

Wybrane Zagadnienia Kryptograficzne - zadanie z RSA

autor:

Jan Lewandowski 136761 (L8)

1. Opis realizacji zadania

Zadanie rozpocząłem od ustalenia liczb pierwszych p oraz q . Podstawowo są one 4-cyfrowe, ale później przedstawię również bardziej realistyczny wariant zadania z większymi liczbami. Do szukania liczb pierwszych użyłem strony <https://bigprimes.org/>. W kodzie w sprawozdaniu wyciąłem zbędne komentarze czy też fragmenty związane z printowaniem do konsoli dla zwiększenia czytelności.

Wyzaczyłem następujące wartości odpowiednich parametrów:

```
p = 4751
q = 9007
n = p * q = 42792257
phi = (p - 1) * (q - 1) = 42778500
e = 9421
```

Następnym krokiem było wyznaczenie parametru d . Był to etap, który przyspożył mi sporo trudności. Na początku wyznaczyłem następujące równanie diofantyczne dla d :

$$(e * d - 1) \bmod \phi = 0 \iff e * d - 1 = x * \phi \iff d = (x * \phi + 1) / e$$

Na początku próbowałem realizować zadanie na gargantuicznie dużych liczbach pierwszych o długości kilkudziesięciu cyfr. Nie potrafiłem samemu wymyślić innej metody na znalezienie takiego x by $(x * \phi + 1) \bmod e$ było równe zero (aby d mogło być całkowite), niż brute force, po prostu iterowanie po kolejnych wartościach x i sprawdzanie czy warunek jest spełniony. Podejście to spaliło na panewce dla dużych liczb, w ciągu 30 min nie udało się wyznaczyć wartości d . Patrząc na następne rezultaty i wielkość liczb, zapewne zajęłoby to tysiące lat. Po zmniejszeniu p i q do wartości 4-cyfrowych podejście to dało wynik $d = 42664981$.

```
x = 2
notFound = True
while notFound:
    numerator = x * phi + 1
    if numerator % e == 0:
        d = numerator // e
        notFound = False
    else:
        x += 1
```

Później dowiedziałem się, że można d bardzo łatwo wyznaczyć korzystając z funkcji `pow` w ciele modułu:

```
d = pow(e, -1, phi)
```

W nowszych wersjach pythona podanie 3 argumentów do funkcji `pow` spowoduje wykonanie jej w ciele modułu (w tym przypadku ϕ). Wystarczyło zatem znaleźć d jako odwrotność e w ciele modułu ϕ , jako, że odwrotność w ciele modułu oznacza, że (w tym wypadku) $e * d \bmod \phi = 1$.

Był to jeden z najtrudniejszych etapów zadania. Metoda z funkcją pow działa również bardzo szybko dla bardzo dużych wartości parametrów, co jest niepraktyczne bez użycia tej metody potęgownia w ciełe modulo.

Wygenerowane zostały 2 klucze:

```
class PublicKey:
```

```
    e = 0
```

```
    n = 0
```

```
class PrivateKey:
```

```
    d = 0
```

```
    n = 0
```

```
[...]
```

```
    publicKey = PublicKey()
```

```
    publicKey.e = e
```

```
    publicKey.n = n
```

```
    privateKey = PrivateKey()
```

```
    privateKey.d = d
```

```
    privateKey.n = n
```

Jako wiadomość do zakodowania zadeklarowałem stringa o długości dokładnie 50 znaków:

```
message = 'Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)'
```

Następnie, przy pomocy napisanych przez siebie funkcji, wyznaczyłem zakodowaną wiadomość oraz odkodowaną wiadomość:

```
encodedMessage = [encode(character, publicKey) for character in message]
```

```
decodedMessage = ''.join([decode(character, privateKey) for character in  
encodedMessage])
```

```
if decodedMessage == message:
```

```
    print('MESSAGES MATCH!')
```

```
else:
```

```
    print('MESSAGES DON`T MATCH!')
```

Każdy znak (char) wiadomości szyfrowałem oddzielnie. Końcowa zaszyfrowana wiadomość encodedMessage to lista intów kodujących każdy kolejny znak. decodedMessage zaś jest połączeniem do jednego stringa rozkodowanych osobno znaków.

