2023/06/20 ver6

Basic SecCap 2023
「セキュリティ基礎論-補助資料-」

宮地 充子

大阪大学大学院

Email: Ta-basicseccap-23@crypto-cybersec.comm.eng.osaka-u.ac.jp

概 要

本講義では，暗号の基本原理，手法, 安全性証明，計算効率の習得を通じて，情報セキュリティを支える数理的な考え方及びその応用手法について理解することを目的とします．全講義を終えて初めて，現代暗号理論が習得できるように講義は構成されて，各理論を技術計算ソフトウェア python を用いて実装します．本講義は板書形式です．板書の速度が頭で理解するのに最適な時間だからです．なお，板書内容は基本的に頭に記憶することが必須です．講義内容は講義と演習の両方の組み合わせで初めて定着しますので，演習をしっかり行うようにしてください．

講義の前に事前課題を出します．講義中に課題について確認しますので，わかる人は積極的に手を上げてください.（挙手で出席確認します）講義後に問，演習，解析の 3 種類の課題を出します．全ての課題は本講義用に開発された学習システム moodle を用いて提出，返却されます．

- 問 (Problem): 講義の理解を深めます．ノートに解答を記載，あるいは Latex や word で記載して，pdf に変換して提出．写真は禁止です．
- 演習 (Implementation): 実装により，理論と実装の間のギャップを理解します．Python を用いて Python に標準搭載の関数も利用しながら作成．提出時は該当の演習が引用するプログラムも一緒に提出するようにしてください．演算速度の測定は指定された回数 (10^4 回等）実施し，平均値を求めるようにして下さい．演習については次の 2 パターンの解答方法を想定しています．各自の進捗に応じて選択してください．
  - 初心者編：Python 標準搭載の関数を利用.
  - 中級者編：Python を用いて，自分で実装.
- 解析 (analysis): 実験結果と理論値から考察します，統計処理などをする際に問を利用します．実験データは excel 等を用いて電子的に作成，実験結果は自分なりに解析する．実験結果を含む解析レポートとなります．これらを 1 つのディレクトリに保存して，zip で圧縮し，以下のファイル名で提出して下さい．

  実装課題の中で利用する数値例は D 章に，実装する関数 API は C 章記載の API に沿ってください．標準搭載関数で利用する関数も C 章に記載されています．なお，作成した関数は E 章記載のテストデータと比較して動作確認を行うとともに，本講義用で構築された moodle システムにある演習検証を実行してください．実装課題はこれらの動作確認を終わらせてから提出してください．python ソースの提出は jupyter notebook のファイルを提出してください．ファイル名は，以下に沿ってください．解析の提出物は，Python のソースファイルと演習で要求された実行結果のファイルです．ソースファイルと実行結果のファイルを 1 つのディレクトリに保存して，zip で圧縮し，以下のファイル名で提出して下さい．なお,「問」「演習」「解析」（プログラムの結果）の提出物のファイル名の演習番号のアルファベットはそれぞれ P，I，A の頭文字をつけて下さい．

大学名頭文字_名前_演習番号 (大学名頭文字：石川高専 → I, 三重大 → M 長崎県立大 → NK，阪大工学部 → OE, 岡山大 → OY, 東北大学 → T, 静岡大 → S)
例: 阪大工学部の宮地が第一回の問，演習 (プログラムの出力値)，解析を提出する場合:
OE_miyaji_P1, OE_miyaji_I1(OE_miyaji_IA1), OE_miyaji_A1

ではインターネットの安心・安全を支えるセキュリティ技術を理論，演習，実装の中で習得していきましょう．

教科書：“代数学から学ぶ暗号理論”（日本評論社） 2,3,7-9,13 章
講義日程及び時間: 6/5, 6/12(演習),6/19, 7/10(演習), 7/24, 8/7(演習）(16:50-18:20)

1. (6/5) 講義 1. 暗号基盤理論（離散数学）の紹介． (教科書 2, 3 章)
   修得知識：知識単位 （群，環，体，有限体，有限体上の演算，位数，指数，Lagrange の定理，べき乗演算）
   実装アルゴリズム（公開鍵暗号・ディジタル署名実装に必須のアルゴリズム）バイナリ法，ユークリッドの互除法，拡張ユークリッドの互除法，素数判定法
2. (6/19) 講義 2. 公開鍵暗号 (教科書 3,7,8 章)
   修得知識：知識単位 （中国人の剰余定理, 公開鍵暗号の仕組み，安全性，実装アルゴリズム）
   実装アルゴリズム（中国人の剰余定理，暗号化，復号，鍵生成アルゴリズム）

3. (7/24) 講義 3. ディジタル署名 (教科書 7,9 章)
   修得知識：知識単位 （ディジタル署名の仕組み，安全性，実装アルゴリズム，ブラインド署名）
   実装アルゴリズム（署名生成，署名検証，鍵生成アルゴリズム）

本講義はセキュリティ・暗号関係の研究者・技術者を目指す受講生からセキュリティ・暗号に興味をもつ受講生までの取り組みを設定します．講義は同一ですが，課題については受講生の目的・レベルに応じて取り組んでください．なお，合格条件は basic mini コースになります．

advanced　セキュリティスペシャリストコース セキュリティ・暗号関係の技術者を目指す方のコースです．全ての事前課題，問と演習と解析に取り組みましょう．演習は標準関数を利用せずに，自分で実装しましょう．

standard コース セキュリティ・暗号関係の知識習得を目指す方の標準的なコースです．全ての事前課題，問と演習に取り組みましょう．演習は標準関数を利用します． PBL 演習を受講する人は standard コース以上が条件です．

basic コース セキュリティ・暗号関係の基礎的な知識習得を目指す方のコースです．すべての事前課題，問と一部の演習に取り組みます．演習は標準関数を利用します． 演習 1.1, 演習 1.3, 演習 2.2, 演習 2.3，演習 2.4, に取り組みましょう．

basic mini コース セキュリティ・暗号関係の最低限の知識習得を目指す方のコースです．全ての事前課題に加えて，以下を行ってください．問 1.2, 問 1.3，演習 2.2, 演習 2.4, 問 2.2, 問 3.1 の提出が単位授与の条件です．

目 次

1 講義 1 暗号基盤理論 (教科書 2, 3, 13 章)

加減乗除ということばを聞きますが，演算としては加法と乗法で，減算と除算はそれぞれの演算の逆元演算になります．本講義では 1 つの演算の逆演算の意味を理解するとともに，元の集まりの集合に演算というルールを与えることで，群という概念が作られることを学びます．また，乗法の逆元演算の計算と不定方程式の解を求める問題の関係についても学びます．

1.1 群・環・体 (教科書 2.4-2.6 章，3.8 章)

例 1 $\mathbb{Z}/3\mathbb{Z}$ における和と積.

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\times$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

例 2 $\mathbb{Z}/4\mathbb{Z}$ における和と積.

$+$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$\times$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

問 1.1 *(証明)* 剰余環 $\mathbb{Z}/m\mathbb{Z}$ において以下を示せ.

$$\mathbb{Z}/m\mathbb{Z} \text{ が体になる.} \iff m \text{ が素数}$$

問 1.2 上で位数 *22* となる元を求めるアルゴリズムをステップ数を評価しながら考える.
(0) 22 を素因数分解せよ.
(1) $U(\mathbb{Z}/23\mathbb{Z})$ において，位数 *2* の元の個数 n_2 を求めよ.
(2) $U(\mathbb{Z}/23\mathbb{Z})$ において，位数 *11* の元の個数 n_{11} を求めよ.
(3) $U(\mathbb{Z}/23\mathbb{Z})$ において，位数 *22* の元の個数 n_{22} を求めよ.

(4) $U(\mathbb{Z}/23\mathbb{Z})$ で，位数 *1*，*2* の元を除いた集合からランダムに生成した元が位数 *11* となる確率を求めよ．
(5) $U(\mathbb{Z}/23\mathbb{Z})$ で，位数 *1*，*2* の元を除いた集合からランダムに生成した元が位数 *22* となる確率を求めよ．
(6) $U(\mathbb{Z}/23\mathbb{Z})$ において，位数 *2* の元を求めよ．
(7) 位数 *22* となる元を求めるアルゴリズムのステップ数の期待値を求めよ．

1. $U(\mathbb{Z}/23\mathbb{Z})$ から位数 *1*，*2* を除いた元からランダムに元 a を生成する．
2. a の位数が *22* であれば，a を出力する．位数が *22* でないとき，ステップ *3* へ．
3. a の位数が *11* であれば，a と位数 *2* の元の積を出力する．

(8) *(7)*）を a を $a = 2$ から順次確認する．位数 *22* と判定される最初の数を求めよ．

1.2 剰余環上の除算 (教科書 2.6, 3.3 章)

アルゴリズム 1 (ユークリッドの互除法)
入力：2 正数 a，$b(a \geq b)$
出力：$\gcd(a, b) = d$
(1) $a_0 = a$，$a_1 = b$，$i = 1$ とする．
(2) $a_i = 0$ ならば，$d = a_{i-1}$ を出力し，終了．
(3) $a_i \neq 0$ のとき，$a_{i-1} = a_i q_i + a_{i+1}$，$0 \leq a_{i+1} < |a_i|$，$i = i + 1$ とし，(2) へ．

事前演習 1 ユークリッドの互除法を実装し，次の数の最大公約数を求めよ．
(1) $gcd(7123456, 345783)$
(2) $gcd(1111110, 2222220)$
(3) $gcd(p_1 - 1, q_1 - 1)$ *(表 D.9)*
(4) $gcd(p_2 - 1, q_2 - 1)$ *(表 D.10)*
(5) $gcd(p_3 - 1, q_3 - 1)$ *(表 D.11)*

アルゴリズム 2 (拡張ユークリッドの互除法)
入力：2 正数 a と $b(a > b)$
出力：$\gcd(a, b) = d$ なる d と $ax + by = d$ なる整数 x，y
(1) $a_0 = a$，$a_1 = b$ とする．
(2) $x_0 = 1$，$x_1 = 0$ とする．
(3) $y_0 = 0$，$y_1 = 1$ とする．
(4) $i = 1$ とする．
(5) $a_i = 0$ ならば，$d = a_{i-1}$，$x = x_{i-1}$，$y = y_{i-1}$ を出力し，終了．
(6) $a_{i-1} = a_i q_i + a_{i+1}$，$0 \leq a_{i+1} < a_i$ により，a_{i+1} と q_i を定める．
(7) $x_{i+1} = x_{i-1} - q_i x_i$ とする．
(8) $y_{i+1} = y_{i-1} - q_i y_i$ とする．
(9) $i = i + 1$ として，(5) へ．

解析 1.1 拡張ユークリッドの互除法は，(a, b) の最大公約数が，$a = bq + r(b > r \geq 0)$ となる (b, r) の最大公約数と一致することを利用する．ここで，*1 ループ (*(5)-(8)*)* において必要となる乗算回数，加算回数，除算回数 *((6)* において，(q_i, a_{i+1}) を求める計算量*)* を求めよ．

問 1.3 拡張ユークリッドの互除法を用いて以下の数に対して，$ax + by = 1$ となる整数 x, y を求めよ．
(1) $a = 23,\ b = 17$
(2) $a = 13,\ b = 8$

演習 1.1 拡張ユークリッドの互除法を実装しよう．以下の数に対して，実装を用いて，$ax + by = 1$ となる整数 x, y が存在するときは解を求めよ．存在しない場合は解がないと出力するように記述せよ．
(1) $a = 1234567$, $b = 234578$
(2) $a = 11111111$, $b = 22222222$

演習 1.2 *[*有限体上の逆元*]* 拡張ユークリッドの互除法を応用して，以下の $\mathbb{F}_p$ 上の逆元を求めよ．
(1) 表 *D.3* にある p_1, g_1 を用いて $g_1^{-1} \pmod{p_1}$ を求めよ．
(2) 表 *D.4* にある p_2, g_2 を用いて $g_2^{-1} \pmod{p_2}$ を求めよ．
(3) 表 *D.1* にある p_4, g_4 を用いて $g_4^{-1} \pmod{p_4}$ を求めよ．
(4) 表 *D.1* にある p_4, g_5 を用いて $g_5^{-1} \pmod{p_4}$ を求めよ．
(5) 上記 *4* つの実行速度を測定せよ．*(*実行速度は 10^3 回実施し，平均値を求める．*)*

解析 1.2 演習 *(1.2)* に基づき，下記の解析を行う．アルゴリズム *2* のループの実行回数 *((5)-(8))* をカウントし，有限体の大きさ *p,* 逆元を求める元 g と逆元を求めるアルゴリズムの計算量の関係を推測せよ．

1.3 べき乗演算 (教科書 13.3 章)

べき乗演算を行うアルゴリズムを記述する．通常，暗号では，k ビットとは k 個の 0 か 1 の数字の列であり，例えば，160 ビットの数には 0 も含まれる．アルゴリズム 3 は g^0 も正しく計算するアルゴリズムである．

アルゴリズム 3 (法 p 上のバイナリ法 1) `ModBinary1(`k`,` y`,` p`)`
入力： 正整数 $k = \sum_{i=0}^{n-1} k_i 2^i \ (k_i = 0, 1)$ と g，p
出力： $y = g^k \bmod p$
`ModBinary1(`k`,` g`,` p`)`
`1.` $y = 1$.
`2.` `for` $i = n-1, \cdots 0$, `do {`
`if` $k_i = 1$, `then` $y =$ `Mod(Mod(`y^2`,p)·`g`,` p`).`
`else` $y =$ `Mod(`y^2`,` p`) }`
`3.` `Output` y

解析 1.3 *(1)* バイナリ法において何回の乗算 *(*法乗算*)* と *2* 乗算 *(*法 *2* 乗算*)*[1] が必要になるか求めよ．ただし, 各ビットに *1* が立つ確率は *1/2* とする．

1. $k = 15$ の場合
2. 160-*bit* 有限体 $\mathbb{F}_p$ 上のランダムな 160-*bit* k を用いる場合．
3. 1024-*bit* 有限体 $\mathbb{F}_p$ 上のランダムな 160-*bit* k を用いる場合．
4. 1024-*bit* 有限体 $\mathbb{F}_p$ 上のランダムな 1024-*bit* k を用いる場合．

(2) 上記の *2-4* の場合の計算量を 160 ビットの乗算の回数で見積もり，評価し比較せよ．但し，見積もりの際には以下を仮定する．

- 160 ビットの乗算の計算量の単位を M_{160} として評価
- 160 ビット*:* 1024 ビット = 1 : 6

[1] n-bit の有限体 $\mathbb{F}_p$ 上の乗算 (法乗算) を実装する際には，n-bit の乗算 (出力結果は $2n$ bits) と $2n$-bit の整数 p を法にした値を求める法演算, すなわち，法乗算が必要である．なお，楕円曲線上の加法公式では実行速度の観点から，乗算の度に法演算を行うとは限らない．このため，楕円曲線上の加法公式では乗算と法演算を分けて考える．また有限体上の逆元 y とは，$\mathbb{F}_p$ の元 x に対して，$xy \equiv 1 \pmod{p}$ かつ $\mathbb{F}_p \ni y$ を求めることを意味する．

- n ビットの乗算の計算量 : mn ビット の乗算の計算量 $= 1 : m^2$
- 加算，減算の計算量は無視
- 乗算の計算量 M : 2 乗算の計算量 $S = 1 : 0.8$, 乗算の計算量 M : 逆元の計算量 $I = 1 : 11$,

演習 1.3 バイナリ法を実装し，以下の $\mathbb{F}_p$ 上のべき乗演算の結果を求めよ．
(1) 表 *D.1* にある p_4, g_4, k_4 を用いて $g_4^{k_4} \pmod{p_4}$ を求めよ．
(2) 表 *D.1* にある p_4, g_4, k_5 を用いて $g_4^{k_5} \pmod{p_4}$ を求めよ．
(3) 表 *D.1* にある p_4, g_5, k_4 を用いて $g_5^{k_4} \pmod{p_4}$ を求めよ．
(4) 表 *D.1* にある p_4, g_5, k_5 を用いて $g_5^{k_5} \pmod{p_4}$ を求めよ．
(5) 上記 *4* つの実行速度を測定する．実測値は 10^3 回実施し，平均値を求める．

解析 1.4 演習 *1.3* に基づき，下記の解析を行う．
(1) 解析 *1.3* の理論値を用いて，*4* つの演算の理論的実行時間を M の回数で求めよう．
(2) (1) により理論値で計算した *4* つの演算時間の比と実際の実測値の比を比較せよ．
(3) 一般に暗号では指数 k_4 や k_5 が秘密情報になり，$g_4, g_5, g_4^{k_4}, g_5^{k_5}$ などから，k_4 や k_5 を求める事が困難であることが安全性の根拠になる．実際には指数の全数攻撃（平方根のオーダ）が最大の安全性となるので，例えば，*160bit* のべき数の場合，鍵の可能性は 2^{160} で，全数攻撃の解読にかかる計算量は 2^{80} と見積もられる．サイドチャネル攻撃とは，実行時間などの情報を測定することで，鍵探索が必要な鍵空間の数を削減する攻撃である．例えば，二乗算は各ビットで必要だが，乗算は *1* が立つビットだけで必要なため，実行時間を測定するとハミング重みを入手出来る．演習 *1.3* の実行時間の測定から k_5 のハミング重みが *15* という情報を利用して，実際に鍵を求める方法を記載せよ．さらにその計算量を乗算量などで評価せよ．なお，k_5 のビットサイズは *157* ビットである．なお，最上位ビットは *1* とする．

2 講義 2 公開鍵暗号 (教科書 3,7,8 章)

命題が真であるとき，対偶は真ですが，逆は必ずしも真になりません．Fermat の小定理に対して，その逆の応用例が確率的素数判定です．数学の各種定理，原理を現実の社会に適用することで，RSA 暗号や素数判定が生まれてます．また，攻撃，解読の形式化についても学びます．完全解読，部分解読，識別可能性，頑強性などの意味を理解しましょう．

2.1 素数・合成数 (教科書 2.4-2.6 章，教科書 3.1 章, 3.8 章)

合成数 n が n と互いに素な任意の自然数 a に対し，$a^{n-1} \equiv 1 \pmod{n}$ を満たすとき，**絶対擬素数** あるいは**カーマイケル数**という．いわゆる，Fermart の小定理を満たす合成数である．演習 2.1 で具体的に絶対擬素数を求めてみよう．

演習 2.1 $2^{15}+1$ から $2^{16}-1$ までの *16* ビットの絶対擬素数を求めよ．ただし，アルゴリズムは次を利用して，効率的に設計すること．

- $a = 2$ あるいは互いに素な素数 $a < \frac{n}{2}$ に対し，$a^{n-1} \equiv 1 \pmod{n}$ を検証.

