

Gilles DUBERTRET

BTS
DUT
Licence

Initiation à la cryptographie avec Python



COURS COMPLET

- Plus de 80 exercices
- Tous les corrigés détaillés
- Programmation avec Python

+ EN LIGNE

OFFERT

Les programmes du livre
en Python

deboeck **B**
SUPÉRIEUR

Gilles Dubertret

Initiation à la cryptographie avec Python

Dans la même collection :

- H. Bersini, P. Francq, N. van Zeebroeck, *Les fondements de l'informatique. Du silicone au bitcoin*, 4^e édition, 2023
- T. Thiry, *Les pratiques de l'équipe agile. Définissez votre propre méthode*, préface de Marc Lainez, 2022
- R. Taillet, *Bien débiter en LaTeX*, 2022
- B. Lubanovic, *Python. Comprendre les bases et maîtriser la programmation*, 2022
- Collectif, traduction de P. Van Goethem & A.-S. Vilret, *Cybersécurité. Sécurisation des systèmes informatiques*, 2021
- R. Taillet, *Python pour la physique. Calcul, graphisme, simulation*, 2020
- S. Monk, traduction de P. Van Goethem & A.-S. Vilret, *Programmation Arduino. Développez rapidement vos premiers programmes*, 2020
- B. Desgraupes, *LaTeX. Apprentissage, guide et référence*, 2019
- R. A. Grimes, traduction de P. Van Goethem & A.-S. Vilret, *Hacking et contre-hacking. La sécurité informatique*, 2019

Pour toute information sur notre fonds et les nouveautés dans votre domaine de spécialisation, consultez notre site web :

www.deboecksuperieur.com

Conception et réalisation de couverture : Primo&Primo

Maquette : Hervé Soulard/Nexeme

Mise en page de l'auteur

Dépôt légal :

Bibliothèque royale de Belgique : 2023/13647/096

Bibliothèque nationale, Paris : septembre 2023

ISBN : 978-2-8073-5143-1

Tous droits réservés pour tous pays.

Il est interdit, sauf accord préalable et écrit de l'éditeur, de reproduire (notamment par photocopie) partiellement ou totalement le présent ouvrage, de le stocker dans une banque de données ou de le communiquer au public, sous quelque forme ou de quelque manière que ce soit.

© De Boeck Supérieur SA, 2023 – Rue du Bosquet 7, B1348 Louvain-la-Neuve
De Boeck Supérieur – 5 allée de la 2^e DB, 75015 Paris

Table des matières

Introduction	9
1 Les nombres premiers	11
1.1 Nombres premiers	11
1.2 Crible d'Ératosthène	13
1.3 Facteurs premiers	14
1.4 Complexité, liste des nombres premiers, spirale d'Ulam	15
1.4.1 Notion de complexité algorithmique	15
1.4.2 Liste des nombres premiers	17
1.4.3 La spirale d'Ulam	18
1.5 Décomposition en facteurs premiers	19
1.6 Exercices	20
2 Éléments d'arithmétique	23
2.1 Congruences dans \mathbb{Z}	23
2.1.1 Introduction	23
2.1.2 Congruence	25
2.1.3 Ensemble quotient $\mathbb{Z}/n\mathbb{Z}$	27
2.1.4 Structure algébrique de $\mathbb{Z}/n\mathbb{Z}$	28
2.1.5 Groupe, anneau et corps	29
2.1.6 Relation d'équivalence	29
2.2 Cryptographie : César, Vigenère, permutation (Programmation)	31
2.2.1 Système de cryptographie de César	31
2.2.2 Système cryptographique de Vigenère	34
2.2.3 Permutations alphabétiques	34
2.3 Divisibilité dans \mathbb{Z}	36
2.3.1 Idéal des multiples de $a : (a)$	36
2.3.2 Divisibilité et idéaux de \mathbb{Z}	37
2.3.3 PPCM	37
2.3.4 PGCD	38
2.3.5 Le Théorème de Gauss	40