Przyjrzyjmy się funkcjom encode i decode:

```
def encode(message, publicKey):
    messageBits = bytearray.bitarray()
    messageBits.frombytes(message.encode('utf-8'))

    messageInt = ba2int(messageBits)

    encodedMessage = pow(messageInt, publicKey.e, publicKey.n)
    return encodedMessage

[...]

encodedMessage = [encode(character, publicKey) for character in
message]
```

W moim przypadku każdym z argumentów message jest pojedynczy znak char, zaś publicKey to oczywiście wspomniana wyżej klasa zawierająca 2 pola z parametrami klucza.

messageBits to obiekt bytearray - konwertuję każdy znak do tablicy pojedynczych bitów. Można tutaj było po prostu zgodnie z ASCII konwertować każdy znak od razu na jego wartość int, ale tutaj wypada wspomnieć o drugiej największej trudności z realizacją zadania. Algorytm RSA działa na liczbach całkowitych, więc nie wiedziałem jak podejść do szyfrowania wiadomości tekstowej. Początkowo próbowałem całą wiadomość przekonwertować na tablicę bitów i utworzyć jednego gargantuicznego inta. Oczywiście nie miało to sensu, przy tak małych p i q oraz e nie byłoby możliwości odkodowania wiadomości, liczba kodowana nie może być większa od modulo.

Zatem uznałem, że podzielę wiadomość na bloki - podszedłem do tego najprościej, czyli zakodowałem każdy znak osobno. Zapewne można to zrobić bardziej optymalnie, maksymalizując ilość kodowanych jednocześnie bajtów - charów, tak by było ich jak najwięcej przy jednoczesnym nieprzekraczaniu rozmiaru modulo. W każdym razie konwertowanie do bytearray, a dopiero potem do inta pozostawiłem, gdyż funkcja i tak była już gotowa. Można de facto dzięki temu kodować nie tylko znaki.

Do zmiennej messageInt konwertuję przy pomocy funkcji ba2int z modułu bytearray tablicę bitów do inta. Następnie zgodnie ze wzorem $c = m^e \bmod n$ wyznaczam zaszyfrowany znak i zwracam go. Miałem tu mały problem z interpretacją wzoru. Początkowo zaimplementowałem go jako $c = \text{pow}(m, e) \% n$ co jest oczywiście niedobre, ponieważ przy dużych wartościach e obliczenie c jest praktycznie niemożliwe. Następnie spróbowałem $c = \text{pow}(m, e \% n)$, co szczerze mówiąc nie wiem nawet czy jest poprawne. W każdym razie $c = \text{pow}(m, e, n)$ załatwia wszystkie problemy i nie sprawi, że obliczenia dla dużych e i n potrwają dłużej niż istnieje wszechświat i zajmą googoliańskie ilości cyfr przed operacją %.

Jak wspomniałem zakodowana wiadomość encodedMessage to oczywiście lista zwróconych przez encode intów dla każdego znaku osobno.

Spójrzmy teraz na decode:

```
def decode(encodedMessage, privateKey):
    decodedMessageInt = pow(encodedMessage, privateKey.d, privateKey.n)

    decodedMessage = chr(decodedMessageInt)

    return decodedMessage

[...]
```

```
decodedMessage = ''.join([decode(character, privateKey) for
character in encodedMessage])
```

