Phòng thí nghiệm An toàn Thông tin InsecLab
Trường Đại học Công nghệ Thông tin (UIT)

BÁO CÁO THỰC HÀNH

Môn học: Mật mã học

Tên chủ đề: Cryptohack

GVHD: Tô Trọng Nghĩa

1. **THÔNG TIN CHUNG:**

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT219.N21.ANTT

STT	Họ và tên	MSSV	Email
1	Trần Minh Duy	21522010	21522010@gm.uit.edu.vn

2. **NỘI DUNG THỰC HIỆN:**[1]

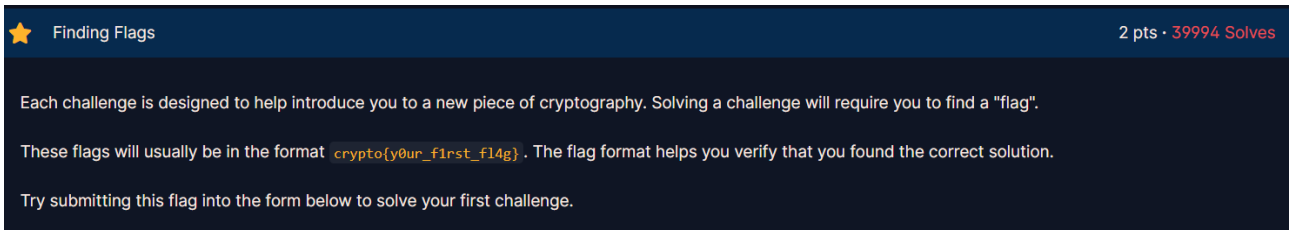
STT	Nội dung	Tình trạng	Trang
1	Giải 10 challenges bất kỳ trên trang cryptohack	>100%	2 – 20
Điểm tự đánh giá			**10/10**

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

[1] Ghi nội dung công việc, các kịch bản trong bài Thực hành

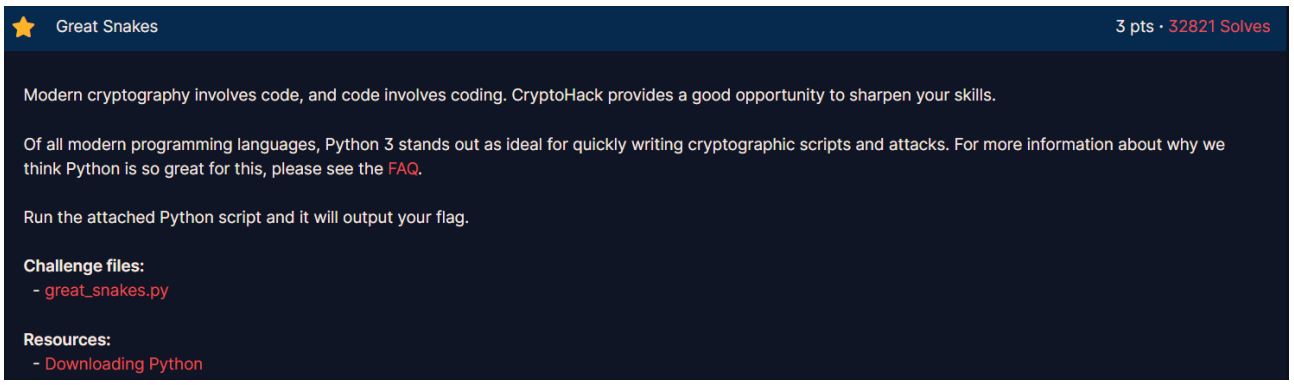
BÁO CÁO CHI TIẾT

1.

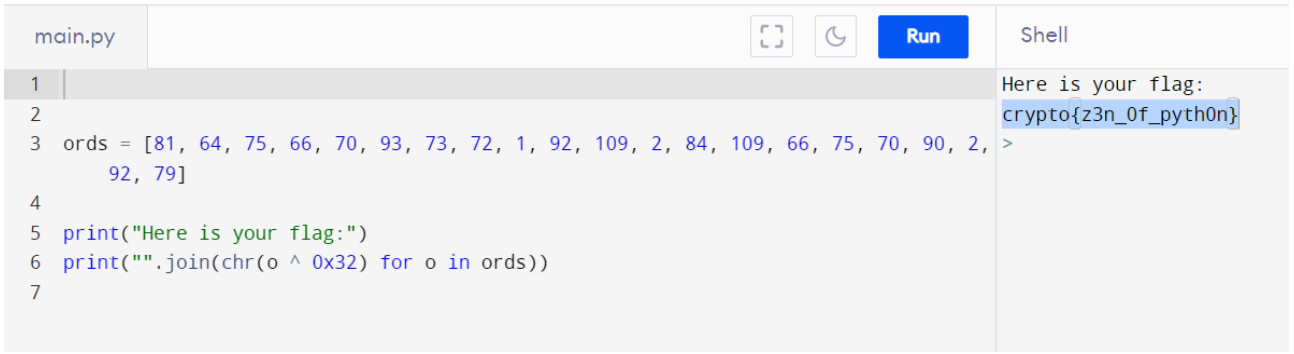


Submit flag following the instruction **crypto{y0ur_f1rst_fl4g}** and pass.

2.

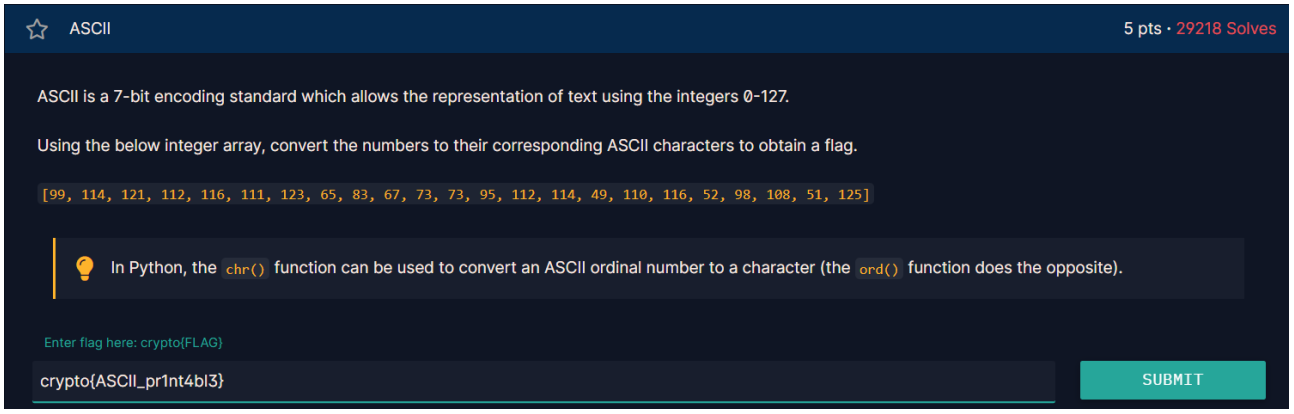


Use https://www.programiz.com/python-programming/online-compiler to run great_snakes.py:

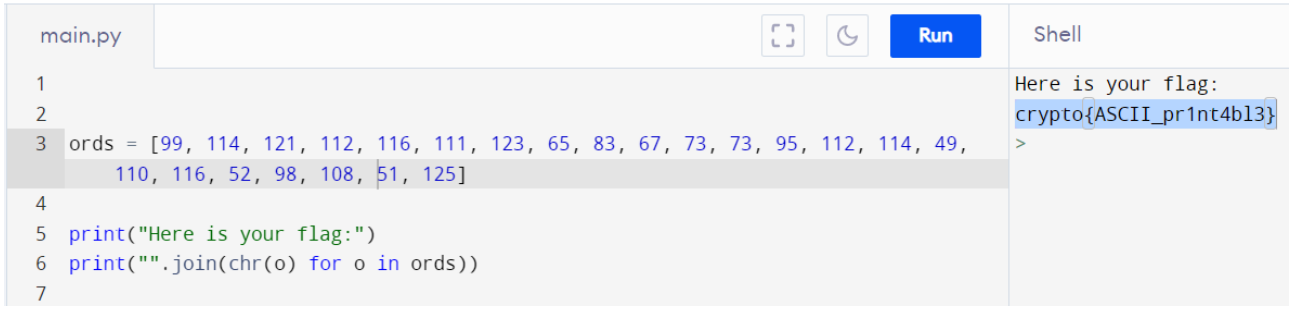


⇨ Flag: crypto{z3n_0f_pyth0n}

3.

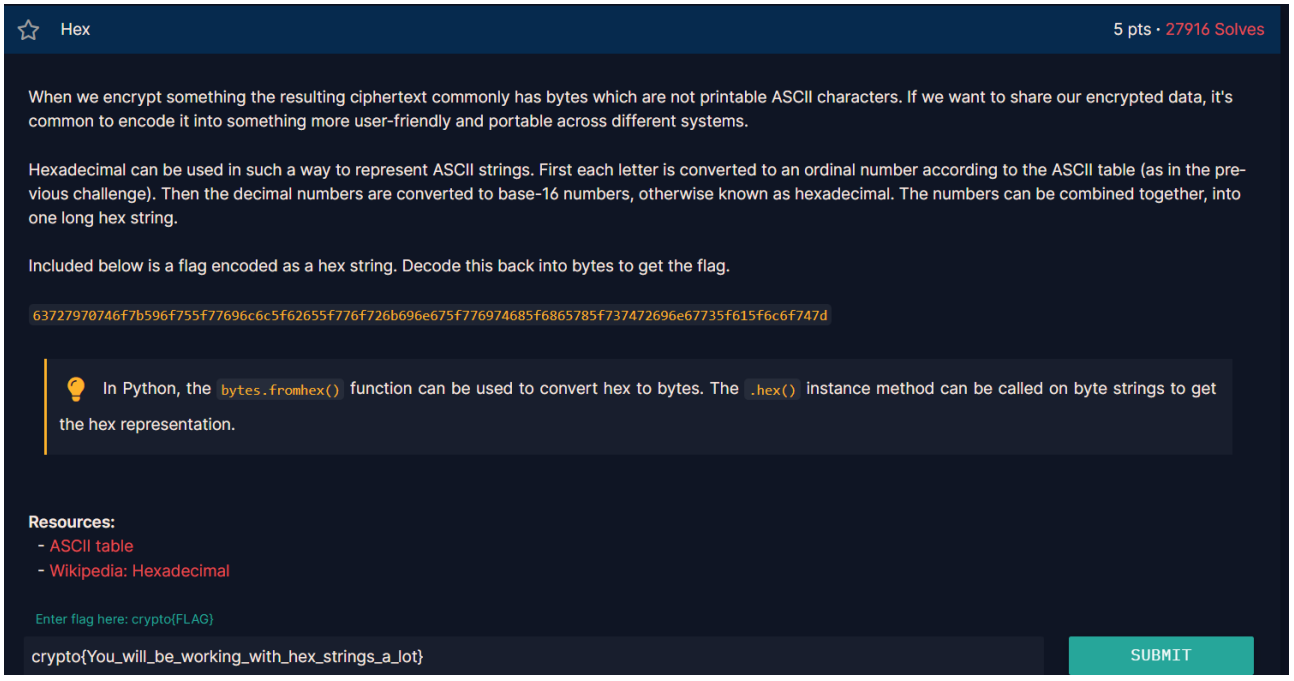


Coincidentally, using the similar code as the previous challenge, we can pass this challenge, just change the ords array:

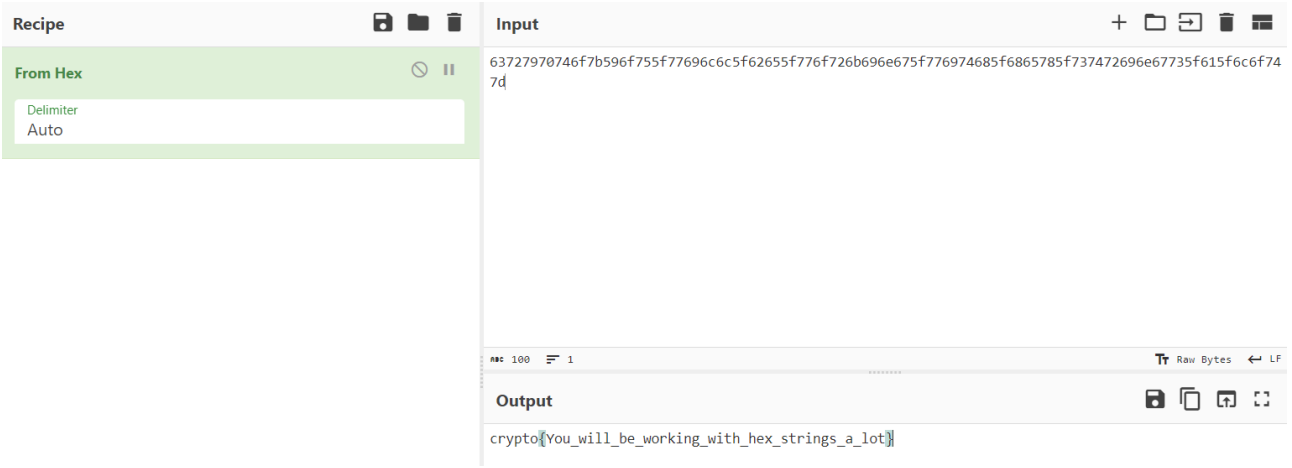


⇨ So flag is: crypto{ASCII_pr1nt4bl3}

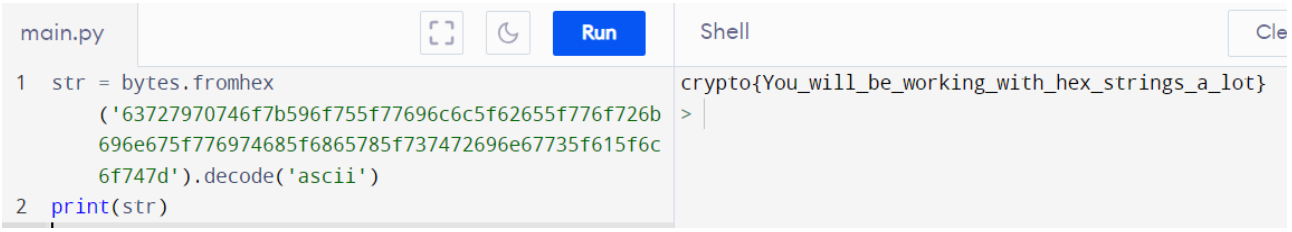
4.