Fermart の小定理を応用した擬素数判定テスト ($\gcd(a, n) = 1)(1 \le a \le n-1)$ に対して $a^{n-1} \equiv 1 \pmod{n}$ を満たすかどうか判定するテスト) を繰り返して，確率的素数判定ができる．これを**フェルマー法**という．本講義では RSA 暗号を構成するが，鍵生成で素数を生成する必要がある．以下で素数を生成するアルゴリズムを実装しよう．素数を生成するアルゴリズムには確率的素数判定法，確定的素数判定法がある．演習 2.1 のアイデアを利用したフェルマー法で確率的に素数を判定し，その精度を求めてみよう．詳細については教科書を見てください．

演習 2.2 フェルマー法の判定を実行する回数を入力して，確率的に素数を判定するフェルマー法を実装し，以下の値を求めよ．
(1) フェルマー法の判定を実行する回数を *10* 回として，*128, 256, 512, 1024* ビットの各素数 p を小さい方から *10* 個生

成しよう．なお，素数判定は小さい素数から順に *10* 回実施する．
(2) (1) で生成した素数が本当に素数であるか，*python* 標準関数を利用して確認し，素数生成確率を各ビットの場合に対して求めよ．

解析 2.1 フェルマー法の実行回数を理論値から検討してみよう．
(1) フェルマー法の判定を実行する回数を *1* 回から *5* 回に変化させ，それぞれフェルマー法で判定される *128, 256, 512, 1024* ビットの素数 p を小さい方から *20* 個生成して出力しよう．なお，フェルマー法の判定の底は素数で実施し，$2, 3, 5, \cdots$ の順に確認する．さらにそのうち実際に素数になる個数を求めよ．
(2) (1) のフェルマー法の実行回数と正しい素数の生成確率をグラフ化し，フェルマー法を実行する回数と正しい素数の生成確率を実験的に求めよ．

2.2 RSA 暗号

【ユーザの鍵生成】ユーザ B は次のように公開鍵と秘密鍵のペアを生成する．
1. 2 素数 p, q を生成し，$n = pq$ を計算する．
2. $\lambda(n) = \mathrm{lcm}(p-1, q-1)$ に対して，$\lambda(n)$ と互いに素な $e \in \mathbb{Z}^*_{\lambda(n)}$ を生成し，

$$ed \equiv 1 \pmod{\lambda(n)}$$

を満たす d を求める．
〈 公開鍵 〉$e,\ n$
〈 秘密鍵 〉d

【暗号化】ユーザ A が平文 $m \in \mathbb{Z}_n$ を暗号化して B に送るとする．A は B の公開鍵を用いて

$$c = m^e \bmod n \tag{1}$$

を計算し，暗号文 c を B に送信する．
【復号】暗号文 c を受信した B は以下のようにして m を復号する．

$$m = c^d \bmod n \tag{2}$$

を得る．

事前問 1 $p = 5, q = 11, n = 55$ を用いて，*RSA* 暗号の公開鍵 $e = 3$ に対する秘密鍵を作れ．次に，公開鍵 $e = 3$ を用いて，$m = 3$ を暗号化した結果を求めよ．

演習 2.3 *RSA* 暗号の鍵生成アルゴリズム，暗号化アルゴリズム，復号アルゴリズムを実装する．*RSA* 暗号の鍵生成で用いる素数生成は演習 *2.2* を利用する．なお，入力はビット数とし，初期値は乱数生成関数で求めること．*API* に沿って作成してください．また，課題では，表 *D.9*，*D.10*，*D.11* の *3* 種類の剰余環 $\mathbb{Z}_n$ と素数 p, q, 公開鍵 e 及び秘密鍵 d を利用する場合を考える．表 *D.9* については (e_{1-1}, d_{1-1}) を利用する．
(1) 各剰余環を用いて，表 *D.2* の 平文 m_1 *(*157 ビット*)* を暗号化した結果を求めよ．また復号できることを確かめよ．
(2) 各剰余環を用いて，各表の平文 $m = a_1, a_2, a_3$ *(*1023 ビット*)* を暗号化した結果を求めよ．

演習 2.4 この問題では，*RSA* 暗号の鍵を構築し，実際に秘匿通信を実施する．
(1) 1024 ビットの自分の *RSA* の公開鍵，秘密鍵を生成し，公開鍵 (n, e) を公開鍵掲示板に掲載しよう．
(2) 担当 *TA* の公開鍵を用いて，以下のどれかの文章を暗号化し，送付しよう．
(a)"Security is main technology for DX."
(b)"Security makes reliable society like traceability of food."

(c)"White hacker is to protect our life from attacker."
なお，*RSA* 暗号は確定的暗号のため，同じ平文の暗号文は同じになる．そこで，各文章の最後に，各自の名前のアルファベットを入れよう．例えば，宮地の場合，アルファベットは *AM* なので，第三の文章を選んだ場合，*"White hacker is to protect our life from attacker by AM."*
となる．*(3)* 復号できたかどうか，*TA* からの返信を確認しよう．*TA* は復号して，*(a)-(c)* のどの文章が暗号化されたか報告してください．

2.3 RSA 暗号の安全性

解析 2.2 (共通法 RSA) *(1)* センターが素数 p, q を秘密に生成し，各ユーザに同じ n を用いた鍵 (e_i, d_i) を配送する方式（共通法 *RSA)* が考えられる．このような方法を利用すると，ある状況において，平文が露呈することを示せ．ここで，各 e_i は互いに素とする．
(2) 表 *D.9* に記載された (n_1, e_{1-1}, d_{1-1}), (n_1, e_{1-2}, d_{1-2}) を用いて暗号化された暗号文

- C_1 = 502576281868419460227447436395560724457976156124316069417118541776714583706110641434048653926894624201334252816047597588984253062261701345492212502532200704312043686622709145321441807923582285535762004881260668768566160362413032353009226035499420680766413461088064426327623822928777185010162824325151274179 49,
- C_2 = 115870729276547586100790621990585092388432886948198758418452055567152196111599860714154493285123211533502961469101571653485618588083828957179608776225135414877457840038659991564339024558687394498346303682373233367132449651862321750838840689347118499374888579761677651619571303928496183017774390146211061621583

を *(1)* の方法を用いて解読してみよう．なお，解読に用いたプログラムと解答の両方を提出すること．

問 2.1 (識別不可能性) *RSA* 暗号は直接攻撃で識別不可能性を満たさないことを示せ．

問 2.2 *RSA* 暗号は適応的選択暗号文攻撃で完全解読できることを以下のステップで証明しよう．つまり，任意の暗号文 C^* が与えられたときに，次の方法でこの暗号文を解読して平文に復号しよう．このとき，各ステップの実行に必要な計算量を求めることで解読に必要な計算量を求めよう．ただし．復号オラクルへの質問は *1* 回とする．オラクルの計算量は考えない．

1. 復号オラクルに質問する暗号文 C を求めよ．
2. 復号オラクルが出力する平文 m を用いて，暗号文 C^* の復号文 m^* を求めよ．
3. $C \neq C^*$ である確率を求めよ．
4. 上記 *1-2* のステップの計算量とステップ *3* の確率から，適応的選択暗号文攻撃で平文 m の復号に必要な計算量を求めよ．計算量は *RSA* を定義する剰余環 $\mathbb{Z}_n$ 上の乗算 M の回数で換算して表す．なお，剰余環上の演算（乗算，*2* 乗算，除算，加減算）の換算式は以下を仮定する．
   - *n bits* の乗算の計算量 *:* *mn bits* の乗算の計算量 $= 1 : m^2$.
   - 加減算と小さい定数との乗算の計算量は無視する．
   - 乗算の計算量 M *:* *2* 乗算の計算量 $S = 1 : 0.8$. 乗算の計算量 M *:* 除算の計算量 $I = 1 : 11$.

解析 2.3 (RSA 暗号の選択暗号文攻撃) *(1)* ユーザの公開鍵を n, e に対して，以下の暗号文 c を入手した．

- n = 101120238836005269809773416981882641266011205065504744716304420651224630140594291762148518493854239011880064057427658763683069244958109350152226679734726981532071377495257561523404449361194579124404455933093256937645064896786745363916150910048949973279743871403067725803396280564233743617093851921899467143641,
- e = 2147483649,

- C = 32895882594000591593742797771500149218003022550037079603315093491447486158015262263141363034156680610271069043198422693708877809022939649632404113517463590100479550135721182799758485945378174083159415425688127829993219211900937547432976167601915191890894997138701599385762943753524662463006179528342436295291

このとき，復号オラクルに一度暗号文を聞ける状況で，暗号文を解読しよう．各人の復号オラクルは公開鍵掲示板に暗号文を掲載すると，その復号結果を入手出来る．

3 講義 3 ディジタル署名 (教科書 7, 9 章)

3.1 RSA 署名とその安全性

公開鍵暗号の重要な応用は署名である．署名は様々な場面で利用される．例えば，電子選挙で，自分の投票する人の名前を記載して，選挙管理委員会に署名をしてもらう必要がある．このとき，投票者の名前は隠して署名してもらう必要があるので，ブラインド署名が必要になる．つまり，ブラインド署名とは中身を見せずに相手に署名をしてもらう概念である．RSA 署名を用いたブラインド署名ではメッセージを隠すために，乱数 r を用いるが，r と n は互いに素である必要がある．これは署名抽出のときに，r の逆元が必要になるからだ．ここで，乱数が n と互いに素にならない確率はどれくらいあるのだろうか？一般に，$n = pq$ で p, q が 500 ビット以上の素数であることから，ほぼ起こらないことがわかる．とはいえ，確率的には互いに素でない乱数 r をとることもある．その時はどうなるだろうか？少し考えてみよう．

RSA 暗号は復号の機能を署名生成，暗号化の機能を署名検証に応用することでディジタル署名を構築できる．まずは RSA 署名の攻撃について考えてみよう．

問 3.1 *RSA* 署名において，ハッシュ関数が衝突困難性を持たないとする．つまり，攻撃者　A はハッシュ値が一致する異なるメッセージを入手できるとする．このとき，選択平文攻撃で　*RSA* 署名は存在的偽造可能であることを示せ．

3.2 ブラインド署名

ブラインド署名とは，あるユーザ A が文章 m の中身を　B に見せることなく，B の署名をもらう方式である．RSA 署名の重要な応用例である．ブラインド署名は電子選挙や電子現金を実現するのに利用される．ここでは，RSA 署名の応用としてブラインド署名を紹介する．

【ユーザの鍵生成】ユーザ V は次のように公開鍵と秘密鍵のペアを生成する．

1. 2 素数 p, q を生成し，$n = pq$ を計算する．
2. $\lambda(n) = \mathrm{lcm}(p-1, q-1)$ に対して，$\lambda(n)$ と互いに素な $e \in \mathbb{Z}^*_{\lambda(n)}$ を生成し，

$$ed \equiv 1 \pmod{\lambda(n)}$$

を満たす d を求める．

3. さらにハッシュ関数

$$H : \{0,1\}^* \longrightarrow \mathbb{Z}_n \setminus \{0\}$$

を公開する．〈 公開鍵 〉$e,\ n$
〈 秘密鍵 〉d

【メッセージの秘匿化】ユーザ V はメッセージ $m \in \mathbb{Z}_n \setminus \{0\}$ に対して，$\gcd(r, n) = 1$ となる乱数 $r \in \mathbb{Z}_n \setminus \{0\}$ を 生成し，

$$t = H(m)r^e \bmod n \tag{3}$$

を計算し，$t \in \mathbb{Z}_n \setminus \{0\}$ を署名者 A に送信する．

【ブラインド署名生成】署名者 A はブラインドされたメッセージ $t \in \mathbb{Z}_n \setminus \{0\}$ に以下のように署名する．

$$\tau = t^d \bmod n \tag{4}$$

署名文 $\tau \in \mathbb{Z}_n \setminus \{0\}$ を V に送信する．

【署名抽出】ユーザ V は署名文 $\tau \in \mathbb{Z}_n \setminus \{0\}$ からメッセージ $m \in \mathbb{Z}_n \setminus \{0\}$ に対する署名 $\sigma \in \mathbb{Z}_n \setminus \{0\}$ を抽出する．

$$\sigma = \tau / r = H(m)^d \bmod n \tag{5}$$

実際，σ　は以下を満たすことから，m の署名となる．

$$H(m) = \sigma^e \bmod n \tag{6}$$

問 3.2 *RSA* ブラインド署名を応用する事例を考えよう．事例では，どのようなユーザを考えるのか，さらに誰がどんな文章に対して誰のブラインド署名を依頼するのかまで記載しよう．

演習 3.1 *RSA* 署名の鍵生成アルゴリズム，署名生成アルゴリズム，検証アルゴリズムを実装する．なお，ハッシュ関数は *shake128* を利用する．なお，ハッシュの出力は *127* バイト *(1016bits)* とする．表 *D.9* の剰余環 $\mathbb{Z}_n$ と素数 p, q, 公開鍵 e 及び秘密鍵 d を利用する場合を考える．表 *D.9* の鍵 $A = (e_{1-1}, d_{1-1})$ について，実施する．なお署名生成と検証の処理時間は 10^3 回実施し，平均値と分散値[2]を求める．*G.H.Hardy* の *"A Mathematicians Apology", 1940* に記載された文章に署名を送付しよう．数論がまさにセキュリティの中心的な理論になることを *Hardy* は想像してなかっただろう．

"Both Gauss and lesser mathematicians may be justified in rejoicing that there is one science, number theory, at any rate, and that their own, whose very remoteness from ordinary human activities should keep it gentle and clean."

(0) メッセージのハッシュ値を求めよ．

(1) A を用いて，*(0)* に署名生成した結果を求めよ．

(2) (1) の署名生成の処理時間を提出せよ．

解析 3.1 *RSA* 署名/暗号では復号に必要な情報は d のみで，n の素因数は不要である．つまり，(n, e, d) のみで暗号化と復号を実現できる．そこで，センターは素数 p, q を生成し，その合成数 $n = pq$ の素因数は秘密にし，同じ n に対して複数の人の公開鍵と秘密鍵 $\{(e_i, d_i)\}$ を構築して利用することが可能である．このとき，安全なだろうか?　異なる *2* 人のユーザが同一の合成数 n 表 *D.9* に同じ合成数 n を利用した *2* 種類の鍵 $A = (e_{1-1}, n_1)$,$B = (e_{1-2}, n_1)$ を持ち，暗号化した同じ契約書の文面，c_1, c_2 を送付したとする．

(1) 2 種類の鍵の情報を用いて，不定方程式 $e_{1-1}x + e_{1-2}y = 1$ の解 (x, y) を求めよ．

(2) もとの契約書の文面を求めよ．

演習 3.2 *RSA* 署名の鍵生成アルゴリズム，署名生成アルゴリズム，検証アルゴリズムを実装する．なお，ハッシュ関数は *shake128* を利用する．なお，ハッシュの出力は *127* バイト *(1016bits)* とする．各大学の *TA* の公開鍵が *moodle* に掲載されている．*RSA* 暗号で暗号化する．*RSA* 暗号の公開鍵は阪大通信 *A* は *Nas,* 阪大情シス *A* は前野，阪大情シス *B* は和泉，阪大情シス *C*，静岡大，長崎県立大学は上杉，阪大電気，量子，三重大は金，石川高専は秀，上記以外は奥村先生の公開鍵で暗号化しよう．なお，暗号化できるビット長は *RSA* が *1024*[3]ビットなので，*42* 文字までの感想文を作成する．本講義の感想を書き，それぞれの公開鍵で，暗号化して提出する．さらに，感想に署名を作成して，提出する．各人の鍵は表 *D.10* の剰余環 $\mathbb{Z}_n$ と素数 p, q, 公開鍵 e 及び秘密鍵 d を利用する．

4　データ秘匿と完全性 (教科書 7 章)

4.1　OAEP (Optimal Asymmetric Encryption Padding) の基本概念

既存の RSA 暗号，Rabin 暗号，ElGamal 暗号は，適応的選択暗号文攻撃のもとで識別不可能性を満たさない．RSA-OAEP 暗号 **[?]** は適応的選択暗号文攻撃に対して安全に RSA 暗号を変換した方式である．**OAEP**(Optimal Asymmetric

[2]分散値の計測は入力によるばらつきの確認のために有効である．

[3]平仮名や漢字だと，1 文字 3 バイトに相当する．

Encryption Padding) は RSA 暗号に限らず，他の方式のパディング方法にも利用できるので，ここでは一般的に記載する．

【初期設定】公開鍵暗号を (Gen, Enc, Dec) とし，Enc の定義域のサイズ，つまり，平文空間のサイズを $k > k_2$ バイト，0 パディングのサイズを $k_1 > 0$ バイトとする．ここで，$k = k_0 + k_1 + k_2$ となる k_0 をとる．またハッシュ関数を公開する．

$$\begin{aligned} G: \quad & \{0,1\}^{8k_0} \longrightarrow \{0,1\}^{8(k_2+k_1)} \\ H: \quad & \{0,1\}^{8(k_2+k_1)} \longrightarrow \{0,1\}^{8k_0} \end{aligned}$$

