

情報工学科 レポート

実験演習記録			判定・指示		
	年月日時	共同作業者			
1	2019/06/24				
2					
3					
4					
レポート提出記録					
	提出年月日	期限年月日			
初	2019/07/08	2019/07/08			
再					
科目名		テーマ担当教員	学年	学期	単位
情報工学実験2		渡辺先生	3	前期	2
テーマ番号 テーマ名		学籍番号 氏 名			
公開鍵暗号入門		16268062 SALIC ERTUGRUL			

1. はじめに

本レポートでは、情報工学実験 2 の第 9 回目のレポートとして、公開鍵についての課題に取り組み、プログラムを作成する。そのプログラムの設計や実装について述べる。

2. 目的

本実験では、代表的な公開鍵暗号である RSA 暗号について学ぶことを目的とする。RSA 暗号の概要と RSA 暗号を実装する際に必要な、素数判定について学ぶ。

3. 演習課題

本実験では、全ての課題を EDEN において、「Python 言語」で行う。課題は次の 8 つの課題である。

- 1- ユークリッド互除法をプログラムで実装する。
- 2- 拡張ユークリッドアルゴリズムを用い、乗法逆元を求めるプログラムを実装する。
- 3- 素数 p, q と e ならびにメッセージ $m (m \in \{1, 2, 3, \dots, N\})$ が与えられた時、RSA 暗号による暗号化ならびに復号化を行うプログラムを作成する。
- 4- テキストの 8.2 節の総当たりによる素数判定法をプログラムで実装し、ビット長 n を大きくしていったときの実行時間の様子を考察する。
- 5- テキストの 8.3 節のフェルマーテストをプログラムで実装し、カーマイケル数以外で誤判定確率を評価する。
- 6- テキストの 8.4 節の Miller-Rabin テストをプログラムで実装し、カーマイケル数に対して誤判定確率を評価する。
- 7- テキストの 8.1 節の素数生成アルゴリズムにいずれかの素数判定法を組合せ、素数をランダムに生成するプログラムを作成する。
- 8- (任意課題) 演習課題 7 の素数生成アルゴリズムと演習課題 3 の暗号化・復号化アルゴリズムを組合せ、RSA 暗号システムを実装する。

3-1. 課題 1

課題 1 の設計

課題 1 では、ユークリッド互除法をプログラムで実装する。ユークリッド互除法は、2 つの自然数 a, b の最大公約数 $\gcd(a, b)$ を効率的に求めるためのアルゴリズムである。そのアルゴリズムのステップを次に述べる。

STEP-1: $a_0 = a, a_1 = b, i = 1$ と初期化する。

STEP-2: $a_{i-1} = q_i a_i + a_{i+1}$ となる商 q_i と剰余 $0 \leq a_{i+1} < a_i$ を計算する。

STEP-3: $a_{i+1} = 0$ ならば、 $\gcd(a, b) = a_i$ とし、 $a_i \neq 0$ ならば $i = i + 1$ とし、STEP-2 に戻る。

課題1では、まず、2つの数値の入力をユーザに求める。そのコードは次のようである。
入力された数値が0より小さい場合の対策も考えられている。

```
1. while True:
2.     print("2つの自然数を入力してください")
3.     a = int(input())
4.     b = int(input())
5.     if a <= 0 or b <= 0:
6.         print("入力が誤っています。再度入力してください")
7.     else:
8.         break
```

ユーザが入力を行った後、その2つの数値の最大公約数を求める。3つのSTEPで説明したアルゴリズムを用いて作成したコードを次に示す。

```
1. # 大きい方をa, 小さい方をbとする。
2. if a < b:
3.     a, b = b, a
4.
5. #ユークリッドの互除法
6. while b:
7.     q = int(a / b)
8.     remainder = a % b
9.     a = b
10.    b = remainder
11.
12. print("最大公約数:", a)
```

課題1の結果および考察

課題1を様々な入力で実装してみた。その結果を図1に示す。

2つの自然数を入力してください 13 5 最大公約数: 1	2つの自然数を入力してください 1234 54 最大公約数: 2	2つの自然数を入力してください 48 96 最大公約数: 48
--	---	--

