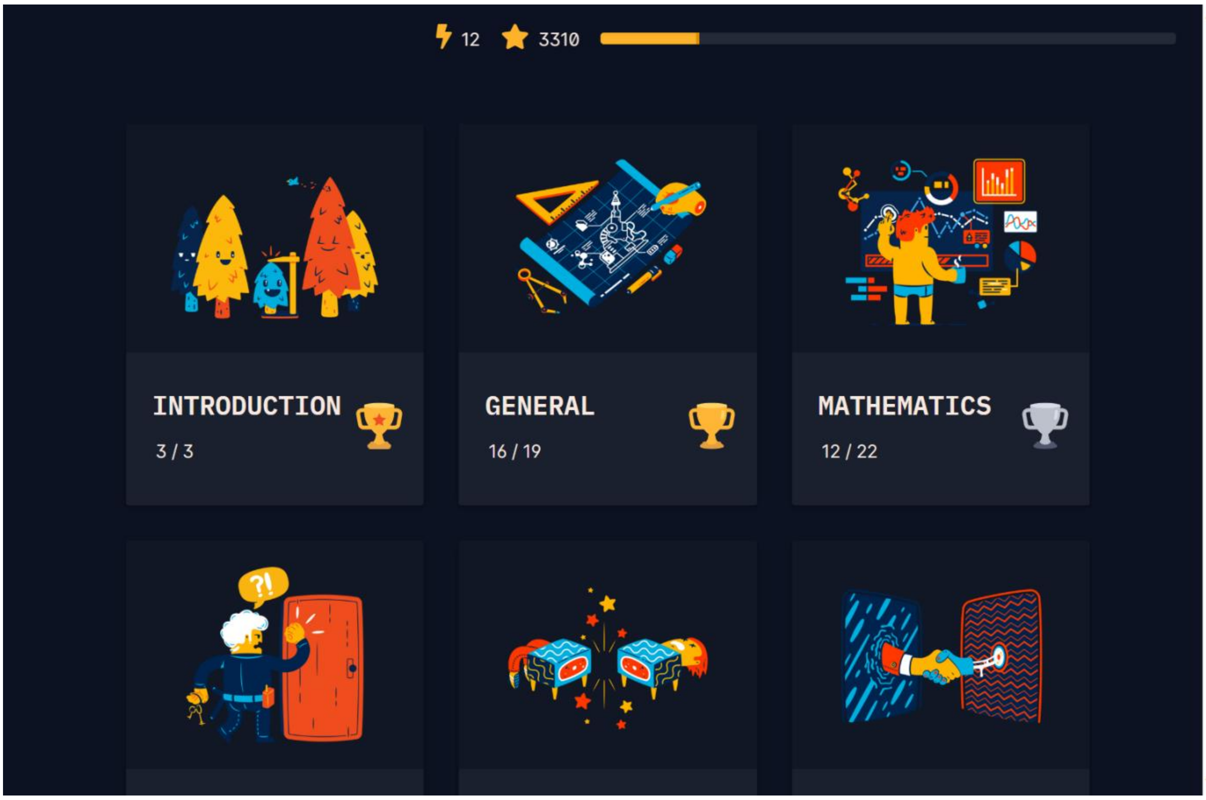
CRYPTOHACK LABS

Score screenshot:

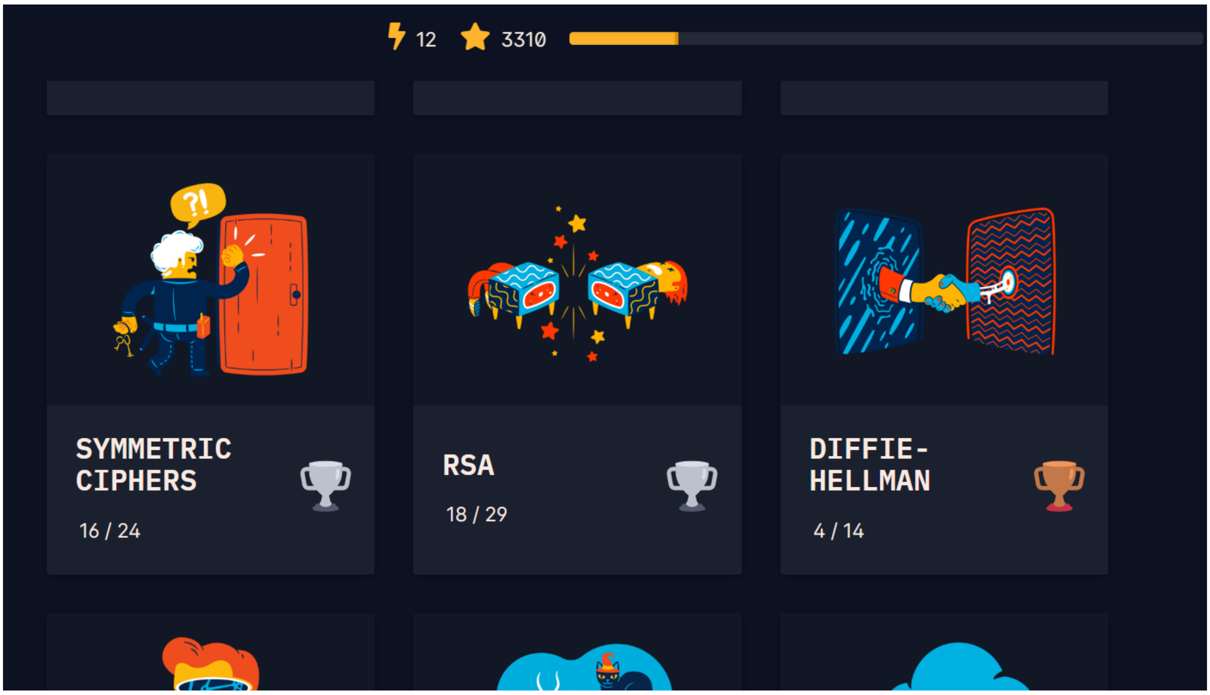


12 3310

INTRODUCTION
3 / 3

GENERAL
16 / 19

MATHEMATICS
12 / 22

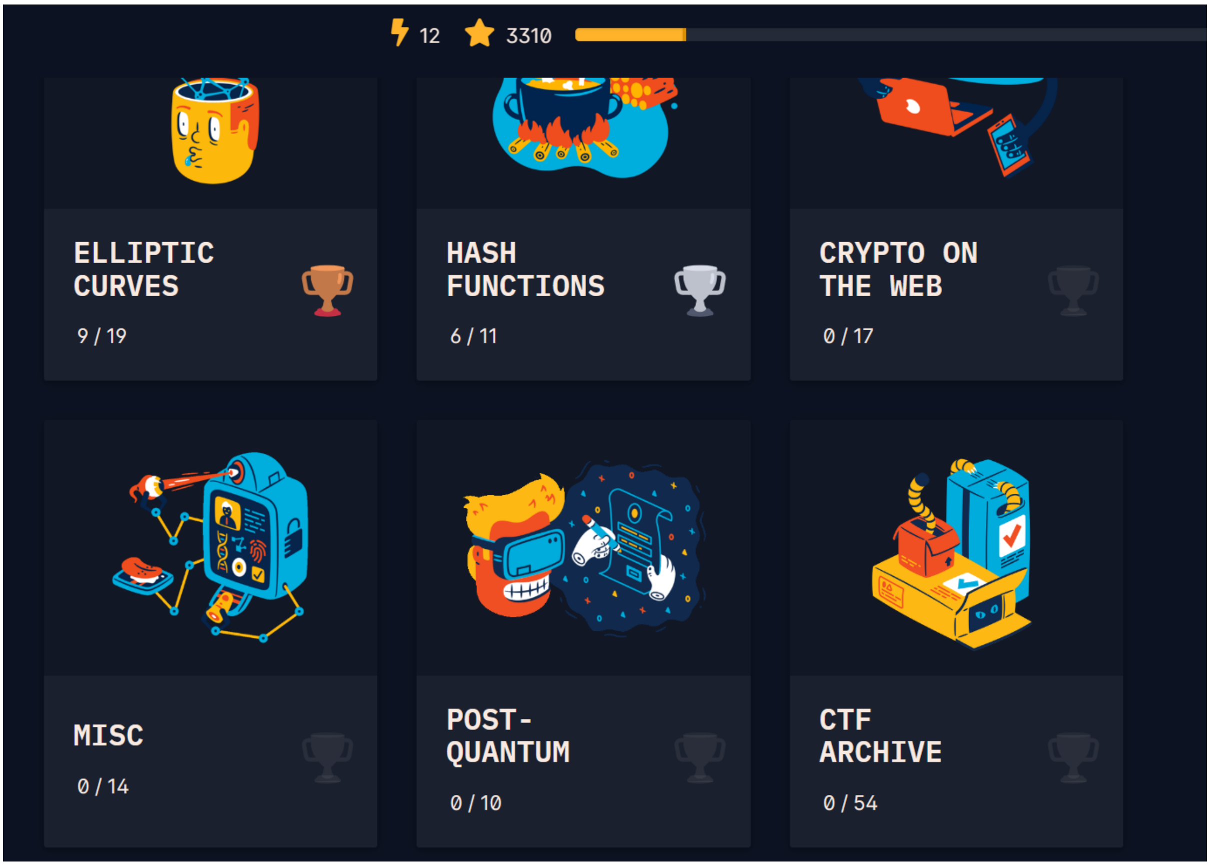


12 3310

SYMMETRIC CIPHERS
16 / 24

RSA
18 / 29

DIFFIE-HELLMAN
4 / 14


12 3310

ELLIPTIC CURVES
9 / 19

HASH FUNCTIONS
6 / 11

CRYPTO ON THE WEB
0 / 17

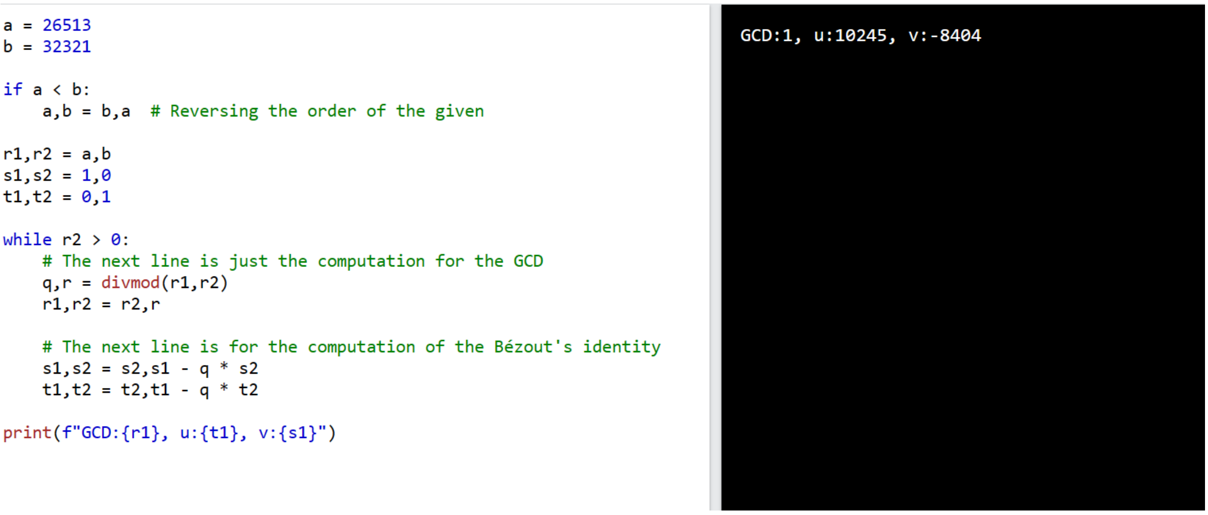
MISC
0 / 14

POST-QUANTUM
0 / 10

CTF ARCHIVE
0 / 54

Solutions for challenges:

Extended GCD


```python
a = 26513
b = 32321

if a < b:
    a,b = b,a  # Reversing the order of the given

r1,r2 = a,b
s1,s2 = 1,0
t1,t2 = 0,1

while r2 > 0:
    # The next line is just the computation for the GCD
    q,r = divmod(r1,r2)
    r1,r2 = r2,r

    # The next line is for the computation of the Bézout's identity
    s1,s2 = s2,s1 - q * s2
    t1,t2 = t2,t1 - q * t2

print(f"GCD:{r1}, u:{t1}, v:{s1}")
```

```
GCD:1, u:10245, v:-8404
```

Encoding challenge

```python
from pwn import * # pip install pwntools
import json
from Crypto.Util.number import bytes_to_long, long_to_bytes
import base64
import codecs
import array
```

```
r = remote('socket.cryptohack.org', 13377, level = 'debug')

def json_recv():
        line = r.recvline()
        return json.loads(line.decode())

def json_send(hsh):
        request = json.dumps(hsh).encode()
        r.sendline(request)

for i in range(0,101):
        received = json_recv()

        if "flag" in received:
                print(received)
                break

        print("\n\n")
        print("Received type: ")
        print(received["type"])
        print("Received encoded value: ")
        print(received["encoded"])

        encoding = received["type"]
        word = received["encoded"]

        if encoding == "base64":#PASSED
                decoded = base64.b64decode(word).decode('utf-8')
        elif encoding == "hex": #PASSED
                decode_hex = codecs.getdecoder("hex_codec")
                decoded = decode_hex(word)[0].decode('utf-8')
        elif encoding == "rot13":#PASSED
                decoded = codecs.encode(word, 'rot_13')
        elif encoding == "bigint":
                # Spent way too long troubleshooting this
                # Its a string so to make it work you have
                # to convert it.
                decoded = long_to_bytes(int(word,16)).decode('utf-8')
        elif encoding == "utf-8": #PASSED
                decoded = array.array('b', word).tobytes().decode('utf-8')

        print("DECODED: "+decoded)

        to_send = {
                "decoded": decoded
        }
        json_send(to_send)
```

diffy hellman starter-2

p = 28151

```
def is_primitive_element(g):
    # Set of powers generated by g
    powers = set()

    # Calculate powers of g modulo p
    for i in range(1, p):
        power = pow(g, i, p)
        if power in powers:
            # If a power is repeated, g is not a primitive element
            return False
        powers.add(power)

    # If all elements in Fp are generated by g, it is a primitive element
    return len(powers) == p - 1

# Iterate over elements of Fp
for g in range(1, p):
    if is_primitive_element(g):
        # Found the smallest primitive element
        smallest_primitive_element = g
        break

# Print the smallest primitive element (the flag)
print("Smallest primitive element of Fp:", smallest_primitive_element)
```

output-7

diffey hellamn starter-1

```
p = 991  # Prime modulus
g = 209  # Element in the finite field Fp

# Calculate the modular multiplicative inverse of g modulo p
d = pow(g, -1, p)
```

```
print(d)

output-569
```

diffey hellman starter-3

```
g = 2

p =
2410312426921032588552076022197566074856950548502459942654116941958108831682612228890093858261341614673227141477904012196503648957050582631942730706805009223062734745341073406696246014589361659774041027169249453200378729434170325843778659198143763193776859869524088940195577346119843545301547043747207749969763750084308926339295559968882457872412993810129130294592999947926365264059284647209730384947211681434464714438488520940127459844288859336526896320919633919

a =
9721074438370337962458643162004582468469045984889816058567658904788530882468973454873284910377102192220389309433658486261941098303091793930182167633275721201247601400180386739998376433775904344138666111324039795471506590538973555933944925869784000443754656572960275929483495892164153637226683613286895889965413700975590903351376764115959493358573417971489261516942995759702928098053144314470434694474859576699499890902023202343378903232934018623049865998847328 15
```

```
# Calculate g^a mod p to obtain the shared secret

shared_secret = pow(g, a, p)


# Print the shared secret

print(shared_secret)

output-
1806857697840726523322586721820911358489420128129248078673933653533930681676181753849411715714173604352323556558783759252661061186320274214883104886050164368129191719707402291577330485499513522368289395359523901406138025022522412429238971591272160519144672389532393673832265070057319485399793101182682177465364396277424717543434017666343807276970864475830391776403957550678362368319776566025118492062196941451265638054400177248572271342548616103967411990437357924
```

diffey hellman starter-4

```
A =
70249943217595468278554541264975482909289174351516133994495821400710625291840101960595720462672604202133493023241393916394629829526272643847352371534839862030 41
```

```
0331485087487331809285533195024369287293217083414424096866925845838641840923193480821332056735592483730921055532222505605661664236182285229504265881752580410194
731633895345823963910901731715743835775619780738974844840425579683385344491015955892106904647602049559477279345982530488299847663103078045601
```

```
b =
120192332529039903445985225357749630203957704094452967240343784334979768401678059705899609622219482909518733877281021159968314544822992432268394909997137634404121779658615087734205322664846191267105664149142275601037153366961932103798505750477303883783482661809349461391004798313398358965834436915293727039545890715077179171369067701220777398142622984886621380856087361034186017508616984173402642138677538346793591914270981958871120645031045104896104482944207 20
```

```
p =
24103124269210325885520760221975660748569505485024599426541169419581088316826122288900938582613416146732271414779040121965036489570505826319427307068050092230627347453410734066962460145893616597740410271692494532003787294341703258437786591981437631937768598695240889401955773461198435453015470437472077499697637500843089263392955599688824578724129938101291302945929999479263652640592846472097303849472116814344647144384885209401274598442888593365268963209196339 19
```

```
shared_secret = pow(A, b, p)


print(shared_secret)
```

output-

```
117413074041382065653383274603484198587730208631638838016598443667230769244371131028501413854520436949547872510288267342789210453912095239378896105199290164969406317985359831147382034121587996534313635143641052285071740844580204300316465834800657740855869350222028570089340467459256762629757122202790263115707214333004311841846709423796559119844080397072660453780714670376357160686144835460750265466470039045379449317679467891735263402971332061586594072083790946 6
```

Gussian reduction

```
import math


def gaussian_lattice_reduction(v1, v2):

    while True:

        # Step (a): Swap vectors if ||v2|| < ||v1||

        if math.sqrt(v2[0]**2 + v2[1]**2) < math.sqrt(v1[0]**2 + v1[1]**2):

            v1, v2 = v2, v1
    # Step (b): Compute m = ⌊ v1·v2 / v1·v1 ⌉
    m = math.floor((v1[0]*v2[0] + v1[1]*v2[1]) / (v1[0]**2 + v1[1]**2))

    # Step (c): If m = 0, return v1, v2
    if m == 0:
        return v1, v2

    # Step (d): v2 = v2 - m*v1
    v2 = (v2[0] - m*v1[0], v2[1] - m*v1[1])

# Define the initial vectors
v = (846835985, 9834798552)
u = (87502093, 123094980)

# Apply Gaussian lattice reduction
v1, v2 = gaussian_lattice_reduction(v, u)

# Calculate the inner product of the new basis vectors
inner_product = v1[0]*v2[0] + v1[1]*v2[1]

# Print the inner product (the flag)
print("Inner product of the new basis vectors:", inner_product)
```

```
output- 7410790865146821
```