【ユーザの鍵生成】ユーザ B は公開鍵暗号 (Gen, Enc, Dec) を用いて，秘密鍵 SK と公開鍵 PK を生成する．

【暗号化】ユーザ A が平文 $m \in \{0,1\}^{8k_2}$ を暗号化して B に送るとする．
1. 乱数 $r \in \{0,1\}^{8k_0}$ に対して，

$$\begin{aligned} s &= G(r) \oplus (0^{8k_1} \parallel m) \\ t &= H(s) \oplus r \\ w &= t \parallel s \\ \mathsf{cipher} &= \mathsf{Enc}(PK, w) \end{aligned} \tag{7}$$

を計算する．
2. cipher を暗号文として B に送信する．

【復号】
1. B は自分の秘密鍵を用いて，

$$\begin{aligned} w &= \mathsf{Dec}(SK, \mathsf{cipher}) \\ s &= [w]_1^{k_1+k_2} \\ t &= [w]_{k_1+k_2+1}^{k} \\ r &= H(s) \oplus t \\ z &= G(r) \oplus s \\ m &= [z]_1^{k_2} \\ \mathsf{chk} &= [z]_{k_2+1}^{k_2+k_1} \end{aligned} \tag{8}$$

を計算する．ここで $[w]_1^{k_1+k_2}$ は w の 1 から $k_1 + k_2$ バイト目，$[w]_{k_1+k_2+1}^{k}$ は w の $k_1 + k_2 + 1$ から k バイト目を意味する．
2. $\mathsf{chk} = 0^{8k_1}$ ならば，m を復号文として出力し，それ以外の場合，エラーを出力する．

4.1 章では OAEP の基本概念について説明した．実際に RSA-OAEP 実用化する際には，ハッシュ関数や m のパディング手法など厳密に規格化されている [**?**]．また，一般に利用されるハッシュ関数は SHA1 などに見られるように出力のサイズが固定値のため，OAEP のように出力サイズが可変となる関数を既存のハッシュ関数から構築する必要がある．これが **MGF**(Message Generation Function) と呼ばれる関数である．

IETF の RFC3447[**?**] においては，既存のハッシュ関数の出力バイト[4]数を k_0, 平文 m のバイトサイズを k_2，RSA の法 n のバイトサイズを k, 0 パディング列 PS のバイトサイズを k_1 とし， 以下のように平文にパディングを設定する．ここで $k = 2k_0 + k_1 + k_2 + 2$ である．ここでは規格化されている RSA-OAEP の簡易版を記載する．詳細は RFC3447[**?**] を参照されたい．

【初期設定】

1. 公開鍵暗号を (Gen, Enc, Dec) とし，Enc の定義域のサイズを k バイト，平文空間のサイズを $k > k_2$ バイト，0 パディングのサイズを $k_1 > 0$ バイトとする．

[4] 8 ビットのことであるが，オクテット (octet) ともいう．

2. **MGF** (message generation function) を以下で定める．

$$\begin{aligned} G_{\mathtt{MGF}}: &\quad \{0,1\}^{8k_0} &&\longrightarrow \{0,1\}^{8(k_0+k_1+k_2+1)} \\ H_{\mathtt{MGF}}: &\quad \{0,1\}^{8(k_0+k_1+k_2+1)} &&\longrightarrow \{0,1\}^{8k_0} \end{aligned}$$

3. 利用するハッシュ関数で決まる固定値を lHASH とする．SHA1 の場合，下記となる．

$$\text{lHASH} = 1245845410931227995499360226027473197403882391305$$

4. 0 パディング列 PS を以下とする．PS は平文 m のサイズにより $\perp$(0 バイト) になることもある．

$$\text{PS} = (0\text{x})00^{k_1}$$

【暗号化】ユーザ A が平文 $m \in \{0,1\}^{k_2}$ を暗号化して B に送るとする．
1. 乱数 $r \in \{0,1\}^{k_0}$ に対して，平文 m のパディング w を以下のように計算し，暗号関数に入力する．

$$\begin{aligned} \text{Pad}(m) &= \text{lHASH} \,\|\, \text{PS} \,\|\, (0\text{x})01 \,\|\, m && (9) \\ s &= G_{\mathtt{MGF}}(r) \oplus \text{Pad}(m)\ (r \in \{0,1\}^{8k_0}) \\ t &= H_{\mathtt{MGF}}(s) \oplus r \\ w &= (0\text{x})00 \,\|\, t \,\|\, s \\ \mathsf{cipher} &= \mathsf{Enc}(PK, w) && (10) \end{aligned}$$

【復号】
1. B は自分の秘密鍵を用いて，

$$\begin{aligned} w &= \mathsf{Dec}(SK, \mathsf{cipher}) && (11) \\ s &= [w]_1^{k_0+k_1+k_2+1} \\ t &= [w]_{k_0+k_1+k_2+2}^{k-1} \\ r &= H(s) \oplus t \\ z &= G(r) \oplus s \\ m &= [z]_1^{k_2} && (12) \end{aligned}$$

を計算する．ここで $[w]_1^{k_0+k_1+k_2+1}$ は w の 1 から $k_0+k_1+k_2+1$ バイト，$[w]_{k_0+k_1+k_2+2}^{k-1}$ は w の $k_0+k_1+k_2+2$ から $k-1$ バイトを意味する．
2. 以下の条件をチェックし，OK なら m を復号文として出力し，それ以外の場合，エラーを出力する．

- (11) の w の 最上位 1 バイトが 0 である．
- (12) が Pad(m) のフォーマット (9) を満たしている．

演習 4.1 RFC3447 に沿って，*OAEP* で *RSA* 暗号を用いる関数を構成しよう．
(1) $k_0 = 160$ *bits (20 bytes),* $k_1 = 560$ *bits(70 bytes),* $k_2 = 128$ *bits(16 bytes)* として *128 bits* のデータを *OAEP* パディングを行い，*RSA* 暗号・復号を行う関数を作成する．
(2) 表 *D.8* の 128 *bits* のデータ K を暗号化した結果を求めよ．また復号できることを確かめよ．なお，*RSA* 暗号の公開鍵 n, e，秘密鍵 p, q, d は *2048* ビット表 *D.18* のデータを用いる．また *OAEP* 用の乱数は表 *D.5* のデータを用いる．

4.2 ハイブリッド暗号

SSL 通信等では鍵共有に公開鍵暗号を利用し，公開鍵暗号で共有した鍵を用いてデータの暗号化を行う．以下の演習で実際に秘匿通信を実装してみよう．

演習 4.2 公開鍵 PK と平文 M の入力に対し，*128* ビット乱数 K を生成し，*RSA* 暗号を用いて K を暗号化し，同時に K を用いて *AES* で平文 M を暗号化する関数を構成しよう．なお，*OAEP* によるパディングは演習 *4.1* と同じ k_0, k_1, k_2 の設定とする．
(0) 1024 ビットの素数 p, q を生成し，自分の公開鍵と秘密鍵を構築し，公開鍵を掲示板に提出しよう．
(1) 128 ビットの乱数 K を生成し，*RSA* 暗号で暗号化する．*RSA* 暗号は公開鍵は阪大通信 *A* は *Nas,* 阪大情シス *A* は前野，阪大情シス *B* は和泉，阪大情シス *C*，静岡大，長崎県立大学は上杉，阪大電気，量子，三重大は金，石川高専は秀の公開鍵，上記以外は奥村先生の公開鍵を利用する．また *OAEP* 用の乱数 $r \in \{0,1\}^{8k_0}$ は表 *D.5* のデータを用いる
(2) (1) で暗号化した K を *AES* の鍵として，次の文章 *(bit coin* の提案論文のアブストラクト*)* から気に入った技術用語 *16* 文字まで）を暗号化して，署名をしてみよう．
“A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.”
(3) (2) の文章を自分の秘密鍵を用いて署名しよう．

参考文献

A Python の 基本操作

Python を使用するにあたって最低限必要と思われる機能及び利用する組み込み関数を紹介する．また本書では，windows 10 上で Anaconda を利用する前提で説明する．

A.1 Windows 上での Python の利用について

Python は，デフォルトでは Windows にインストールされておらず，別途入手してインストールする必要がある．本章では,「Anaconda 4.4」(https://www.continuum.io/) をインストールすることを想定している．Anaconda は修正 BSD ライセンスで頒布されており無償利用が可能である．Anaconda は Python ディストリビューションの一つで，以下のようなライブラリのほかに，対話的にに Python を実行する「IPython」Python プログラムの記述と実行，メモの作成などをブラウザ上で行なう「Jupyter Notebook」，統合開発環境の「Spyder」などが自動的にインストールされる利点がある．

- NumPy：数値データの作成と操作に関するライブラリ
- SciPy：科学技術計算に関するライブラリ
- SymPy：代数計算に関するライブラリ

以降では，Python のバージョンは 3.6 として説明する．Python を利用するには以下ような方法がある．

スクリプト言語としての Python: Perl や Ruby と同様の使い方であり，テキストエディタを用いて「***.py」というファイルを作成し，コマンドプロンプトで実行する．[スタート] - [Anaconda3] - [Anaconda Prompt] でコマンドプロンプトを起動し,「python ***.py」を入力して実行する．コンパイルは不要である．

IPython: Python を対話的 (Interactive) に実行するシェルである．[スタート] - [Anaconda3] - [IPython] で実行し，一行ずつ入力して出力を得ることができる．エディタを使用しないため，簡易的な計算を行うのに向いている．

Jupyter Notebook: ウェブブラウザ上で Python を実行・表示できるツールである．ノートのように扱うことができ，式と答えが表示された形式のまま保存できる利点がある．ただし，これで保存されたファイルはノートブック形式であり,「***.ipynb」となる．つまり，このファイルは Jupyter Notebook 以外で実行できない．

A.2 Anaconda のインストール

Anaconda のインストール方法は https://docs.continuum.io/anaconda/install/windows に記載されている．Anaconda 4.4 時点での手順を以下に示す．

(1) Anaconda を新規にインストールする

1. https://www.continuum.io/downloads からインストーラをダウンロードする．ここでは，Python 3.6 バージョンをダウンロードする．
2. ダウンロードしたインストーラを起動する．
3. Next をクリックする．
4. ライセンスを確認し，”I Agree”を押す．
5. インストール対象ユーザーを選ぶ．ここでは，”Just Me” を選択して ”Next” を押す．

6. インストールフォルダを選択して ″Next″ を押す．デフォルト設定でよい．
7. Python の環境変数の設定，Python 3.6 をデフォルトの Python として登録するかを問われるが，デフォルト設定で良い．
8. ″Install″ を押すとインストールが始まる．
9. ″Finish″ を押し，インストーラを閉じる．

(2) 既にインストールしている Anaconda に Python3 系を導入する

1. 「スタート」 -「Anaconda3」 -「Anaconda Prompt」からコマンドプロンプトを起動する．
2. 「conda create -n py36 python=3.6 anaconda」と入力して，Enter を押す．
3. 「Proceed ([y]—n)?」と聞かれるので ″y″ を押す．
4. インストール完了後,「activate py36」と入力して Enter を押すと，Python3.6 環境が利用できる．
5. 元のバージョンの環境に戻したい場合は,「deactivate」と打ち込んで非アクティブ化することができる．
6. 詳細は https://conda.io/docs/using/envs.html#share-an-environment を参照．

A.3 Python プログラムの実行方法

ここでは，プログラムファイルの実行と対話実行の二種類について解説する．

(1) Python プログラムファイルの実行

1. プログラムをテキストエディタ等で作成し，拡張子 ″.py″ で保存する．ここでは，作成したプログラムのパスを ″`C:\work\test.py`″ と仮定する．
2. 「スタート」 -「Anaconda3」 -「Anaconda Prompt」からコマンドプロンプトを起動する．
3. コマンドプロンプト上で「cd /d `C:\work`」と入力して Enter を押すと，作業フォルダへ移動できる．
4. 「python test.py」と入力し Enter を押すとプログラムが実行される．

A.4 Python プログラムの対話実行

1. 「スタート」 -「Anaconda3」 -「Anaconda Prompt」からコマンドプロンプトを起動する．
2. 「python」と入力し Enter を押すと，Python 対話モードが起動する．
3. 実行したい Python のステートメントを記述して Enter キーを押すと実行される．

A.5 基本的な文法

if 文

C 言語と比較して注意点が 3 箇所ある．1 つは各条件文の最後にコロン「：」が必要なこと，2 つ目は elif となっていること，さらに条件文が真のときに実行される範囲はインデントされている範囲だけである．if-elif-else の分岐処理のサンプルコードを以下に記す．

```
n = 0
if n < 0:
   print('n<0')
elif n == 0:
   print('n==0')
else:
   print('n>0')
```

結果：n==0

for 文

Python の for 文は C 言語と少し異なり，書式は次のようになる．インデントされている範囲だけ繰り返されることに注意する．

```
for 変数 in オブジェクト:
   実行する処理 1
   実行する処理 2
   ⋮
```

通常のカウンタを取る for 文のサンプルコードを以下に記す．range 関数は連番でリストを作成する関数であり，range(5) は 0≤x<5 となる整数 x のリストを作成する．

```
a = 0
for i in range(5):
   a = a + i
print(a)
```

結果：10

while 文

書式は次のようになる．for 文と同様，インデントされている範囲だけ繰り返されることに注意する．

```
while 条件式:
   条件式が真の時に実行する処理 1
   条件式が真の時に実行する処理 2
   ⋮
```

while 文のサンプルコードを以下に記す．

```
a = 4
b = 0
while a > 0:
   b = b + a
   a = a - 1
print(b)
```

結果：10

リスト

リストとは任意のオブジェクトのシーケンスであり，その要素は 0 から始まる整数で番号付けされる．リストを作るには [ ] を使い，リストにはどのような値も持たせることができる．例えば enc=[’RSA’,[’AES’,’DES’,’RC4’]] のように書く．このとき，enc[0] が RSA，enc[1] が [’AES’,’DES’,’RC4’] である．さらに，enc[1][2] は RC4 である．リストのサイズは len で調べることができる．例えば，len(enc) は 2 である．

```
# リストの作成 a = [] # 空リストの作成
b = [’RSA’,’AES’,’DES’] # 複数要素からなるリスト作成
c = [’RSA’,[’AES’,’DES’,’RC4’]] # リストの入れ子
len(b) # リストの要素数 3

array = [1, 2, 3, 4]
# 任意の要素を取り出す
first = array.pop(0) # first == 1, array == [2, 3, 4]
# 任意の要素を追加する
array.insert(0, 5) # array == [5, 2, 3]
# 末尾を取り出す
last = array.pop() # last == 3, array = [5, 2]
# 末尾に追加
array.append(9) # array == [5, 2, 9]
# 末尾にリストを追加
array.extend([0, 1]) # array == [5, 2, 9, 0, 1]
```

変数の初期化

Python では変数は必ず初期化して使うものと想定されている．つまり，暗黙の初期値が存在しない．初期化すべき値がない場合には「null」に相当する「None」を代入する

多倍長演算

Python では，Perl や Ruby と同様に，整数演算で値の範囲が固定長の範囲を越えるものを，自動的に多倍長整数に変換する．

print 関数

プログラム実行中にコンソール画面に文字列を表示させるには `print()` を用いる．また，文字列中に別の文字列を挿入するには，`format()` を用いる．次の `print()` 関数の呼び出しでは，いずれも ”Hello, world.”を表示する．

```
s1 = ”Hello, world.”
s2 = ”Hello, {}.”.format(”world”)
a = ”Hello”
b = ”world”
s3 = ”{}, {}.”.format(a, b)
print(”Hello, world.”)
print(s1)
print(s2)
print(s3)
```

`format()` は数値に対しても有効である．次のプログラムを実行すると 「2 times 3 equal 6」と表示される．

```
print(”{} times {} equal {}”.format(2,3,6))
```

コメント

一行コメント： #から行末までがコメントアウトされる．

```
print(”abc”) # コメント

```

複数行コメント： 3 重クオート文字列で囲うと，その内部がコメントアウトされる．

```
”””
この中は
コメント
”””
```

数値

A.6 パッケージについて

パッケージから利用する関数は下表の通りである．プロセッサ時刻を求める関数 clock() は time パッケージに含まれる．そのため，clock() を使用する際には，time パッケージをあらかじめ import しておく必要がある．下表は，本稿で利用するパッケージ及びその組み込み関数名を記載している．ただし，chi2 は単純に import することはできず，from を利用する import 方法をとる必要がある．つまり，import の際には「from scipy.stats import chi2」と書く．この場合，chi2 を使用する際は from と import の間に書いた部分「scipy.stats」をプログラム内で省略できる．

パッケージ名	組み込み関数名
sympy	gcd, invert, factorint, randprime
Base64	b64encode, b64decode
binascii	hexlify, unhexlify
os	urandom
random	seed, randrange, getrandbits
hashlib	sha1, sha224, sha256, sha384, sha512, shake_128, shake_256
pickle	dump, load
csv	writer, reader
chi2 (in scipy.stats)	ppf
time	clock, perf_counter

以下にサンプルコードを記す．関数 gcd() を使用する際には sympy.gcd() と書く．ただし，import は一度でよい．

```
import sympy
sympy.gcd(6, 9)
```

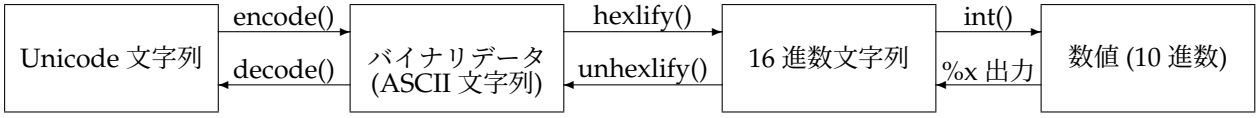
結果：3

A.7 演算における注意点

Python では，ビット演算以外の演算は 10 進数の数値で行う必要がある．しかし，乱数生成関数である os.urandom(n) の出力はバイナリデータで得られる．そのため，バイナリデータを 10 進数の数値に変換した後に演算を行わなければならない．バイナリデータを 10 進数の数値に変換するためには，下図の通り，まずバイナリデータを 16 進数の文字列に変換する（バイナリデータは 16 進数の文字列とバイト単位で対応）．その後，16 進数の文字列を 10 進数の数値に変換する．

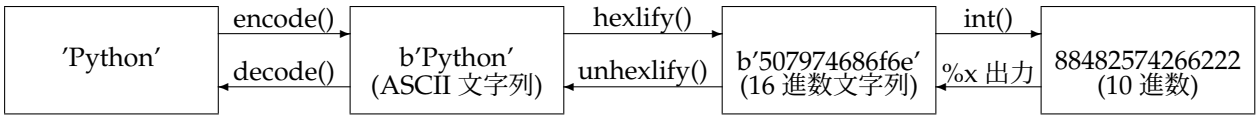
16 進数文字列は単なる中継の役割ではなく，主なメリットが 2 つある．1 つは，バイナリデータをバイト単位で可視化できる点であり，もう 1 つは文字列として保存する際に 10 進数に比べてかなり短くできる点である．もちろんバイナリデータで保存するのが最もデータサイズを短くできるが，テキストデータの方が扱いやすいという場合がある．

なお, Python 3 から 'Python' といったテキスト (Unicode) 文字列はバイト型ではなく str 型で扱われるため, そのような文字列を整数値などに変換するには, encode() を用いて ASCII 文字列 (バイト型) に変換する必要がある. 逆に, ASCII 文字列をテキスト文字列に変換するには decode() を用いる.

