici une autre méthode de factorisation qui peut entraîner beaucoup de calculs (encore plus qu'au 1) !) mais qui est peu coûteuse précisément dans le cas où celle du 1) est très coûteuse. Soit $m \in \mathbb{N}$ le nombre qu'on cherche à factoriser.

Idée de base : On considère l'équation (F) : $x^2 - y^2 = m$ d'inconnue $(x, y) \in \mathbb{N}^2$.

Elle s'écrit encore $(x-y)(x+y) = m$. Donc si on trouve une solution $(x, y) \in \mathbb{N}^2$ telle que $x - y \neq 1$, on saura que m n'est pas premier et on aura une factorisation non-triviale de m.

Géométriquement l'ensemble des solutions de (F) est l'ensemble des points à coordonnées entières sur $\mathcal{H} = \{(x, y) \in (\mathbb{R}^+)^2,\ x^2 - y^2 = m\}$ qui est donc l'intersection d'une hyperbole avec le premier quadrant.

2.1) La version la plus naïve de l'algorithme :

L'intersection de $\mathcal{H}$ avec l'axe des abscisse est le point $(\sqrt{m}, 0)$. Si ce point est à coordonnées entières i.e. si $\sqrt{m} \in \mathbb{N}$, on a gagné : $m = (\sqrt{m})^2$ avec $\sqrt{m} \in \mathbb{N}$ Sinon on va tester pour chaque $x_k = \lceil \sqrt{m} \rceil + k$ pour $k = 1, 2, \ldots$ si le point d'abscisse x_k sur $\mathcal{H}$ a une ordonnée entière, autrement dit si $x_k^2 - m$ est un carré d'entier.

N.B. Pour un réel x, $\lceil x \rceil$ désigne la « partie entière supérieure » c'est à dire le plus petit entier supérieur ou égal à x. Si x n'est pas un entier, $\lceil x \rceil = \lfloor x \rfloor + 1$. Si x est entier $\lceil x \rceil = x$. Dans le module `math`, elle s'appelle `ceil`.

a) Comment tester si un entier est un carré d'entier ? La fonction `sqrt` du module `math` (ou le $**(1/2)$) renvoie un flottant, donc on ne peut pas complètement lui faire confiance. Néanmoins le calcul fait par `sqrt` a l'avantage d'être très rapide (il repose, nous le verrons plus tard, sur une méthode

d'analyse qui converge très vite), donc on va quand même l'utiliser mais après on doit vérifier son résultat.. comment ? Ecrire ainsi une fonction `TestCarre` rapide et fiable qui prend un entier n en argument et renvoie `True` ou `False` suivant que n est un carré ou pas.

b) **Remarque (inutile pour la suite)** Fermat connaissait une C.N. sur les deux derniers chiffres d'un carré. Il savait que les deux derniers chiffres de l'écriture décimale d'un carré sont dans une liste L qui commence par $00, 01, 04, 09, 16, 21, ...$ Cela lui permettait d'éliminer très vite des nombres dans l'algorithme précédent qu'il effectuait, lui, sans ordinateur. Ecrire un programme Python qui permet d'obtenir la liste L.

c) Ecrire une fonction `DecompFermat(m)` qui prend en argument un entier m, met en oeuvre l'algorithme de Fermat décrit ci-dessus et s'arrête au premier k tel que $x_k^2 - m$ est un carré qu'on note y_k^2. La fonction renvoie alors le couple $(x_k - y_k, x_k + y_k)$.

d) Montrer que si m est un entier impair la fonction `DecompFermat` s'arrête toujours. Que renvoie-t-elle si m est premier ? Que renvoie-t-elle pour $m = 3^2 \times 7$ (il s'agit moins ici de mettre en oeuvre l'algorithme que de comprendre quelle décomposition de m est renvoyée).

2.2) Une petite amélioration pour ne pas avoir de carrés à calculer

a) Avec les notations du 2.1. trouver une relation simple entre $x_{k+1}^2 - m$ et $x_k^2 - m$.

b) En déduire une fonction `DecompFermat2` où, une fois qu'on a calculé x_0^2, on n'a plus besoin de calculer de carré (sauf pour `TestCarre`).

2.3) Complexité de l'algorithme

On suppose toujours que m est un entier impair. La méthode de Fermat peut donner des calculs très longs, mais bien sûr elle est intéressante précisément dans le cas où le 1) était très long, à savoir si le plus petit facteur premier de m est proche de $\sqrt{m}$. Bien sûr si m est un carré parfait, l'algorithme termine immédiatement.

Sinon, notons $m = pq$ avec $1 < p < \sqrt{m}$ et $\sqrt{m} < q < m$, tels que p soit le plus grand diviseur de m inférieur à $\sqrt{m}$ (et donc q le plus petit diviseur de m supérieur à $\sqrt{m}$) et $\mu = (p+q)/2$.

Donner alors une formule donnant exactement le nombre de tours de boucle en fonction de l'écart entre μ et $\lceil \sqrt{m} \rceil$.

3) L'algorithme $p-1$ de Pollard

L'algorithme naïf du 1) est efficace seulement si un nombre m a au moins un « petit » facteur premier, l'algorithme de l'hyperbole du 2) au contraire si m a des facteurs proches de $\sqrt{m}$ mais ces deux algorithmes sont trop lents pour de nombreux grands nombres m dont le plus petit facteur premier est de taille « intermédiaire ».