図1 課題1の結果 (最大公約数)

図1より、課題1のプログラムが正常に動いたことが分かる。

3-2. 課題 2

課題 2 の設計

課題 2 では、拡張ユークリッドアルゴリズムを用い、乗法逆元を求めるプログラムを実装する。

課題 2 では、まず、ユーザから法にする数字と逆元を求める数字を求める。そのコードを次に示す。

```
1. while True:
2.     print("法とする数字:")
3.     N = int(input())
4.     print("逆元を求める数字:")
5.     a = int(input())
6.
7.     if a <= 0 or N <= 0:
8.         print("入力が誤っています。再度入力してください")
9.     else:
10.         break
```

次に、与えられた入力に対して、乗法逆元を求める。本プログラムではそれを `modInverse` 関数で計算する。`modInverse` 関数は拡張ユークリッドアルゴリズムを用いて処理を行う。そのコードを次に示す。

```
1. def modInverse(N, a):
2.     N0 = N
3.     y = 0
4.     x = 1
5.     if N == 1:
6.         return 0
7.
8.     while a > 1:
9.         q = int(a / N)
10.        t = N
11.        # N は現在余り
12.        # ユークリッドアルゴ
13.        N = a % N
```

```
14.        a = t
15.        t = y
16.        # y と x を更新
17.        y = x - (q * y)
18.        x = t
19. # 求めた x が負の数であれば正にする
20.     if x < 0:
21.         x = x + N0
22.     return x
23.
24. print("逆元:", modInverse(N, a))
```

`modInverse` 関数が返す `x` の値が逆元である。

課題 2 の結果および考察

課題 2 を実行し、様々な入力を与えてみた。その実装結果を図 2 に示す。

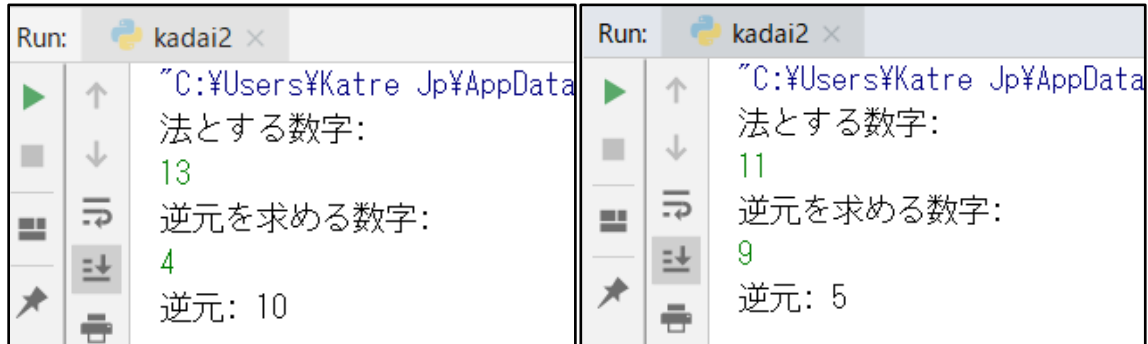


図 2 課題 2 の実装結果

図 2 より、プログラムが正しく逆元を求めることができたと分かる。なぜならば、 $10 \times 4 = 44 \equiv \text{mod}(13)$ 、また、 $9 \times 5 = 45 \equiv \text{mod}(11)$ だからである。

3-3. 課題 3

課題 3 の設計

課題 3 では、素数 p, q と e ならびにメッセージ $m(m \in \{1, 2, 3, \dots, N\})$ が与えられた時、RSA 暗号による暗号化ならびに復号化を行うプログラムを作成する。

RSA 暗号による暗号化は次のアルゴリズムで実装されている。

1) **鍵生成**を行う。受信者は次のアルゴリズムを行う。

- 2 つの大きな素数 p と q を秘密裏に選び、 $N = pq$ を計算する。
- $\text{gcd}((p-1)(q-1), e) = 1$ となる自然数 e を適当に選ぶ。
- 拡張ユークリッドのアルゴリズムを $((p-1)(q-1), e)$ に対して用いて、
$$ed \equiv 1 \pmod{(p-1)(q-1)}$$
となる自然数 d を求める。