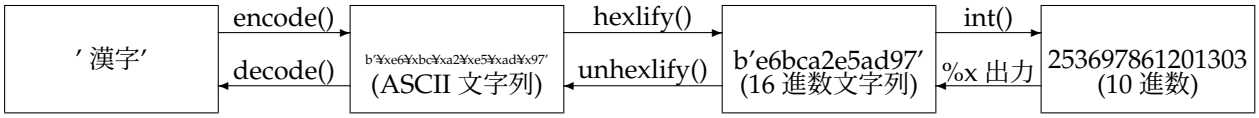


A.8 文字列における注意点

16 進数文字列というのはあくまでもオリジナルのデータをバイト毎に可視化している文字列に過ぎないということに注意する．例えば'Python' という文字列（内部ではバイナリデータとして扱われる）は，16 進数文字列では「707974686f6e」であり，10 進数の数値では「123666946355054」である．つまり，オリジナルのデータである'Python' から見ると，16 進数文字列は単なる 16 進数で表現された文字列であり，これをオリジナルデータの文字列として関数に入力してはいけない．x='Python' の x にはバイナリデータが入っていることを考えると，オリジナルデータの文字列というのは下図の一番左に位置するバイナリデータを指すと思って差し支えない．



文字列から変換した整数値は, 文字列がアルファベットの場合, 1 文字につき 1 バイトであるが, 文字列が漢字やひらがな, カタカナの場合は, 1 文字につき 3 バイトとなる.



A.9 組み込み関数

Enthought Python Distribution でデフォルトで組み込まれている最低限必要な関数を例を用いて紹介する．

商の計算

100 を 13 で割った商の計算は次のように入力する．

```
100 // 13
```

結果：7

mod の計算

100 (mod 13) = 9 の計算は次のように入力する．

```
100 % 13
```

結果：9

べき乗法演算

2^{10} (mod 13) = 10 の計算は次のように入力する．

```
pow(2, 10, 13)
```

結果：10

最大公約数 (GCD)

12 と 30 の最大公約数の計算は次のように入力する．ただし，sympy の import が必要．

```
sympy.gcd(12, 30)
```

結果：6

逆元演算

法 13 における 7 の逆元演算は次のように入力する．ただし，sympy の import が必要．

```
sympy.invert(7, 19)
```

結果：11

素因数分解

素因数分解は次のように入力すると，結果が小さい素数でソートされリスト形式で返される．例えば，96525 を素因数分解すると $3^3 \times 5^2 \times 11^1 \times 13^1$ となる．最も大きな素因数を取り出したい場合は，例のように，len() を使ってリストの長さを導出し，リストの最後の素因数を取り出せばよい．ただし，sympy の import が必要である．

```
v = sorted(sympy.factorint(96525).items())
print(v)
n = len(v)
print(v[n-1])
```

結果：[(3, 3), (5, 2), (11, 1), (13, 1)], (13, 1)

文字列 (10/16 進数) から 10 進数の数値への変換

`int()` を使用すると，次のように 10 進数の文字列を 10 進数の数値に変換できる．

```
int('14') #→ 14
int('84') #→ 84
```

さらに，`int()` の 2 番目の引数に基数を指定できる．例えば，14 を 16 進数の文字列として捉えて 10 進数の数値に変換すると 20 である．

```
int('14', 16) #→ 20
int('a', 16) #→ 10
```

10 進数の数値から文字列 (10/16 進数) への変換

10 進数の数値を 10 進数の文字列に変換するサンプルコードを次に記す.

```
val = 255
s = ″%d″ %val
print(s)
```

結果：255

さらに，10 進数の数値を 16 進数の文字列へ変換するサンプルコードを次に記す.

```
val = 255
s = ″%x″ %val
print(s)
```

結果：ff

バイナリデータと 16 進数の文字列の相互の変換

```
binascii.hexlify(data)
binascii.unhexlify(hexstr)
```

`hexlify()` はバイナリデータを 16 進数の文字列に変換し，逆に `unhexlify()` は 16 進数の文字列をバイナリデータに変換する．次にサンプルコードを示す.

```
import binascii
v1 = binascii.hexlify('Python'.encode())
if (len(v1)%2 == 0):
   v2 = binascii.unhexlify(v1)
else:
   v2 = binascii.unhexlify('0'+v1)
print(v1)
print(v2)
```

結果 1：′507974686f6e′， 結果 2：′Python′

バイナリデータと 16 進数文字列はバイト単位で対応するため，`unhexlify()` の入力は偶数個の 16 進数文字列でなければならない．そこで，上記サンプルコードに示されているように，奇数個の場合は 16 進数文字列としての 0 をパディングする.

16 進数文字列の一部を取り出す

ハッシュ関数 SHA1 の出力 20 バイトに対して，上位の 10 バイトと下位の 10 バイトに分けて取り出すには次のように入力する．ただし，16 進数文字列は 0〜9 及び a〜f の中の 2 文字で 1 バイトを表すことに注意する.

```
import hashlib
v = hashlib.sha1('Python'.encode()).hexdigest()
v1 = v[0:20]
v2 = v[20:40]
print(v1)
print(v2)
```

結果 1：′6e3604888c4b4ec08e28′， 結果 2：′37913d012fe2834ffa83′

16 進数文字列の連結

「+」を利用すると，次のように簡単に文字列の連結ができる.

```
v1 = '6e3604888c4b4ec08e28'
v2 = '37913d012fe2834ffa83'
v = v1 + v2
print(v)
```

結果：'6e3604888c4b4ec08e2837913d012fe2834ffa83'

乱数生成

```
os.urandom(n)
random.seed(x)
```

`urandom(n)` は，暗号の使用に適している疑似乱数を生成する関数であり，n バイト（n は整数）からなる乱数をバイナリデータで返す．この関数は，Windows API である CryptGenRandom を使う．本関数は，プロセス ID やスレッド ID，システムクロック等のシステム情報を利用することで擬似乱数を生成するものであり，乱数のシードとしても利用できる．`seed(x)` は乱数生成器を初期化する関数であり，整数 x を乱数のシードにセットする．urandom() を使用する際には文頭で「os」をインポートし，seed() を使用する際には文頭に「random」をインポートしておく．

以下に乱数を生成するサンプルコードを示す．このサンプルコードは，8 バイト乱数を 16 進数の文字列で出力する．

```
import os
import binascii
myseed = binascii.hexlify(os.urandom(8))
print(myseed)
```

上記サンプルコードでは，16 進数の文字列で表示させるために hexlify() を使用している．

素数生成

```
sympy.randprime(a, b)
```

`randprime(a, b)` は, 整数 a 以上 b 未満の素数の中でランダムなものを 10 進数の数値で返す．これはチェビシェフの定理を実装したものである．使用する際には，文頭で sympy をインポートしておく．

以下に素数乱数を生成するサンプルコードを示す．このサンプルコードは，2 以上 100 未満の素数をランダムに 1 つ出力する．また，乱数のシードとして 1234 を与えている．

```
import sympy
import random
random.seed(1234)
v = sympy.randprime(2, 100)
print(v)
```

ハッシュ関数

```
hashlib.sha1(s.encode()).hexdigest()
hashlib.sha224(s.encode()).hexdigest()
hashlib.sha256(s.encode()).hexdigest()
hashlib.sha384(s.encode()).hexdigest()
hashlib.sha512(s.encode()).hexdigest()
hashlib.shake_128(s.encode()).hexdigest(length)
hashlib.shake_256(s.encode()).hexdigest(length)
```

sha1(s.encode()).hexdigest() は，入力である任意長の文字列 s（入力のバイナリデータに相当，A.8 節を参照）に対して，160 ビットの 16 進数文字列を出力するハッシュ関数 SHA1 であり，sha224(s.encode()).hexdigest() は，入力である任意長の文字列 s に対して，224 ビットの 16 進数文字列を出力するハッシュ関数 SHA224 である．その他，256 ビット，384 ビット，及び 512 ビットの 16 進数文字列を出力するハッシュ関数も用意されている．また，可変長の戻り値を返す shake_128(s.encode()).hexdigest(length)，shake_256(s.encode()).hexdigest(length) が用意されている．hashlib を import しておく必要がある．

以下にハッシュ値を生成する 2 種類のサンプルコードを示す．1 つ目のサンプルコードは，文字列'Python' のハッシュ値を導出するものである．'Python' は内部では Unicode 文字列として扱われるため, encode() 関数によりバイナリデータに変換する必要がある．

```
import hashlib
v=hashlib.sha1('Python'.encode()).hexdigest()
print(v)
```

結果 1：'6e3604888c4b4ec08e2837913d012fe2834ffa83'

次のサンプルコードは，10 進数の数値とメッセージをビット連結したもののハッシュ値を導出するものである．数値もメッセージもパソコン内部ではビット列として扱われるのでビット連結できることは自然である．ただし，値の右側に最下位ビット (LSB) があることに注意する．

```
import hashlib
u = 123456789
m = 'Python'
tmp = "%x" %u
if (len(tmp)%2 == 0):
   s1 = binascii.unhexlify(tmp)
else:
   s1 = binascii.unhexlify('0'+tmp)
s = s1 + m
v = hashlib.sha1(s.encode()).hexdigest()
print(v)
```

結果 1：'9be4fd318b9d7d6a6104769ef012618d9f45bda7'

カイ自乗分布表の導出

```
chi2.ppf(1-α, n)
```

chi2.ppf(1-α,n) は，自由度 n，有意水準 α のカイ自乗統計量を導出する関数である．以下に，α=0.005，n=10 のときのカイ自乗統計量を導出するサンプルコードを示す．ただし，この関数の import 方法が他と異なることに注意する．

```
from scipy.stats import chi2
v=chi2.ppf(1-0.005, 10)
print(v)
```

結果：'23.209251159'

時間計測

時間計測には time.perf_counter() を利用する．この関数は、スリープ中の経過時間も含め、パフォーマンスカウンターの値 (小数点以下がミリ秒) を返し、その値はシステム全体で一意である。次に pow() の演算に要するの時間の測定の悪い例と良い例を示す．測定誤差を減らすために，1000 回の処理の平均を取ることにしている．当然入力値は同じ値にしないので，ここでは乱数生成関数を利用する．まずは悪い例を示す．この悪い例では，pow() の演算時間だけでなく，os.random() の演算時間も含まれてしまう点が悪い点である．

[悪い例]

```
p = 184908779667576050572947262326548513689
t0 = time.perf_counter()
for i in range(1000):
    a = int(binascii.hexlify(os.urandom(20)), 16)
    pow(2, a, p)
t1 = time.perf_counter() - t0
print(t1/1000)
```

結果：'0.000131047400794'

次に良い例を示す．ここでは，a の 1000 個のデータを予めメモリに用意しておき，純粋に pow() の演算時間だけを計算している．もちろん，メモリアクセス等の時間を要するが乱数生成関数の演算に要する時間に比べたら無視できるほど小さい．測定結果を見ても，悪い例では 0.13msec，良い例では 0.12msec と明らかに差があることが分かる．

[良い例]

```
p = 184908779667576050572947262326548513689
a = [ ]
for i in range(1000):
    a.append(int(binascii.hexlify(os.urandom(20)), 16))
t0 = time.perf_counter()
for i in range(1000):
    pow(2, a[i], p)
t1 = time.perf_counter() - t0
print(t1/1000)
```

結果：'0.00012201551483'

A.10 自作関数

他のプログラミング言語と同様に，Python においても自身で関数を定義できる．関数を定義するには def を使用する．ただし，関数本体のインデントは必須であり，関数の範囲を意味する．次の関数は，2 つの入力を入れ替える関数例である．

```
def ex(a, b):
    t = a
    a = b
    b = t
    return (a, b)
```

```
print(ex(1, 3))
```

結果：(3, 1)

Python では変数の定義を明示的にしないが，変数は自作関数の内部/外部で区別される．また，変数の型は何を代入するかによって自動的に決まるという特徴を持つ．異なる型の値を代入すれば，その都度変数の型が変わる．

A.11 ファイル入出力

ファイル入出力では，データがテキストであるかバイナリであるかで扱いが異なる．これに対して，以下の dump() と load() をペアで使用することによってデータの種類を意識することなくファイル入出力ができる．データ d をファイル data.txt に出力するには以下のように書く．data.txt が存在しない場合は新たにファイルが作成され，存在する場合は上書きされる．pickle を import しておく必要がある．

```
import pickle
f = open('data.txt', 'wb')
   pickle.dump(d, f)
f.close()
```

次に，ファイル data.txt からデータを読み込む場合は以下のように書く．

```
f = open('data.txt', 'rb')
   pickle.load(f)
f.close()
```

A.12 CSV ファイル入出力

大量の演算データを CSV 形式でファイル出力したり，ファイルから入力したりできると便利である．基本的な CSV ファイル出力は次のように書く．このサンプルコードは，リストデータをコンマ繋がりの CSV 形式に変換して，指定のファイルに 1 行ずつ書くコードである．ファイルを開く際には，バイナリモード (”wb”) を指定した方が無難である．csv を import しておく必要がある．

```
import csv
fh = open(”file.csv”, ”wb”)
writer = csv.writer(fh)
writer.writerow([1, 'RSA', 'A'])
writer.writerow([2, 'AES', 'S'])
writer.writerow([3, 'RC4', 'S'])
fh.close()
```

```
ファイル内の結果：
1,RSA,A
2,AES,S
3,RC4,S
```

さらに，上記ファイルをリスト形式で読み込むサンプルコードは以下である．

```
import csv
a = [ ]
fh = open(”file.csv”, ”rb”)
reader = csv.reader(fh)
for row in reader:
   a.append(row)
fh.close()
print(a)
```

結果：[['1', 'RSA', 'A'], ['2', 'AES', 'S'], ['3', 'RC4', 'S']]

A.13 グラフ描画

基本的なグラフ描画は次のように書く．このサンプルコードは sin(x) のグラフを描くコードである．`plt.axis` は XY 座標の範囲を指定する関数で，下記では $0 \leq x < 5, -1.5 \leq y < 1.5$ が指定される．そして，`plt.grid` は目盛りを付ける関数である．また，`np.arange` は x の値を設定する関数であり，下記では 0 から 5 未満まで 0.01 ずつ増加する値がリスト形式で x にセットされる．つまり，`y` にはリスト形式で `sin(x)` の出力が格納される．さらに，`plt.plot` 及び `plt.show` で xy 座標にリストの組をプロットする．

```
plt.axis([0.0, 5.0, -1.5, 1.5])
plt.grid(True)
x = np.arange(0, 5, 0.01)
y=sin(x)
plt.plot(x, y)
plt.show()
```

楕円曲線を描画するには，等高線を描く contour 関数を使う工夫が必要である．このサンプルコードは楕円曲線 $y^2 = x^3 + x + 1$ のグラフを描くコードである．ここでは，x と y の両方に値を設定し，meshgrid を使って 2 次元のリストに変換し，新たな変数 z を導入して $z = y^2 - x^3 - x - 1$ をプロットする．ただし，contour 関数の第 4 引数を [0] にすることで，$z = 0$ のときのグラフだけを描くことができる．

```
pylab.grid(True)
x=np.arange(-1,1,0.01)
y=np.arange(-2,2,0.01)
(X,Y)=np.meshgrid(x,y)
Z=Y**2-X**3-X-1
plt.contour(X,Y,Z,[0])
plt.show()
```

ただし，上記グラフ描画ではパッケージを import する必要がない．

A.14 Python2 から 3 での変更点

Python2 から 3 でのバージョンアップ変更点で重要なものを掲載する．

print が文から関数に変更

Python3 から print 文は関数に変更された．このため，引数は括弧 () で括らなければならない．

Python2
```
print ”Hello”    # ”Hello” と表示される
```

Python3
```
print(”Hello”)    # ”Hello” と表示される
```

a/b の仕様変更

int 型同士の割り算 (/) は Python2 では int 型を返す演算子だったが，Python3 では float 型を返す演算子に変更された．Python3 で int 型を返したいときは，// 演算子を使う．