2.4	PGCD, PPCM et Python (Programmation)	41
2.5	Retour aux nombres premiers	43
2.6	Exercices	44
2.7	Éléments inversibles de $\mathbb{Z}/n\mathbb{Z}$	48
2.7.1	Indicateur d'Euler	48
2.7.2	Petit Théorème de Fermat	50
2.8	Applications et pratique	50
2.8.1	Cryptographie et algèbre linéaire	50
2.8.2	Calcul de $a^x \bmod n$ et le théorème de Fermat	51
2.8.3	Test de non primalité	52
2.8.4	Calcul de a^x « à la main ». Notion de cycle	53
3	L'algorithme d'Euclide étendu	55
3.1	Présentation de l'algorithme	55
3.2	Euclide étendu, inverse de a dans $\mathbb{Z}/n\mathbb{Z}$ (Programmation)	57
3.2.1	Euclide étendu, ou Bezout	57
3.2.2	Inverse de a dans $\mathbb{Z}/n\mathbb{Z}$	58
3.2.3	Python et le package 'galois'	58
3.3	Exercices	61
4	Le logarithme discret	63
4.1	Racine primitive	63
4.2	Critère de primalité de Lehmer	65
4.3	Racine primitive, grands nombres premiers (Programmation)	65
4.3.1	Recherche de racine primitive	65
4.3.2	Recherche de grands nombres premiers	66
5	Cryptosystèmes	69
5.1	Exemples de cryptosystèmes classiques	70
5.1.1	Trois exemples	70
5.1.2	N-gramme substitution	70
5.1.3	Permutation d'ordre d	70
5.1.4	Playfair Cipher	71
5.1.5	Transformation linéaire	71
5.1.6	La machine Enigma	71
5.2	Casser un cryptosystème	77
5.3	Différents niveaux d'attaque	78
5.4	Masque jetable, Vernam (One time pad)	79
5.5	Cryptographie quantique	80
5.6	La Cryptographie militaire (1883), Kerckhoffs	81
5.7	<i>Communication Theory of Secrecy Systems</i> , Shannon	82

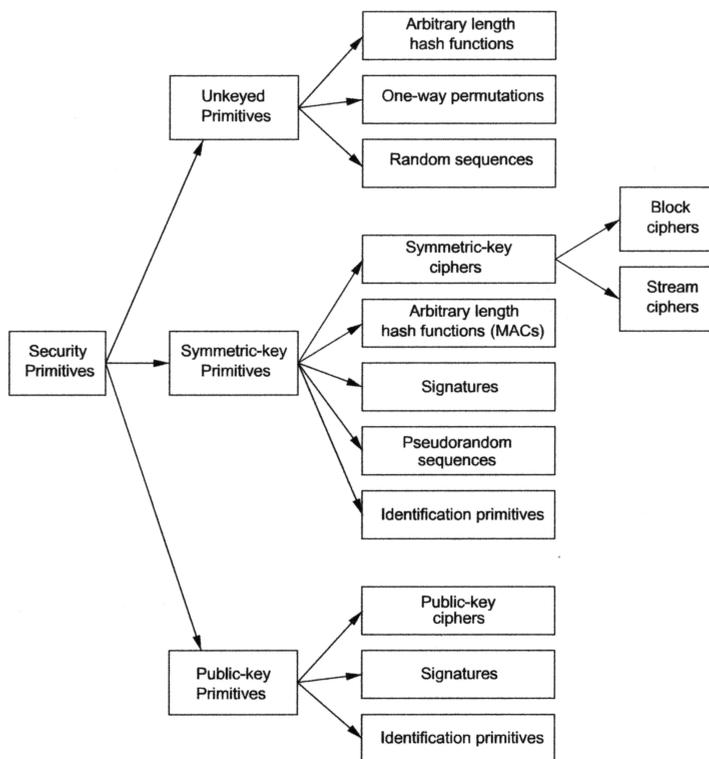
5.8	Convertir du texte en nombre (Programmation)	83
5.9	Python et l'utilisation du code ASCII	83
5.10	Python et la conversion texte-nombre :	84
6	Fonctions à sens unique	87
6.1	Fonctions à sens unique	87
6.2	Sac à dos, Protocole DH, ..., chiffre de Rabin	89
6.2.1	Partage de clés : protocole DH	89
6.2.2	Un cryptosystème sans clé	90
6.2.3	Algorithme du sac à dos	90
6.2.4	Le chiffre de Rabin	91
6.3	Implémentation avec Python (Programmation)	92
6.3.1	Le sac à dos	92
6.3.2	Partage de clés	94
6.3.3	Cryptosystème sans clé	95
6.4	Le théorème chinois et le chiffre de Rabin (Théorie et programmation)	97
6.4.1	Le théorème du reste chinois	97
6.4.2	Le chiffre de Rabin	99
7	Le RSA et le chiffrement Elgamal	103
7.1	Le système RSA	103
7.2	RSA et Python (Programmation)	105
7.3	Chiffrement Elgamal	106
8	Le DES	109
8.1	L'algorithme LUCIFER : notion de ronde	109
8.2	Le DES	111
8.3	IDEA	114
8.4	Modes de chiffrement par bloc. Mode ECB, CBC, CFB, OFB	115
8.5	Ou exclusif et addition modulo 2 (Programmation)	118
8.6	Addition modulo 2^{16}	119
9	Advanced Encryption Standard (AES)	121
9.1	Introduction	121
9.2	Les corps finis (Théorie)	121
9.2.1	Construction de $GF(2^8)$	123
9.2.2	L'anneau $GF(2^8)[x]/(x^4 + 1)$	126
9.3	AES	126
9.3.1	Les rondes	126
9.3.2	La génération des clés de rondes (Key Expansion)	128
9.3.3	Déchiffrement	128
9.4	Python et le corps de Galois $GF(2^8)$ (Programmation)	129