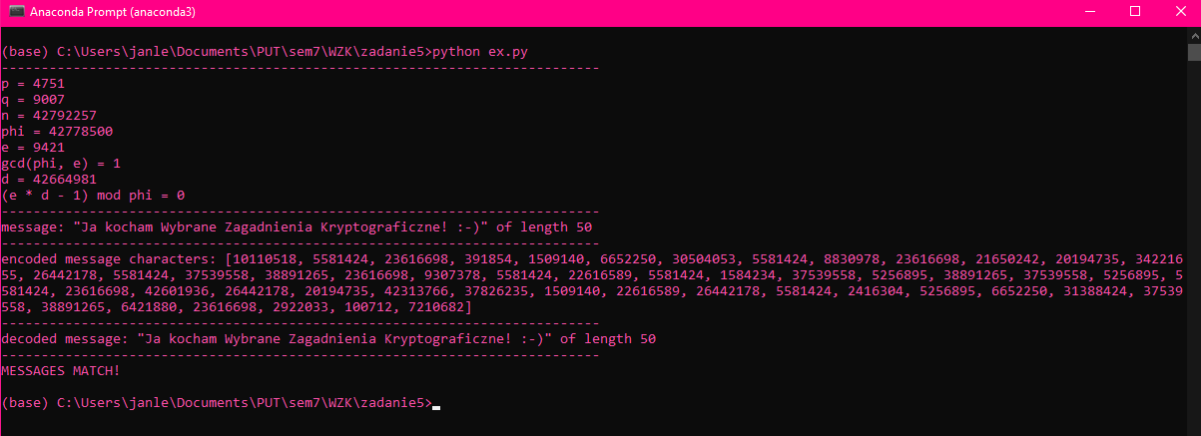
Sprawa wygląda tutaj już prosto. Tutaj już nie korzystałem z bytearray, więc funkcja jest przystosowana stricte do zmiennych 1-bajtowych (char). Rozkodowuję inty zgodnie ze wzorem $m = c^d \bmod n$, konwertuje je na zmienne char i zwracam. Następnie przy przypisaniu do decodedMessage następuje złączenie wyników w jeden string.

```
if decodedMessage == message:
    print('MESSAGES MATCH!')
else:
    print('MESSAGES DON`T MATCH!')
```

Na końcu porównuję wiadomości i mogę powiedzieć, że całość działa i śmiga :-)

2. Wyniki działania programu

Wyniki w konsoli były następujące:



```
Anaconda Prompt (anaconda3)
(base) C:\Users\janle\Documents\PUT\sem7\WZK\zadanie5>python ex.py
p = 4751
q = 9007
n = 42792257
phi = 42778500
e = 9421
gcd(phi, e) = 1
d = 42664981
(e * d - 1) mod phi = 0

message: "Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)" of length 50

encoded message characters: [10110518, 5581424, 23616698, 391854, 1509140, 6652250, 30504053, 5581424, 8830978, 23616698, 21650242, 20194735, 342216
55, 26442178, 5581424, 37539558, 38891265, 23616698, 9307378, 5581424, 22616589, 5581424, 1584234, 37539558, 5256895, 38891265, 37539558, 5256895, 5
581424, 23616698, 42601936, 26442178, 20194735, 42313766, 37826235, 1509140, 22616589, 26442178, 5581424, 2416304, 5256895, 6652250, 31388424, 37539
558, 38891265, 6421880, 23616698, 2922033, 100712, 7210682]

decoded message: "Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)" of length 50
MESSAGES MATCH!
(base) C:\Users\janle\Documents\PUT\sem7\WZK\zadanie5>
```

3. Test dla większych liczb

Przetestowałem również działanie algorytmu dla większych wartości parametrów:

$p = 440652540385366476885648349606092631044387368497923129947867$

$q = 848678487181295759680383727940935479949828086390833874248117$

$n = p * q = 37397233134684765560808849943192458322846640294105262212634461734402599255917479759088968690353910933891470880032916439$

$\phi = (p - 1) * (q - 1) = 37397233134684765560808849943192458322846640294105262212633172403375032593680913727011421662242916718436582123028720456$

$e = 331217644374414450300103554646128062125802050357921772483749$

Jak widać liczby są dość duże. p, q i e są liczbami pierwszymi mającymi 60 cyfr. Widać tutaj siłę potęgowania w ciele modulo. Obliczenia są natychmiastowe, użycie tracycyjnego potęgowania z takimi wykładnikami, a dopiero potem liczenie modulo byłoby szaleństwem.