Python2
```
3/2    # 結果は 1
```

Python3
```
3/2    # 結果は 1.5
3//2    # 結果は 1
```

文字列型が Unicode に

Python2 では文字列型はバイト列であったが，Python3 では文字列型が Unicode に変更された．Python3 でバイト列を扱いたいときは，バイト型 (bytes) を使う．

Python2

```
"normal"　# Python2 での文字列型はバイト列
type("normal")　# <type 'str'>
u"unicode"　# Python2 で Unicode を使うときは，文字リテラルの前に u を付与
type(u"unicode")　# <type 'unicode'>
```

Python3

```
"normal"　# Python3 では文字列型が Unicode に
type("normal")　# <type 'str'>
b"bytedata"　# Python3 でバイト列を扱いたいときは，文字リテラルの前に b を付与
type(b"bytedata")　# <type 'bytes'>
"str".encode()　# b'str'
b"bytes".decode()　# b'bytes'
```

B ライブラリ API

B.1 ライブラリ API 一覧

表 B.1: ライブラリ API 一覧

パッケージ	関数	内容
標準	pow(x, y, z)	累乗計算
標準	int.bit_length()	n を 2 進数表現したときの桁数を返す
標準	int.from_bytes(s, order)	与えられたバイト列の整数表現を返す.
標準	int.to_bytes(n, order)	整数を表すバイト列を返す.
標準	(str).encode()	Unicode 文字列をバイト列に変換する
標準	(byte).decode()	バイト列を Unicode 文字列に変換する
標準	bin()	整数を先頭に 0b がついた 2 進文字列に変換
base64	base64.b64decode(s, altchars, validate)	Base64 でエンコードされた s をデコードする.
base64	base64.b64encode(s, altchars)	Base64 を使って s をエンコードする
binascii	binascii.hexlify(s)	ASCII バイト列を 16 進数を表すバイト列に変換する
binascii	binascii.unhexlify(s)	16 進数を表すバイト列を ASCII バイト列に変換する
hashlib	hashlib.shake_128(m)	バイト列 m を入力とする SHAKE128 のオブジェクトを返す
hashlib	hash.shake_128.hexdigest(n)	shake_128 オブジェクトによる出力を返す.
os	os.urandom(n)	長さ n のランダムなバイト列を生成する.
random	random.getrandbits(k)	k 桁の乱数を生成する.
random	random.randrange(a, b)	[a, b) に含まれる乱数を返す.
random	random.seed(a, version)	乱数関数を初期化する.
random	random.SystemRandom()	エントロピープールを利用する乱数生成オブジェクトを返す.
random	SystemRandom.randrange(a, b)	[a, b) に含まれる乱数を返す.
sympy	sympy.randprime(a, b)	[a, b) に含まれる素数をランダムに返す.
time	time.perf_counter()	パフォーマンスカウンターの値 (小数点以下がミリ秒) を返す.

B.2 ライブラリ API 詳細

Algorithm 1 累乗計算 pow(x, y[, z])

Input: x, y, z (int 型)

Output: x の y 乗. z があれば，x の y 乗に対する z の剰余を返す.

Object: int

Algorithm 2 ビット長出力 int.bit_length()

Output: 整数のビット数 (int 型)

Object: int

Algorithm 3 バイト列を整数へ変換 int.from_bytes(s, order)

Input: s: バイト列 (bytes 型), order: バイトオーダー ('big' or 'little')

Output: 整数

Remark: Python3.2 から追加.

Algorithm 4 整数をバイト列へ変換 int.to_bytes(n, order)

Input: n: 整数 (int 型), order: バイトオーダー ('big' or 'little')

Output: バイト列 (bytes 型)

Remark: Python3.2 から追加.

Algorithm 5 Base64 へのエンコード base64.b64encode(s, altchars=None)

Input: s: 入力バイト列 (bytes 型), altchars=None : 変更する文字 (bytes 型)

Output: Base64 にエンコードされたバイト列 (bytes 型)

Package: Base64

Algorithm 6 Base64 によるデコード base64.b64decode(s, altchars=None, validate=False)

Input: s: 入力バイト列 (bytes 型), altchars=None: 変更する文字 (bytes 型), validate=False: 検証するか (bool 型)

Output: Base64 によってデコードされたバイト列 (bytes 型)

Package: Base64

Algorithm 7 Unicode 文字列のエンコード (str).encode()

Input: str: 入力文字列 (Unicode 型)

Output: バイト列 (bytes 型)

Package: 標準

Algorithm 8 バイト列のデコード (bytes).decode()

Input: bytes: 入力バイト列 (byte 型)

Output: 文字列 (Unicode 型)

Package: 標準

Algorithm 9 整数の 2 進文字列への変換 bin()

Input: 整数

Output: 入力の 2 進文字列 (先頭に 0b がついている)

Package: 標準

Algorithm 10 ASCII バイト列を 16 進数バイト列へ変換 binascii.hexlify(s)

Input: s: ASCII バイト列 (bytes 型)

Output: 16 進数表現バイト列 (bytes 型)

Package: binascii

Algorithm 11 16 進数バイト列を ASCII バイト列へ変換 binascii.unhexlify(s)

Input: s: 16 進数表現バイト列 (bytes 型)

Output: ASCII バイト列 (bytes 型)

Package: binascii

Algorithm 12 Shake128 によるハッシュ値 hashlib.shake_128(m).hexdigest()

Input: m : バイト列 (bytes 型)

Output: m のハッシュ値 (bytes 型, 16 進数表示)

Package: hashlib

Algorithm 13 ハッシュアルゴリズム SHAKE128 のオブジェクト生成 hashlib.shake_128(m)

Input: m : バイト列 (bytes 型)

Output: SHAKE128 アルゴリズムを扱うオブジェクト (hash 型)

Package: hashlib

Algorithm 14 ランダムなバイト列を生成 os.urandom(n)

Input: n: バイト数 (int 型)

Output: ランダムな長さ n のバイト列 (bytes 型)

Package: os

Remark: エントロピープールを利用するため，乱数初期化子は持たない．Unix の/dev/urandom に相当する．

Algorithm 15 k ビットの乱数を生成 random.getrandbits(k)

Input: k: 出力乱数のビット数 (int 型)

Output: 乱数 (int 型)

Package: random

Remark: random.seed で初期化される．

Algorithm 16 [a, b) の範囲内にある乱数を生成 random.randrange(a, b)

Input: a, b: 乱数の生成範囲 (int 型)

Output: 乱数 (int 型)

Package: random

Remark: random.seed で初期化される．

Algorithm 17 乱数関数の初期化 random.seed(a=None, version=2)

Input: a: 乱数シード (int or str or bytes 型), version: 乱数生成アルゴリズムのバージョン (int 型)

Package: random

Remark: random.randrange, random.getrandbits を初期化する．

Algorithm 18 乱数生成オブジェクトの生成 random.SystemRandom()

Package: random

Remark: エントロピープールを利用する乱数生成オブジェクトを返す．

Algorithm 19 乱数生成オブジェクトの生成 random.SystemRandom.randrange(a, b)

Input: a, b: 乱数の生成範囲 (int 型)

Output: 乱数 (int 型)

Package: random

Object: random.SystemRandom

Remark: 乱数の生成生成オブジェクトを返す．

Algorithm 20 [a, b) に含まれる素数をランダムに生成．sympy.randprime(a, b)

Input: a, b: 素数の生成範囲 (int 型)

Output: 素数 (int 型)

Package: sympy

Algorithm 21 時間計測 time.perf_counter()

Input: なし

Output: パフォーマンスカウンターの値 (小数点以下がミリ秒)

Package: time

C　作成関数 API

演習で作成する関数は他の演習で作成した関数を用いる必要があります．この際，関数名や関数の入出力変数を統一しておくと，作業がスムースに進みます．なお，参考関数として利用する Python の標準関数を記載します．それ以外の Python の標準関数は利用しないで，自分でアルゴリズムを考えて実装してください．また，自作した関数を利用する場合は，自作関数として記載していますので，自作関数を利用するようにしてください．なお，文字列・リスト・ビットに関連した関数や制御関数については，明示的に指定されていなくても使用することができます．

Algorithm 22 ユークリッドの互除法 euclid(a, b)

Input: 整数: a, b

Output: 整数: a, b の最大公約数

関連演習：事前演習 1

この関数は sympy.gcd(a, b) で代替可能です．

Algorithm 23 拡張ユークリッドの互除法 exEuclid(a, b)

Input: 整数: a, b

Output: 整数のリスト: $[d = ax + by, x, y]$(d は a, b の最大公約数)

関連演習：演習 1.1, 演習 1.2

この関数は sympy.gcdex(a, b) で代替可能です．ただし，返り値は (x, y, d) の順です．

Algorithm 24 逆元 inv(a, n)（ユークリッドの互除法を利用）

Input: 整数: a, n (gcd(a, n)= 1, n は素数と合成数のどちらも取りうる)

Output: 整数: $a^{-1} \bmod n$

自作関数：exEuclid

関連演習：演習 1.2

Algorithm 25 実行時間の計測 time_check(t_s, t_e)

Input: 計測開始地点：t_s, 計測終了地点の時刻：t_e

Output: $t_e - t_s$

参考関数：time.perf_counter

関連演習：演習 1.2, 演習 1.3

Algorithm 26 拡張ユークリッドの互除法のループ回数計測 exEuclidExp(a, b)

Input: 整数: a, b

Output: 整数のリスト: $[d = ax + by, x, y, counter]$, counter はループの実行回数, d は a, b の最大公約数.

関連演習：解析 1.2

Algorithm 27 逆元計算のループ回数計測 invExp(a, n)

Input: 整数: a, n (gcd(a, n)= 1, n は素数と合成数のどちらも取りうる)

Output: 整数のリスト: $[a^{-1} \bmod n, counter]$, counter はループの実行回数.

自作関数：exEuclidExp

関連演習：解析 1.2

Algorithm 28 法 n 上のバイナリ法 mod_binary(g, k, n)

Input: 整数: g, k, n (n は素数と合成数のどちらも取りうる)

Output: $g^k \bmod n$

参考関数：bin

関連演習：演習 1.3

この関数は pow(g, k, n) で代替可能です.

Algorithm 29 法 n 上のバイナリ法での法乗算回数計測 modBinaryExp(k, g, n)

Input: 整数: k, g, n (n は素数と合成数のどちらも取りうる)

Output: [[n のビット数, 法乗算回数, 法 2 乗算回数], $g^k \bmod n$]

参考関数：bin, int.bit_length

関連演習：解析 1.4

Algorithm 30 整数の最小公倍数 lcm(a, b)

Input: 正の整数：a, b ($a < b$)

Output: lcm(a, b) = $ab/\gcd(a, b)$ (sympy.lcm(a, b) で代用可能)

自作関数：euclid

関連演習：演習 2.3, 2.4

Algorithm 31 カーマイケル数探索 search_carmichael(a, b)

Input: 正の整数：a, b ($a < b$)

Output: カーマイケル数のリスト $[n_1, n_2, \ldots, n_k]$ ($a \le n_i \le b$)

自作関数：mod_binary, euclid

参考関数：sympy.isprime

関連演習：演習 2.1

Algorithm 32 フェルマーテスト (素数判定) fermat_test(n, a)

Input: 正の整数：n, a ($\gcd(a, n) = 1$

Output: $a^{n-1} \equiv 1 \pmod{n}$：True

$a^{n-1} \not\equiv 1 \pmod{n}$：False

自作関数： mod_binary

関連演習：演習 2.2, 解析 2.1

Algorithm 33 フェルマーテストによる素数生成 fermat_primegen(iv, k)

Input: フェルマーテストの繰り返し回数：k, 初期値：iv (演習 2.3 では iv は乱数生成関数で生成)

Output: iv 以上の最小の素数

自作関数：fermat_test, euclid

参考関数：

関連演習：演習 2.2, 2.3, 解析 2.1

D 数値例

演習問題利用する数値例を列挙する.

表 D.13: 実験 **??**

$$p^{1024}_{205,1} = 493595662\ 2392582906\ 0159543226\ 4097389883\ 0219276387\ 0876133442543$$
$$p^{1024}_{205,2} = 456808137\ 5831629151\ 5717355536\ 5566850489\ 5496119860\ 5929142386251$$
$$p^{1024}_{205,3} = 506122852\ 3243889890\ 9959304006\ 4579176710\ 1990872376\ 8699682562461$$
$$p^{1024}_{205,4} = 464070881\ 5961179432\ 5372916676\ 9152885541\ 1813776945\ 0524712275639$$
$$p^{1024}_{204,5} = 183708392\ 8689162312\ 8623749268\ 8259016032\ 6233453942\ 4149602250757$$
$$q^{1024}_{204} = 203918313\ 8486605892\ 0559002137\ 1223440022\ 4409438062\ 2706168647991$$
$$p^{1024}_{128,1} = 327727013\ 0697280848\ 8944386996\ 3828059711$$
$$p^{1024}_{128,2} = 327252369\ 8083701070\ 2068872026\ 5095547587$$
$$p^{1024}_{128,3} = 307454904\ 9880630415\ 6742678423\ 2785496057$$
$$p^{1024}_{128,4} = 319924500\ 4586458424\ 6505038090\ 3060228263$$
$$p^{1024}_{128,5} = 302452415\ 7870358694\ 8842693281\ 9841899867$$
$$p^{1024}_{128,6} = 338758195\ 2055110602\ 9709400454\ 1911629483$$
$$p^{1024}_{128,7} = 287458879\ 5521621931\ 8417952737\ 8191280963$$
$$p^{1024}_{128,8} = 305456423\ 3327921932\ 5853935801\ 9720643289$$
$$q^{1024}_{128} = 332279858\ 4304493661\ 1686731274\ 3835330019$$
$$p^{1024}_{102,1} = 495993670\ 4085803091\ 909934580737$$
$$p^{1024}_{102,2} = 469814670\ 4762017122\ 018325831239$$
$$p^{1024}_{102,3} = 486667865\ 5871057795\ 337652859779$$
$$p^{1024}_{102,4} = 489797520\ 2654528612\ 188886803549$$
$$p^{1024}_{102,5} = 446464625\ 6452858241\ 550294992549$$
$$p^{1024}_{102,6} = 475909429\ 5450630354\ 542422115239$$
$$p^{1024}_{103,7} = 973019323\ 5597482959\ 391696091919$$
$$p^{1024}_{103,8} = 101118188\ 9708315373\ 4384206604517$$
$$p^{1024}_{103,9} = 949832607\ 6865856811\ 759823202011$$
$$p^{1024}_{103,10} = 985114119\ 5354528562\ 161235301321$$
$$q^{1024}_{106} = 746930625\ 8946365431\ 6918233318301$$
$$p^{1024}_{64,1} = 1793527218\ 8932513967$$
$$p^{1024}_{64,2} = 1800312522\ 9117196033$$
$$p^{1024}_{64,3} = 1827421461\ 7505544791$$
$$p^{1024}_{64,4} = 1751576479\ 3748280049$$
$$p^{1024}_{64,5} = 1811950497\ 7260936187$$
$$p^{1024}_{64,6} = 1821399703\ 9822583309$$
$$p^{1024}_{64,7} = 1618704703\ 5808728619$$
$$p^{1024}_{64,8} = 1728773677\ 3492607021$$
$$p^{1024}_{64,9} = 1836897792\ 5200211593$$
$$p^{1024}_{64,10} = 1776317044\ 3295043023$$
$$p^{1024}_{64,11} = 1774898105\ 1160024049$$
$$p^{1024}_{64,12} = 1747302870\ 7383089513$$
$$p^{1024}_{64,13} = 1716872894\ 0836252187$$
$$p^{1024}_{64,14} = 1746710983\ 3925754251$$
$$p^{1024}_{64,15} = 1773343856\ 6473322513$$
$$p^{1024}_{64,16} = 1763074430\ 9897329279$$
$$q^{1024}_{64} = 1690799673\ 5427429109$$
$$p^{512}_{102,1} = 499986983\ 1519335186\ 491039887011$$

$$p_{102,2}^{512} = 434391841\ 2294302841\ 333376743907$$
$$p_{102,3}^{512} = 454007051\ 3713951418\ 173066378121$$
$$p_{103,4}^{512} = 923279523\ 4423487512\ 704957441133$$
$$p_{103,5}^{512} = 738245583\ 9101146879\ 478406938167$$
$$q_{104}^{512} = 192491032\ 0067926071\ 2302169662261$$
$$p_{64,1}^{512} = 1800561438\ 4815733387$$
$$p_{64,2}^{512} = 1651409709\ 4544156203$$
$$p_{64,3}^{512} = 1701721994\ 2406764657$$
$$p_{64,4}^{512} = 1737059845\ 9556484827$$
$$p_{64,5}^{512} = 1822958800\ 2006510251$$
$$p_{64,6}^{512} = 1634346051\ 7761035033$$
$$p_{64,7}^{512} = 1658670383\ 1293716201$$
$$p_{64,8}^{512} = 1835786494\ 4608595657$$
$$q_{64}^{512} = 1754844339\ 2621680141$$
$$p_{51,1}^{512} = 2230411555\ 766393$$
$$p_{51,2}^{512} = 2245702431\ 005177$$
$$p_{51,3}^{512} = 2246104575\ 278881$$
$$p_{51,4}^{512} = 1968007279\ 343447$$
$$p_{51,5}^{512} = 2248619464\ 978387$$
$$p_{51,6}^{512} = 2249329656\ 268613$$
$$p_{51,7}^{512} = 2024614374\ 484313$$
$$p_{51,8}^{512} = 2056690612\ 038619$$
$$p_{52,9}^{512} = 4146564166\ 452341$$
$$p_{52,10}^{512} = 4472076290\ 677579$$
$$q_{53}^{512} = 8729924371\ 238659$$
$$p_{32,1}^{512} = 4262526571$$
$$p_{32,2}^{512} = 4243343609$$
$$p_{32,3}^{512} = 4268376751$$
$$p_{32,4}^{512} = 4138507397$$
$$p_{32,5}^{512} = 4275976433$$
$$p_{32,6}^{512} = 4184037401$$
$$p_{32,7}^{512} = 4273997501$$
$$p_{32,8}^{512} = 4111394753$$
$$p_{32,9}^{512} = 4231168379$$
$$p_{32,10}^{512} = 3826886339$$
$$p_{32,11}^{512} = 4211145737$$
$$p_{32,12}^{512} = 4236261749$$
$$p_{32,13}^{512} = 4276081549$$
$$p_{32,14}^{512} = 4031143681$$
$$p_{32,15}^{512} = 4071011653$$
$$p_{32,16}^{512} = 4198142857$$
$$q_{32}^{512} = 4062055801$$