- $P = (N, e)$ を公開鍵として公開し、 $S = d$ を秘密鍵として保持する。

2) **暗号化**を行う。送信者は公開鍵 $P = (N, e)$ および $m \in \{0, 1, \dots, N-1\}$ を入力とし、暗号文 $C = m^e \pmod{N}$ を計算する。

3) **復号化**を行う。受信者は秘密鍵 d および暗号文 C を入力とし、平文 $\tilde{m} \equiv C^d \pmod{N}$ となる $\tilde{m} \in \{0, 1, \dots, N-1\}$ を計算する。

そのアルゴリズムに従ってプログラムを作成する。今回は p, q, e の初期値を次のようにする。

- $p = 101$
- $q = 107$
- $N = p \cdot q$
- $e = 3$

また、メッセージ `m` と、暗号文を格納するリスト型の `C` 変数、復号化されたメッセージを格納するリスト型 `m_new` 変数を次のように定義する。

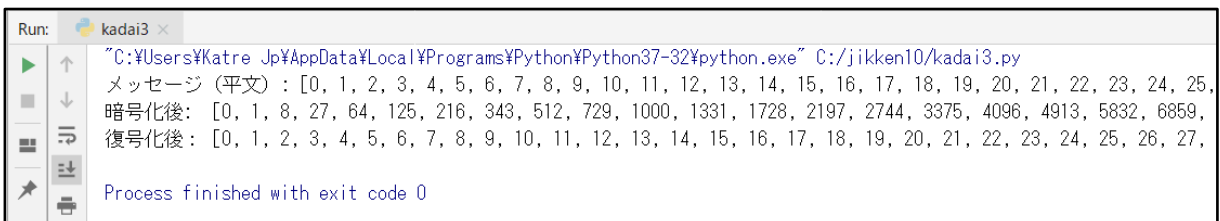
```
1. # 平文のメッセージ
2. m = list(range(0,N-1))
3. print(m)
4.
5. C = [] # 暗号文
6. m_new = [] # 復号化された平文
```

次に、上記のアルゴリズムを用い、RSA 暗号化および復号化を行う。そのコードを次に示す。なお、本課題では、課題 2 で作成した `modInverse` 関数を用いる。

```
1. # 秘密鍵を計算する
2. d = modInverse((p-1)*(q-1),3)
3.
4. # C 暗号文を求める
5. for i in range(0, N-1):
6.     C.append(pow(m[i], 3) % N)
7. print(C)
8.
9. # 復号化する
10. for i in range(0, N-1):
11.     m_new.append(pow(C[i],d) % N)
12. print(m_new)
```

課題 3 の結果および考察

課題 3 の実装結果を図 3 に示す。



```
Run: kaday3
"C:\Users\Katre_Jp\AppData\Local\Programs\Python\Python37-32\python.exe" C:/jikken10/kaday3.py
メッセージ (平文) : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
暗号化後: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859,
復号化後: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
Process finished with exit code 0
```

図 3 課題 3 の結果 (RSA 暗号化・復号化)

図 3 より、暗号化される前のメッセージ `m` と復号化後のメッセージ `m_new` が同じであることが分かる。よって、プログラムが正常に動いたことが分かる。

3-4. 課題 4

課題 4 の設計

課題 4 では、テキストの 8.2 節の総当たりによる素数判定法をプログラムで実装し、ビット長 n を大きくしていったときの実行時間の様子を考察する。

総当たり法による方法では、 $k = 2, \dots, \lfloor \sqrt{p} \rfloor$ に対して順番に割り切れるかどうか、 $p \equiv 0 \pmod{k}$ を確認する。本プログラムではそれを `prime_number(number)` 関数で実装する。そのコードを次に示す。

```
1. # 素数判定
2. def prime_number(number):
3.     flag = 0 # number が素数である場合は flag=0 のままである
4.     if number == 0 or number == 1: # number は 0 か 1 の時は調べない
5.         print(number, "は素数ではない")
6.         return
7.     # 2 からルート (number) までは割り切れるかどうか調べる
8.     for j in range(2, int(sqrt(number))+1):
9.         # 割り切れれば flag=1 (素数ではない)
10.            if number % j == 0:
11.                flag = 1
12.                break
```