4. Wyniki działania programu dla większych liczb:

```
Anaconda Prompt (anaconda3)

(base) C:\Users\janle\Documents\PUT\sem7\WZK\zadanie5>python ex.py

-----
p = 440652540385366476885648349606092631044387368497923129947867
q = 848678487181295759680383727940935479949828086390833874248117
n = 37397233134684765560808849943192458322846640294105262212634461734402599255917479759088968690353910933891470880032916439
phi = 37397233134684765560808849943192458322846640294105262212633172403375032593680913727011421662242916718436582123028720456
e = 331217644374414450300103554646128062125802050357921772483749

gcd(phi, e) = 1
d = 34247892498303506446967716711989638012079627014421022318423294080026637055930618354966398827478637759720799864776878213
(e * d - 1) mod phi = 0

-----
message: "Ja Kocham Wybrane Zagadnienia Kryptograficzne! :~)" of length 50

encoded message characters: [36612848020465345770187452045322015813026592472763712593865796265414495938486864313385735917462578316400748732059928881
7, 197513129972780173304207321177036579104101512899479093518197118454194735672327297583243595846045196870655155608787779349, 29635656311399480404314
3335774980194284511836225493637539393416821245061561041110186302831241989481782733286008749349335, 1732771009543689498532787519637173841504109699717
701171631047851020594750237198259694528647417588620560453501457821914460, 413252826117775561343085701852513078030704048865961777200398519426458764881
9108878780151108899388447839585109904773836, 355815847548548117054682100891458653875425301727468391893521380322129293479867755832274695534566061010
446717581406154721, 1024002425870254028743355157008690458821546048758845763116574669475715136702636464408472806389487019631654847626453254, 197513
129972780173304207321177036579104101512899479093518197118454194735672327297583243595846045196870655155608787779349, 16238517479210423630635918581012
7201226535687509784073903969231433264917281414002250116587808156615113570841447396384, 29635656311399480404314333577498019428451183622549363753939
2425661561041110186302831241989481782733286008749349335, 1767613078082846341947173043590999130513400535137615744373631357965439508855826145757
922734332752063101117723072177620987, 6183857489771741823822579273645306040879417234783702324529615377993712494172020667681390859278651587480146232805
82963843, 2412685876393244021516207215192719881943744288845440351508027638038585843327800618352076535155626918870791455648639391, 1202598324916047
30430869841242042630729816130214910208818424977862719053754673645750877650460291447920838457806574121, 1975131299727801733042973211770365791041015
512899479093518197118454194735672327297583243595846045196870655155608787779349, 32591269309783569103265641597608329037400409396007401563900597490470
120140910710998168621769838987813974287251413037436, 1043445429846099577035447256925313139911313671039462421140096319754817011678130443691518365735
24014261177756260466770060, 296356563113994804043143335774980194284511836225493637539393416821245061561041110186302831241989481782733286008749349335
, 2771928864413684395141015795529024396916612674513308166372383104164024612968844003076679233246092722483027574073836131, 197513129972780173304297
321177036579104101512899479093518197118454194735672327297583243595846045196870655155608787779349, 21344581943256723602125373125899268525912563553857
3252341898532619095972338605110791976872994694376587600704816935534192, 1975131299727801733042973211770365791041015128994790935181971184541947356723
7297583243595846045196870655155608787779349, 17215899811107554009617829727772774739661392629104583436100383930317651219461793932639396187234747691