Size and basis

```
import math

# Define the vector
v = (4, 6, 2, 5)

# Calculate the size (norm) of the vector
```

```
size = math.sqrt(sum(component ** 2 for component in v))

# Print the size of the vector
print("The size of the vector is:", size)
```

output- 9

vectors

```
# Define the vectors v, w, and u
v = (2, 6, 3)
w = (1, 0, 0)
u = (7, 7, 2)

# Calculate the expression 3*(2*v - w) · 2*u

# Step 1: Calculate the vector 2*v - w
vector_1 = (2 * v[0] - w[0], 2 * v[1] - w[1], 2 * v[2] - w[2])

# Step 2: Multiply each component of vector_1 by 3
vector_2 = (3 * vector_1[0], 3 * vector_1[1], 3 * vector_1[2])

# Step 3: Multiply each component of vector_2 by 2*u and calculate the dot product
result = vector_2[0] * 2 * u[0] + vector_2[1] * 2 * u[1] + vector_2[2] * 2 * u[2]

# Print the result
print("The result of the expression is:", result)
```

output- 702

quadratic residue

```
p = 29
ints = [14, 6, 11]
```

```
def find_quadratic_residue(p, ints):
    quadratic_residue = None
    for a in range(1, p):
        if (a**2) % p in ints:
            quadratic_residue = (a**2) % p
            break
    return quadratic_residue


def calculate_square_root(p, quadratic_residue):
    a = 1
    while (a**2) % p != quadratic_residue:
        a += 1
    return a


quadratic_residue = find_quadratic_residue(p, ints)
square_root = calculate_square_root(p, quadratic_residue)


print(square_root)
```

output-8

legendry symbol

```
import math


# The prime number (p) and the list of integers (ints)
p =
101524035174539890485408575671085261788758965189060164484385690801466167356667036677932998889725476582421738788500738738503134356158197247473850273565349249573867251280253564698939768700489401960767007716413932851838937641880157263936985954881657889497583485535527613578457628399173971810541670838543309159139
ints =
[25081841204695904475894082974192007718642931811040324543182130088804239047149283334700530600468528298920930150221871666297194395061462592781551275161695411167049544771049769000895119729307495913024360169904315078028798025169985966732789207320203861858234048872508633514498384390497048416012928086480326832803,
4547176518033043906050464748062144963490419283938389721280980833961984163382653 4
```

```
856109999027962620381874878086991125854247108359699799913776917227058286090426484
545834938813893550429960920037789905271666335118866409630267271207850860131172586
36782238741578611631963403910086344193485739758415783593559315905 55,
```

```
85610999902796262038187487808699112585424710835969979991377691722705828609042648
```

```
        p (int): The prime number 'p' modulo which the Legendre symbol is calculated.

    Returns:
        int: The Legendre symbol (a/p). It can be 1, -1, or 0.

    """
    legendre_symbol = pow(a, (p-1)//2, p)
    return legendre_symbol


def find_quadratic_residue(p, ints):
    """
    Find the quadratic residue among the given integers modulo prime 'p'.

    Args:
        p (int): The prime number 'p' modulo which the quadratic residue is calculated.
        ints (list): List of integers among which to find the quadratic residue.

    Returns:
        int: The quadratic residue if found, otherwise None.

    """
    quadratic_residue = None
    for a in ints:
        legendre_symbol = calculate_legendre_symbol(a, p)
        if legendre_symbol == 1:
            quadratic_residue = a
            break
    return quadratic_residue
```

```
def calculate_square_root(p, quadratic_residue):
    """
    Calculate the square root of a quadratic residue modulo prime 'p'.

    Args:
        p (int): The prime number 'p' modulo which the square root is calculated.
        quadratic_residue (int): The quadratic residue for which to calculate the square root.

    Returns:
        int: The square root of the quadratic residue modulo 'p'.

    """
    square_root = pow(quadratic_residue, (p+1)//4, p)
    return square_root



# Find the quadratic residue
quadratic_residue = find_quadratic_residue(p, ints)

if quadratic_residue is not None:
    # Calculate the square root
    square_root = calculate_square_root(p, quadratic_residue)
    print("The square root of the quadratic residue is:", square_root)
else:
    print("No quadratic residue found in the given integers.")
```

output-

```
93291799125366706806545638475797430512104976066103610269938025709952247020061090
80487018619528599872768020097985384871858912676574255085595480529025359214420955
21230621614585845750609394813682106886298620369588576047074683723842780497413691
53506182660264876115428251983455344219194133033177700490981696141526
```

diffusion through permutation

```
def shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]

# The inv_shift_rows function is the inverse operation of shift_rows.
# It reverses the shift performed in shift_rows, restoring the original state matrix.

def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]

# The mix_single_column function performs the MixColumns operation for a single column of the state matrix.
# It uses multiplication in the Rijndael's Galois field to ensure diffusion and non-linearity.


def mix_single_column(a):
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)

# The mix_columns function applies the mix_single_column operation to each column of the state matrix.

def inv_mix_columns(s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)



# The inv_mix_columns function performs the inverse operation of mix_columns.
# It reverses the mixing by applying inverse transformations to each column of the state matrix.

def inv_mix_columns(s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)




state = [
    [108, 106, 71, 86],
    [96, 62, 38, 72],
    [42, 184, 92, 209],
    [94, 79, 8, 54],
]


inv_mix_columns(state)
inv_shift_rows(state)


result = []
for row in state:
    result.extend(row)


flag = bytes(result)
print(flag)
```

Mode of operation starter:

```
import requests

# request encrypted flag
r =
requests.get('http://aes.cryptohack.org/block_cipher_starter/encrypt_fl
ag/')
res = r.json()['ciphertext']
# print(res)

# request plaintext/decrypting flag
endpointdec = 'http://aes.cryptohack.org/block_cipher_starter/decrypt/'
+ res
dec = requests.get(endpointdec)
res1 = dec.json()['plaintext']
# print(res1)

by = bytes.fromhex(res1)
finalres = by.decode()
print(finalres)
```

crypto{bl0ck_c1ph3r5_4r3_f457_!}

Bringing it altogether:

```
N_ROUNDS = 10

key        = b'\xc3,\\\xa6\xb5\x80^\x0c\xdb\x8d\xa5z*\xb6\xfe\\'
ciphertext = b'\xd1O\x14j\xa4+O\xb6\xa1\xc4\x08B)\x8f\x12\xdd'
```

```
s_box = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)

def bytes2matrix(text):
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    out = []
    for r in matrix:
        for c in r:
            out.append(c.to_bytes(2,byteorder='little').decode())
    return ''.join(out)

def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]

def inv_sub_bytes(s, sbox=inv_s_box):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (sbox[s[i][j]])


def add_round_key(s, k):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = (s[i][j] ^ k[i][j])

xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def mix_single_column(a):
    # see Sec 4.1.2 in The Design of Rijndael
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)


def mix_columns(s):
    for i in range(4):
        mix_single_column(s[i])


def inv_mix_columns(s):
    # see Sec 4.1.3 in The Design of Rijndael
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    mix_columns(s)

def expand_key(master_key):
    """
    Expands and returns a list of key matrices for the given master_key.
    """

    # Round constants https://en.wikipedia.org/wiki/AES_key_schedule#Round_constants
    r_con = (
        0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
        0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A,
        0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A,
        0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,
    )

    # Initialize round keys with raw key material.
    key_columns = bytes2matrix(master_key)
    iteration_size = len(master_key) // 4

    # Each iteration has exactly as many columns as the key material.
    i = 1
    while len(key_columns) < (N_ROUNDS + 1) * 4:
        # Copy previous word.
        word = list(key_columns[-1])

        # Perform schedule_core once every "row".
        if len(key_columns) % iteration_size == 0:
            # Circular shift.
            word.append(word.pop(0))
            # Map to S-BOX.
            word = [s_box[b] for b in word]
            # XOR with first byte of R-CON, since the others bytes of R-CON are 0.
            word[0] ^= r_con[i]
            i += 1
        elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:
            # Run word through S-box in the fourth iteration when using a
            # 256-bit key.
            word = [s_box[b] for b in word]

        # XOR with equivalent word from previous iteration.
        word = bytes(i^j for i, j in zip(word, key_columns[-iteration_size]))
        key_columns.append(word)

    # Group key words in 4x4 byte matrices.
    return [key_columns[4*i : 4*(i+1)] for i in range(len(key_columns) // 4)]


def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and work backwards through them when decrypting

    # Convert ciphertext to state matrix
    state = bytes2matrix(ciphertext)
    # Initial add round key step
    add_round_key(state,round_keys[-1])


    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(state)
        inv_sub_bytes(state, inv_s_box)
        add_round_key(state,round_keys[i])
        inv_mix_columns(state)

    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(state)
    inv_sub_bytes(state, inv_s_box)
    add_round_key(state,round_keys[0])

    # Convert state matrix to plaintext
    plaintext = matrix2bytes(state)

    return plaintext

print(decrypt(key, ciphertext))
```