課題 4 では、1 から 10000 までの数値の素数判定を行い、数値のビット数ごとにその平均実行時間を計算するコードもプログラムに入れる。

課題 4 の結果および考察

課題 4 では総当たり法による素数判定プログラムを作成した。そのプログラムで 1 から 99 までの数値の素数判定を行った。その実行結果を図 4 に示す。

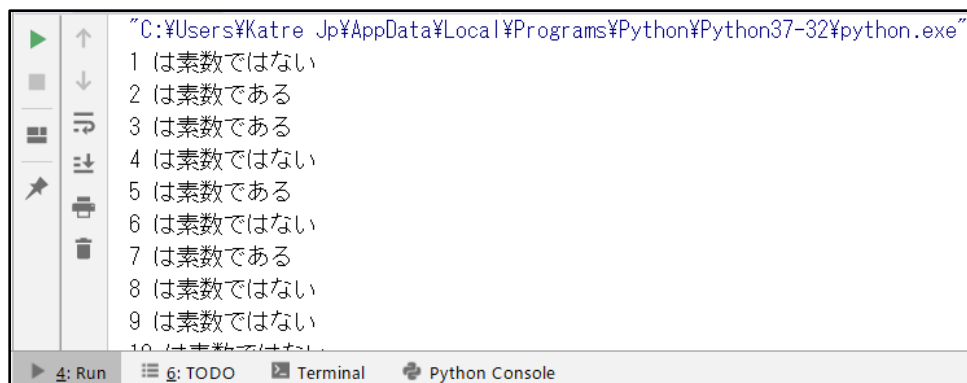


図 4 課題 4 の実行結果 (素数判定)

図 4 には 1~99 の素数判定結果を示したが、全てが図に入りきれなかったのでその結果全体を次にも示す。

1 は素数ではない	21 は素数ではない	41 は素数である	61 は素数である	81 は素数ではない
2 は素数である	22 は素数ではない	42 は素数ではない	62 は素数ではない	82 は素数ではない
3 は素数である	23 は素数である	43 は素数である	63 は素数ではない	83 は素数である
4 は素数ではない	24 は素数ではない	44 は素数ではない	64 は素数ではない	84 は素数ではない
5 は素数である	25 は素数ではない	45 は素数ではない	65 は素数ではない	85 は素数ではない
6 は素数ではない	26 は素数ではない	46 は素数ではない	66 は素数ではない	86 は素数ではない
7 は素数である	27 は素数ではない	47 は素数である	67 は素数である	87 は素数ではない
8 は素数ではない	28 は素数ではない	48 は素数ではない	68 は素数ではない	88 は素数ではない
9 は素数ではない	29 は素数である	49 は素数ではない	69 は素数ではない	89 は素数である
10 は素数ではない	30 は素数ではない	50 は素数ではない	70 は素数ではない	90 は素数ではない
11 は素数である	31 は素数である	51 は素数ではない	71 は素数である	91 は素数ではない
12 は素数ではない	32 は素数ではない	52 は素数ではない	72 は素数ではない	92 は素数ではない
13 は素数である	33 は素数ではない	53 は素数である	73 は素数である	93 は素数ではない
14 は素数ではない	34 は素数ではない	54 は素数ではない	74 は素数ではない	94 は素数ではない
15 は素数ではない	35 は素数ではない	55 は素数ではない	75 は素数ではない	95 は素数ではない
16 は素数ではない	36 は素数ではない	56 は素数ではない	76 は素数ではない	96 は素数ではない
17 は素数である	37 は素数である	57 は素数ではない	77 は素数ではない	97 は素数である
18 は素数ではない	38 は素数ではない	58 は素数ではない	78 は素数ではない	98 は素数ではない
19 は素数である	39 は素数ではない	59 は素数である	79 は素数である	99 は素数ではない
20 は素数ではない	40 は素数ではない	60 は素数ではない	80 は素数ではない	