Use cyberchef tool:

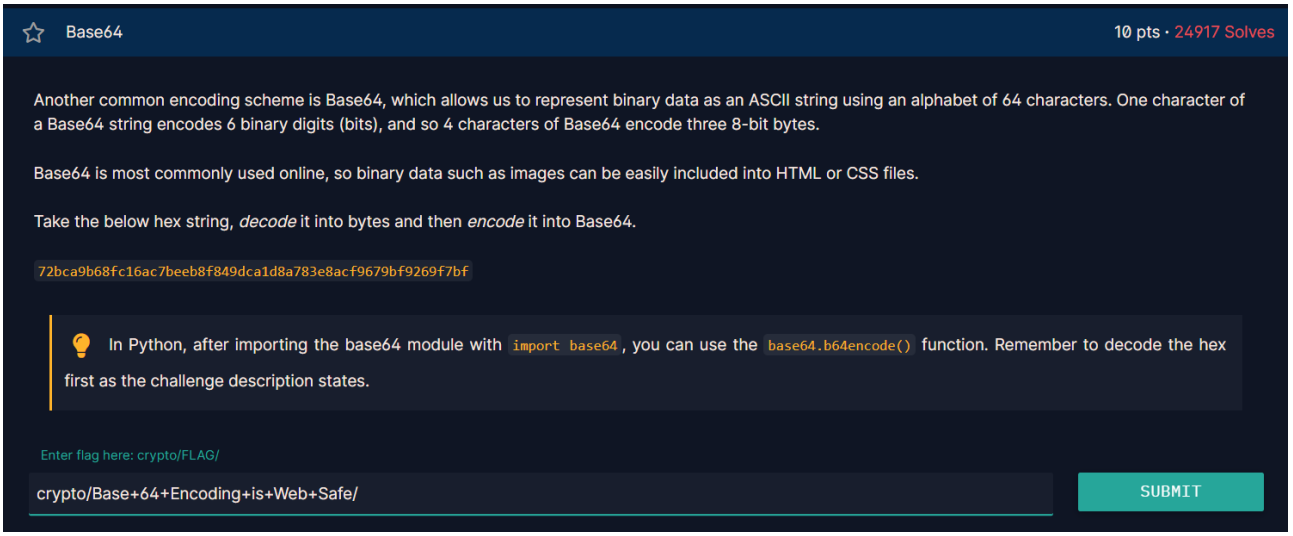


Or python code:

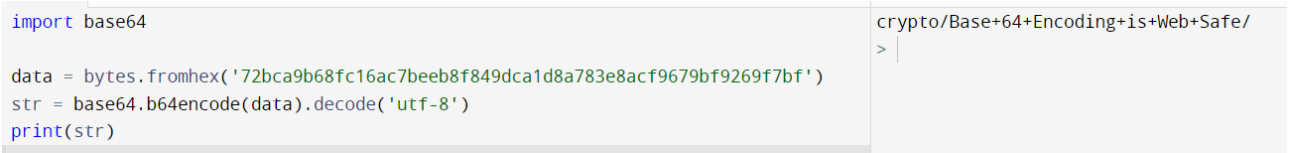


⇨ The flag: crypto{You_will_be_working_with_hex_strings_a_lot}

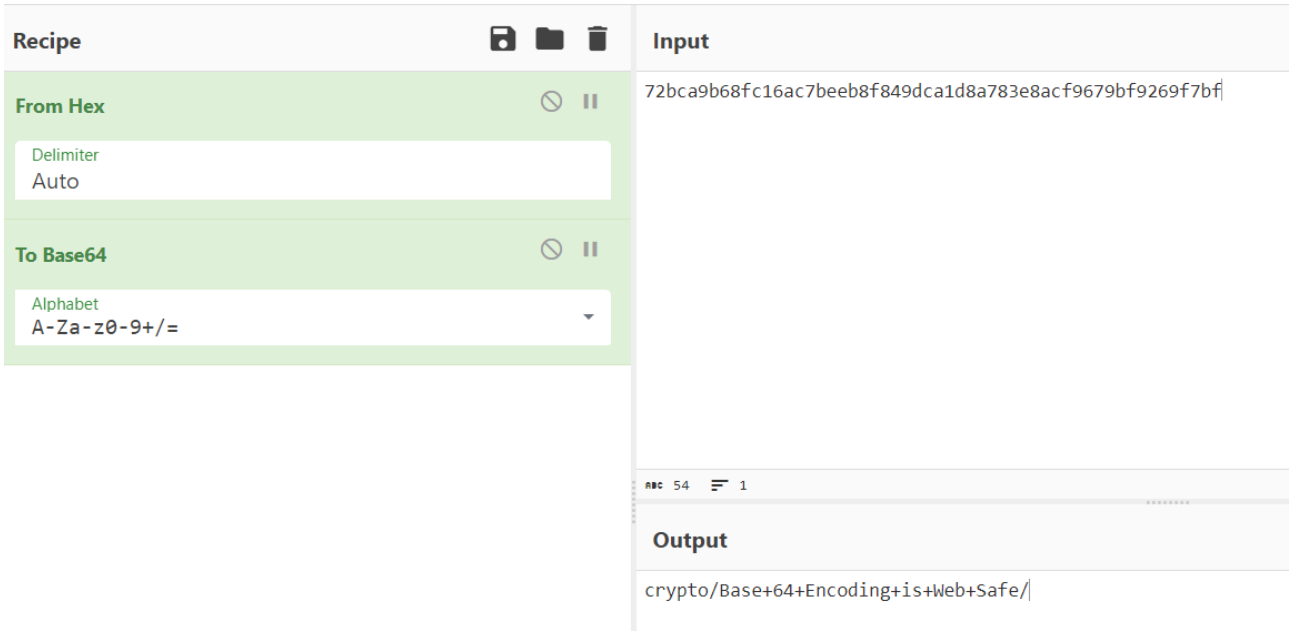
5.