9.5	Implémentation de l'AES (Programmation)	132
9.5.1	Le corps de Galois $GF(2^8)$	132
9.5.2	Les routines	132
9.5.3	KeyExpansion	136
9.5.4	Le chiffrement	138
10	Courbes elliptiques	141
10.1	Introduction	141
10.2	Courbes elliptiques sur Z/pZ (p premier)	143
10.3	Courbes elliptiques sur Z/nZ (n composé)	144
10.4	Application à la cryptographie	144
10.5	Application à la décomposition des grands nombres	145
10.6	Courbes elliptiques et Python (Programmation)	147
10.6.1	Courbes elliptiques sur R	147
10.6.2	Racine carrée dans Z/pZ	148
10.6.3	Courbes elliptiques sur Z/pZ (p premier)	149
10.6.4	Courbes sur Z/nZ (n composé)	151
11	Fonction de Hachage	155
11.1	Protocole	155
11.2	Empreinte (Hash Code)	156
11.3	KECCAK ou SHA-3	158
11.4	Preuve de travail	158
11.5	Générateur Pseudo-aléatoire	159
12	Protocole ZK : Zero Knowledge	161
12.1	Le démon de Quisquater et Guillou	161
12.2	Protocole de Fiat-Shamir	162
12.3	Graphes et cryptographie	163
12.4	Complexité	165
13	Identification, Authentification, Signature	167
13.1	Authentification	168
13.2	Identification	169
13.3	Signature	171
13.4	Signature Elgamal	172
13.5	Conclusion	173
14	Horodatage et Blockchain	175
14.1	Horodatage	175
14.2	Blockchain et le BitCoin	177

15 Exemples d'applications de la cryptographie	181
15.1 PKI	181
15.2 L'argent n'a pas d'odeur	183
15.3 Organiser une partie de poker sur internet	184
15.4 HTTPS	184
15.5 Carte bancaire	185
15.6 PGP	187
15.7 Voter via internet	187
15.8 Chiffrement homomorphe	189
15.9 Secret partagé, Clé partagée	190
15.10 Le WIFI	191
15.11 Chiffrement par flot	192
15.12 La lettre recommandée avec AR	193
15.13 Tatouage numérique	194
15.14 Conclusion	196
16 Cryptanalyse	197
17 La cryptographie à travers l'histoire	199
17.1 L'antiquité	199
17.2 La mécanisation	200
17.3 Systèmes symétriques	200
17.4 Systèmes à clé publique (asymétriques)	201
17.5 Mars 2000 : la signature numérique a valeur légale en France	201
Bibliographie	203
Index	205

Introduction

Contrairement à une idée fort répandue, la cryptographie n'a pas pour seule finalité la confidentialité des communications, même si cet aspect est absolument essentiel. Le tableau suivant copié de [15] paru en 1996 le montre bien.



Remarquons tout de suite que les protocoles de **preuve à divulgation nulle** n'y apparaissent pas : les développements récents ont conduit la cryptologie vers des considérations mathématiques très théoriques.

Cette science est un beau mélange de :

- pratiques souvent empiriques ;
- de mathématiques élémentaires utilisés avec les systèmes à clé publique ;

- de mathématiques au plus haut niveau théorique avec la théorie des langages et la complexité ;
- d'informatique tout ce qu'il y a de plus pratique afin d'implémenter correctement les primitives utilisées ;
- d'informatique théorique, en particulier pour la validation des primitives utilisées ;
- de probabilité, par exemple avec le problème des générateurs pseudo-aléatoires ;
- ...