flag = crypto{MYAES128}

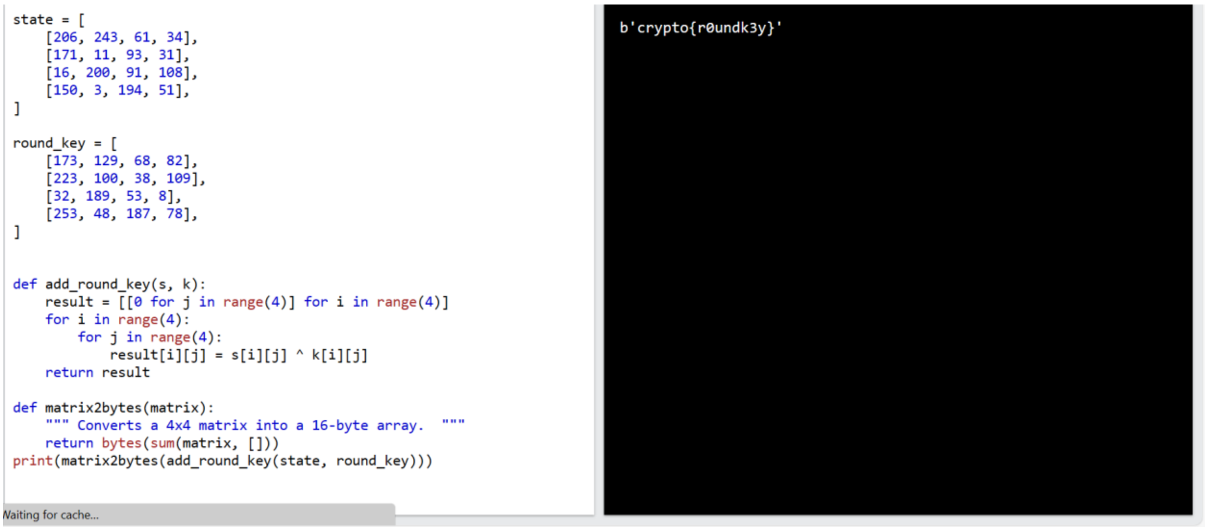
Structure of AES:

```
def bytes2matrix(text):
    """ Converts a 16-byte array into a 4x4 matrix.  """
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array.  """
    text = ''
    for i in range(len(matrix)):
        for j in range(4):
            text += chr(matrix[i][j])
    return text

matrix = [
    [99, 114, 121, 112],
    [116, 111, 123, 105],
    [110, 109, 97, 116],
    [114, 105, 120, 125],
]

print(matrix2bytes(matrix))
```

Solutions:

`crypto{inmatrix}`

round keys



```
state = [
    [206, 243, 61, 34],
    [171, 11, 93, 31],
    [16, 200, 91, 108],
    [150, 3, 194, 51],
]

round_key = [
    [173, 129, 68, 82],
    [223, 100, 38, 109],
    [32, 189, 53, 8],
    [253, 48, 187, 78],
]


def add_round_key(s, k):
    result = [[0 for j in range(4)] for i in range(4)]
    for i in range(4):
        for j in range(4):
            result[i][j] = s[i][j] ^ k[i][j]
    return result

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array.  """
    return bytes(sum(matrix, []))
print(matrix2bytes(add_round_key(state, round_key)))
```

```
b'crypto{r0undk3y}'
```

Confusion through substitution:

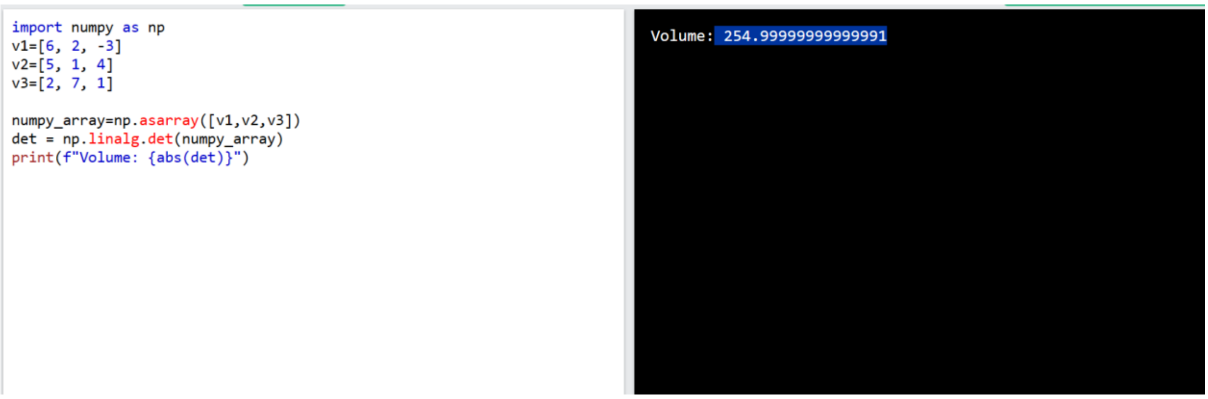
s_box = (

0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,

```
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)


inv_s_box = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)


state = [
    [251, 64, 182, 81],
    [146, 168, 33, 80],
    [199, 159, 195, 24],
    [64, 80, 182, 255],
]
def sub_bytes(s, sbox=s_box):
    #result = [[0 for j in range(4)] for i in range(4)]
    for i in range(4):
        for j in range(4):
            print(chr(sbox[s[i][j]]), end="")
print(sub_bytes(state, sbox=inv_s_box))
```

output: crypto{l1n34rly}

whats a lattice



```
import numpy as np
v1=[6, 2, -3]
v2=[5, 1, 4]
v3=[2, 7, 1]

numpy_array=np.asarray([v1,v2,v3])
det = np.linalg.det(numpy_array)
print(f"Volume: {abs(det)}")
```

```
Volume: 254.99999999999991
```

Flipping cookie:

Flag: crypto{4u7h3n71c4710n_15_3553n714l}

Lazy CBC:

```
def encrypt(plaintext):
    plaintext = bytes.fromhex(plaintext)
    if len(plaintext) % 16 != 0:
        return {"error": "Data length must be multiple of 16"}

    cipher = AES.new(KEY, AES.MODE_CBC, KEY)
    encrypted = cipher.encrypt(plaintext)

    return {"ciphertext": encrypted.hex()}


@chal.route('/lazy_cbc/get_flag/<key>/')
def get_flag(key):
    key = bytes.fromhex(key)

    if key == KEY:
        return {"plaintext": FLAG.encode().hex()}
    else:
        return {"error": "invalid key"}


@chal.route('/lazy_cbc/receive/<ciphertext>/')
def receive(ciphertext):
    ciphertext = bytes.fromhex(ciphertext)
    if len(ciphertext) % 16 != 0:
        return {"error": "Data length must be multiple of 16"}

    cipher = AES.new(KEY, AES.MODE_CBC, KEY)
    decrypted = cipher.decrypt(ciphertext)

    try:
        decrypted.decode() # ensure plaintext is valid ascii
    except UnicodeDecodeError:
        return {"error": "Invalid plaintext: " + decrypted.hex()}

    return {"success": "Your message has been received"}
```

Flag: `crypto{50m3_p30pl3_d0n7_7h1nk_IV_15_1mp0r74n7_?}`

Privacy enhanced mail:

```
Output: from Crypto.PublicKey import RSA

f = open('privacy_enhanced_mail_1f696c053d76a78c2c531bb013a92d4a.pem','r')
a = RSA.importKey(f.read())
print(a.d)

output:
15682700288056331364787171045819973654991149949197959929860861228180021707316851924
45620554366556581089267419005983133023143697091447477456271494562051914438978515890
89941819513488460174325064641635649609937842541533954067991013147600334450651934295
92512349952020982932218524462341002102063435489318813316464511621736943938440710470
69491233623768021974620459512895916180059521636623753829644733537581887195252002699
31021483288970835471842864932411915059536016688589411297909669092369411278513702024
21135897091086763569884760099112291072056970636380417349019579768748054760104838790
424708988260443926906673795975104689
```

from Crypto.Util.number import inverse, long_to_bytes n =
110581795715958566206600392161360212579669637391437097703685154237017

351570464767725324182051199901920318211290404777259728923614917211291
562555864753005179326101890427669819834642007924406862482343614488768
256951616086287044725034412802176312273081322195866046098595306261781
7882765709204678401720045308737677
e = 1 ct =
449812307182121836042747859257931454426554650252645540460282513111644
94127485

print(long_to_bytes(ct))
flag: crypto{saltstack_fell_for_this!}

Scalar multiplication

```
import math


O = 'Origin'


def inv_mod(x, p):

    return pow(x, p-2, p)


# Calculate S = P + Q

def ecc_points_add(P, Q, a, p):


    if P == O:

        return Q

    if Q == O:

        return P


    if P[0] == Q[0] and P[1] == -Q[1]:

        return O


    if P != Q:

        lam = (Q[1]-P[1])*inv_mod(Q[0]-P[0], p)

    else:
        lam = (3*pow(P[0],2)+a)*inv_mod(2*P[1], p)

    x3 = pow(lam, 2) - P[0] - Q[0]
    x3 %= p
    y3 = lam*(P[0]-x3)-P[1]
    return (int(x3), int(y3%p))

# Calculate Q = nP
def scalar_mul(P, n, a, p):
    R = O
    Q = P

    while n > 0:
        if n % 2 == 1:
            R = ecc_points_add(R, Q, a, p)
        Q = ecc_points_add(Q, Q, a, p)
        n = math.floor(n/2)
    return R

# E: Y2 = X3 + 497 X + 1768, p: 9739
a = 497
b = 1768
p = 9739

n = 1337
X = (5323, 5438)
S = scalar_mul(X, n, a, p)
print(S, S == (1089, 6931))
```

```
P = (2339, 2213)
n = 7863
S = scalar_mul(P, n, a, p)
print(S)
```

Output:(9467, 2742)