Use code:

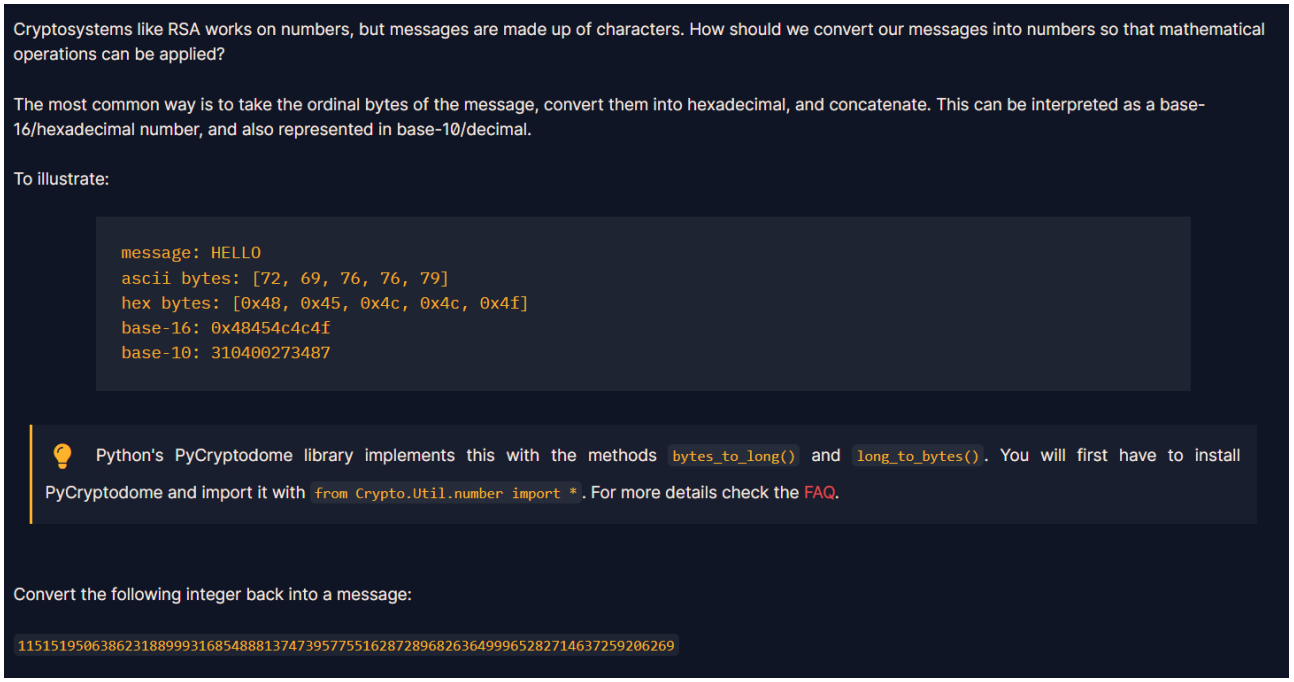


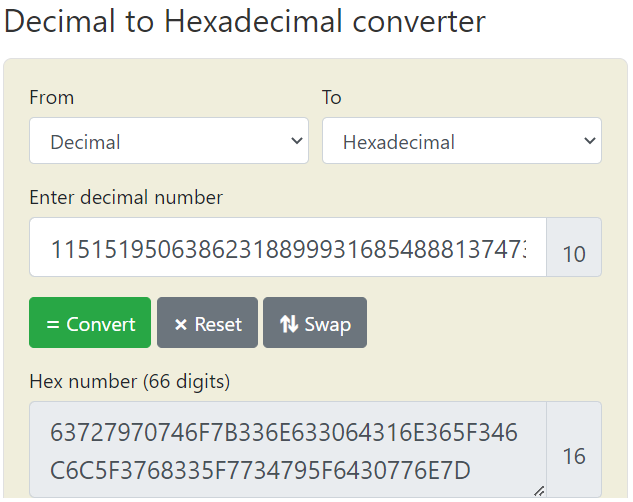
Use cyberchef tool:



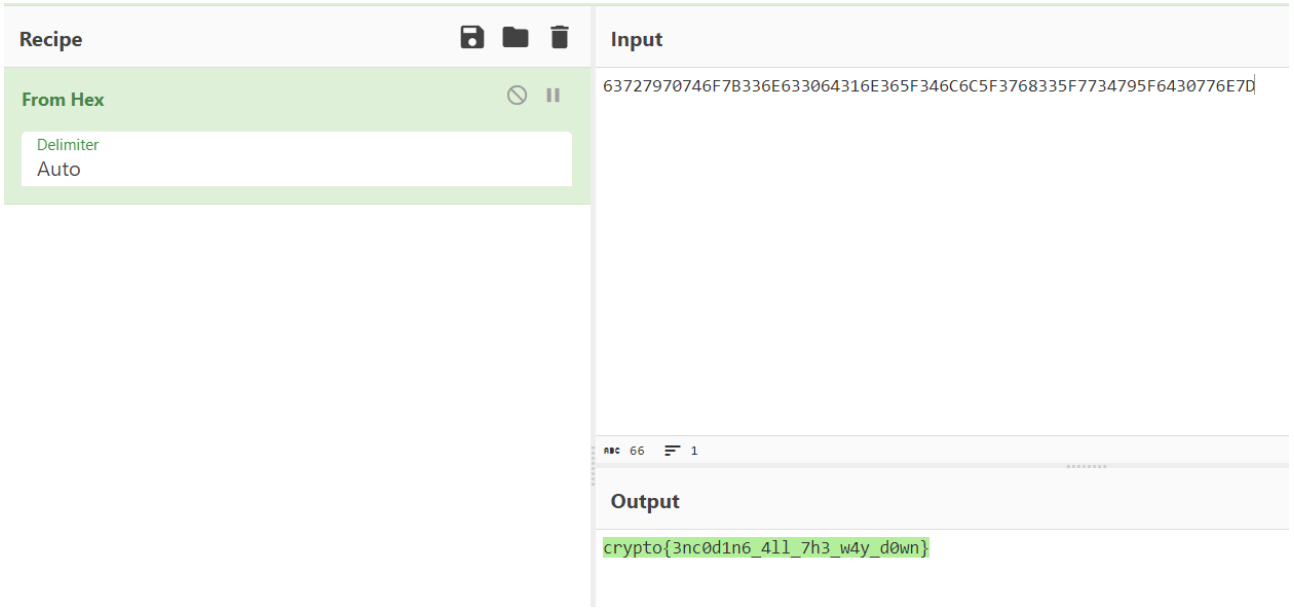
⇨ The flag: crypto/Base+64+Encoding+is+Web+Safe/

6.