Le test de Miller-Rabin présenté à la fin du TP permet de détecter « la plupart » des nombres non premiers, mais pour ces nombres, il ne fournit pas de diviseur. L'algorithme qui suit va permettre d'en trouver assez efficacement.

3.0. Un bébé exemple et deux outils

On considère $m = 403$. On considère[1] les nombres successifs de la forme $a_k = 2^{k!}\,[m]$ à partir de $k = 1$.

A chaque fois, on calcule le pgcd de $a_k - 1$ et de m, on s'arrête dès qu'on obtient autre chose que 1. Ici, on obtient :

$$\begin{array}{ll} a_1 \equiv 1\ [403] & 1 \wedge 403 = 1, \\ a_2 = 2^2 \equiv 4\ [403] & 3 \wedge 403 = 1, \\ a_3 = 2^6\ \equiv 64\,[403] & 63 \wedge 403 = 1, \\ a_4 = 2^{24} \equiv 326\,[403] & 325 \wedge 403 = 13. \end{array}$$

On vient de trouver un diviseur premier $p = 13$ de 403. Que s'est-il passé ?

On sait, grâce au petit théorème de Fermat, que pour tout nombre premier $p \neq 2$, $2^{p-1} \equiv 1\,[p]$. Autrement dit que $2^{p-1} - 1$ est un multiple de p.

Dans l'exemple précédent, pour $p = 13$, $2^{12} - 1$ est divisible par 13. Mais $a_4 - 1 = 2^{24} - 1 = (2^{12} - 1)(2^{12} + 1)$ donc $a_4 - 1$ est bien un multiple de $p = 13$.

L'essentiel : le $p - 1 = 12$ se trouve dans l'exposant 4! de a_3.

1. On pourrait remplacer 2 par un autre nombre premier, comme 3, le principe est toujours celui des témoins de Fermat, cf. T.P. et fin de ce problème.

Ainsi $p = 13$, qui est un diviseur premier de 403, sera un diviseur commun à 403 et $a_3 - 1$.
Outils :

a) L'algorithme d'Euclide. Ecrire une fonction `Euclide` qui prend en argument deux entiers naturels a et b et renvoie $\text{pgcd}(a, b)$ calculé par l'algorithme d'Euclide.

b) L'exponentiation rapide. La fonction $**$ de Python (raccourci syntaxique de la fonction `pow`) réalise une exponentiation rapide. Rappeler *l'idée* (sans écrire l'algorithme en détail ni son implémentation en Python) de cet algorithme et l'ordre de grandeur du nombre d'opérations pour le calcul de a^m par la méthode d'exp. rapide

3.1. Explication de la méthode dans le cas général

Donnons nous un nombre m, non premier, dont on cherche un diviseur non trivial (si possible même un diviseur premier).

a) On considère la suite des nombres $a_k = 2^{k!}$ successifs à partir de $k = 1$. Déterminer une relation de récurrence simple entre a_k et a_{k-1} pour tout $k \geq 2$. Cette relation permettra d'économiser des calculs dans la boucle.

b) A chaque étape k, on calcule $g_k = \text{pgcd}((a_k \% m) - 1, m)$ où $a_k \% m$ désigne le reste de la division euclidienne de a_k par m.
Justifier que $g_k = \text{pgcd}(a_k - 1, m)$. Pourquoi calculer plutôt $\text{pgcd}((a_k \% m) - 1, m)$?

c) Montrer qu' à chaque étape g_k divise g_{k+1}.

d) La suite des (g_k) commence avec $g_1 = 1$, elle peut rester à 1 pendant un certain un temps, mais si on fixe un certain diviseur premier p de m, il existe un rang n tel que le nombre $p - 1$ divise l'exposant $n!$ de a_n. Démontrer qu'alors on aura $p | g_n$.
On arrête l'algorithme pour la première valeur de n telle que $g_n \neq 1$, en espérant que la valeur g_n obtenue ne soit pas m lui-même mais bien un diviseur non trivial de m.

3.2. Script Python et cas de tests négatifs.

a) Ecrire une fonction `Pollard` qui prend en argument un entier m et renvoie le premier $g_k \neq 1$ de l'algorithme précédent.

b) Si on teste cette fonction on va voir qu'elle marche *souvent* i.e. le g_k renvoyé sera souvent différent de m pour un nombre m non premier. Cependant si on considère $m = 65$, on obtient (tableau à recopier et compléter sur votre copie) :

$a_1 \equiv 2 [65]$ $\quad 1 \wedge 65 = 1,$
$a_2 = 2^2 \equiv 4 [65]$ $\quad 3 \wedge 65 = 1,$
$a_3 = \quad \equiv \quad [65]$ $\quad \wedge 65 = \quad ,$
$a_4 = \quad \equiv \quad [65]$ $\quad \wedge 65 = \quad .$ Echec : on n'a pas trouvé de diviseur non trivial.

c) Expliquer cet échec à partir de la décomposition $m = 65 = 5 \times 13$

d) Généraliser le problème d'échec rencontré au b) et c) : pour quel type de nombres se produira-t-il ?

3.3. Parade possible quand l'algorithme ne fonctionne pas avec $a_1 = 2$

On peut essayer de remplacer 2 par 3 dans la définition de la suite a_k.Autrement dit $a_k = 3^{k!}$.
a) Reprendre l'algorithme pour $m = 65$ en remarquant que $3^{2\times 3} \equiv 14 [65]$.
b) Expliquer ce qui se passe : pourquoi l'algorithme réussit-il ici ?