145245084867675378, 32591269309783569103265641597608329037400409396007401563900597490470120140910710998168621769838987813974287251413037436, 1624591
1018213980633038769093731865135543298097021670038728615578560793459829104640682808974741583191605675648585744127, 104344542984609957703544725692531
313991131367103946242114009631975481701167813044369151836573524014261177756260466770060, 32591269309783569103265641597608329037400409396007401563900
507490470120140910710998168621769838987813974287251413037436, 1624591018213980633638769093731865135543298097021670038728615578560793459829104640682808974741583191605675648585744127, 184344542984609957703544725692531
313991131367103946242114009631975481701167813044369151836573524014261177756260466770060, 32591269309783569103265641597608329037400409396007401563900
8089747015831916056756485857244127, 1975131299727801733042973211770365791041015128994790935181971184541947356723272975832435958460451968706551556087
87779349, 296356563113994804043143335774980194284511836225493637539393416821245061561041110186302831241989481782733286008749349335, 1401510466179338
72028399009865298391863333430777466209514129516885833700700391646234859153380837551459292935876177967849, 120259832491604739430609841242042630729816
1302149102088184249778627865158748014623280582963843, 17034319120351889775143509976085075445406518992328028719954328079145038248208320055276367284363
124941720206676813905827865158748014623280582963843, 17034319120351889775143509976085075445406518992328028719954328079145038248208320055276367284363
9350423160773885730006353, 32795171845744519544016920596476346677058691847558542238845673674766731833492356805808911399794255470993220265612129771,
41325282631777556134308570185251307803070404886596177720039851492645876488101088787891455108899388447839585109904773836, 213445819432567236021253731
25899268525912563553857325241808537650460291447920038457806574121, 12025983249160473943060984124204263072981613021491020
8818424977862719053754673645750877650460291447920038457806574121, 1975131299727801733042973211770365791041015128994790935181971184541947356723272
97583243595846045196870655155608787779349, 4956149745993898934073104128700626142392906278674920377621205841545332569620541420876544656496958275337
84841198912664, 16245910182139806336387690937318651355432980970216700387286155785607934598291046406828089747415831916056756485857244127, 35581584754
854817054682100891458653875425301727468391893521380322129293479867755832274605534566061010446717581406154721, 3563751511367108800650009677815440567
560106508080174687813281636081687216053998225442972491816587795305828784449091074272, 325912693097835691032656415976083290374004093960074015639005074
90470120140910710998168621769838987813974287251413037436, 10434454298460995770354472569253131399113136710394624211400963197548170116781304436915183
657352401426117756260466770060, 278618255393710079615600239564597606626236633264748835292571161430739777448435426668056873870400478697913511747801
40396, 296356563113994804043143335774980194284511836225493637539393416821245061561041110186302831241989481782733286008749349335, 1540986024265208523
71378350447375960534453600350036737408641012653665351414220535590771251968437780309921657855689981403793, 224987905686552773489583953833262631633358021
360017804617459605344536003500367374086415505217540969380969632275593834835, 18248715493223448124822777896652005022668316741674827713397198706931233
7786722994652689885164789494047397656481144350790]

-----
decoded message: "Ja Kocham Wybrane Zagadnienia Kryptograficzne! :~)" of length 50

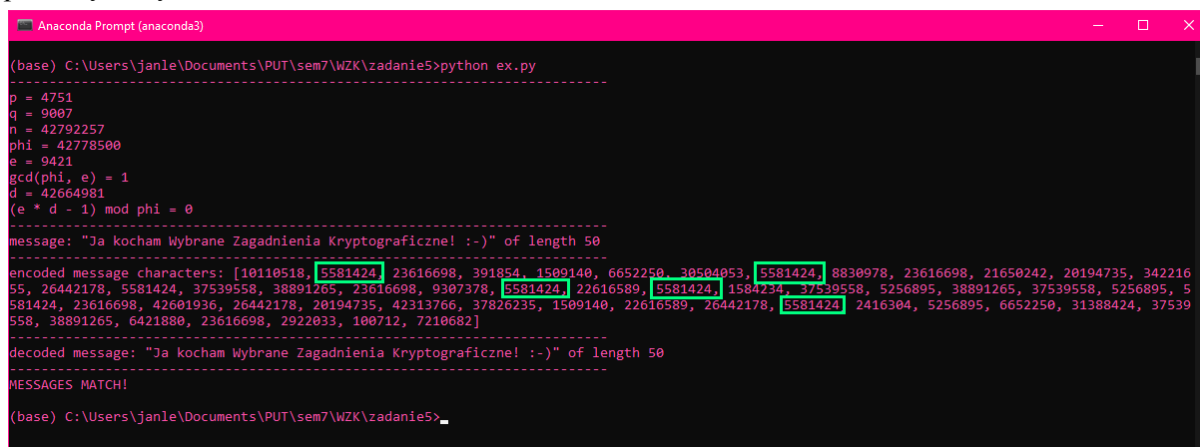
MESSAGES MATCH!

(base) C:\Users\janle\Documents\PUT\sem7\WZK\zadanie5>
```