A l'origine cet ouvrage n'était qu'un prétexte pour faire un peu d'arithmétique avec les étudiants en informatique de l'IUT de Paris. Au fil des ré-éditions, il a grossi, s'est complété, mais il fallait garder l'esprit ce qui a fait son succès : un ouvrage d'initiation, présentant quelques résultats intéressants, mais surtout proposant au lecteur des idées d'approfondissement et d'exploration.

Et peut-être de se lancer dans des études plus complètes de ce domaine vraiment pluri-disciplinaire.

[9] propose une première approche, moins technique, de la cryptographie.

Cette nouvelle édition a obligé l'auteur à apprendre le langage Python, à reprendre tous les programmes d'illustration à l'origine écrits pour MAPLE¹. Ce qui ne signifie pas que MAPLE soit inintéressant, bien au contraire !

Python offre un premier avantage (que l'on ne trouvait dans les années 1990 qu'avec MAPLE) : faire des calculs avec des nombres entier aussi grands que de besoin, et la cryptographie mathématique ne se conçoit pas sans cette possibilité. Est-il besoin de parler des langages de programmation dans lesquels les calculs avec des entiers étaient limités à 65535² ?

Une petite difficulté rencontrée avec Python : le typage dynamique. On crée une donnée, liste, tableau ..., et celle-ci n'est pas toujours du type désiré, ce qui oblige parfois à quelques artifices. Mais peut-être l'auteur a-t-il gardé de vieux réflexes issus d'autres langages ? Il faut savoir rester modeste !

Mais surtout l'univers de Python a l'intérêt d'être particulièrement ouvert, accessible et très dynamique. Chacun pourra trouver dans des « tutos », des « packages » ... de quoi approfondir les points qui ont retenu l'attention, et pourquoi pas trouver très simplement une solution à un « vieux » problème de l'arithmétique. Cela s'est déjà vu !

1. Les programmeurs de MAPLE ont utilisé des idées mathématiques reprises à l'occasion dans ce livre. Seules ces idées sont expliquées. Il n'est pas besoin de connaître MAPLE pour les trouver intéressantes.

2. Pourquoi 65535 ? Question pour les débutants en informatique.

Chapitre 1

Les nombres premiers

1.1 Nombres premiers

Dans ce chapitre, on ne considère que l'ensemble \mathbb{N} des entiers naturels.

Définition 1.1 Un nombre p est premier s'il admet **exactement** deux diviseurs, 1 et lui-même.

Remarque : *cette définition exclut 1 de la liste des nombres premiers.*

Il est facile de vérifier de tête que 7, 13 ou 31 sont des nombres premiers. Quelle méthode adopter pour montrer que 4999 est premier ?

Premier algorithme

Essayer toutes les divisions de 4999 par D , pour D allant de 2 à 4998.

Si aucune division ne tombe juste, alors on peut affirmer que 4999 est premier.

Deuxième algorithme

En général, l'idée vient assez rapidement de s'arrêter à la moitié de 4999 (le lecteur est invité à vérifier cette affirmation sur son entourage).

En effet, les quotients pour des diviseurs au-delà de 2500 sont inférieurs à 1, et ne peuvent donc pas être entiers.

Mais il y a mieux : examinons la suite des quotients successifs des divisions de 29 (29 pour simplifier le tableau).

Diviseur	2	3	4	5	6
Quotient	14.5	9.6	7.2	5.8	4.8

Le tableau ci-dessus nous indique que les quotients vont en décroissant. À partir de 6, les quotients sont plus petits que le diviseur. Aucun de ces quotients pour un diviseur supérieur à 6 ne peut être entier puisque cela signifierait que 29 est divisible par ce quotient. Mais aucune division par un nombre inférieur à 6 n'a donné de quotient entier. D'où la simplification suivante :

Essayer toutes les divisions de 29 par D , pour D allant de 2 à $\sqrt{29}$.

(Plus précisément à la partie entière de $\sqrt{29} + 1$.)

Avec 4999, on est passé de 4997 divisions à 70 divisions, ce qui est un gain appréciable de temps.

Proposition 1.1 Pour vérifier que N est premier, il suffit de tester toutes les divisions de N par D , avec D allant de 2 à $\sqrt{N} + 1$

Cela donne le programme suivant avec python :

Code Python

www.lienmini.fr/51431-python1



```
def isprime(n):
    """teste si un nombre est premier,
       teste toutes les divisions impaires jusqu'à la racine de n+1
       time.time() permet de mesurer le temps d'exécution"""
    start = time.time()
    if n == 2:
        return True
    elif n % 2 == 0:
        return False
    max = int(n**(0.5))+1
    for i in range(3, max, 2):
        if n % i == 0:
            end = time.time()
            return False,end-start
    else:
        end = time.time()
        return True,end-start
```

Remarque : (La fonction `time()` nécessite d'importer le package "time". Elle n'est pas essentielle ici, mais utile que pour estimer le temps de calcul de cet algorithme. Le lecteur est invité à la supprimer, ou à l'utiliser à sa convenance avec d'autres fonctions de ce chapitre.)

