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 SUPÉRIEUR

Gilles Dubertret

Initiation à la cryptographie avec Python

deboeck SUPÉRIEUR

Dans la même collection :

H. Bersini, P. Francq, N. van Zeebroeck, *Les fondements de l'informatique. Du silicone au bitcoin*, 4e édition, 2023

T. Thiry, *Les pratiques de l'équipe agile. Définissez votre propre méthode*, préface de Marc Lainez, 2022

R. Taillet, *Bien débuter 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

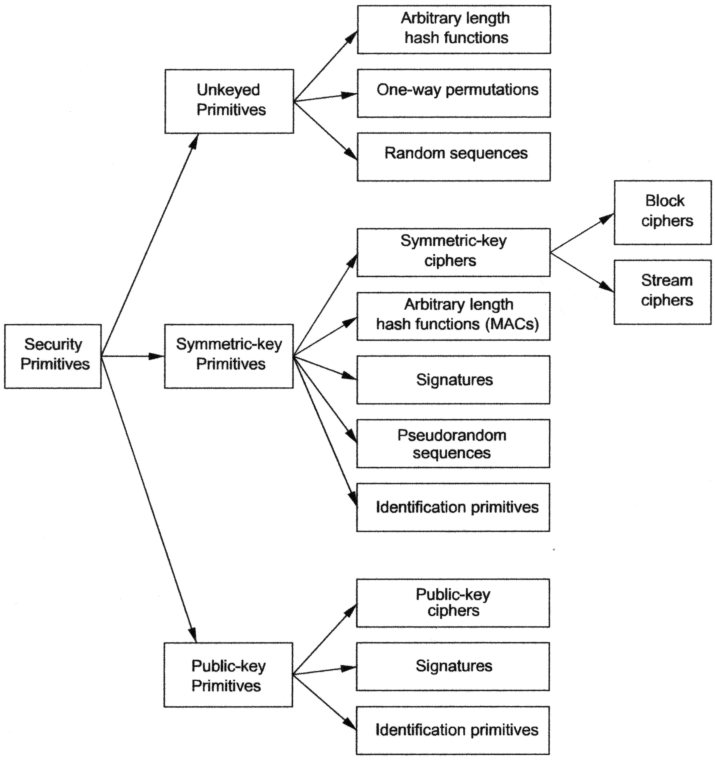
Conception et réalisation de couverture : Primo&Primo
Maquette : Hervé Soulard/Nexeme
Mise en page de l'auteur

Dépôt légal :
Bibliothèque royale de Belgique : 2023/13647/096
Bibliothèque nationale, Paris : septembre 2023
ISBN : 978-2-8073-5143-1

Table des matières

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 MAPLE [1]. 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 65535 [2] ?

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 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 division 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 Ératosthène [1] 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 Euclide [2].

Théorème 1.1 *Tout nombre N entier admet au mois 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 $p_1, p_2, ...p_n$, alors $N = p_1 \times p_2 \times ... \times p_n + 1$ est aussi premier puisqu'il n'est divisible par aucun des $p_1, p_2, ... p_n$. Comme il est plus grand que chacun des nombres $p_1, p_2, ... p_n$, 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) :* $n = p_1^{a_1} \times p_2^{a_2} \times ... \times p_r^{a_r}$.

(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 $n = p_1^{a_1} \times p_2^{a_2} \times ... \times p_r^{a_r}$

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,..., P+n sont composés.

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

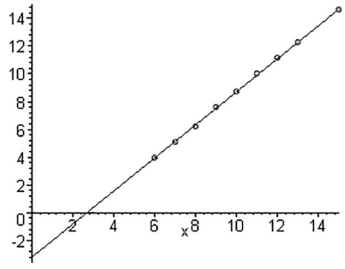
1.4 Complexité, liste des nombres premiers, spirale d'Ulam

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'auteur [3], 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
100000000000031	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 on 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 :

$$Temps = e^{(1.2 \times nombredechiffres - 3.2)}$$

Cela signifie, en supposant que cette relation reste exacte pour d'autre 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 un 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 un 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 $n \times Log(n)$ 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×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 Ulam [4] 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 $\prod(n)$, et démontré par des méthodes analytiques qu'il n'est pas question de développer ici : $\prod(n) \approx n/Log(n)$ 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_premiers(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_premiers(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

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 351431

deboeck SUPÉRIEUR

www.deboecksuperieur.com

Conception graphique : Primo&Primo®