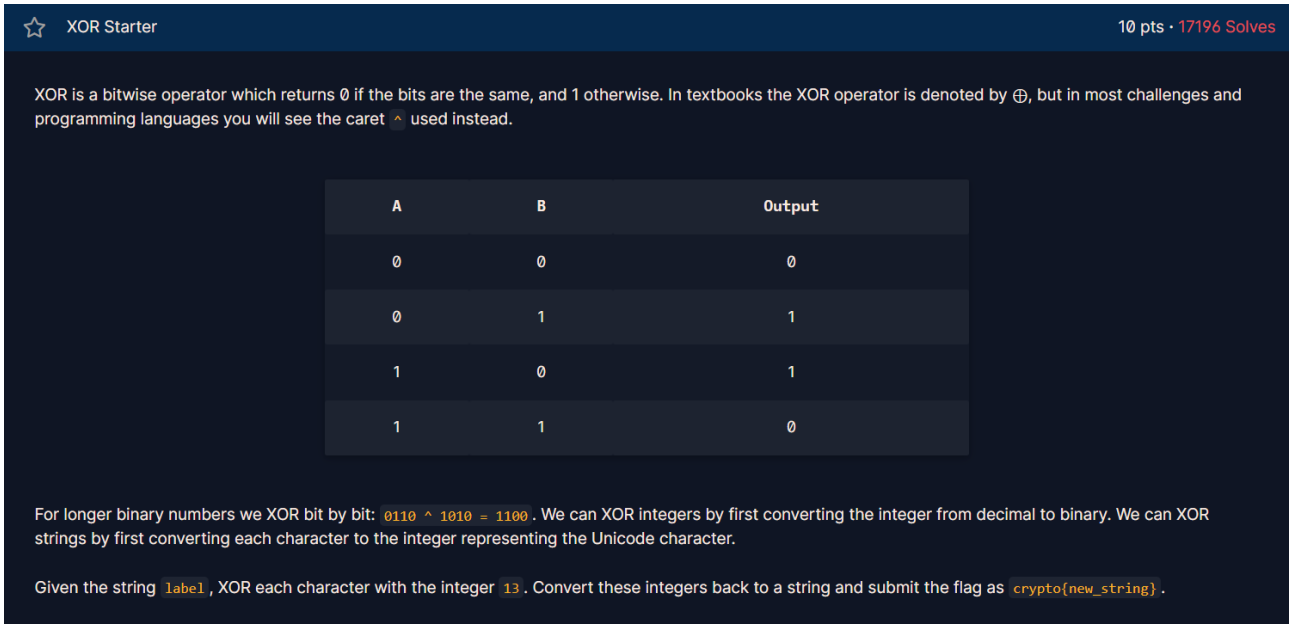


After num is converted to base64, we continue to hex decode to message:



⇨ Flag: crypto{3nc0d1n6_4ll_7t3_w4y_d0wn}

7.



XOR Starter — 10 pts · 17196 Solves

XOR is a bitwise operator which returns 0 if the bits are the same, and 1 otherwise. In textbooks the XOR operator is denoted by ⊕, but in most challenges and programming languages you will see the caret ^ used instead.

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

For longer binary numbers we XOR bit by bit: `0110 ^ 1010 = 1100`. We can XOR integers by first converting the integer from decimal to binary. We can XOR strings by first converting each character to the integer representing the Unicode character.

Given the string `label`, XOR each character with the integer `13`. Convert these integers back to a string and submit the flag as `crypto{new_string}`.

My python code and result:

main.py — Run — Shell

```
def xor_with_13(label):
    result = ""
    for char in label:
        result += chr(ord(char) ^ 13)
    return result

label = "label"
encrypted_label = xor_with_13(label)
print(encrypted_label)
```

```
aloha
>
```

⇨ The flag: crypto{aloha}

8.

XOR Properties — 15 pts · 14136 Solves

In the last challenge, you saw how XOR worked at the level of bits. In this one, we're going to cover the properties of the XOR operation and then use them to undo a chain of operations that have encrypted a flag. Gaining an intuition for how this works will help greatly when you come to attacking real cryptosystems later, especially in the block ciphers category.

There are four main properties we should consider when we solve challenges using the XOR operator

```
Commutative: A ⊕ B = B ⊕ A
Associative: A ⊕ (B ⊕ C) = (A ⊕ B) ⊕ C
Identity: A ⊕ 0 = A
Self-Inverse: A ⊕ A = 0
```

Let's break this down. Commutative means that the order of the XOR operations is not important. Associative means that a chain of operations can be carried out without order (we do not need to worry about brackets). The identity is 0, so XOR with 0 "does nothing", and lastly something XOR'd with itself returns zero.

Let's put this into practice! Below is a series of outputs where three random keys have been XOR'd together and with the flag. Use the above properties to undo the encryption in the final line to obtain the flag.

```
KEY1 = a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313
KEY2 ^ KEY1 = 37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e
KEY2 ^ KEY3 = c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1
FLAG ^ KEY1 ^ KEY3 ^ KEY2 = 04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf
```

My python code and result:

main.py — Run — Shell

```
1  KEY1 = 0xa6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313
2  KEY2_KEY1 = 0x37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e
3  KEY2_KEY3 = 0xc1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1
4  FLAG_KEY1_KEY3_KEY2 = 0x04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf
5
6  key2 = KEY1 ^ KEY2_KEY1
7  key3 = key2 ^ KEY2_KEY3
8  flag = FLAG_KEY1_KEY3_KEY2 ^ KEY1 ^ key2 ^ key3
9
10 print(bytes.fromhex(hex(flag)[2:]))
```

```
b'crypto{x0r_i5_ass0c1at1v3}'
>
```

⇨ Flag: crypto{x0r_i5_ass0c1at1v3}

9.

Favourite byte — 20 pts · 13868 Solves

For the next few challenges, you'll use what you've just learned to solve some more XOR puzzles.