Troisième algorithme

On peut même encore améliorer la méthode en n'essayant que les divisions par les nombres premiers inférieurs à $\sqrt{N} + 1$, à condition de disposer d'une liste des nombres premiers.

En effet, si la division par 2 ne tombe pas juste, il est inutile d'essayer les divisions par les multiples de 2. On les raie tous de la liste des divisions à tester.

Si la division par 3 ne tombe pas juste, il est inutile d'essayer les divisions par les multiples de 3...

Il ne reste plus qu'à essayer les divisions par les nombres **premiers** inférieurs à $\sqrt{N} + 1$. Si vous avez une telle liste jusqu'à 1000, cela permet de tester rapidement si un nombre inférieur à 1000^2 est premier.

Avec 4999, il ne reste plus que 20 divisions à tester.

Ces considérations nous amènent tout naturellement au crible d'Ératosthène.

1.2 Crible d'Ératosthène

Comment créer une liste des nombres premiers ?

La méthode proposée par Ératosthène¹ connue sous le nom de «Crible d'Ératosthène» donne une solution.

Écrire tous les nombres de 2 à 1000. 2 est premier, on le souligne et on raie tous les multiples de 2. Le premier nombre non rayé est 3. Il est donc premier, on le souligne et on raie tous ses multiples. Etc.

Arrivé à 32, qui est supérieur à la racine carrée de 1000, on a terminé : tous les nombres qui n'ont pas été rayés sont premiers.

La méthode peut paraître efficace, mais elle devient inutilisable pour une table des nombres premiers jusqu'à 10.000.000 par exemple.

On peut aussi utiliser le programme ci-dessous qui renvoie le résultat dans la variable L :

```
def listprime1(n):
    """Retourne la liste des nb premier jusqu'à n"""
    Listprime = []
    for i in range(2, n):
        if isprime(i):
            Listprime.append(i)
    return(Listprime)
L = listprime1(100)
```

Cependant on constate vite que les calculs, rapides au début, deviennent de plus en plus lents.

Toutes ces méthodes deviennent inefficaces avec des nombres très grands. Estimons par exemple le nombre de divisions à effectuer pour tester $2^{67} - 1$, qui vaut approximativement 1.47×10^{20} , c'est-à-dire qui s'écrit avec 21 chiffres. Sa racine carrée vaut approximativement 1.200.000.000. Il faudra donc un peu plus d'un milliard de divisions, certaines à dix chiffres ! Bon courage !

(F. Cole a calculé en 1903 que $2^{67} - 1 = 193707721 \times 761838257287$)

Le package NumTheory de Python fournit la procédure `isprime(n)`, et bien d'autres, qu'il convient de tester maintenant. Et bien sûr des versions de toutes les fonctions étudiées dans ce chapitre et le suivant.

En bref, toutes ces méthodes ne permettent pas de traiter les « grands » nombres premiers. Si on dispose aujourd'hui de tests rapides de primalité, ces tests ne fournissent pas de diviseurs des nombres qui ne sont pas premiers : la décomposition en facteurs premiers reste un problème difficile. Nous y reviendrons (voir 4.2 et 4.3.2).

1. Ératosthène : astronome, mathématicien grec (Cyrène, v. 284 - Alexandrie, v. 192).

1.3 Facteurs premiers

Existe-t-il de grands nombres premiers ?

La réponse a été fournie par Euclide².

Théorème 1.1 *Tout nombre N entier admet au moins un facteur premier (sauf 0 et 1 bien sûr).*

(C'est-à-dire tout nombre N entier est divisible par un nombre premier.)

En effet, soit N est premier et il n'est divisible que par lui-même (et 1), soit N n'est pas premier et il admet un certain nombre de diviseurs. Appelons P le plus petit de ces diviseurs. P est premier car sinon P aurait un diviseur D , plus petit que P , et qui diviserait N .