Stream of consciousness

```
def xor_all(ciphers, test_key):
    for cipher in ciphers:
        cipher = bytes.fromhex(cipher)
        for i in range(len(test_key)):
            if i >= len(cipher): break
            a = test_key[i] ^ cipher[i]
            if not (a > 31 and a < 127):
                return False
            print(chr(a), end='')
        print()
        print('cipher', bytes.hex(cipher))
    return True

prefix = b'crypto{'
key = []
encrypted_flag = b''
for c in ciphers:
    c = bytes.fromhex(c)
    k = []
    for i in range(len(prefix)):
        k.append(prefix[i] ^ c[i])
    if xor_all(ciphers, k):
        print('found', k, len(k))
        key[:] = k[:]
        encrypted_flag = c
        break

    if key: break
def guess_next(cipher, key, guess):
    cipher = bytes.fromhex(cipher)
    for i in range(len(key)):
        if i >= len(cipher): break
        a = key[i] ^ cipher[i]
        print(chr(a), end='')
    print()
    if i + 1 < len(cipher) and guess:
        key.append(ord(guess) ^ cipher[i+1])


def test_key(cipher, key):
    for i in range(len(key)):
        if i >= len(cipher): break
        b = key[i] ^ cipher[i]
        print(chr(b), end='')
    print()
```

Triple DES

```
    def encrypt(key, plain):
        url = "http://aes.cryptohack.org/triple_des/encrypt/"
        rsp = requests.get(url + key + '/' + plain + '/').json()
        if rsp.get("error", None):
            raise ValueError(rsp["error"])
        return rsp["ciphertext"]
    def encrypt_flag(key):
        url = "http://aes.cryptohack.org/triple_des/encrypt_flag/"
        rsp = requests.get(url + key + '/').json()
        if rsp.get("error", None):
            raise ValueError(rsp["error"])
        return rsp["ciphertext"]


    key = b'\x00'*8 + b'\xff'*8
    flag = encrypt_flag(key.hex())
    flag_sz = 34
    cipher = encrypt(key.hex(), flag)
    print_blk(cipher, 16)
    print(bytes.fromhex(cipher))
```

crypto{n0t_4ll_k3ys_4r3_g00d_k3ys}

Point addition

```
import math
O = 'Origin'


def inv_mod(x, p):
    return pow(x, p-2, p)


def ecc_points_add(P, Q, a, p):


    if P == O:
        return Q
    if Q == O:
        return P

    if P[0] == Q[0] and P[1] == -Q[1]:
        return O

    if P != Q:
        #lam = (Q[1]-P[1])/(Q[0]-P[0])
        lam = (Q[1]-P[1])*inv_mod(Q[0]-P[0], p)
    else:
        #lam = (3*pow(P[0],2)+a)/(2*P[1])
        lam = (3*pow(P[0],2)+a)*inv_mod(2*P[1], p)

    x3 = pow(lam, 2) - P[0] - Q[0]
    x3 %= p
    y3 = lam*(P[0]-x3)-P[1]
    return (int(x3), int(y3%p))


if __name__ == '__main__':
    P = (493, 5564)
    Q = (1539, 4742)
    R = (4403, 5202)

    # E: Y2 = X3 + 497 X + 1768, p: 9739
    a = 497
    b = 1768
    p = 9739

    # test
```

```
X = (5274, 2841)
Y = (8669, 740)
S = ecc_points_add(X, X, a, p)
print(S, S == (7284, 2107))
S = ecc_points_add(X, Y, a, p)
print(S, S == (1024, 4440))

# S(x,y) = P + P + Q + R
S = ecc_points_add(P, P, a, p)
print('P+P', S)
S = ecc_points_add(S, Q, a, p)
print('S+Q', S)
S = ecc_points_add(S, R, a, p)
print('S+R', S)
print(S == (4215, 2162))
```

Flag:Crypto{4215, 2162}

Ron was wrong

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Util import number
import gmpy
from itertools import combinations

grps = {'n':[],'c':[],'e':[]}
for i in range(1, 51):
    key = RSA.importKey(open(f"keys_and_messages/{i}.pem", 'r').read())
    cipher = open(f"keys_and_messages/{i}.ciphertext", 'r').read()
    cipher = number.bytes_to_long(bytes.fromhex(cipher))
    grps['n'].append(key.n)
    grps['c'].append(cipher)
    grps['e'].append(key.e)

N = 0
for i in range(len(grps['n'])):
    for j in range(i+1, len(grps['n'])):
        if i == j: continue
        gcd = gmpy.gcd(grps['n'][i], grps['n'][j])
        if gcd != 1:
            print(i, j, gcd)
            N = int(gcd)
            ind = i

e = grps['e'][ind]
p = N
q = grps['n'][ind]//N
phi = (p-1)*(q-1)
d = number.inverse(e, phi)

key = RSA.construct((grps['n'][ind], e, d))
cipher = PKCS1_OAEP.new(key)
flag = number.long_to_bytes(grps['c'][ind])
flag = cipher.decrypt(flag)
print(flag)
```

Flag:crypto{3ucl1d_w0uld_b3_pr0ud}

Jacks birthday hash

n=11

```
lamb=0.75
from math import log,sqrt,ceil
 t=2**((n+1)/2)*sqrt(log(1/(1-lamb)))
 n=2**11
p=1
i=0
while p>0.5:
    i=i+1
    p=(((n-1)/n)**i)
#print(p,i) p is basically the probability of i people to have different birthdat=y then our target

print("We would need {0} different hashes to have 1 collision with 75% and we would need {1} hashes to collide with 1 specific hash".format(ceil(t),i))
```

Flag:1420

Smooth criminal

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Util.number import *
import hashlib

a = 497
b = 1768
p = 9739
G = (1804,5368)

def add_point(p1, p2):
    if p1 == (0, 0):
        return p2
    if p2 == (0,0):
        return p1

    x1, y1 = p1
    x2, y2 = p2

    if x1 == x2 and y1 == -y2:
        return (0, 0)

    lamda = 0
    if p1 == p2:
        lamda = ((3*pow(x1,2,p)+a) * inverse(2*y1, p))
    else:
        lamda = ((y2-y1) * inverse(x2-x1, p))

    x3 = (pow(lamda, 2) - x1 - x2) % p
    y3 = (lamda*(x1 - x3) - y1) % p
    return (x3, y3)

def Scalar_Mul(P, n):
    Q = P
    R = (0, 0)
    while n > 0:
    #If n ≡ 1 mod 2, set R = R + Q.
        if n % 2 == 1:
            R = add_point(R, Q)
        #Set Q = 2 Q and n = ⌊n/2⌋.
        Q = add_point(Q, Q)
        n = n//2
    return R
```

```
def is_pkcs7_padded(message):
    padding = message[-message[-1]:]
    return all(padding[i] == len(padding) for i in range(0, len(padding)))

def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    # Decrypt flag
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)

    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('ascii')
    else:
        return plaintext.decode('ascii')

#E: Y^2 = X^3 + 497 X + 1768, p: 9739, G: (1804,5368)
q_x = 4726
nB = 6534
y_2 = (pow(q_x,3) + 497*q_x + 1768) % p
q_y = pow(y_2, (p+1)//4, p)
Q = (q_x, q_y)

shared_secret = Scalar_Mul(Q, nB)[0]
```

```
iv = 'cd9da9f1c60925922377ea952afc212c'
ciphertext = 'febcbe3a3414a730b125931dccf912d2239f3e969c4334d95ed0ec86f6449ad8'

print(decrypt_flag(shared_secret, iv, ciphertext))
```

Flag:crypto{n07_4ll_curv3s_4r3_s4f3_curv3s}

Exceptional curve:

```
from Crypto.Cipher import AES
from Crypto.Util.number import inverse
from Crypto.Util.Padding import pad, unpad
from collections import namedtuple
import hashlib
import os

# Create a simple Point class to represent the affine points.
Point = namedtuple("Point", "x y")

# The point at infinity (origin for the group law).
O = 'Origin'

def check_point(P: tuple):
    if P == O:
        return True
    else:
        return (P.y*2 - (P.x*3 + a*P.x + b)) % p == 0 and 0 <= P.x < p and 0 <= P.y < p

def point_inverse(P: tuple):
    if P == O:
        return P
    return Point(P.x, -P.y % p)


def point_addition(P: tuple, Q: tuple):
    # based of algo. in ICM
    if P == O:
        return Q
    elif Q == O:
        return P
    elif Q == point_inverse(P):
        return O
    else:
        if P == Q:
            lam = (3*P.x**2 + a)*inverse(2*P.y, p)
            lam %= p
        else:
            lam = (Q.y - P.y) * inverse((Q.x - P.x), p)
            lam %= p
    Rx = (lam**2 - P.x - Q.x) % p
    Ry = (lam*(P.x - Rx) - P.y) % p
    R = Point(Rx, Ry)
    assert check_point(R)
    return R


def double_and_add(P: tuple, n: int):
    # based of algo. in ICM
    Q = P
    R = O
    while n > 0:
        if n % 2 == 1:
            R = point_addition(R, Q)
        Q = point_addition(Q, Q)
        n = n // 2
    assert check_point(R)
    return R


def gen_shared_secret(Q: tuple, n: int):
    # Bob's Public key, my secret int
    S = double_and_add(Q, n)
    return S.x


def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    # Decrypt flag
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)

    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('ascii')
    else:
        return plaintext.decode('ascii')


# Define the curve
#E: Y^2 = X^3 + 2X + 3
```

```
    p = 310717010502520989590157367261876774703

    a = 2

    b = 3


    # Generator

    g_x = 179210853392303317793440285562762725654

    g_y = 105268671499942631758568591033409611165

    G = Point(g_x, g_y)


    # My secret int, different every time!!

    n = 47836431801801373761601790722388100620


    # Send this to Bob!

    # public = n*g

    public = double_and_add(G, n)


    # Bob's public key

    b_x = 272640099140026426377756188075937988094

    b_y = 51062462309521034358726608268084433317

    B = Point(b_x, b_y)


    # Calculate Shared Secret

    shared_secret = gen_shared_secret(B, n)


    iv = '07e2628b590095a5e332d397b8a59aa7'

    enc_flag =
'8220b7c47b36777a737f5ef9caa2814cf20c1c1ef496ec21a9b4833da24a008d0870d3ac3a6a
d80065c138a2ed6136af'


    flag = decrypt_flag(shared_secret, iv, enc_flag)
```

```
print(flag)
```