実行結果より、素数判定が正しく行われていることが分かる。

また、課題 4 のプログラムを用い、1~10000 までの数値の素数判定を行い、数値のビット数ごとにその平均実行時間を計算した。その結果を図 5 に示す。

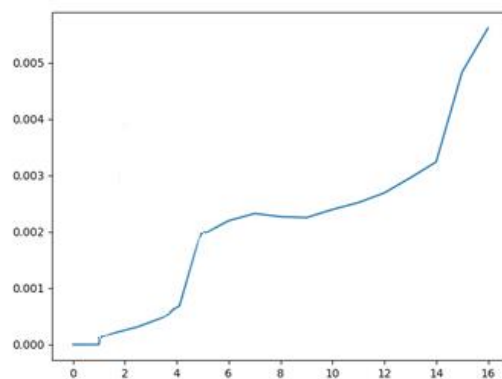


図 5 課題 4 総当たり法による素数判定法 ビット数毎の平均実行時間

図 5 より、総当たり法による素数判定法では、繰り返す回数が長さ n に対して、指数的に増加してしまっていることがわかる。よって、その方法は、課題 5 と課題 6 で取り組む素数判定法と比べて、効率的ではないと考えられる。

3-5. 課題 5

課題 5 の設計

課題 5 では、テキストの 8.3 節のフェルマーテストをプログラムで実装し、カーマイケル数以外で誤判定確率を評価する。フェルマーテストのアルゴリズムは次のようになっている。

STEP-1: $a \in \{1, 2, \dots, p-1\}$ をランダムに選ぶ。

STEP-2: もし $a^{p-1} \neq 1 \pmod{p}$ ならば、「素数ではない」と判定し終了。

STEP-3: $i = S$ ならば、「素数である」と判定。そうでなければ、 i を増加させ STEP1 へ戻る。

そこで、計算（処理）をより早くするために、STEP-2 の前に、 $\gcd(a, p)=1$ であることを確認する。もしそれが満たされていないならば、素数でないと判定し、STEP-1 に戻る。

$\gcd()$ は課題 1 で作成したプログラムを関数化したものである。つまり、最大公約数を求める。また、本課題では、与えられた数値が本当に素数であるかを `prime_number` 関数で求める。それは課題 4 で作成した関数である。

STEP-1, 2, 3 を実装したのが `fermat_test(p)` 関数である。`fermat_test(p)` のコードを次に示す。

```
1. # 素数判定
2. def fermat_test(p):
3.     # p=0,1 の場合素数ではない。
4.     if p < 2:
5.         return 0
6.     k = 0 # Temporary 変数 (LOOP 用)
7.     s = 5*p # ループ繰り返し回数
8.
9.     # テスト開始
10.    while k < s:
11.        a = random.randint(1, p - 1) # ランダム数
12.        if gcd(a, p) != 1:
13.            return 0
14.        if (pow(a, p-1) % p) != 1:
15.            return 0
16.        k = k + 1
17.    # 素数でないと確認できなかったら素数と判断
18.    return 1
```

また、誤判定確率を計算するために 1 から 100000 までの数値をフェルマーテストの対象とし（カーマイケル数以外）、結果が誤っているかどうかを確認する。そのコードを次に示す。

```

1. error = 0 # 誤差数
2. # カーマイケル数
3. carmichael = [561, 1105, 1729, 2465,
4.               2821, 6601, 8911, 10585,
5.               15841, 29341, 41041, 46657,
6.               52633, 62745, 63973, 75361]
7.
8. variable = 100000 # どこまで素数を調べるか指定
9.
10. for i in range(variable):
11.     if i in carmichael:
12.         continue
13.     result = fermat_test(i) # Fermat テストの結果
14.     prime = prime_number(i) # 素数かどうか確認
15.     if result != prime:
16.         error += 1 # 結果が誤っていたら誤差数を増やす
17.
18. print(error/variable)