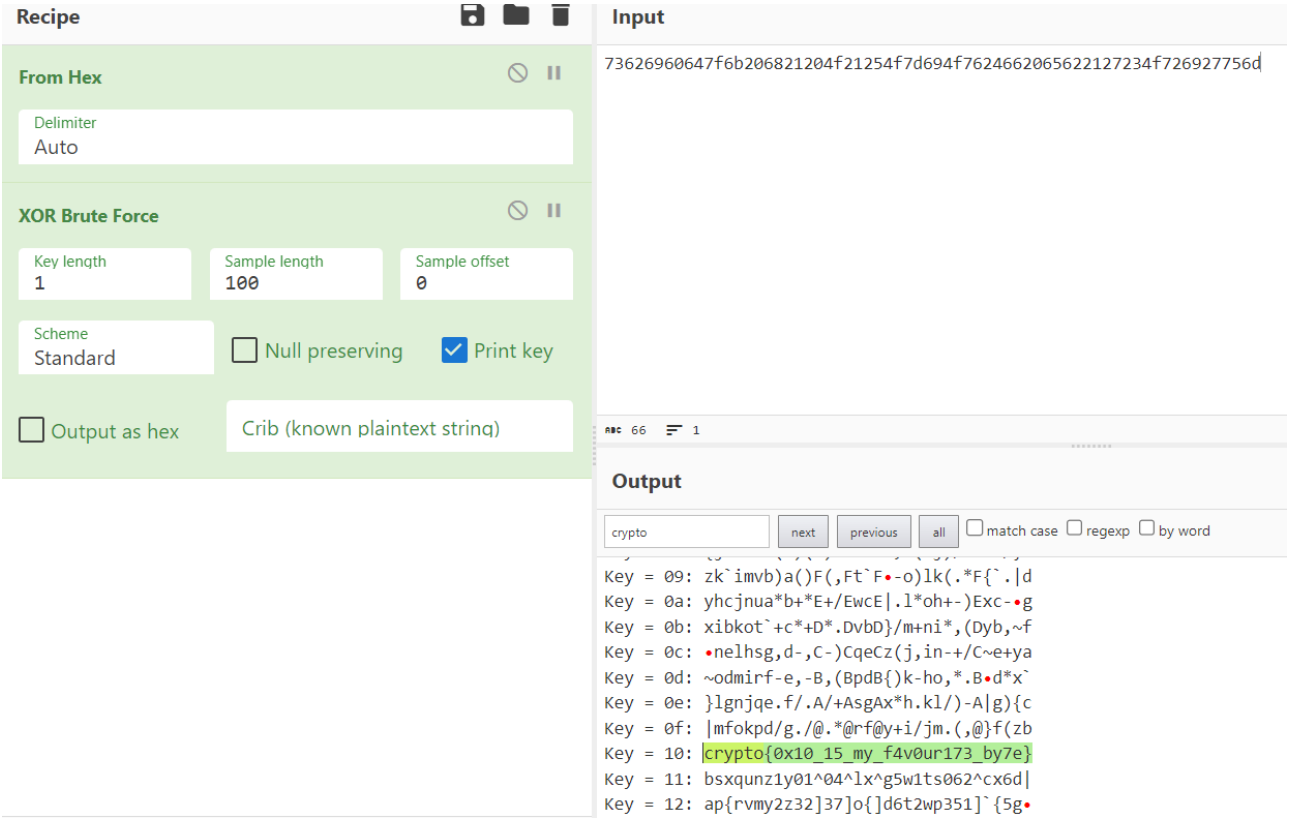
I've hidden some data using XOR with a single byte, but that byte is a secret. Don't forget to decode from hex first.

```
73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d
```

Enter flag here: crypto{FLAG}

crypto{0×10_15_my_f4v0ur173_by7e} SUBMIT

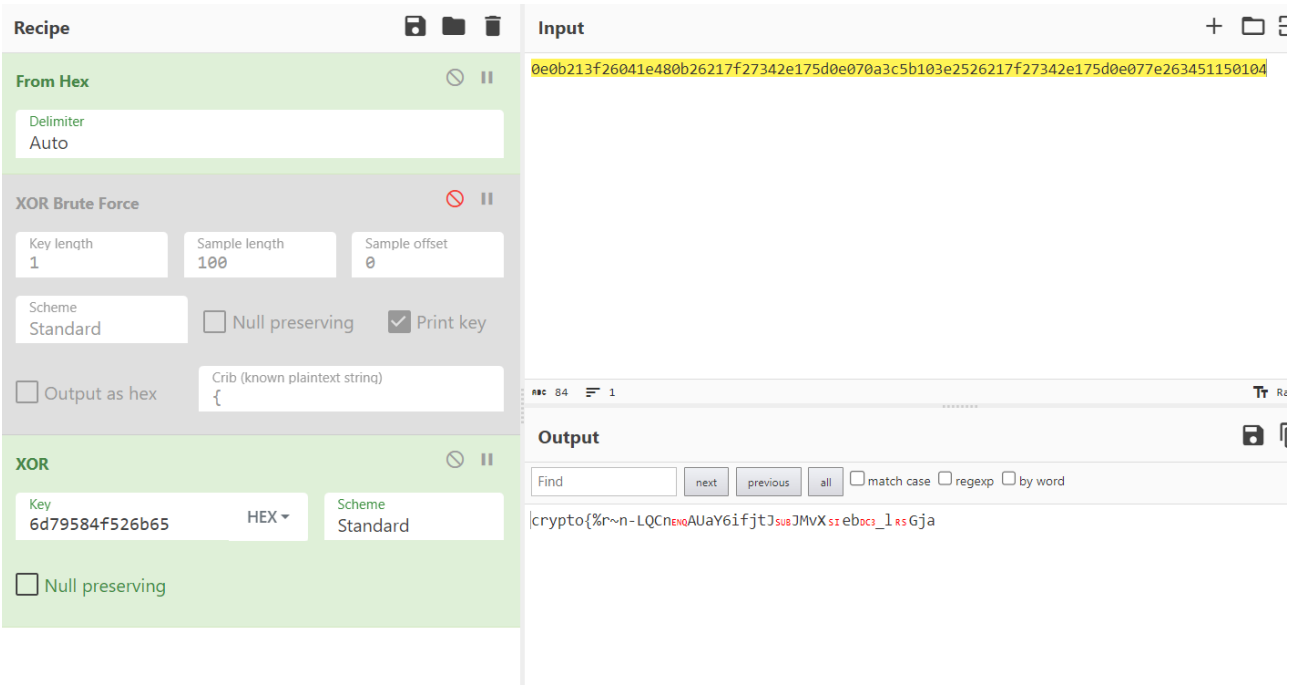
Use cyberchef tool to brute-force:

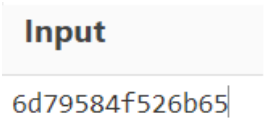


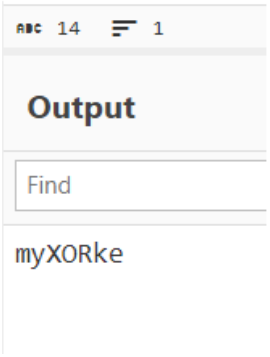
Here we get a meaning result with xor key is 0x10, it's also in flag format too, so there is flag: crypto{0x10_15_my_f4v0ur173_by7e}

10.