Flag:crypto{H3ns3l_lift3d_my_fl4g!}

Micro transmission

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import hashlib
from sage.all import *


def is_pkcs7_padded(message):
    padding = message[-message[-1]:]
    return all(padding[i] == len(padding) for i in range(0, len(padding)))


def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    # Decrypt flag
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)

    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('ascii')
    else:
        return plaintext.decode('ascii')
```

```
p = 99061670249353652702595159229088680425828208953931838069069584252923270946291
```

```
p = 99061670249353652702595159229088680425828208953931838069069584252923270946291
a = 1
b = 4
E = EllipticCurve(GF(p), [a,b])
G = E(43190960452218023575787899214023014938926631792651638044680168600989609069200, 20971936269255296908588589778128791635639992476076894152303569022736123671173)
P_A = E.lift_x(ZZ(87360200456784002948566700858113190957688355783112995047798140117594305287669))
P_B = E.lift_x(ZZ(6082896373499126624029343293750138460137531774473450341235217699497602895121))


primes = [p for p, _ in E.order().factor()][:-2]
dlogs = []
for fac in primes:
    t = int(G.order()) // int(fac)
    dlog = (t*G).discrete_log(t*P_A)
    dlogs += [dlog]
nA = crt(dlogs, primes)
shared_secret = (nA*P_B).xy()[0]


iv = "ceb34a8c174d77136455971f08641cc5"
ciphertext = "b503bf04df71cfbd3f464aec2083e9b79c825803a4d4a43697889ad29eb75453"
print(decrypt_flag(shared_secret, iv, ciphertext))
```

Flag:crypto{d0nt_l3t_n_b3_t00_sm4ll}

Inferious prime:

Flag:crypto{N33d_b1g_pR1m35}

Gram schmidt:

```
from Crypto.Util.number import getPrime, inverse, bytes_to_long, long_to_bytes, GCD


e = 3

n = 742449129124467073921545687640895127535705902454369756401331

ct = 39207274348578481322317340648475596807303160111338236677373  ## this is the cipher text


## we have N given .. now we can factorise it to get p and q and from there we get phi = (p-1)*(q-1) ...

## and then we can use d = inverse(e,phi) to get d

## then a = pow(c,d,n)

## then m = long_to_bytes(a)  and print(a)


## from factor.db we got :

p = 752708788837165590355094155871

q = 986369682585281993933185289261


phi = (p-1)*(q-1)


d = inverse(e,phi)


a = pow(ct,d,n)

m = long_to_bytes(a)

print(m)


from math import sqrt

# Hàm nhân vector

def dot_product(v1, v2):
    return sum(a*b for a, b in zip(v1, v2)) # tương đương với a in v1 * b in v2


# Hàm tính norm ~ 2
def vector_norm(v):
    return sqrt(dot_product(v, v))


vectors = [[4, 1, 3, -1], [2, 1, -3, 4], [1, 0, -2, 7], [6, 2, 9, -5]]
# Gram smith
def gram_smith(vectors):
    u = []
    for i in range(len(vectors)):
        ui = vectors[i]
        for j in range(i):
            muj = dot_product(vectors[i], u[j]) / vector_norm(u[j])**2
            ui = [ui[k] - muj * u[j][k] for k in range(len(ui))]
        u.append(ui)
    return u


flag = round(gram_smith(vectors)[3][1],5)
print(flag)
```

ANSWER:0.91611

MODULAR INVERTING

PRINT(POW(3, -1, 13))
OUTPUT:9

Unencryptable:
N =
0x7fe8cafec59886e9318830f33747cafd200588406e7c42741859e15994ab6241
0438991ab5d9fc94f386219e3c27d6ffc73754f791e7b2c565611f8fe5054dd132
b8c4f3eadcf1180cd8f2a3cc756b06996f2d5b67c390adcba9d444697b13d12b2b
adfc3c7d5459df16a047ca25f4d18570cd6fa727aed46394576cfdb56b41
e = 0x10001

```
c =
0x5233da71cc1dc1c5f21039f51eb51c80657e1af217d563aa25a8104a4e84a423
79040ecdfdd5afa191156ccb40b6f188f4ad96c58922428c4c0bc17fd538445685
3e139afde40c3f95988879629297f48d0efa6b335716a4c24bfee36f714d34a4e8
10a9689e93a0af8502528844ae578100b0188a2790518c695c095c9d677b

p =
823983539720851611172036284794942540104567236582993760211748044
931669455822662220011005753587380213296354891420146838354567626
2090246827792522994758916609
q =
109008243533344718300073075299373579261603864619678844461603152
186306877933414710791707505485547079266115420198592966051885354
13447791710067186432371970369
d = pow(e, -1, (p-1)*(q-1))
print(bytes.fromhex(hex(pow(c, d, N))[2:]).decode())
crypto{R3m3mb3r!_F1x3d_P0iNts_aR3_s3crE7s_t00}
```

Marins secret:

```
n =
65841627483018454412502751992144351578988826415607473309924404012621368249
77140327981163992881765024628292557845259777229030187144343096981082083886
64768262754316426220651576623731617882923164117579624827261244506084274371
25027784935163167944117101841801849803999647254989315057718930287152031171
51797307143121814562450978484916697959972898306129880585239683848088228283
70900198489249243399165125219244753790779764466236965135793576516193213175
06140166738862222836204271705401467903295344103402150685601708106261757235
11954185058993887157097959920295590421197834235973247071006940646759092387
17573058764118893225111602703838080618565401139902143069901117174204252871
94884686443677180861643245710284453484385719873524200530907393905143379094
67266722346432593495351862685716290779375978388013379730922856087442099515
33199868228040004432132597073390363357892379997655878857696334892216345070
22764674985138120855404494044418286402651370944982348959343901736635886964
81682387350875938083444843651362842197252338116053318150074245828908218872
60682886632543613109252862114326372077785369292570900594814481097443781269
56264730367142889576422408440225960510960036309895009199889137581283952361
32956672538139784348791727812172856528954691941812183430787545016947465987
38215243769747956572555989594598180639098344891175879455994652382137038240
166358066403475457
e = 65537
c =
40028046308893043231928035911519497758251736361053246429521066953040787075
34391274554013845697054256214459439929633809830849173854286312230469088378
04126399345875252917090184158440305503817193246288672986488987883177380307
```

```
37702507926603026265093257520514185341330255846036424235553127296748140941
47836345587911758278165407675459445342381890790301928432885969349796935179
64655661507346729751987928147021620165009965051933278913952899114253301044
74758731083041919062328257893158958750455500536157157256191686606345881296
53144741604990675250674951401500921196209283630074673909201307175211691051
67963364154636472055084012592138570354390246779276003156184676298710746583
10470051646609103451076502716795611786905193811645737038473744096510961957
82274220498065660605718310176108770724842627247895710765295864274057801210
96546942812322324807145137017942266863534989082115189065560011841150908380
93735430124315320642889632057660990436193703526398534898479420819889261589
89070059554035294708471242695123161917539502037945786560293245066882934465
71598506042198219080325747328636232040936761788558421528960279832802127562
11585230494686762831650295956227448548386748173114933820900975322946392485
59301032711978313709824887034564633859148012468286622126220069473801155495
29820197355738525329885232170215757585685484402344437894981555179129287164
971002033759724456
p = 2**2203-1
q = 2**2281-1
print(bytes.fromhex(hex(pow(c, pow(e, -1, (p-1)*(q-1)), n))[2:]).decode())
```

crypto{Th3se_Pr1m3s_4r3_t00_r4r3}

Everything is still big:

```
n =
0x665166804cd78e8197073f65f58bca14e019982245fcc7cad74535e948a4e0258b2e919bf3
720968a00e5240c5e1d6b8831d8fec300d969fccec6cce11dde826d3fbe0837194f2dc64194c7
8379440671563c6c75267f0286d779e6d91d3e9037c642a860a894d8c45b7ed564d341501ce
df260d3019234f2964ccc6c56b6de8a4f66667e9672a03f6c29d95100cdf5cb363d66f2131823
a953621680300ab3a2eb51c12999b6d4249dde499055584925399f3a8c7a4a5a21f095878e8
0bbc772f785d2cbf70a87c6b854eb566e1e1beb7d4ac6eb46023b3dc7fdf34529a40f5fc5797f
9c15c54ed4cb018c072168e9c30ca3602e00ea4047d2e5686c6eb37b9
```