```

課題 5 の結果および考察

課題 5 で作成したプログラムの実装結果を図 6 に示す。

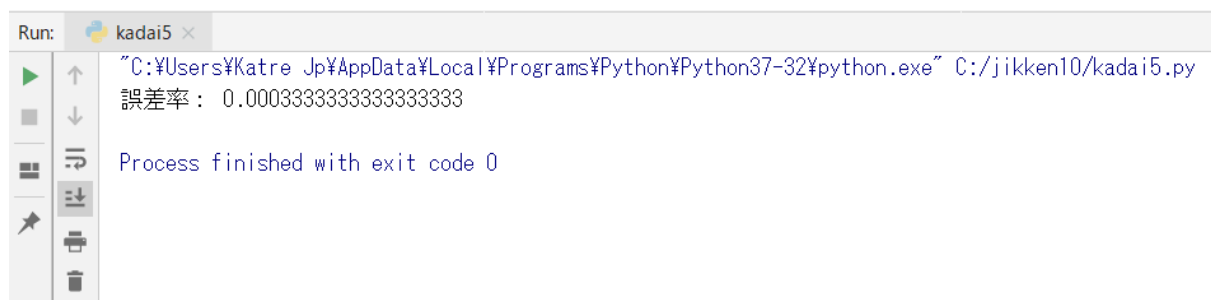


図 6 課題 5 の実行結果

3-6. 課題 6

課題 6 の設計

課題 6 では、テキストの 8.4 節の Miller-Rabin テストをプログラムで実装し、カーマイケル数に対して誤判定確率を評価する。10 万までのカーマイケル数を次のリストで定義する。

```

1. # カーマイケル数
2. carmichael = [561, 1105, 1729, 2465, 2821, 6601, 8911,
3.               10585, 15841, 29341, 41041, 46657, 52633,
4.               62745, 63973, 75361, 101101, 115921, 126217,
5.               162401, 172081, 188461, 252601, 278545,
6.               294409, 314821, 334153, 340561, 399001,
7.               410041, 449065, 488881, 512461]

```

課題 6 では、次ステップで miller-rabin テストを実装する。

$p > 2$ かつ偶数であれば「素数ではない」と判定し終了。

$p - 1 = 2^u v$ となる整数 $u \geq 0$ と奇数 $v \geq 3$ を見つける。 $1 \leq i \leq S$ の間以下を繰り返す。

1. $a \in \{1, 2, \dots, p-1\}$ をランダムに選ぶ。

2. もし $a^{p-1} \not\equiv 1 \pmod{p}$ ならば、「素数ではない」と判定し終了。

3. $j=1, \dots, u$ に対して、 $a^{2^j v} \equiv 1 \pmod{p}$ かつ $a^{2^{j-1} v} \not\equiv \pm 1 \pmod{p}$ となる j が存在すれば、「素数ではない」と判定し終了。

4. $i=S$ ならば、「素数である」と判定し終了。 そうでなければ、 i を増加させステップ 1 へ戻る。

そのアルゴリズムをコーディングした結果を次に示す。

```

1. def miller_rabin(p):
2.     # p=0,1 の場合素数ではない。
3.     if p < 2:
4.         return 0
5.     # p>2 かつ偶数であれば「素数ではない」
6.     if p > 2 and p % 2 == 0:
7.         return 0
8.
9.     k = 0 # Temporary 変数 (LOOP 用)
10.    s = 10 # ループ繰り返し回数
11.
12.    # v と u の計算 (u=0, v=p-1 で初期値を設定)
13.    u = 0
14.    v = p - 1
15.    # p-1 (つまり v) が 2 で割り切れなくなるまで 2 で割る
16.    # 割れた回数が u, 最後に割り切れない時の v が目的の v となっている
17.    while v % 2 == 0:
18.        v = v / 2

```

```

19.         u += 1
20.     v = int(v)
21.
22.     # テスト開始
23.     while k < s:
24.         a = random.randint(1, p - 1) # ランダム数
25.         if (pow(a, p-1) % p) != 1:
26.             return 0
27.         for j in range(u+1):
28.             m = n = a % p
29.             #pow(a, pow(2, j))が非常に大きい数字になってしまうため、gcd(p,
pow(a, pow(2, j)))=1 という特徴を使う
30.             # つまりあまりの乗を求める
31.             for i in range(2, pow(2, j)*v + 1):
32.                 m = m * (a % p)
33.                 m = m % p
34.             for i in range(2, int(pow(2, j-1)*v) + 1):
35.                 n = n * (a % p)
36.                 n = n % p
37.             if m == 1 and (n == 1 or n == p-1):
38.                 return 0
39.
40.         k = k + 1
41.     # 素数でないと確認できなかったら素数と判断
42.     return 1