Théorème 1.2 *L'ensemble des nombres premiers est infini.*

En effet, si on suppose que cet ensemble est fini, il est composé de n nombres p_1, p_2, \dots, p_n , alors $N = p_1 \times p_2 \times \dots \times p_n + 1$ est aussi premier puisqu'il n'est divisible par aucun des p_1, p_2, \dots, p_n . Comme il est plus grand que chacun des nombres p_1, p_2, \dots, p_n , il y a contradiction.

On peut donc trouver des nombres premiers aussi grands que l'on veut.

En 1983, le record du nombre premier le plus grand était : $2^{86243} - 1$, soit un nombre de trente mille chiffres. Le record a été battu depuis avec un nombre premier comportant un million de chiffres décimaux.

Théorème 1.3 *Tout entier peut se décomposer en produit de facteurs premiers (sauf 0 et 1 bien sûr) : $n = p_1^{a_1} \times p_2^{a_2} \times \dots \times p_r^{a_r}$.*

(L'unicité de cette décomposition sera démontrée plus tard)

En effet, si n est premier la décomposition est toute trouvée.

Sinon n admet un facteur premier p_1 , et un quotient $q_1 : n = p_1 \times q_1$ avec $q_1 < n$.

On recommence avec q_1 jusqu'à obtenir un quotient premier.

Conclusion : tout entier n peut s'écrire $n = p_1^{a_1} \times p_2^{a_2} \times \dots \times p_r^{a_r}$

Un autre problème qui s'est posé très tôt est celui de la répartition des nombres premiers. S'ils apparaissent régulièrement dans la suite des entiers, il sera facile de déterminer si N est premier ou pas. Hélas...

Par exemple, on montre facilement que la suite des nombres premiers comporte des « trous » de longueur aussi grande que l'on veut : si on note P le produit des n premiers nombres premiers, alors tous les nombres $P+2, P+3, P+4, \dots, P+n$ sont composés.

2. Euclide : mathématicien grec du troisième siècle av. J-C.

1.4 Complexité, liste des nombres premiers, spirale d'Ulam

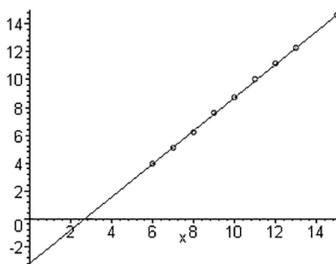
1.4.1 Notion de complexité algorithmique

La fonction `isprime1()` donnée ci-dessus permet de connaître le temps de calcul pour vérifier que n est premier. Sur l'ordinateur de l'auteur³, les essais ont donné le tableau suivant :

Nombre	Nombre de chiffres	Temps de calcul en ms
13	2	0
1009	4	0
10007	5	0
100003	6	55
1000003	7	164
10000019	8	494
100000007	9	1922
1000000007	10	6536
10000000019	11	22464
100000000003	12	72116
1000000000039	13	225689
10000000000031	15	2397419

Quelle est la nature de la relation entre le nombre de chiffres de n et le temps de calcul ?

Traçons le nuage de points correspondant à ces données, en prenant le logarithme népérien du temps de calcul (les trois premiers résultats ont été exclus). (Ce livre n'ayant pas pour objet le calcul statistique, les deux calculs ci-dessous ne seront pas détaillés.) Le lecteur est invité à trouver dans Numpy les fonctions d'affichage et les fonctions statistiques permettant de traiter les données ci-dessous : [6,7,8,9,10,11,12,13] : nombre de chiffres [4,5,1,6,2,7,6,8,7,10,11,2,12,3] : Logarithme du temps de calcul



On est frappé par l'alignement des points.

3. En 1998, mais cela ne change rien au problème posé!

Un ajustement linéaire s'impose.

On obtient : $y = 1.192419355 x - 3.201129032$

y est le logarithme népérien du temps de calcul en fonction du nombre de chiffre de n . On en déduit facilement la relation suivante entre le temps de calcul et le nombre de chiffres du nombre premier :

$$Temps = e^{(1.2 \times \text{nombre de chiffres} - 3.2)}$$

Cela signifie, en supposant que cette relation reste exacte pour d'autres valeurs que celles testées, que pour vérifier par cette méthode qu'un nombre de 100 chiffres est premier, il faudrait environ 5×10^{50} s, soit environ $1,5 \times 10^{43}$ années, ce qui rend la méthode totalement inutilisable.

Même en supposant que l'utilisateur se lance dans la course à l'ordinateur le plus performant, équipé du dernier modèle de processeur ..., le temps de calcul restera rédhibitoire.