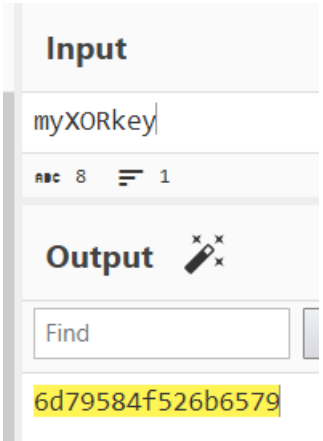
Encode the string "crypto{" and xor with data string of the challenge, we will have first 14 hex digit is: 6d 79 58 4f 52 6b 65

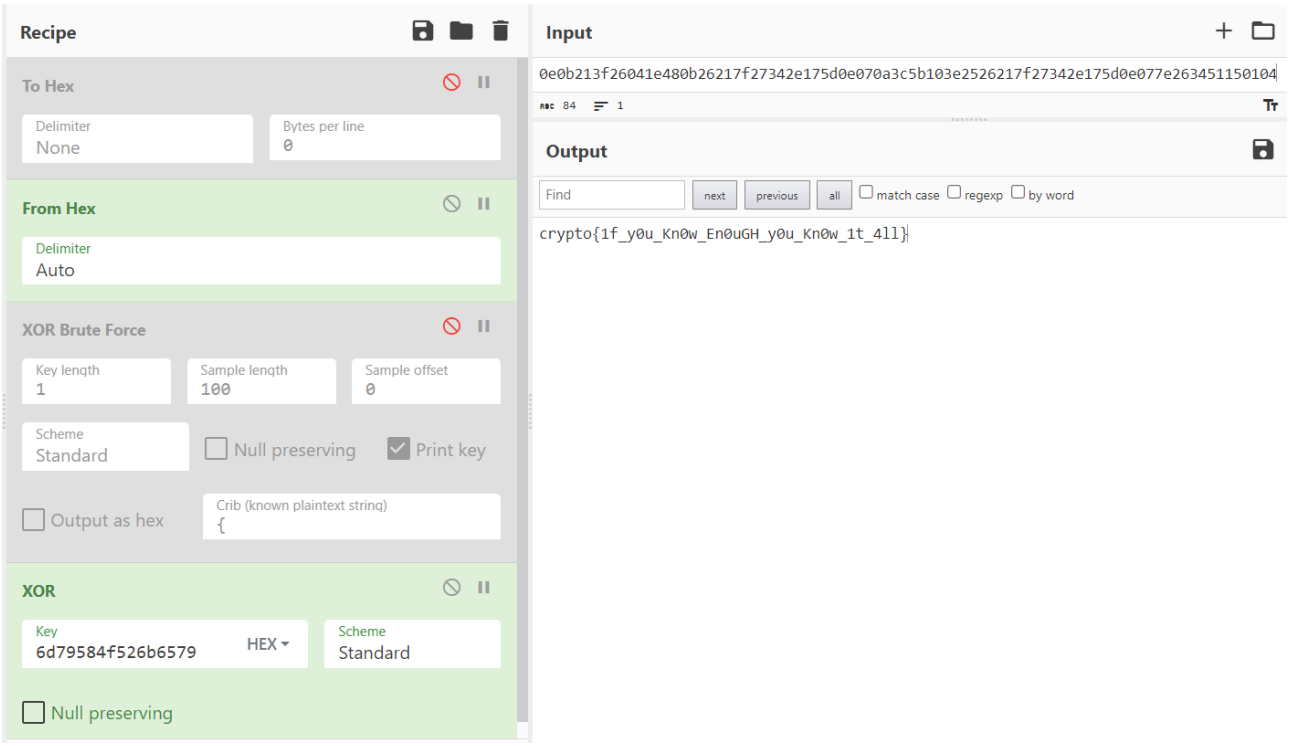






=> Guess the key: “myXORkey”

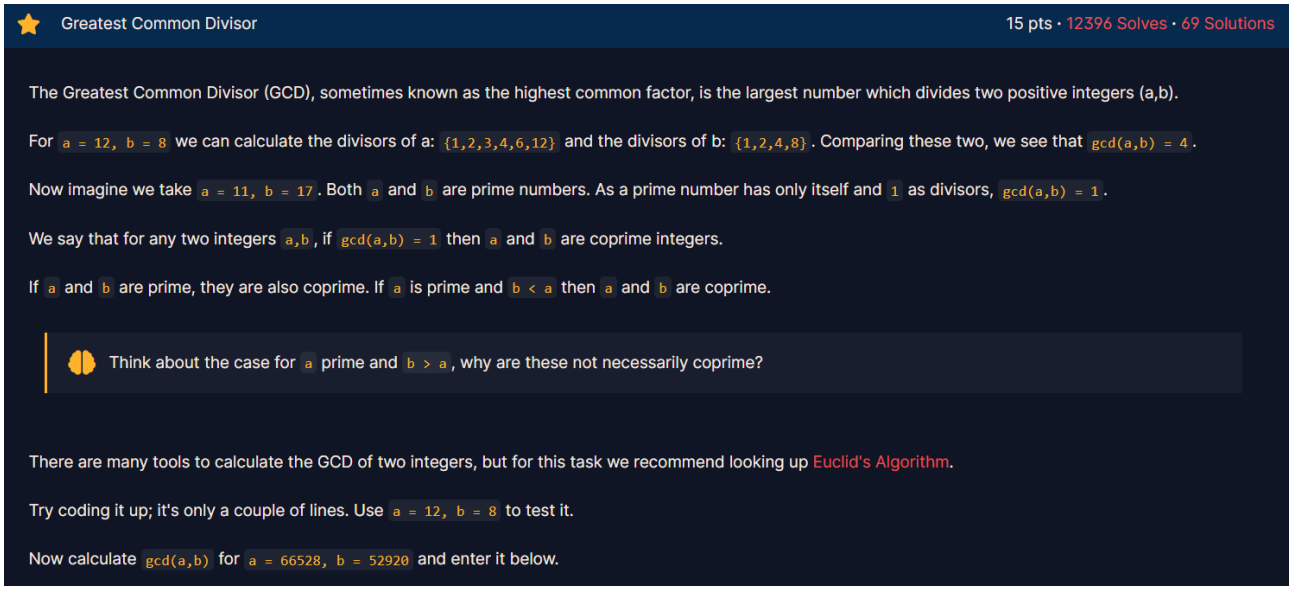




Yeah, it seems my guess is true, sometimes simple is best. By the way, the flag is:

crypto{1f_y0u_Kn0w_En0uGH_y0u_Kn0w_1t_4ll}

11.



Belows are my code and result*(the top right corner of the picture)*:

```python
def gcd(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

print(gcd(52920,66528))
```

```
1512
>
```

⇨ Flag is '1512'.

12.

Extended GCD — 20 pts · 10015 Solves · 48 Solution

Let `a` and `b` be positive integers.

The extended Euclidean algorithm is an efficient way to find integers `u,v` such that

`a * u + b * v = gcd(a,b)`

> Later, when we learn to decrypt RSA, we will need this algorithm to calculate the modular inverse of the public exponent.

Using the two primes `p = 26513, q = 32321`, find the integers `u,v` such that

`p * u + q * v = gcd(p,q)`

Enter whichever of `u` and `v` is the lower number as the flag.

> Knowing that `p,q` are prime, what would you expect `gcd(p,q)` to be? For more details on the extended Euclidean algorithm, check out this page.

My code and result:

```python
def extended_gcd(a, b):
    if b == 0:
        return (1, 0)
    else:
        x, y = extended_gcd(b, a % b)
        return (y, x - (a // b) * y)
```

```
1   extended_gcd(26513, 32321)
(10245, -8404)
```

⇨ Flag is '–8404' (because it is the lower number).

13.