```

課題 6 の結果および考察

課題 6 で作成したプログラムの実行結果を図 7 に示す。

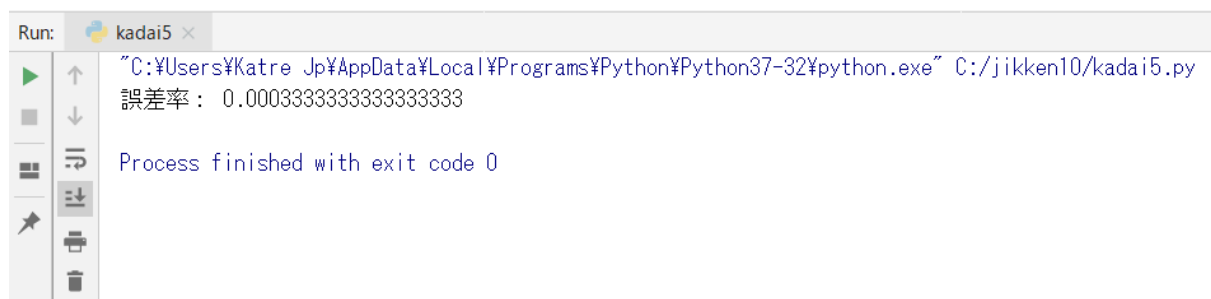


図 7 課題 6 の実行結果

3-7. 課題 7

課題 7 の設計

課題 7 では、素数生成アルゴリズムにフェルマーテストを組合せ、素数をランダムに生成するプログラムを作成する。そのためには次の STEP のアルゴリズムを用いる。

STEP-1: $n-1$ ビットの整数を $\{0, 1\}^{n-1}$ からランダムに選ぶ。

STEP-2: 選んだ整数の先頭に「1」を付け加えて候補にする

STEP-3: 候補が素数かどうかをフェルマーテストで判定する、判定されれば終了。判定されなかったら STEP-1 に戻る。

上のアルゴリズムを `create_prime_number(length)` 関数で実装する。そのコードを次に示す。

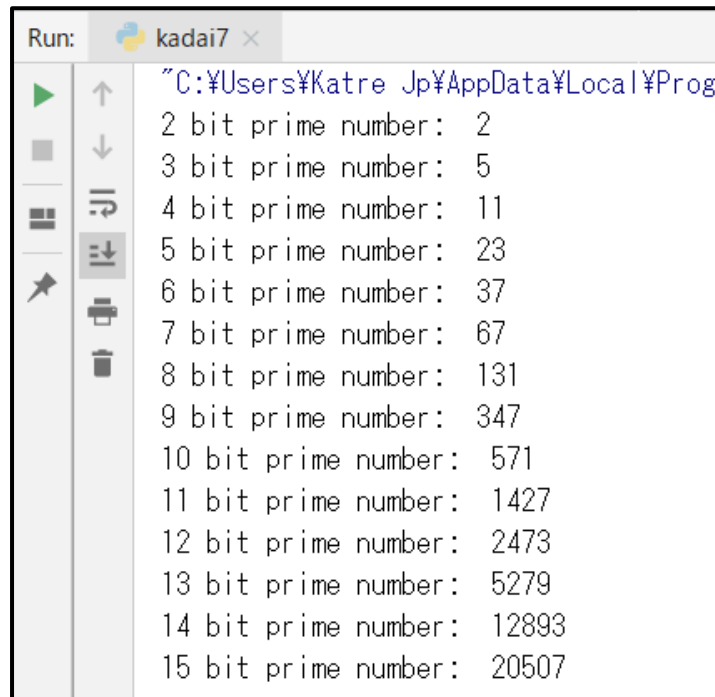
```
1. def create_prime_number(length):
2.     flag = 0 # 候補が判定されたかどうかをチェックするフラグ
3.     candidate = 0 # 候補
4.     K = 3*length**2 # 最大繰り返し数
5.     i = 1 # 繰り返し変数
6.     # STEP-1: 整数をランダムに選ぶ
7.     while flag == 0 and i <= K:
8.         bits = ["0", "1"]
9.         number = ""
10.         for j in range(length - 1): # n-1 ビットの 2 進数整数をランダムに選ぶ
11.             number = number + bits[random.randint(0, 1)]
12.         # STEP-2: 選んだ整数の先頭に「1」を付け加えて候補にする
13.         number = "1" + number
14.         candidate = int(number, 2) # 10 進数にする
15.         # 素数かどうかを判定する、判定されれば終了
16.         flag = fermat_test(candidate)
17.         i += 1
18.     return candidate
```