```
e =
0x2c998e57bc651fe4807443dbb3e794711ca22b473d7792a64b7a326538dc528a17c79c72e
425bf29937e47b2d6f6330ee5c13bfd8564b50e49132d47befd0ee2e85f4bfe2c9452d62ef838
d487c099b3d7c80f14e362b3d97ca4774f1e4e851d38a4a834b077ded3d40cd20ddc45d5758
1beaa7b4d299da9dec8a1f361c808637238fa368e07c7d08f5654c7b2f8a90d47857e9b9c0a8
1a46769f6307d5a4442707afb017959d9a681fa1dc8d97565e55f02df34b04a3d0a0bf98b779
8d7084db4b3f6696fa139f83ada3dc70d0b4c57bf49f530dec938096071f9c4498fdef9641dfbf
e516c985b27d1748cc6ce1a4beb1381fb165a3d14f61032e0f76f095d
```

```
c =
0x503d5dd3bf3d76918b868c0789c81b4a384184ddadef81142eabdcb78656632e54c9cb22a
c2c41178607aa41adebdf89cd24ec1876365994f54f2b8fc492636b59382eb5094c46b5818cf8
d9b42aed7e8051d7ca1537202d20ef945876e94f502e048ad71c7ad89200341f8071dc73c2cc
1c7688494cad0110fca4854ee6a1ba999005a650062a5d55063693e8b018b08c4591946a3fc9
```

61dae2ba0c046f0848fbe5206d56767aae8812d55ee9decc1587cf5905887846cd3ecc6fc069e
40d36b29ee48229c0c79eceab9a95b11d15421b8585a2576a63b9f09c56a4ca1729680410da
237ac5b05850604e2af1f4ede9cf3928cbb3193a159e64482928b585ac

p =
98444549679044409506244239144443867459824227934526036052949278261505813439
01529745920037910875244423523266721313846407641509548690728828263059562228
72372158014709401468863715156799093220908714734123848945406423999500102962
14525469622505798526072170187467562765920044646574445427364231529083610955
760228212701

q =
13120530470771769980002321905708200798628604582368357166311201461218860671
00790387518534162737097290396229088619335271114696169001888759124304872645
76215232569029320804579614330240773622645122871884209068761138439268551367
19879800979063666289214806358313574794560477174045835289920242870464525679
0931460695949

print(bytes.fromhex(hex(pow(c, pow(e, -1, (p-1)*(q-1)), n))[2:]).decode())

crypto{bon3h5_4tt4ck_i5_sr0ng3r_th4n_w13n3r5}

Everything is big:
n =
0x8da7d2ec7bf9b322a539afb9962d4d2ebeb3e3d449d709b80a51dc680a14c87ffa863edfc7
b5a2a542a0fa610febe2d967b58ae714c46a6eccb44cd5c90d1cf5e271224aa3367e5a13305f7
44e2e56059b17bf520c95d521d34fdad3b0c12e7821a3169aa900c711e6923ca1a26c71fc5ac
8a9ff8c878164e2434c724b68b508a030f86211c1307b6f90c0cd489a27fdc5e6190f6193447e
0441a49edde165cf6074994ea260a21ea1fc7e2dfb038df437f02b9ddb7b5244a9620c8eca85
8865e83bab3413135e76a54ee718f4e431c29d3cb6e353a75d74f831bed2cc7bdce553f25b61
7b3bdd9ef901e249e43545c91b0cd8798b27804d61926e317a2b745
e =
0x86d357db4e1b60a2e9f9f25e2db15204c820b6e8d8d04d29db168c890bc8a6c1e31b9316c
9680174e128515a00256b775a1a8ccca9c6936f1b4c2298c03032cda4dd8eca1145828d31466
bf56bfcf0c6a8b4a1b2fb27de7a57fae7430048d7590734b2f05b6443ad60d89606802409d2fa
4c6767ad42bffae01a8ef1364418362e133fa7b2770af64a68ad50ad8d2bd5cebb99ceb13368f
b31a6e7503e753f8638e21a96af1b6498c18578ba89b98d70fa482ad137d28fe701b4b77baa2
5d5e84c81b26ee9bddf8cbb51a071c60dd57714de379cd4bc14932809ba18524a0a18e41336
65cfc46e2c4fcfbc28e0a0957e5513a7307c422b87a6182d0b6a074b4d
c =
0x6a2f2e401a54eeb5dab1e6d5d80e92a6ca189049e22844c825012b8f0578f95b269b19644c
7c8af3d544840d380ed75fdf86844aa8976622fa0501eaec0e5a1a5ab09d3d1037e55501c4e2
70060470c9f4019ced6c4e67673843daf2fd71c64f3dd8939ae322f2b79d283b3382052d076e
be9bb50b0042f1f7dd7beadf0f5686926ade9fc8370283ead781a21896e7a878d99e77c3bb1f
470401062c0e0327fd85da1cf12901635f1df310e8f8c7d87aff5a01dbbecd739cd8f36462060d
0eb237af8d613e2d9cebb67d612bcfc353ef2cd44b7ac85e471287eb04ae9b388b66ea8eb324
29ae96dba5da8206894fa8c58a7440a127fceb5717a2eaa3c29f25f7

p =
115507290436804681853972513785855229092334080356874717883434238235532664441400698329642751264652299576298636563034566154936812894976200811116395627642824129881201879664681775402664283913508399125714656956248098339209538838857780042711388040763913184276611985732635893909527514214876581238108724839614805837919
q =
154815838830735756266839897002132314538675909135249954852542100179590192055257100601759523401409380049463266265323780406554692248012224485469468355325698979825072726357567352976501667578692853621821116743497998613306224857520639271685772211046709014876104481766909036522349223506326852943709940047836080845531
print(bytes.fromhex(hex(pow(c, pow(e, -1, (p-1)*(q-1)), n))[2:]).decode())

crypto{s0m3th1ng5_c4n_b3_t00_b1g}

RSA backdoor viability:
n =
709872443186761582125747585668724501268558458558798673014673483766300964836479167241315660053878650421761726639872089885502004902487471946410918420927682586362111137364814638033425428214041019139158018673749256694555341525164012369589067354955298579131735466795918522816127398340465761406719060284098094643289390016311668316687808837563589124091867773655044913003668590954899705366787080923717270827184222673706856184434629431186284270269532605221507485774898673802583974291853116198037970076073697225047098901414637433392658500670740996008799860530032515716031449787089371403485205810795880416920642186451022374989891611943906891139047764042051071647203057520104267427832746020858026150611650447823314079076243582616371718150121483335889885277291312834083234087660399534665835291621232056473843224515909023120834377664505788329527517932160909013410933312572810208043849529655209420055180680775718614088521014772491776654380478948591063486615023605584483338460667397264724871221133652955371027085804223956104532604113969119716485142424996255737376464834315527822566017923598626634438066724763559943441023574575168924010274261376863202598353430010875182947485101076308406061724505065886990350185188453776162319552566614214624361251463
c =
6084846173161381264432756605242630255081353837456651754332295985174330300037042616581725823705437582776855475338340858995410361565954892063692797392109041547164645956574219486075699204988156315031972357023330178249935763268601666528453346175797985364420661849535509754870317210851057576678008381722259470012244951263905879503468229785196776735681215954278279801953324647470315774319259373142093914334076848457971711870065864550123647021609881471089898223929869666890579068846914992342983510036660199575287380943303897750544857314482745953303229768868755285252293375129099523910412800064260033007205477210727251685001046519619702927713823906477514504458923613113320746638953755449591931481146354768278553274218123075627424814878129652104062315075248308893754190455420578586796092653898693323318112186014403731217974

613189319768906743368075281071154239151527092652375903583483487165436839000
084640921475797266390455366908727400038393697480363793285799860812451995497
744422167439037225559951457819448752388203823448787222354051300473403913524
384955131506529773753511252544009417139303962299256151917084996289164519611
130753734119462168979728249628130229702602513174342320554419353669910333
8587843100187637572006174858230467771942700918388
e = 65537
p =
2036502927612137448623909363751805659117315356081608870497493422513763102602
100627872817226306709337512779951702164268302645394189208554959641555963
2837140072587743305574479218628388191587060262263170430315761890303990233871
5768605511661621105655750882431224118408754916145719317697891732168965276
6831843457114023104384188324674599747450017667192615361616877915240030631
3362477888262997093036136582318881633235376026276416829652885223234411339116
36273259031473139177094243362599271047539402167557257502744585237140073650977
2725581130537614203735350104770971283827769016324589620678432160581245381
480093375303381611323
q =
34857423162121791604235470898471761566115159084585269586007822559458774716
27716488251035886947629393917628761027489950978673682446174060361859854994
52730294798252904590623704246574461516239056536321816780659754729682428228
59926902463043730644958467921837687772906975274812905594211460094944271575
69800492037290572179885642980604009969883147170977409900344111156884344945
24075427993274679446856302587480288751034447601525874935437991856466926840
32460858150960790495575921455423185709811342689185127936111993248778962219
41345125854586308440372113563342849104647454047202959261313412576786400649
5572504245538373207974181
print(bytes.fromhex(hex(pow(c, pow(e, -1, (p-1)*(q-1)), n))[2:]).decode())
Flag: crypto{I_want_to_Break_Square-free_4p-1}

Infinite decent:
n =
38334771233087704045223861932952484176339252614684057223292692464209489145
39792463837989133941143053683604268670216236496670242172665290008597035425
90316063318592391925062014229671423777966797987471312505524553560618347195
12365575593221216339005132464338847195248627639623487124025890693416305788
16090576201182507933688056746103332224001577110292969635016193795038742769
63858504437277779964835844646100463807227367907901880619643112221539856142
87276995741553706506834906746892708903948496564047090014307484054609862129
53026210866956783472635207806008188971210941207373102603046630006034173750
42238220147140564137521658417493681595105881786040961919567509410783914156
34472219765129561622344109769892244712668402761549412177892054051266761597
33066054570431721056775982875715690477849560896878574799805985746744012815
60683917469196842582276828660836623452636595580668641092124572861145062284
70930775092735385388316268663664139056183180238043386636254075940621543717
53167099582341707066600593045283638981212946205177164604849839719515740538
69234468938865930486809848969898091358022768929110385880087019267292698112

45322689177654603766358389362547925264304251719695899026637674167651463108
9466493864064316127648074609662749196545969926051
e = 65537
c =
98280456757136766244944891987028935843441533415613592591358482906016439563
07615052611636984221310333348050670599363390199410728189018724849550727086
86213846522076976070198991664921324083487892525551964286086613206718774127
10489782358282011364127799563335562917707783563681920786994453004763755404
51054157450217624389675683991799184842809159491911144802394852776636830450
31006503799141530581911400725280958985760188938298301043621249271405551079
94114143042266758709328068902664037870075742542194318059191313468675939426
81098823907942482349531746403525232552191759204519815264353322301595270264
92494947533951009735345417662855518918596493203711785622002522287793953939
74169736998523394598517174182142007480526603025578004665936854657294541338
69751352100781855225481179756686076344260436574459644473599173279092634372
01022934534299367342062461099688171588157499270635618352746361951497023174
15680401987150336994583752062565237605953153790371155918439941193401473271
75303818056012978419280035164972446555373320145158152517353673167452414502
79319232049612743698263793250516012383086351925402234840550962032934004198
16024111797903442864181965959247745006822690967920957905188441550106930799
896292835287867403979631824085790047851383294389
p =
19579267410474709598749314750954211170621862561006233612440352022286786882
37261913007163982410978354056451242908167413233681197240456395702546503402
57812064666317307845163372102913343563964717321687427397904641098810392194
52504456611589154349427303832789968502204300316585544080003423669120186095
18847848076110816829937032692812788878681939237247706951531817975170298580
90242101642434095446927086842150422269320810528310285700603089630932176221
83111643335692361019897449265402290540025790581589980867847884281862216603
57153625538229803533786588515332816963417832327900474991519727012032334041
6965014136429743252761521
q =
19579267410474709598749314750954211170621862561006233612440352022286786882
37261913007163982410978354056451242908167413233681197240456395702546503402
57812064666317307845163372102913343563964717321687427397904641098810392194
52504456611589154349427303832789968502204300316585544080003423669120186095
18847848076110816829937032692812788878681939237247706951531817975170298580
90242101642434095446927086842150422269320810528310285700603089630932176221
83111643335692362635203582868526178838018946986792656819885261069890315500
55080230362255102982105845916370275189379867644341568114442909698966447370
5850619792495553724950931
print(bytes.fromhex(hex(pow(c, pow(e, -1, (p-1)*(q-1)), n))[2:]).decode())
Flag:crypto{f3rm47_w45_4_g3n1u5}

Crossed wires:
e = 0x10001