Algorithm 34 共通法 RSA への攻撃 common_mod_attack(e_1, e_2, n, c_1, c_2)

Input: RSA 暗号の公開鍵：(e_1, n), (e_2, n), 暗号文：c_1, c_2

Output: 平文：m ($m^{e_i} \equiv c_i \pmod{n}$)

自作関数：ex_euclid, inv, mod_binary

関連演習：解析 2.2

Algorithm 35 RSA への適応的選択暗号文攻撃 1(復号オラクルに入力する暗号文計算） rsa_cca_attack1(e, n, c)

Input: RSA 暗号の公開鍵：(e, n), 暗号文：c

Output: 復号オラクルに入力する暗号文：c'

自作関数：euclid, mod_binary

関連演習：解析 2.3

Algorithm 36 RSA への適応的選択暗号文攻撃 1(平文の復号） rsa_cca_attack2(n, m')

Input: RSA 暗号の公開鍵：n, 復号オラクルが出力した c' に対応する平文：m'

Output: 復号対象の平文：c

自作関数：inv

関連演習：解析 2.3

Algorithm 37 RSA 暗号-鍵生成 rsaKeygen(p, q)

Input: 素数 p, q

Output: RSA 暗号の公開鍵 (整数の組）：$(n = pq, e)$ $(\gcd(e, \mathrm{lcm}(p-1, q-1)) = 1)$, 秘密鍵：$d$ $(ed \equiv 1 \pmod{\mathrm{lcm}(p-1, q-1)})$

自作関数：inv, lcm

関連演習：演習 2.3, 2.4

Algorithm 38 RSA 暗号-暗号化 rsaEnc(m, n, e)

Input: メッセージ (整数)：m, 公開鍵：n, e $(m < n)$

Output: 暗号文 (整数): $c \equiv m^e \bmod n$

自作関数：mod_binary

関連演習：演習 2.3, 2.4, 演習 4.1, 4.2

Algorithm 39 RSA 暗号-復号 rsaDec(c, n, d)

Input: 暗号文 (整数): c, 法：n, 秘密鍵：d

Output: 平文（整数）: $m \equiv c^d \bmod n$

自作関数： mod_binary

関連演習：演習 2.3, 2.4, 演習 4.1, 4.2

Algorithm 40 平均と分散 mv(T)

Input: 実数値のリスト：$T = [t_1, \ldots, t_k]$

Output: $t_1, \ldots, t_k$ の平均と分散

参考関数： numpy.mean, numpy.var

関連演習：演習 3.1

Algorithm 41 RSA 署名-鍵生成関数 rsaSignGenKey(p, q, e)

Input: 素数 p, q, 正の整数 e ($\gcd(e, \mathrm{lcm}(p-1, q-1)) = 1$

Output: RSA 署名の公開鍵: ($n = pq, e$)(整数の組), 秘密鍵：d ($ed \equiv 1$ (mod. $\mathrm{lcm}(p-1, q-1)$))

自作関数：inv, lcm

関連演習：演習 3.1, 4.2

Algorithm 42 shake128 によるハッシュ値の計算関数 shake128(m, h_size)

Input: メッセージ（通常の文字列）：m, ハッシュ値のサイズ (バイト): h_size

Output: shake128 による m のハッシュ値 (整数値）

関連演習：演習 3.1, 3.2, 4.2

※この関数はこちらから提供します

Algorithm 43 RSA 署名-署名関数 rsaSignGen(m, n, d)

Input: メッセージ: m(通常の文字列), 公開鍵: n(整数), 秘密鍵: d(整数)

Output: RSA 署名 (整数の組) (m, σ)

参考関数：

自作関数：mod_binary, shake128

関連演習：演習 3.1, 3.2, 演習 4.2

Algorithm 44 RSA 署名-署名検証関数 rsaSignVerify(m, sigma, n, e)

Input: メッセージ: m(通常の文字列), 署名: sigma(整数), 公開鍵: n, e (整数)

Output: 署名が正しければ True, 誤っていれば False

参考関数：

自作関数：mod_binary, shake128

関連演習：演習 3.1, 3.2, 演習 4.2

Algorithm 45 MGF(Message Generation Function) mgf(D, oLen)

Input: データ (16 進バイト列): D, 出力バイト長: oLen

Output: D の MGF 出力値 (16 進バイト列)

関連演習：演習 4.1, 4.2

※この関数はこちらから提供します

E テストデータ

演習で作成する関数の動作確認用のデータです．演習で作成した関数はまず，以下のデータと合致するか確認してください．

演習 1.3, (べき乗算) : p : 1024bit 素数, $q = (p-1)/2$: 1023bit 素数, $k \in [1, q]$

Algorithm 46 補助関数 xor(x, y)

Input: 16 進数バイト列 x, y

Output: 16 進数バイト列 $x \oplus y$

参考関数 binascii.unhexlify

関連演習：演習 4.1, 4.2

※この関数はこちらから提供します

Algorithm 47 RSA-OAEP 暗号-暗号化 rsa_oaep_Enc(m, k_0, k_1, k_2, IHASH, r, e, n)

Input: メッセージ (16 進数バイト列): m, OAEP ハッシュ値 IHASH のバイト長: k_0,
パディングバイト長: k_1, メッセージバイト長: k_2, OAEP ハッシュ値: IHASH(16 進バイト列),
OAEP 用乱数 (k_0 バイト 16 進バイト列): r, RSA 暗号の公開鍵 (整数の組): (e, n)

Output: RSA-OAEP 暗号の暗号文 (整数): C

参考関数：xor, mgf, int

自作関数：rsaEnc

関連演習：演習 4.1, 4.2

Algorithm 48 RSA-OAEP 暗号-復号 rsa_oaep_Dec($C, k0, k1, k2$, IHASH, r, d, n)

Input: 暗号文 (整数): C, OAEP ハッシュ値 IHASH のバイト長: k_0, パディングバイト長: k_1,
メッセージバイト長: k_2, OAEP ハッシュ値: IHASH(16 進バイト列),
OAEP 用乱数 (k_0 バイト 16 進バイト列): r. 法 n, RSA 暗号の秘密鍵 (整数): d

Output: 復号できれば復号文 (k_2 バイト 16 進バイト列) m を返す．復号失敗時は None を返す

参考関数：xor, mgf, hex

自作関数：rsaDec

関連演習：演習 4.1, 4.2

Algorithm 49 ハイブリッド暗号-暗号化（RSA-AES) rsa_aes_hybrid_enc(K, m, k_0, k_1, k_2, r, IHASH, e, n)

Input: AES の鍵 (16 進バイト列) : K, 平文 (通常の文字列): m, OAEP ハッシュ値 IHASH のバイト長: k_0,
パディングバイト長: k_1, メッセージバイト長 k_2, OAEP ハッシュ値: IHASH(16 進バイト列),
OAEP 用の乱数 (k_0 バイト 16 進バイト列): r, OAEP ハッシュ値: IHASH, RSA 暗号の公開鍵: (e, n)

Output: K の RSA 暗号の暗号文 (整数): C_K, m の AES による暗号文 (16 進バイト列): C_m, AES の復号に使うタグ: tag, AES のオブジェクトのナンス値: AES_object.nonce

参考関数：encode, AES.new, AES_object.encrypt_and_digest

自作関数：rsa_oaep_Enc

関連演習：演習 4.2

※ AES はこちらから提供します

- p = 1 6172 2843 4731 0795 1878 1965 2301 0278 7808 9812 9304 8648 7693 9285 1319 9489 2003 4541 6630 4728 6149 4261 6153 8957 9993 6946 2947 8786 8572 6549 2519 7743 3635 2325 7076 9590 4138 9911 4723 4072 4603 5258 7039 7715 7115 6154 9962 9292 1342 2994 5032 8755 2800 1617 8818 1774 6715 1989 9908 0848 5445 2802 5379 0904 5880 8020 8105 5347 7605 9442 2746 7751 5598 9001 6491 8419 2227
- q = 8086 1421 7365 5397 5939 0982 6150 5139 3904 4906 4652 4324 3846 9642 5659 9744 6001 7270 8315 2364 3074 7130 8076 9478 9996 8473 1473 9393 4286 3274 6259 8871 6817 6162 8538 4795 2069 4955 7361 7036 2301 7629 3519 8857 8557 8077 4981 4646 0671 1497 2516 4377 6400 0808 9409 0887 3357 5994 9954 0424 2722 6401 2689 5452 2940 4010 4052 7673 8802 9721 1373 3875 7799 4500 8245 9209 6113
- g = 3

Algorithm 50 ハイブリッド暗号-復号（RSA-AES) aes_rsa_hybrid_dec(C_K, C_m, tag, AES_object_nonce, k_0, k_1, k_2, r, IHASH, d, n)

Input: K の暗号文 (整数): C_K, m の暗号文 (16 進バイト列): C_m, AES の復号に使うタグ: tag, AES のオブジェクトのナンス値: AES_object_nonce,
OAEP 乱数バイト長: k_0, パディングバイト長: k_1, メッセージバイト長: k_2, OAEP 用の乱数 (k_0 バイト 16 進バイト列): r, OAEP ハッシュ値: IHASH, RSA 暗号の法: n, RSA 暗号の秘密鍵: d

Output: 復号成功時は AES の秘密鍵と復号文のリスト $[K, m]$ (K は 16 進バイト列, m は通常の文字列 を返す. 復号失敗時は None を返す)

参考関数：decode, AES.new, decrypt_and_verify

自作関数：rsa_oaep_Dec

関連演習：演習 4.2

※ AES はこちらから提供します

- k = 10 4571 8704 5037 5446 0826 0300 9247 8128 3536 0745 0058 8985 4016 0509 0290 7977 8978 1827 0556 5319 3714 3909 2550 6161 3097 3670 9929 3941 1661 3083 3827 3198 5660 1144 6487 7867 3546 1877 1349 3977 1924 7348 8299 8510 8204 6399 0623 8414 4311 4889 5630 6610 8294 2534 7290 7226 0408 4630 0068 9528 8051 6877 1990 5030 9111 1158 3719 1715 0969 4344 4779 2040 9435 4942 9418 3981
- $g^q \bmod p = 1$
- $g^k \bmod p$ = 6860 4621 4128 4475 2900 7754 9546 8089 0419 2606 7214 1901 5868 8634 3977 2491 7254 4902 5245 1024 3317 5133 4422 6544 1957 4139 5994 2010 8286 1887 3286 0160 5282 9148 6753 5222 4607 9097 2165 3422 8589 4946 8773 0126 7774 2870 0925 4615 3505 8405 6594 2421 2897 9651 9355 7233 4995 9685 8509 3272 0501 2483 1922 5542 6083 7080 8623 3334 9544 6586 2453 1840 0265 5446 7343 2182 1794

演習 1.1 (拡張ユークリッドの互除法) : p : 1024bit 素数, $a \in [1, p]$, $x, y \in \mathbb{Z}$ s.t. $ax + py = 1$

- p = 1 3587 3444 2648 8802 0826 0557 7702 2778 8944 4208 9288 0418 2342 1678 5295 5250 7472 9079 4928 0760 0315 7447 1230 0388 3623 4493 6751 7410 6810 4295 1880 9553 3153 2883 5948 9052 5616 7722 9406 0981 0061 0136 0991 2715 9476 9103 8097 6831 2172 5981 3417 2440 4727 2971 5036 8876 4137 5029 2574 4760 3615 7404 1615 0271 6910 3928 2461 2195 5659 5972 2386 5183 7205 4336 1321 5536 0047
- a = 6713 3154 0796 5730 2823 6948 9926 0332 0052 0715 2011 2103 4639 8405 3173 0534 7604 3698 6831 0127 5989 0346 4948 5654 7874 9762 5976 7670 2846 1558 6934 7399 4611 7443 7115 7541 8167 5473 7152 1263 3208 5508 1560 6312 4662 5936 5382 5342 0239 0266 3842 1665 8059 4617 2670 1799 8621 8207 3437 9121 1754 4587 0277 7423 3472 7209 9964 4933 4420 4352 1554 4869 9894 7615 5622 2891 4576
- x = 4136 5361 7575 9588 7529 8657 8634 9595 2112 3856 3490 0194 8803 1242 5318 6918 1102 3075 6357 7808 0689 4165 2337 0079 0345 6444 8978 5388 8655 0998 4047 2840 5025 8008 7528 8510 1999 4257 9088 8380 5202 2752 7227 3531 0700 5111 4618 1596 8615 4081 9797 7339 3291 7020 6019 3134 3130 7257 1268 7521 9777 8200 7469 2566 8019 8443 4352 3958 3689 3678 5316 8887 1259 5806 2859 3581 5686
- y = -2043 8042 3228 2252 2661 5588 0729 7356 1954 6917 2801 8434 6172 6071 7421 2341 1210 1321 3211 2542 6799 1874 5191 3517 7336 5280 2843 3329 0575 8599 5443 6193 3533 4390 8551 9851 6269 6112 9068 3992 0346 8982 6466 4559 5144 5176 6467 4484 7750 7018 0528 6971 9153 5999 6300 5376 6508 2671 3782 6803 7451 1704 1762 9703 6698 3072 5766 0591 9207 4062 8729 7840 0429 7005 1982 1629 8705

なお上記の解 (x, y) に対し, $x' = x - pt$, $y = y + at$ ($t \in \mathbb{Z}$) も解になります.

演習 2.4 (RSA 暗号) :

表 D.1: 160 ビット素数

p_4	=	73075081 8665451459 5239617144 9964083306 2084344321
g_4	=	2
g_5	=	63075081 8665451459 5239617144 9964083306 2084344321
k_4	=	17235797 9665757759 7237577777 9967773376 2237577327
k_5	=	10111008 8010111000 8000000000 1100000000 11 (16 進)

表 D.2: 157 ビット整数

m_1	=	123 75081 11111 51459 12345 17144 99640 33333 55555 44444
r_1	=	123 45678 91234 56789 12345 67891 23456 78912 34567 89123
r_2	=	113 25678 91234 56789 12345 67891 23456 78912 34567 89123

表 D.3: 1,024 ビット素体

p_1	=	179769313 4862315907 7083915679 3787453197 8602960487 5601170644 4423684197
		1802161585 1936894783 3795864925 5415021805 6548598050 3646440548 1992391000 5079287700
		3355816639 2295531362 3907650873 5759914822 5748625750 0742530207 7447712589 5509579377
		7842444242 6617334727 6292993876 6870920560 6050270810 8429076929 3201912819 4467627007
ℓ_1	=	89884656 7431157953 8541957839 6893726598 9301480243 7800585322 2211842098
		5901080792 5968447391 6897932462 7707510902 8274299025 1823220274 0996195500 2539643850
		1677908319 6147765681 1953825436 7879957411 2874312875 0371265103 8723856294 7754789688
		8921222121 3308667363 8146496938 3435460280 3025135405 4214538464 6600956409 7233813503
g_1	=	2
g_2	=	3
k_1	=	2137858233 0649381417 4696927539 8479291412 7976226092
		0485325804 8108487164 4119000600 1415349290 3789795134 2787125074 0540083892 2914995687
		6406489134 1265294738 0207374900 6458927006 5399155390 0954617037 8468358492 9753269110
		9084168706 9786303767 0868022176 5537411507 2258330322 9140472735 5023569089 8518749239
k_2	=	5653423359 3039407425 4433017009 9880725338 4559379100
		7410819211 5937645050 5499824740 0741425627 8367541365 1201024417 3844247923 5536228759
		0802671976 5311636330 8565847410 7994638347 8718740305 4790304127 7983163637 4101856231
		5240667128 4971720906 9816526027 8072896856 6018316270 2665463088 5659143409 3339991506

表 D.4: 1,018 ビットの例

p_2	=	2808 89552322 23686058 27039360 60785114 62780890 29597354
		01989734 50180895 73059460 95254894 85699581 62617750 33000177 93729905
		21213418 59013772 52597264 50741103 74178319 34026233 34763523 20744222
		21812694 70220616 45442112 63282151 38096104 41160098 25230298 92352200
		42558067 73517294 46660909 99917571 77887455 67263052 44265037 8502727
ℓ_2	=	235602549 7647385004 6586536675 5585567600 1569676039
g_2	=	223430576 4178904874 2226942690 6195161767 8328806534 2384853944 5139969853
		9878215171 3855299851 8623144538 1932618038 7501745856 9632252569 2516975899 9729335545
		4814586334 5982301876 4919047828 9169251991 3262751174 3981332237 7796421797 5243365085
		8754451978 6674834759 3695654408 7470953611 5867164313 8122477155 5933732847 0627806180
k_2	=	40845473 3839582623 3244734395 0544815575 2423456693