Un tel algorithme pour lequel le temps de calcul s'exprime comme une fonction exponentielle de la taille des données est dit de **complexité exponentielle**.

Par contre, si le temps de calcul s'exprime comme une fonction linéaire de la taille des données, l'algorithme sera dit de **complexité linéaire**.

Et bien sûr de complexité en n^2 si le temps de calcul est une fonction de degré 2 de la taille des données. (Exemple : l'élévation au carré d'un nombre est de complexité en n^2 , où n est le nombre de chiffres du nombre à élever au carré). On parlera alors de complexité Polynomiale, notée P . Plus subtil, et cela fonde la cryptographie comme on le verra, certains problèmes peuvent être de Complexité Exponentielle, mais si on connaît le résultat, celui-ci est facile à vérifier. C'est le cas par exemple de la décomposition d'un entier en facteurs premiers.

Exercice 1.1. Écrire une procédure qui calcule le carré de 1.000 nombres de 100, 1.000, 10.000 ... chiffres (utiliser une boucle "for i in range(100) ... n^2 ...), et vérifier cette affirmation avec les méthodes exposées ci-dessus.

Résumé

Un algorithme de complexité exponentielle est en pratique inutilisable.

Des algorithmes de complexité linéaire, en $n \times \text{Log}(n)$ ou en n^2 sont considérés comme « rapides ».

Pour terminer signalons que MAPLE fournit la procédure « isprime(n) » qui teste avec une rapidité surprenante si n est premier. La documentation indique que le test est « probabiliste ». En est-il de même avec la fonction équivalente de Python ?

Quelle est la signification de cette indication ?

Pour y répondre, supposons que vous soyez candidat à un jeu télévisé et qu'on vous propose un nombre de 3 chiffres. Question : est-il premier ? Si vous répondez au hasard, vous avez une probabilité faible de répondre juste.

Le test de divisibilité par 2 est immédiat. Si vous l'utilisez, votre probabilité de répondre juste va augmenter.

Si vous avez le temps, vous appliquerez les tests de divisibilité par 3, puis 5, puis 11... en augmentant à chaque fois vos chances de réussite.

Si votre nombre n'est divisible ni par 2 ni par 3 ni par 5, vous affirmerez très sérieusement que le nombre est premier.

Signalons ici un article datant de 2004 intitulé 'PRIMES is in P', que l'on pourra trouver dans [1], tendant à prouver que le problème de la primalité d'un nombre entier est "rapide".

Bien sûr les tests utilisés par « isprime(n) » sont plus sophistiqués que ceux décrits ci-dessus (voir 2.8.3), et donnent de très bons résultats. On n'obtient cependant pas une certitude mathématique de primalité. Certains auteurs parlent de **nombres premiers industriels** pour de tels nombres. Leur utilisation en cryptographie est amplement satisfaisante.

Insistons : s'il est rapide de vérifier avec une probabilité très proche de 1 qu'un nombre de 100 chiffres est premier, ou mieux encore si on peut trouver rapidement des nombres premiers de 200 chiffres en disposant d'une preuve mathématique que le nombre trouvé est bien premier (voir 4.3.2), **il est par contre pratiquement impossible de trouver les diviseurs de $n=p \times q$, avec p et q premiers, p et q s'écrivant avec 100 chiffres.**

1.4.2 Liste des nombres premiers

La liste des nombres premiers (jusqu'à n) peut s'obtenir à la main par le crible d'Eratosthène. Pour pouvoir disposer de listes plus complètes, on pourra utiliser l'une ou l'autre des deux procédures qui suivent. La première utilise isprime(n), la deuxième crée la liste des nombres premiers et l'utilise pour minimiser le nombre de divisions à effectuer pour trouver le nombre premier suivant.

Avec MAPLE la seconde est sensiblement plus rapide. Avec Python, ce serait l'inverse. Le lecteur est invité à utiliser la fonction "time" de Python pour les comparer avec des nombres 'grands'.

```
def listprime1(n):
    """Retourne la liste des nombres premier jusqu'à n"""
    Listprime = []
    for i in range(2, n):
        if isprime(i):
            Listprime.append(i)
    return(Listprime)
L = listprime(100)

def listprime2(n):
    """Construit la liste des nombres premiers,
    et l'utilise pour continuer
    Retourne la liste des nombres premier jusqu'à N"""
    l = [2]
    for i in range(3,n):
        max = int(n**(0.5))
        for d in l[:max]:
            if i % d == 0:
                break
        else:
            l.append(i)
    return l
```

1.4.3 La spirale d'Ulam

L'idée due à Stanislaw Ulam⁴ est de représenter la suite des nombres premiers en spirale. On « enroule » la demi droite $[0, +\infty[$ autour de 0. L'option `coords=polar` de la procédure `plot` nous permet de faire cela facilement : on fait correspondre à un nombre premier p le point du plan de coordonnées polaires $(p, p/100)$.