```
n =
217113082253463155427068446184415657410464982777169799434783605980531449713799569165753703434489886019058545720296358466262594872979503052316611098558549474942091352055892586435179615215949243684986720642932082308024410773901936829580951119220826778131758047756288843777243776474283858418312770592741729822805452377655599692287075068575612152684910240970639203377217836730605301816371615774015891265585561825468967833073705172750465227040473857861114894470647942100108027617086159072455234925858962863749960880893178261627982785282962069779002744318298292061032271718392708874764368994944283713238746890556907299867771
d =
273441167725114803072313800571610973383886654537552760201825515931963102665319078367049310793640160398142917188050436056049477101724646870290264737095422031245254134285874759057627377510787045085353371711668432697626300643573338204580797189076201874772957402105743033177803398235918483815974733123653850184996532926477492760757041034701941840745193787568437345498230692317840316121681723789096265121471883195421520063765110390720934790085782472265321717954814814568718137722054486452180823012273096745298143535533493210426548807577763860804132525677627520006754153302252796474347855494879257805770852235081215488809 7
```

```
n =
2171130822534631554270684461844156574104649827771697994347836059805314497137995691657537034344898860190585457202963584662625948729795030523166110985585494749420913520558925864351796152159492436849867206429320823080244107739019368295809511192208267781317580477562888437772437764742838584183127705927417298228054523776555996922870750685756121526849102409706392033772178367306053018163716157740158912655855618254689678330737051727504652270404738578611148944706479421001080276170861590724552349258589628637499608808931782616279827852829620697790027443182982920610322717183927088747643689949442837132387468905569072998677 1
d =
2734411677251148030723138005716109733838866545375527602018255159319631026653190783670493107936401603981429171880504360560494771017246468702902647370954220312452541342858747590576273775107870450853533717116684326976263006435733382045807971890762018747729574021057430331778033982359184838159747331236538501849965329264774927607570410347019418407451937875684373454982306923178403161216817237890962651214718831954215200637651103907209347900857824722653217179548148145687181377220544864521808230122730967452981435355334932104265488075777638608041325256776275200067541533022527964743478554948792578057708522350812154888097
c =
2030461027957818673817276622422479311988507126246446444886346118409222573605474797698517967390544150268912621628289770450874540379905473412158396885399979160428161515410073625913145342438536432463022967118534377817280726264070930183827482460310169248566272622690212110559113743733146320188126424556221401216087517716744201095243936062339665897441390046909383679475227039952007459632905872587483408218869737759794940577903913919419606536442621320834546140703077108978752920005710574658449355472279059253047286958131011730034346120775082173784004274553087639179348403502464447553535322785132150553739888810685501274611 7
print(bytes.fromhex(hex(pow(c, pow(106979*108533*69557*97117*103231, -1, e*d-1),
n))[2:]).decode())
```

```
    Break
```

Ans:1420

Jacks birthday confusion

```
from math import factorial
n = 2048
for i in range(n):
        probability = 1 - factorial(n) / (factorial(n - i)*pow(n,i))
        if probability > 0.75:
                print(i)
                Break
```

Ans:76

Collider

```
from utils import listener


FLAG = "crypto{???????????????????????????????????}"



class Challenge():
  def __init__(self):
    self.before_input = "Give me a document to store\n"
    self.documents = {
      "508dcc4dbe9113b15a1f971639b335bd": b"Particle physics (also known as high energy physics) is a branch of physics that studies the nature of the particles that constitute matter and radiation. Although the word particle can refer to various types of very small objects (e.g. protons, gas particles, or even household dust), particle physics usually investigates the irreducibly smallest detectable particles and the fundamental interactions necessary to explain their behaviour.",
            "cb07ff7a5f043361b698c31046b8b0ab": b"The Large Hadron Collider (LHC) is the world's largest and highest-energy particle collider and the largest machine in the world. It was built by the European Organization for Nuclear Research (CERN) between 1998 and 2008 in collaboration with over 10,000 scientists and hundreds of universities and laboratories, as well as more than 100 countries.",
        }

    def challenge(self, msg):
        if "document" not in msg:
            self.exit = True
            return {"error": "You must send a document"}

        document = bytes.fromhex(msg["document"])
        document_hash = hashlib.md5(document).hexdigest()

        if document_hash in self.documents.keys():
            self.exit = True
            if self.documents[document_hash] == document:
                return {"error": "Document already exists in system"}
            else:
                return {"error": f"Document system crash, leaking flag: {FLAG}"}

        self.documents[document_hash] = document

        if len(self.documents) > 5:
            self.exit = True
            return {"error": "Too many documents in the system"}

        return {"success": f"Document {document_hash} added to system"}
```

```
"""
When you connect, the 'challenge' function will be called on your JSON
input.
"""
listener.start_server(port=13389)
```

Ans:

Hash stuffing:

```
from pwn import *
import json
block1 = b'a'*32
block2 = b'b'*32

m1 = (block1 + block2).hex()
m2 = (block2 + block1).hex()
payload = json.dumps({"m1" : m1, "m2" : m2}).encode()
r = remote("socket.cryptohack.org", 13405)
r.sendlineafter(b'in JSON: ', payload)
data = r.recvline()
print(data)
r.close()
```

Ans:**crypto{Always_add_padding_even_if_its_a_whole_block!!!}**

MD0

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import os
from utils import listener
```

```
FLAG = "crypto{???????????????}"


def bxor(a, b):
    return bytes(x ^ y for x, y in zip(a, b))


def hash(data):
    data = pad(data, 16)
    out = b"\x00" * 16
    for i in range(0, len(data), 16):
        blk = data[i:i+16]
        out = bxor(AES.new(blk, AES.MODE_ECB).encrypt(out), out)
    return out


class Challenge():
    def __init__(self):
        self.before_input = "You'll never forge my signatures!\n"
        self.key = os.urandom(16)

    def challenge(self, msg):
        if "option" not in msg:
            return {"error": "You must send an option to this server."}

        elif msg["option"] == "sign":
            data = bytes.fromhex(msg["message"])
            if b"admin=True" in data:
                return {"error": "Unauthorized to sign message"}
            sig = hash(self.key + data)
            return {"signature": sig.hex()}
        elif msg["option"] == "get_flag":
            sent_sig = bytes.fromhex(msg["signature"])
            data = bytes.fromhex(msg["message"])
            real_sig = hash(self.key + data)

            if real_sig != sent_sig:
                return {"error": "Invalid signature"}

            if b"admin=True" in data:
                return {"flag": FLAG}
            else:
                return {"error": "Unauthorized to get flag"}

        else:
            return {"error": "Invalid option"}

"""
When you connect, the 'challenge' function will be called on your JSON
input.
"""
listener.start_server(port=13388)
```

Ans:crypto{l3ngth_3xT3nd3r}