関数でランダムに素数を生成することができると考えられる。その結果を見るために、2 から 15 までのビット数で素数を生成してみる。そのために、プログラムに次のコードを付け加える。

```
1. for i in range(2, 12):
2.     print(i, "bit prime number: ", create_prime_number(i))
```

課題 7 の結果および考察

課題 7 では、素数生成アルゴリズムにフェルマーテストを組合せ、素数をランダムに生成するプログラムを作成した。その結果を図 8 に示す。



```
Run: kadai7 x
"C:\Users\Katre_Jp\AppData\Local\Prog
2 bit prime number: 2
3 bit prime number: 5
4 bit prime number: 11
5 bit prime number: 23
6 bit prime number: 37
7 bit prime number: 67
8 bit prime number: 131
9 bit prime number: 347
10 bit prime number: 571
11 bit prime number: 1427
12 bit prime number: 2473
13 bit prime number: 5279
14 bit prime number: 12893
15 bit prime number: 20507
```

図 8 課題 7 の実行結果（素数生成プログラム）

図 8 より、課題 7 で作成したプログラムは、決めたビット数で素数をランダムに生成することができていることが分かる。

3-8. 任意課題

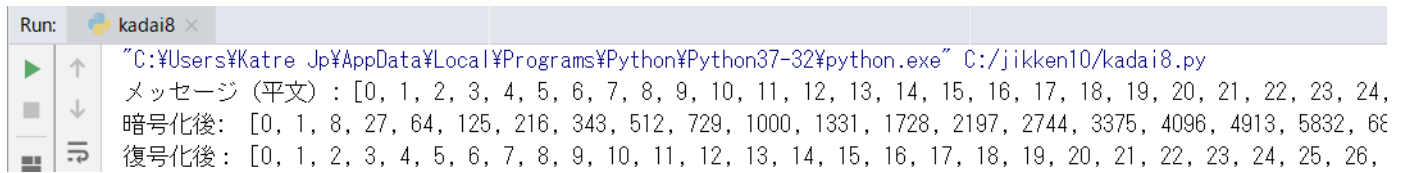
任意課題の設計

任意課題では、課題 3 のプログラムの p と q に素数を代入するところで、課題 7 において作成した素数生成プログラムを用いることによって、RSA 暗号化を行った。つまり、課題 3 で定義した p と q 変数それぞれに素数を代入する代わりに、次のコードを用いた。

```
1. p = create_prime_number(7)
2. q = create_prime_number(7)
```

任意課題の結果および考察

任意課題の実行結果を図 9 に示す。



```
Run: kadai8 ×
"C:\Users\Katre_Jp\AppData\Local\Programs\Python\Python37-32\python.exe" C:/jikken10/kadai8.py
メッセージ (平文) : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
暗号化後: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859,
復号化後: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
```

図 9 任意課題の実行結果

図 “” より、プログラムが正常に動いたことがわかる。よって、RSA 暗号化ができていると考えられる。

4. おわりに

本実験では、代表的な公開鍵暗号である RSA 暗号について学べた。RSA 暗号の概要と RSA 暗号を実装する際に必要な、素数判定について学んだ。

5. 参考文献

[1] 渡辺 峻, 東京農工大学, 情報工学科, 情報工学実験 2, 2019 年度第 9 回講義資料