5. Wnioski

Jeśli chodzi o odpowiedzi na pytanie 1., to są one zawarte już wyżej. Niemniej, w formie podsumowania, mogę napisać, że moim zdaniem najtrudniejsze etapy implementacji algorytmu to wyznaczenie d (o ile nie znamy krótkiej metody) oraz ustalenie optymalnego systemu szyfrowania wiadomości typu string.

Właśnie ze stringami wiąże się podstawowa wada mojego algorytmu - każdy znak jest szyfrowany przez tą samą wartość int:



```
Anaconda Prompt (anaconda3)
(base) C:\Users\janle\Documents\PUT\sem7\WZK\zadanie5>python ex.py
-----
p = 4751
q = 9007
n = 42792257
phi = 42778500
s = 9421
gcd(phi, e) = 1
d = 42664981
(e * d - 1) mod phi = 0
-----
message: "Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)" of length 50
-----
encoded message characters: [10110518, 5581424, 23616698, 391854, 1509140, 6652250, 30504053, 5581424, 8830978, 23616698, 21650242, 20194735, 342216
55, 26442178, 5581424, 37539558, 38891265, 23616698, 9307378, 5581424, 22616589, 5581424, 1584234, 37539558, 5256895, 38891265, 37539558, 5256895, 5
581424, 23616698, 42601936, 26442178, 20194735, 42313766, 37826235, 1509140, 22616589, 26442178, 5581424, 2416304, 5256895, 6652250, 31388424, 37539
558, 38891265, 6421880, 23616698, 2922033, 100712, 7210682]
-----
decoded message: "Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)" of length 50
-----
MESSAGES MATCH!
-----
(base) C:\Users\janle\Documents\PUT\sem7\WZK\zadanie5>
```

Na powyższym zrzucie ekranu każda z zaznaczonych wartości koduje jeden znak - małą literę "a". Jest to duża wada, gdyż przez to szyfr de facto redukuje się do zwykłego szyfru podstawieniowego. Najprostszym rozwiązaniem byłoby zwiększenie rozmiaru bloku, tak aby objąć więcej bajtów - znaków, albo nawet cały string kodować jako jednego inta. Wielkość takiej wiadomości byłaby ograniczona wielkościami modulo. Implementacja byłaby praktycznie identyczna, trzeba by jedynie pilnować wzajemnych rozmiarów oraz ewentualnie pilnować zer na początku strumienia bitów, w razie potrzeby dopisać je na początku rozkodowanej wiadomości by dopełnić ilość bitów do wielokrotności liczby 8. Inną możliwością jest użycie funkcji haszujących.

Notabene, string o długości 50 przekonwertowany na int, będący konkatencją bitów poszczególnych znaków ma następującą postać:

```
'Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)' =
7502585898292004480449492194025403635471319308941800755459830544853554220008422045
39562167034984446699029911044787678505
```

Jeśli chodzi o pytanie 2., to o bezpieczeństwie algorytmu decyduje przede wszystkim rozmiar liczb p i q . Wartość n jest dostępna w kluczu publicznym. Wiadomo, że n jest iloczynem 2 liczb pierwszych. Jeśli więc są one małe, to znalezienie tych czynników n jest trywialne. Jako, że e również jest publiczne, to znając p , q i e można łatwo obliczyć d , a wtedy zyskujemy dostęp do klucza prywatnego. Bezpieczeństwo szyfrowania opiera się zatem na trudności faktoryzacji dużych liczb złożonych. Cytując Wikipedię: "Dotychczas największym kluczem RSA, jaki rozłożono na czynniki pierwsze, jest klucz 768-bitowy. Liczby pierwsze zostały znalezione 12 grudnia 2009, a informacje o przeprowadzonej faktoryzacji opublikowano 7 stycznia 2010 roku. Wykorzystano do tego klaster komputerów; czas zużyty na obliczenia był o 2 rzędy wielkości krótszy od prognozowanego."