Les points correspondant à des nombres premiers seront marqués d'une croix.

Ulam(10000), coordonnées polaires : (n,n/100)



Ulam(100000), coordonnées polaires : (n,n/100)



La même spirale, sur laquelle ne sont tracés que les nombres premiers jusqu'à 100.000 permet de deviner certaines régularités.

Bien sûr d'autres façons de construire la spirale sont possibles et permettent de trouver des idées intéressantes sur les nombres premiers.

Signalons enfin un résultat très important sur le nombre de nombres premiers inférieurs à n , noté habituellement $\Pi(n)$, et démontré par des méthodes analytiques qu'il n'est pas question de développer ici : $\Pi(n) \approx n/\text{Log}(n)$ pour n « grand ».

4. Stanislaw Ulam 1909-1984.

Exercice 1.2. Le lecteur est invité à vérifier expérimentalement ce résultat à l'aide de son ordinateur.

1.5 Décomposition en facteurs premiers

L'idée est qu'un nombre est soit premier, soit admet un diviseur premier. S'il admet un diviseur premier, on effectue la division et on recommence avec le quotient obtenu.

Ce qui donne la procédure suivante :

```
def pluspetitdiviseur(n):
    """Fournit le plus petit diviseur de n (sauf 1)"""
    max = int(n**(0.5))+1
    for i in range(2, max):
        if n % i == 0:
            return i
        break
    else:
        return n

def facteurs_preiers(n):
    """Décomposition de n en produit de facteurs premiers
    Exercice : obtenir un affichage plus joli
    que la liste de facteurs premiers"""
    L = []
    while True:
        d = pluspetitdiviseur(n)
        L = L+[d]
        if d == n:
            return L
        else:
            n = n//d
```

```
In[8] : facteurs_preiers(3**2*5**3*11**4)
```

```
Out[8]: [3, 3, 5, 5, 5, 11, 11, 11, 11]
```

Exercice 1.3. Chercher dans Python, et en particulier Numpy, toutes les fonctions similaires à celles exposées ci-dessus. Cet exercice vaut aussi pour le chapitre suivant concernant l'arithmétique.

Initiation à la cryptographie avec Python

Considérée comme la science du secret, la cryptographie fait aujourd'hui partie de notre vie quotidienne : cartes à puce, Internet, courrier électronique... ne faisons-nous pas déjà depuis de longues années de la cryptographie sans le savoir ? Riche de multiples possibilités et méthodes, cette discipline – servant à assurer la sécurité et la confidentialité des communications et des données – s'impose à tous.

L'objectif de ce manuel est de rendre accessible, aux étudiants en premier cycle des études supérieures des cursus mathématiques et informatique, les possibilités et les méthodes de la cryptographie moderne, à l'aide de Python.

Ce cours complet de cryptographie accueille également de nombreux exercices, dont certains sont intégralement corrigés. Des compléments numériques vers les programmes du livre en Python viennent compléter l'ensemble.

SOMMAIRE

Introduction

1. Les nombres premiers
2. Éléments d'arithmétique
3. L'algorithme d'Euclide étendu
4. Le logarithme discret
5. Cryptosystèmes
6. Fonctions à sens unique
7. Le RSA et le chiffrement Elgamal
8. Le DES
9. Advanced Encryption Standard (AES)

10. Courbes elliptiques
11. Fonctions de Hachage
12. Protocole ZK : Zero Knowledge
13. Identification, Authentification, Signature
14. Horodatage et Blockchain
15. Exemples d'application de la cryptographie
16. Cryptanalyse
17. La cryptographie à travers l'histoire

Bibliographie

Index

LES PLUS

- Nouvelles méthodes de cryptographie (AES, chiffrement homomorphe, etc.)
- Compléments numériques vers les programmes du livre en Python

Gilles Dubertret, Professeur certifié de mathématiques, a enseigné les mathématiques et l'informatique au lycée de Sèvres et en IUT à l'Université Paris V / René Descartes.

20,90 €

ISBN : 978-2-8073-5143-1



9 782807 135143 1

deboeck **B**
SUPÉRIEUR

www.deboecksuperieur.com