表 D.5: 1,024 ビット素数 p_3, ベースポイント $\mathbb{F}_p \ni g_3(\mathrm{ord}(g_3) = \ell_3)$, 1,023 ビット乱数 k_3,r_3 と OAEP 用乱数

p_3	=	135873444 2648880208 2605577702 2778894442 0892880418 2342167852 9552507472
		9079492807 6003157447 1230038836 2344936751 7410681042 9518809553 3153288359 4890525616
		7722940609 8100610136 0991271594 7691038097 6831217259 8134172440 4727297150 3688764137
		5029257447 6036157404 1615027169 1039282461 2195565959 7223865183 7205433613 2155360047
g_3	=	2
ℓ_3	=	67936722 1324440104 1302788851 1389447221 0446440209 1171083926 4776253736
		4539746403 8001578723 5615019418 1172468375 8705340521 4759404776 6576644179 7445262808
		3861470304 9050305068 0495635797 3845519048 8415608629 9067086220 2363648575 1844382068
		7514628723 8018078702 0807513584 5519641230 6097782979 8611932591 8602716806 6077680023
k_3	=	57936722 1324440104 1302788851 1389447221 0446440209 1171083926 4776253736
		4539746403 8001578723 5615019418 1172468375 8705340521 4759404776 6576644179 7445262808
		3861470304 9050305068 0495635797 3845519048 8415608629 9067086220 2363648575 1844382068
		7514628723 8018078702 0807513584 5519641230 6097782979 8611932591 8602716806 6077680023
r_3	=	47936722 1324440104 1302788851 1389447221 0446440209 1171083926 4776253736
		4539746403 8001578723 5615019418 1172468375 8705340521 4759404776 6576644179 7445262808
		3861470304 9050305068 0495635797 3845519048 8415608629 9067086220 2363648575 1844382068
		7514628723 8018078702 0807513584 5519641230 6097782979 8611932591 8602716806 6077680023
r_{oaep_512}	=	84629 4561031082 0848364283 6464927423 0275240543 9834618616 0752812353 2454141256
		1247803792 8634961254 2153256413 5746078706 8970698750 8744563522 7490097466 782894686
r_{oaep_160}	=	788255724614721016190591162463944054696650907899

表 D.6: 1,024 ビット素数 p_4, 160 ビット位数のベースポイント $\mathbb{F}_p \ni g_4(\mathrm{ord}(g_4) = \ell_4)$, 158, 160 ビット乱数 k_4,r_4)

p_4	=	141108755 3329747116 0681521826 3958123381 1845882120 6101844813 6404826965
		8894330794 5378916621 8230378522 2285640581 2786036719 0611065605 3750255462 5753148936
		9344062782 5218069782 1880894009 1447658298 3518536032 3706998059 7505163602 4730956156
		7099846439 1197300372 9331477720 0949382303 7167642459 3784527310 9255717090 9406945309
g_4	=	79207621 7877600382 3576323926 9746451281 5520975586 2576344005 0213854787
		2406330846 6725739742 1010854631 6235969173 6492935768 1934505810 8657967082 6832189488
		4518347711 0927089585 9682955591 8931536779 2520597630 8332008486 7242870421 4841371963
		6524425738 8686997557 1374550464 4699780995 3054632950 1856786371 3795563229 9024284915
ℓ_4	=	136211592 3099293242 3699222613 0521234356 1846087883
k_4	=	31647783 2003765415 2477353792 6352571815 0288987267
r_4	=	131647783 2003765415 2477353792 6352571815 0288987267

表 D.7: 小さい有限体の例 ($\mathbb{F}_p$, $\ell \mid (p-1)$, $ord(g) = \ell$)

	$\mathbb{F}_p$	ℓ	g	h
(1)	983	491	2	981
(2)	1187	593	3	1184
(3)	9987	4939	2	5
(4)	10079	5039	3	11

表 D.8: 128 ビット整数

K_1(Decimal)	=	184 221027 78699 61103 64334 85937 28750 60126
K_1(Hex)	=	8a 97 ad eb da 4b c6 5e 07 2c 25 a8 70 dd 93 9e

表 D.9: 512 ビットの素数の組 - 1 (p_1, q_1:512-bit, $n_1 = p_1 \times q_1$,e_1: 32-bit, $gcd(a_1, n_1) = 1, |k_1| = |n_1|, k_1 < n_1$)

n_1	=	1228340282 9371229672 0762660347 5421461646 5209348167 0201947376 3249930095 5769193300 3435610085 9768987586 8733831669 2229094216 9535836110 2860006897 5461290111 5941975224 8671316105 8298860890 9985552715 9045423477 4552183738 8939563882 4495933403 4615659391 1239817569 4572958404 0032409298 3089026564 0578775821 4276110040 436127519
e_{1-1}	=	576545171
d_{1-1}	=	3240119253 6282284985 2937640680 1561754108 8107068689 9129710462 9180360828 1162104458 5516888062 8895806434 8844691177 5283356386 5361311664 1193673091 5287548583 9148644717 9012634626 9963262730 4171809163 9197392417 5886127177 0064850130 8653623794 2258743958 0303085709 8539375261 9828361163 7092685818 0336861819 3220918317 49480363
e_{1-2}	=	4170652133
d_{1-2}	=	4400877548 0361668477 6773407914 8433938575 8776325274 4162519273 4300666321 9106026995 1887543716 4590262996 6377504884 3670717399 1430338401 4636764437 7666294961 3111452313 0465800850 7271486013 1440747661 6710751735 2605176547 9187252258 8693284457 7558783953 8673210507 4595941943 5464851966 3281747671 7970541520 5792541224 47719913
a_1	=	5517136991 0051437990 4126907809 7691886400 4594400468 0579369956 2639916993 5459645309 2936382738 3901143322 0903844032 9782230806 5100196316 5983148138 9038664650 7034968771 7707483416 9602476092 5769897674 8740151772 0715057633 3584463725 8758436116 1658474605 4957823806 8993246164 2739285921 0459388248 9289102848 3858898273 48866771
k_1	=	4790803212 4466708640 5082821139 6166041485 1281601799 7379478442 0229758404 2029548314 4737351219 2239347898 9236470550 2874127684 5024428026 2453424885 2004107095 2254994809 2068728231 9494673491 3741339098 4868117642 2885271685 2146572363 2643962930 7682906468 8036543959 7335446193 2808904382 8142255591 0552145824 5050864185 30684390

- $e = 65537$
- d = 1314 4330 4793 9274 2815 1332 2215 1110 8704 8800 6794 7255 3856 1657 4001 1065 4335 8350 5126 9502 8610 5916 8266 4665 9981 4975 9408 7235 0366 8627 3602 9419 4220 3190 2814 0588 3458 2135 8616 7270 2832 5663 4412 3191 3951 9923 8284 5289 4484 7397 4065 2028 8349 2432 5486 9474 6970 3072 7830 5986 9110 9663 0577 7683 5942 6701 4527 7055 0634 5806 0568 6885 5006 9304 5782 5343 83513
- p(512 bits) = 1206 2671 6082 2359 2785 5576 1048 8590 6100 8357 3423 8736 6140 1669 6310 1629 7538 2086 8976 8836 8639 0733 7656 7629 6141 5197 6813 3657 6681 3326 0302 5235 6115 3684 2905 1500 5877 3339083
- q(512 bits) = 1257 5048 0619 9176 2258 1325 1545 6708 8066 2668 8217 7513 6218 9646 1515 4315 9731 4091 0180 3505 6460 9356 3622 4797 3606 1395 5798 7683 1444 7862 1639 0463 8882 4332 3252 3292 3770 8577867
- n(1024 bits) = 1516 8867 5229 4351 4454 5762 6307 6551 3843 8409 0531 8881 4264 3341 0993 3178 2940 7927 7610 5818 0842 6192 4822 6698 6356 8833 9185 9396 4183 7133 8494 7350 9537 1904 9387 6558 1690 4550 7678 5281 8023 3325 1445 8438 2935 8013 4439 8779 8968 7034 2021 4994 2241 1253 8038 7654 9156 1264 5588 1007 4903 3611 6762 8966 4911 1675 3399 8425 5409 2820 3432 8119 9277 1620 6914 8998 75961

表 D.10: 512 ビットの素数の組 - 2 (p_2, q_2:512-bit, $n_2 = p_2 \times q_2$, e_2:64-bit, $gcd(a_2, n_2) = 1$, $|k_2| = |n_2|, k_2 < n_2$)

p_2	=	1185308576 3133505990 2862012693 6392113175 5361282207 7718064542 1682390646 3045956110 4040185181 1887198856 1332197770 0151754580 2509656687 5663947404 2315970486 49687
q_2	=	1324147987 9786700069 5376730134 0228371346 1303895018 6387250437 1572833335 1182535991 7916987304 5490567956 9343041660 9104011044 9999896454 1968688709 3605287712 69943
n_2	=	1569523966 4591850295 5193834718 7763885068 5739921301 9211056913 5670254477 8362395342 4981098100 3203717970 3739071352 7200065108 0485043023 4788150574 2461209041 7644696580 5210049306 3095556772 1591369643 3784788243 3273567579 1585286253 0141525314 5239724411 4433931271 4812961056 4195773940 5631844445 8695333074 1471865748 319457841
e_1	=	16906953396398285955
d_1	=	7243120132 7459355910 4718233668 1310111949 9609639791 1006981852 1891729612 3904894886 3282836141 9263587738 2223140871 2979119342 9476334875 0948968199 9208855556 6997541879 1136760295 4918028434 9351662954 6984571593 4583725830 1381051407 1426513902 7719788715 2694674835 9902661782 3666098202 9008713958 6496712874 3435198316 095145
a_2	=	8273172872 5194908289 2236176663 6745746952 8377629992 3444868922 2799947380 8545734393 6014816382 1388375508 2805006377 0654397886 6124357334 0349538848 2362613359 2304202545 9102737381 9491870149 9926161168 4851439601 1676206647 5445546988 0526500359 5439269501 5391883077 9325957499 2519364080 1344009225 7831969264 5061721961 99532825
k_2	=	1510611613 3679978202 7909948821 5310586749 0950348922 5543744191 8572391793 0120874747 9918027528 6680672639 8883021452 9226358772 3560597448 1819755657 0073572539 5153114171 7413494388 4071414804 6233931768 3191431043 6415866358 5331414086 2052956067 9609405497 0175524650 6679069530 0301408503 5728199079 1590484047 2327542139 70906224
r_2	=	1113587397 5794446838 4163550742 2549378044 1326065243 8479685343 4545266502 4892484614 0194858368 0152878868 2775275081 7988170103 5410791946 6010736321 8574957085 62039

- *message*(1020 bits) = 9380 0268 1455 1100 0395 1680 5740 3795 8517 3246 9117 6630 2786 1995 5375 3827 8130 2089 7087 6198 1337 1981 7277 4031 4623 2740 5830 0468 4100 7656 3368 6228 4802 8671 2159 1758 3366 1085 1174 7268 6236 3297 6213 3266 0034 9018 3723 5938 1406 7167 0583 7687 6777 3503 1671 0176 7574 8670 5474 3880 6960 5213 0236 5172 8708 6261 5423 8459 3797 2545 6067 7419 9244 0521 3408 8576 557
- *cipher* = 1239 6790 8186 5578 9642 5405 9623 5486 6844 0982 3142 1600 8174 4953 5819 6363 7038 6369 1701 2916 6369 4436 9885 3804 5422 6106 9823 1039 1218 7564 9076 0912 3506 8030 6321 1324 0401 7484 2792 7534 2191 9960 7275 1468 8574 6587 0013 0143 2634 4452 7712 9764 4751 4197 5020 8630 9446 7198 5582 8540 5642 9300 4223 4783 2474 1826 9158 6957 7453 0854 3340 9322 2649 0751 7184 1991 15712

演習 4.1 (RSA-OAEP 暗号) :

- n(1024 bits) = 1319 9664 9081 9883 0981 5009 4122 3160 6409 9988 7200 8467 2203 5670 4480 6582 0632 9986 0177 4142 5592 7395 9878 4901 1474 9026 2698 2832 6520 2147 5938 1792 6551 9984 5793 6217 7299 8404 3905 4838 0689 8514 0623 3864 9654 3388 2904 5552 6885 8728 5851 6219 4605 3376 3923 1268 0578 7956 9268 2905 5995 9042 2046 7205 8771 0762 9271 3074 0460 4424 3853 3124 0538 4889 8103 7901 24491
- $e = 17$
- d = 1164 6763 1542 9308 6160 1478 8931 4553 5067 6460 6353 6882 8414 9120 9835 8748 8793 8222 9568 3066 9640 6525 8716 3148 0713 1493 7674 9558 2223 7189 0533 6875 8722 3515 8053 1956 8205 7415 6366

表 D.11: 512 ビットの素数の組 - 3 (p_3, q_3:512-bit, $n_3 = p_3 \times q_3$, e_3:96-bit, $gcd(a_3, n_3) = 1$, $|k_3| = |n_3|, k_3 < n_3$)

p_3	=	9599724472 0774591709 3840929535 1348996733 2868455933 8072367828 7015797869 3765829946 9871755715 9063112480 4878040576 6720230511 2706638969 5431443647 0919347304 8347
q_3	=	1215903644 9739250060 6096309888 3622687320 0838844190 0239984788 8102552041 1437687569 8709068905 7956108167 0068532247 4247654605 6472174913 0863519839 4434242804 86653
n_3	=	1167233997 6344370570 8568459695 4891354662 7640956374 5323740425 7136259107 5486777858 4330574304 2743079604 8925073072 9776954865 8802927423 4069501543 2095205032 8146497566 5678480630 4411866553 7806278220 5022188868 5962926806 0292775049 8387055511 2017587086 2496314002 4150305756 4836383137 7331056181 5698080034 2118804904 557212591
e_3	=	394203512154731469664702999 31
d_3	=	1393788582 3096711794 5389439208 3928985141 9878930943 6919363645 3686076511 5415925578 7954214246 2456048851 5326679482 8769294798 3386170094 6333089661 0365306575 7028177682 1642299039 2185974956 8500878836 6679109934 8267920306 0551049688 3010641915 8921960476 8948555056 8282679006 4888550118 6860360638 8727010900 6453180955 69480383
a_3	=	8767142442 1652388950 6426896438 9619900644 5185897064 6380740588 9616808859 9903264970 8622796868 4417061150 6580163120 4547948483 9313426559 8192406101 5335286113 5763140223 4248276942 3026303535 6103843587 6081389801 0277191736 2219512572 2971702016 0175850000 5347742292 5129871008 0476404456 6077962563 3162226643 5475737467 43342095
k_3	=	8775639933 8693425772 3588635722 2170288853 0279473215 3150830915 4201012623 1959240646 1540908092 6736780704 6443318462 9189948399 7539515433 7145392059 2734621419 0589329251 9252249862 1356874120 8948106741 1690879810 9795740849 4921032742 8566805387 3689318129 5101313209 0781006976 4941637729 7019783074 2959418297 8513273171 579104
r_3	=	9252603648 9950447750 1920424993 8037126999 6940321772 1673727433 0948174216 3968887187 9856045244 6856381282 8779931444 2168464249 3763696467 3275215265 8852560434 6551

表 D.12: 256 ビット，341 ビット，342 ビットの素数

$p_{256,1}$	=	9941433401 5203114960 9526738843 3997810424 3180737243 6577343921 1785285298 5243877
$p_{256,2}$	=	11188162371 5119180788 5222912270 5022208805 8587137547 7054079219 3728086967 85333197
$p_{256,3}$	=	1024251801 4062364432 9684971746 6346801594 2762407940 7752936443 1626609164 33842869
$p_{256,4}$	=	9699716940 1433342338 0020821998 2175684873 2698444814 9706418755 6601377921 5176849
$p_{256,5}$	=	1121860325 9323072578 9036535291 0752951628 9143173589 0576369178 5637290233 96691719
$p_{256,6}$	=	9859465706 2062438738 9997689888 8105295949 9441773971 2724620644 1568882854 3936849
$p_{341,1}$	=	3891959031 0264267341 3700056810 8937900354 1022208879 8978305407 4296698666 3269387607 6331389547 1080407684 987
$p_{341,2}$	=	4427097703 0367742933 4184751354 2455648045 4499812140 3253376000 5166058967 8024194060 9945892334 5179373991 057
$p_{342,3}$	=	8177691406 2846101635 6186958356 4507224208 5812741772 4007701629 2017078599 5286020516 0221865297 6276439249 321
$p_{341,3}$	=	3921014898 8611599345 1163009455 0579065995 8724470558 5260627058 0083365821 3687560719 0150346213 9246480070 829
$p_{342,2}$	=	7269804724 8172173272 9811908562 0946690548 2882694064 6216307713 4902269530 0339558444 9252380964 1760407629 981