6. Kod programu:

```
import random
import sys
import numpy as np
from math import gcd
import bitarray
from bitarray.util import ba2int

class PublicKey:
    e = 0
    n = 0

class PrivateKey:
    d = 0
    n = 0

def encode(message, publicKey, display = True):
    messageBits = bitarray.bitarray()
    messageBits.frombytes(message.encode('utf-8'))
    if display:
        print('message as bits:', messageBits)

    print('-----')

    messageInt = ba2int(messageBits)
    if display:
        print('message as int:', messageInt)

    print('-----')

    encodedMessage = pow(messageInt, publicKey.e, publicKey.n)
    return encodedMessage

def decode(encodedMessage, privateKey, display = True):
    decodedMessageInt = pow(encodedMessage, privateKey.d, privateKey.n)
    if display:
        print('decoded message as int:', decodedMessageInt)

    print('-----')

    decodedMessage = chr(decodedMessageInt)
    if display:
        print('decoded message as char:', decodedMessage)
```

```

print('-----
-----')

    return decodedMessage

def main():

print('-----
-----')

    p = 440652540385366476885648349606092631044387368497923129947867 #60
    q = 848678487181295759680383727940935479949828086390833874248117 #60
    #p = 4751 #4
    #q = 9007 #4
    print('p =', p)
    print('q =', q)

    n = p * q
    print('n =', n)

    phi = (p - 1) * (q - 1)
    print('phi =', phi)

    e = 331217644374414450300103554646128062125802050357921772483749 #60
    #e = 9421 #4
    print('e =', e)
    print('gcd(phi, e) =', gcd(phi, e))

    ...
    (e * d - 1) mod phi = 0 <==> e * d - 1 = x * phi <==> d = (x * phi
+ 1) / e
    ...

    #x = 2
    #notFound = True

#print('-----
-----')

    #pierwsza metoda do wyznaczania d (brute force):
    #print('searching for x...')
    ...

    while notFound:
        if x % 1000 == 0:
            pass

```



```

        #print(x)
        numerator = x * phi + 1
        if numerator % e == 0:
            d = numerator // e
            notFound = False
        else:
            x += 1
    ...

#druga metoda, korzystająca z odwrotności modulo:
d = pow(e, -1, phi)

#print('x =', x)
#print('d = (x * phi + 1) / e')
#print('d = (', x, ' * ', phi, ' + 1) / ', e, sep='')
print('d =', d)
print('(e * d - 1) mod phi =', (e * d - 1) % phi)

print('-----')
print('-----')

publicKey = PublicKey()
publicKey.e = e
publicKey.n = n

privateKey = PrivateKey()
privateKey.d = d
privateKey.n = n

message = 'Ja kocham Wybrane Zagadnienia Kryptograficzne! :-)'
print('message: "', message, '" of length ', len(message), sep='')

print('-----')
print('-----')

encodedMessage = [encode(character, publicKey, False) for character
in message]
print('encoded message characters:', encodedMessage)

print('-----')
print('-----')

```

```
decodedMessage = ''.join([decode(character, privateKey, False) for
character in encodedMessage])
    print('decoded message: "', decodedMessage, '" of length ',
len(decodedMessage), sep='')

print('-----
-----')

    if decodedMessage == message:
        print('MESSAGES MATCH!')
    else:
        print('MESSAGES DON`T MATCH!')

if __name__ == '__main__':
    main()
```