表 D.14: 1024 ビット RSA 公開鍵と暗号文

e	=	3
$n_{1,1024}$	=	1246424028 1595120124 2747411048 4548817765 5482865688 7198056462 2526090307 4883138802 6169367886 0579097227 2349519097 8994958671 2501040015 0427991456 9506822012 5866775284 7150923689 3066329423 9894595245 0596032798 7094888333 2207128158 6978694758 1477630705 5697445656 0883416827 6981860601 4379983270 1591647584 5782191229 582548671
$n_{2,1024}$	=	1237863469 0741288296 4224724042 5419819269 8133253860 0075571606 9588423881 8598472030 2656988531 5509973507 7662798128 7953491799 1585529667 1584860366 3159475522 4552690063 9500214460 5702485087 4617781911 7802710430 3037933642 0260049069 4377244926 5720354131 4818561339 1009125141 4421228217 3278068615 6155013399 4126667410 810563519
$n_{3,1024}$	=	1064418998 9059955334 7630116702 4849141577 1282674064 9823955954 4338235351 3660437238 7646213947 4028499878 3804836338 0853826216 7484206192 1832754289 4536428751 7842431103 4086734609 3680931513 8345730774 2663068860 6825876526 6265585620 8539941172 9642335117 5599110065 0502643328 4012784100 5067880507 6279630607 3185853145 846863439
c_1	=	2100950624 9788720465 1237831931 4715382676 7557898188 5699760993 1718957066 3494538606 8910263847 5923880354 1740356375 2411464035 0716587886 1319008822 2120540910 3570740523 2499036650 4845919640 7190262181 1364677555 5623045502 1232905215 0787503325 7137330502 2074867401 0826106297 8156877512 2929190962 6926911763 3392467558 21527505
c_2	=	4556366548 5889359707 4708385715 5607402658 7199909856 3901415900 6675693178 1719919662 5451177626 4689372837 1510284791 5497932211 4334870964 3853818922 4419713601 0555631921 3962946763 3268925800 6266106198 1902736550 1020905726 5425967389 1898635555 7002192580 8597370512 1245127078 2318433520 4110576710 4333960048 9978366319 74780954
c_3	=	6839735635 7072083459 8789613303 2691362866 1403009598 6242426683 4581313167 5542210793 5495017335 0823604850 0748483820 5378347211 6780014827 8125841553 2899328876 2973217361 1418702231 4133204950 7480764848 6904063885 7260414209 6207092226 3405917155 4458251955 4279510671 7051600255 0166966933 2096655158 5876408888 2784613834 36211255

8437 3315 6436 1630 9716 4007 9679 0490 0300 7752 2365 8035 4323 3292 3992 4506 4743 9719 6947 3468 3045 3671 4979 0102 1988 1003 3962 3586 1837 0829 4418 9542 5705 7285 2387 4962 1070 52993

- r(160 bits) = 976172171425244805716717114288565309992579413391
- *message*(128 bits) = 282081456775998934689133617587168685129
- *cipher* = 1287 0107 4370 3005 4101 4829 3473 8564 5297 5385 3252 4014 8798 3564 6041 8427 8059 2622 7601 7913 5505 2449 2861 9141 9379 9268 2181 8698 2531 0459 4013 9614 8196 5788 5439 9251 3353 9694 0841 4762 9392 2557 4343 5568 7615 8351 8349 8057 7594 8578 8756 7504 8670 6621 3670 3602 0679 0489 8189 4026 0911 0471 9058 3254 6285 8524 4432 0269 5189 6761 2457 4483 7776 0141 0937 9749 7173

演習 ??(ハイブリッド暗号) :

- RC4 key(hex): 8a 97 ad eb da 4b c6 5e 07 2c 25 a8 70 dd 93 9e
- e = 16906953396398285955
- d = 7243 1201 3274 5935 5910 4718 2336 6813 1011 1949 9609 6397 9110 0698 1852 1891 7296 1239 0489 4886 3282 8361 4192 6358 7738 2223 1408 7129 7911 9342 9476 3348 7509 4896 8199 9208 8555 5669 9754 1879 1136 7602 9549 1802 8434 9351 6629 5469 8457 1593 4583 7258 3013 8105 1407 1426 5139 0277 1978 8715 2694 6748 3599 0266 1782 3666 0982 0290 0871 3958 6496 7128 7434 3519 8316 095145

表 D.15: p : 1024 ビット素体 ($(p-1)/2$: 素数) と 128,256,512,768,1024 ビットの元 (その 1)

p_1	=	8988465674 3115795386 4652595394 5123668089 8848947115 3286367150 4057886633 7902750481
		5663542386 6120376801 0560056939 9356966788 2939488440 7208311246 4237153197 3706218888
		3946712432 7426381511 0980062304 7059726541 4760425028 8441907534 1171231440 7369565552
		7041361858 1675255342 2931491199 7362296923 9858152417 6781648121 13740223
$g_{1,128}$	=	2552117751 9070384759 7530955573 826162347
$g_{1,256}$	=	8684406692 7987146567 6782387565 1593088995 2488499230 4230295931 8800593484 7271147
$g_{1,512}$	=	6703903964 9712985497 8701249910 2923063739 6829102961 9668886178 0721860882 0150368892
		8049017446 5278875284 8300246170 0109663569 3909502908 4762428253 2031682019 4359
$g_{1,768}$	=	1164388569 2255317013 6173461634 6876916442 6645128375 2245835428 9028519538 2145147960
		7255608874 9218584432 4878640038 6897578427 0449277426 4638343550 2693722031 0959401380
		1431063138 0433259038 8996081885 3196698010 0327809929 8133116068 7893683471 79
$g_{1,1024}$	=	8988465674 3115795386 4652595394 5123668089 8848947115 3286367150 4057886633 7902750869
		6958773138 3843755525 5432172565 5745109003 3367405855 9993074255 9302280579 0866981827
		4596257237 0038316637 2993914345 7307393904 9634469093 1157838013 4754021809 5687344708
		8871699796 1490511695 5592352172 3079247292 1099216175 3478101745 73488207

表 D.16: p : 1024 ビット素体 ($(p-1)/2$: 素数) と 128, 256, 512, 768, 1024 ビットの元 (その 2)

p_2	=	1348269851 1467369307 9697889309 1768550213 4827342067 2992955072 5608682995 0685412572
		2349531357 9918056520 1584008540 9903545018 2440923266 1081246686 9635572979 6055932833
		2592006864 9113957226 6647009345 7058958981 2214063754 3266286130 1175684716 1105434832
		9056204278 7251288301 3439723679 9604344538 5978722862 6517247218 169050179
$g_{2,128}$	=	1701411834 6046923176 8580791863 303232283
$g_{2,256}$	=	5789604461 8658097711 7854925043 4395392697 5274699741 2204831921 6661138833 3043903
$g_{2,512}$	=	1173183193 8699772462 1272718734 3011536154 4445093018 3442055081 1626325654 3526314469
		3967908773 2709146407 0213786563 8210285644 8170028802 4492743180 6255034292 94719
$g_{2,768}$	=	7762590461 5035446757 4489744231 2512776284 4300855834 8305569526 0190130254 7634320185
		0973369147 0114228130 0024207119 4693015968 9063824154 3513075716 5865401550 6642323711
		1859962478 7279773807 6833618935 6207375228 6041172683 8663514724 2081213868 3
$g_{2,1024}$	=	8988465674 3115795386 4652595394 5123668089 8848947115 3286367150 4057886633 7902751257
		8254003890 1567134250 0304288191 2133251218 3795323271 2777837265 4367407960 8026403985
		7315859444 1654511513 5187181774 0075695447 8584579380 1517624120 6572782104 9693276738
		7003840586 1449063025 6550426820 4737525528 1030836284 0296372329 25182547

- p = 1185 3085 7631 3350 5990 2862 0126 9363 9211 3175 5361 2822 0777 1806 4542 1682 3906 4630 4595 6110 4040 1851 8118 8719 8856 1332 1977 7001 5175 4580 2509 6566 8756 6394 7404 2315 9704 8649 687
- q = 1324 1479 8797 8670 0069 5376 7301 3402 2837 1346 1303 8950 1863 8725 0437 1572 8333 3511 8253 5991 7916 9873 0454 9056 7956 9343 0416 6091 0401 1044 9999 8964 5419 6868 8709 3605 2877 1269 943
- r = 1316 4778 3200 3765 4152 4773 5379 2635 2571 8150 2889 87267
- Encrypted RC4 key(base64): zuBD yh4E V6Il Mx5G XPn0 H6sq C6os i0nt aZMg Y54F udMv Kf0g SEc2 1wli afQ9 WOtm RtAr viQy upmv BuC5 V4Zj q7PA WVaQ iOJ2 mYZ4 eNSg lOu3 NBAX p2BS xYd6 e2jX 6Qfi nVG5 aJjo f0gT SLj6 qJBJ 2YHh DOnZ 9lDL gnVw gspG rrA=
- Encrypted RC4 key(整数): 145273291712204045158962174120528973828600675542426587081732584507

表 D.17: $n = pq$: 1024 ビット剰余環 $(p, q, (p-1)/2, (q-1)/2$ は素数) と 128,256,512,768,1024 ビットの元

p_1	=	1173183193 8699772462 1272718734 3011536154 4445093018 3442055081 1626325654 3526314353 6047016400 1089604049 9228777875 9131752944 9703464746 0453248703 1294428803 95983
q_1	=	1089384394 3078360143 4038953110 4224997857 6984729231 3196194003 9367302393 3274434756 9186515228 6726060903 4998150884 7765199163 1867502978 4706588081 4773398176 73279
n_1	=	1278047463 0661777156 5130290907 6572271556 5305084667 9607905329 1983230755 7420547334 0977159933 0963991076 4001508096 1471068715 2105458512 6649931755 3508720223 7375851935 9024330520 1232849812 3855614623 9684904495 5738773699 7331687379 3628315859 6706058528 2374756062 8380733094 0868908971 4058628589 4958057351 3082585825 238038257
a_{128}	=	2552117751 9070384763 4424443721 245264803
a_{256}	=	5789604461 8658097718 0625942397 3063469081 1064122948 8868993314 1554399978 6647923
a_{512}	=	6703903964 9712985497 8701249910 2923063779 0849164925 9116807405 9761961025 6288419690 1976063991 1946823578 2342718002 7423083692 5875051909 7304113745 3916897497 7699
a_{768}	=	7762590461 5035446757 4489744231 2512776284 4300855834 8305569526 0190130254 7634318844 3165439204 4143232389 7524386534 8565576005 0919864165 3661947476 2243781134 3956493414 9795345709 2907345638 6239242514 8345903233 7150007697 9077070121 1247732673 9
a_{1024}	=	8988465674 3115795386 4652595394 5123668089 8848947115 3286367150 4057886633 7902751257 8254003890 1567134250 0304288191 2133251218 3795323271 2777837265 4367407960 8025063204 9385916847 0658771263 5366597161 2635731633 4624590395 0389383758 4952365836 3862980532 2387071148 9020893966 2174006034 3265530638 9865850325 3851769245 89179087

360033231838732069965955941276525033418731548911844582253463336814638856542466362558462321
015791186905279637447084044500685291198196698311594573582292236616661023712917628388188800
497383245071945795063028483560718986012802944929382396103929520

- message: "Cryptography is the science of writing in secret code and is an ancient art; the first documented use of cryptography in writing dates back to circa 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription. Some experts argue that cryptography appeared spontaneously sometime after writing was invented, with applications ranging from diplomatic missives to war-time battle plans. It is no surprise, then, that new forms of cryptography came soon after the widespread development of computer communications. In data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about any network, particularly the Internet."
- cipher(base64): 6jc0 yqPw Z9YO iTGr 7DMT 23Qd CQ88 PL6m fVXZ 9nPP r6gA 5Rol BkwZ TjOG 3x9x aVw+ OhKb Zvcl KI0Y 4xHR zTSn ArcV B9ZL Og9o tBKR upJr 7uQw suiz sWt9 PB35 Qvvf 1GQI WZml GQO7 vKF1 eMd0 zNN7 V6uw zK68 DfLF TUAv wAdI GTl1 BblO A0Lh FrKc uwyU ZYnc nrr9 KfvH F9TA UGIJ 8g/o UEKW cdSA Nvzh 6ZP/ XTbx ZOQq +F/Z gau0 SOFO Hub/ FD7y rry0 MlAb cUKu 7Oea K4JC Z7NH iRwL J8PV O7jW nBNH 85T+ Yeu2 cW+I fgUd c1Es IBPK 0spB qHGv zB8f mQPB 5bDP Jn/n 0P8f DwH0 if7R +K1/ +9k9 GkYY 0Cwc H+xn NG+lB RWi3 1h1S 5+3Z tTpV 6VtH ndGf VV74 cmmQ PIH6 2Sr4 xcQX FliI q215 AMHy 9Wp9 FPiI hVR+ l44j xfkq kbM2 3qjO K4iQ DO7d sM/d 8nUx Wz81 jXtn e8gL CEYr qFcJ Kx5p Jfv/ W6as PPt+ UUHG 4Os9 uRN3 /iNs XsVn p+uL MmY7 JS02 G8vS czRj pHEv /gqk czy5 k0my 5vR5 Nlyc jbBm +nJX 6DFB TQjo 7ACk zaew unVg FDbK uLUf hxW4 NSiB Y2P3 hlY6 EqWy 0Gwg WsEK +U+t 7m/w vugE jBAy /KQX j/wT pDtg Uqup YBML V2wU +Xy3 JfMO qRu6 OmvP ojoa ftcg +xL0 YkXO es0L i4b5 ukyW iJVP 4PZ2 Z1ih vT8z aTYO sTZq 4Q+W bfdc 5Ldt RlD8 99XF XeeU UDF0 /vp5 TlO6 gWVm 9xgY fE5a OoE+ 9wKp T5cK ro1b CBtt Xuct M0go

表 D.18: $n = pq$: 2048 ビット剰余環

p	=	1454023946 9065080801 5809767452 5358241658 9338425616 2897433916 7119414033 1168523618
		8685351381 5221293811 7435379669 7291558561 9222073289 3486580852 4552819391 3262618487
		3817537808 2413244334 3053229824 0917686403 1114669374 2503156736 7736159827 1478894003
		0346830169 4479529723 5780926144 0742257188 6322032030 6572401200 175488317
q	=	1765469362 7607133308 6679739854 6684286422 1620281042 9645260978 7424824130 2336440471
		2031675341 2056484628 3347667711 5173708154 9910128299 2676191085 3727177687 8834199205
		8351538570 1738267410 5038784167 2127179634 4720036269 1146540769 9690636019 9373164857
		6968326311 2988996763 3835236762 3454959117 9347209781 4057318289 503503493
n	=	2567034730 9838500937 3589869551 0084673178 8156598839 5695902597 0932225662 3677033941
		5186684386 0566562742 2161027494 1450184187 6050517246 0665734801 7795808716 9265003863
		4892670155 7841773179 8459484985 6702689560 9649811889 1208354113 4907229931 8530916907
		4619851314 4637323504 8505809545 7955761859 6096558122 1323586742 5229578626 0220619278
		7363483086 4407197052 1860058968 5028382544 3387837957 5196127456 2062282796 2478853249
		7792871645 8013172201 7111765176 0753078221 5823571274 0958574844 0039412137 0469752697
		1924253079 6157957533 2159615488 3141141051 2012417050 7533698574 1409703442 9157025737
		8844252796 2248875663 8339952826 1918793897 5568019359 0191281
e	=	1690695339 6398285953
d	=	1940291962 1840482310 0878341486 5995865539 2980349578 1922540631 0949656839 7273271936
		9714347378 4848029933 6739284763 2878880315 9046751898 9052297118 0699668529 2939788058
		3619596106 6636722393 4713308112 1763703850 7178478277 1580592856 8288691674 7230509049
		4240441749 0908646585 7952582031 6530374109 8952585728 2944526744 0547107971 1414576069
		4671801287 9196846053 0470260766 9535495216 2771504883 4687410669 2722634363 7409463263
		3497660656 5202715426 8409638505 8741446565 5111976049 8157013813 6520027950 7956532158
		4997707772 4126728531 2592949708 0052697488 9153145995 1700760347 3104499240 0874789311
		1677726101 6158255837 5382543873 3962117573 5078410582 19373

kH8l fMD9 bdAH NAor Cr2Q eN4J tDQH zGXH cpZX bV02 S14+ Xscl wru/ oSMI 5oXj xk1J fPj6 TZUY nrUY 5t9K C3uM wn9U A9TA 1zAf qtS5 B3K7 /P8

- cipher(16 進数): ea3734caa3f067d60e8931abec3313db741d090f3c3cbea67d55d9f673cfafa800e51a25064c194e3386df1f71695c3e3a129b66f725288d18e311d1cd34a702b71507d64b3a0f68b41291ba926beee430b2e8b3b16b7d3c1df942fbdfd464085999a51903bbbca17578c774ccd37b57abb0ccaebc0df2c54d402fc0074819397505b94e0342e116b29cbb0c946589dc9ebafd29fbc717d4c0506209f20fe850429671d48036fce1e993ff5d36f164e42af85fd981abb448e14e1ee6ff143ef2aebcb432501b7142aeece79a2b824267b347891c0b27c3d53bb8d69c1347f394fe61ebb6716f887e051d73512c2013cad2ca41a871afcc1f1f9903c1e5b0cf267fe7d0ff1f0f01f489fed1f8ad7ffbd93d1a4618d02c1c1fec67346fa50515a2df58754b9fb766d4e957a56d1e77467d557be1c9a640f207eb64abe317105c596222adb5e40307cbd5a9f453e2221551fa5e388f17e4aa46ccdb7aa338ae224033bb76c33f77c9d4c56cfcd635ed9def202c2118aea15c24ac79a497effd6e9ab0f3edf945071b83acf6e44ddff88db17b159e9fae2cc998ec94b4d86f2f49ccd18e91c4bff82a91ccf2e64d26cb9bd1e4d9727236c19be9c95fa0c5053423a3b00293369ec2e9d58050db2ae2d47e1c56e0d4a2058d8fde1958e84a96cb41b0816b042be53eb7b9bfc2fba0123040cbf2905e3ff04e90ed814aaea5804c2d5db053e5f2dc97cc3aa46ee8e9af3e88e869fb5c83ec4bd1891739eb342e2e1be6e9325a22553f83d9d99d6286f4fccda4d83ac4d9ab843e59b7dd7392ddb51943f3df5715779e5140c5d3fbe9e5394eea05959bdc6061f13968ea04fbdc0aa53e5c2aba356c206db57b9cb4cd20a241fc95f303f5b7401cd028ac2af641e37826d0d01f31971dca595db574d92d78f97b1c970aeefe848c239a178f193525f3e3e936

54627ad4639b7d282dee3309fd500f53035cc07eab52e41dcaeff3fc