Modular Arithmetic 1 — 20 pts · 10048 Solves · 13 Solutions

Imagine you lean over and look at a cryptographer's notebook. You see some notes in the margin:

```
4 + 9 = 1
5 - 7 = 10
2 + 3 = 5
```

At first you might think they've gone mad. Maybe this is why there are so many data leaks nowadays you'd think, but this is nothing more than modular arithmetic modulo 12 (albeit with some sloppy notation).

You may not have been calling it modular arithmetic, but you've been doing these kinds of calculations since you learnt to tell the time (look again at those equations and think about adding hours).

Formally, "calculating time" is described by the theory of congruences. We say that two integers are congruent modulo m if `a ≡ b mod m`.

Another way of saying this, is that when we divide the integer `a` by `m`, the remainder is `b`. This tells you that if m divides a (this can be written as `m | a`) then `a ≡ 0 mod m`.

Calculate the following integers:

```
11 ≡ x mod 6
8146798528947 ≡ y mod 17
```

The solution is the smaller of the two integers.

My code and result:

```
1   min(11%6, 8146798528947%17)
```

```
4
```

⇨ Flag is '4' (Is this challenge too simple? I don't know :>)

14.

Modular Arithmetic 2 — 20 pts · 9552 Solves

We'll pick up from the last challenge and imagine we've picked a modulus `p`, and we will restrict ourselves to the case when `p` is prime.

The integers modulo `p` define a field, denoted F_p.

> If the modulus is not prime, the set of integers modulo `n` define a ring.

A finite field F_p is the set of integers $\{0,1,...,p-1\}$, and under both addition and multiplication there is an inverse element `b` for every element `a` in the set, such that `a + b = 0` and `a * b = 1`.

> Note that the identity element for addition and multiplication is different! This is because the identity when acted with the operator should do nothing: `a + 0 = a` and `a * 1 = a`.

Lets say we pick `p = 17`. Calculate $3^{17} \bmod 17$. Now do the same but with $5^{17} \bmod 17$.

What would you expect to get for $7^{16} \bmod 17$? Try calculating that.

This interesting fact is known as Fermat's little theorem. We'll be needing this (and its generalisations) when we look at RSA cryptography.

Now take the prime `p = 65537`. Calculate $273246787654^{65536} \bmod 65537$.

Did you need a calculator?

Again, my code and result:

```
1   pow(273246787654, 65536, 65537)
1
```

⇨ Flag is '1'.

But there is a question here: "*Did you need a caculator?*"

Looking at Fermat's little theorem...

if p is prime, for every integer a:

pow(a, p) = a mod p

and, if p is prime and abs(a) is an integer less than p:

pow(a, p-1) = 1 mod p

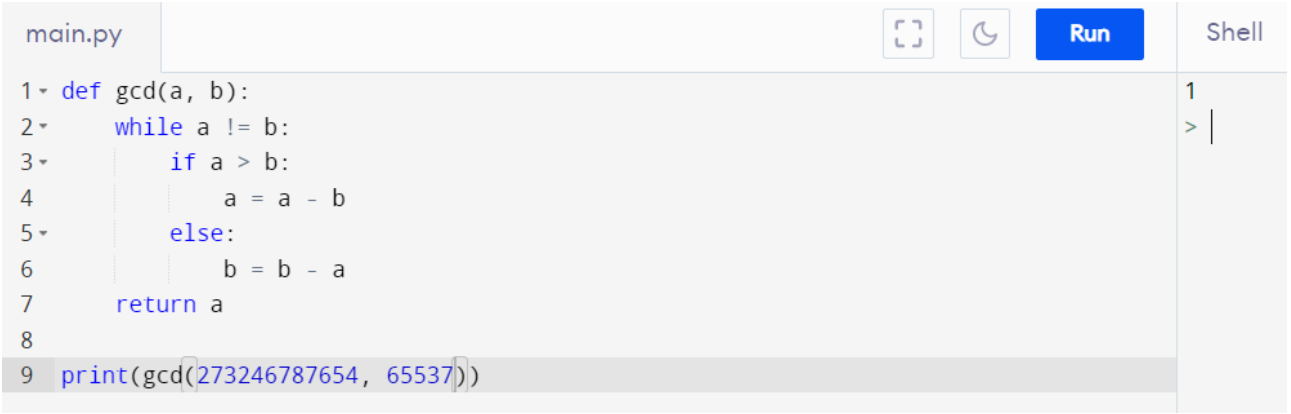
So lets check

pow(273246787654, 65536) mod 65537

Notice that 65536 is exactly 65537-1,

If 273246787654 and 65537 are coprime,

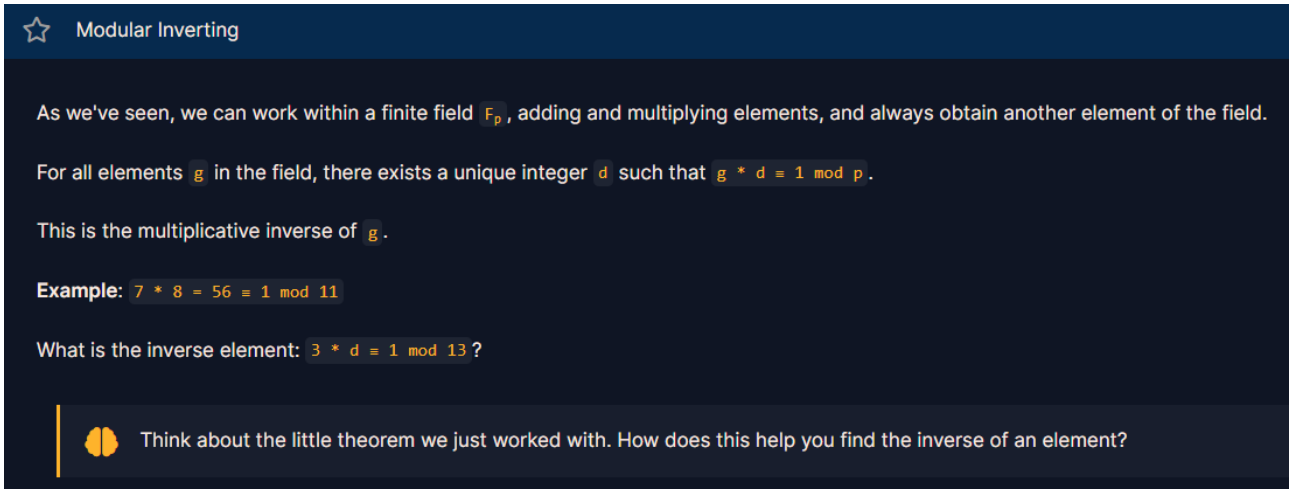
then the result or flag is '1':



Look at the result, those 2 numbers are coprime => the flag is 1.

So we just need to check if 2 numbers, 273246787654 and 65537, are coprime or not, this can be calculated by hand, don't need a calculator...

15.



My code and result:

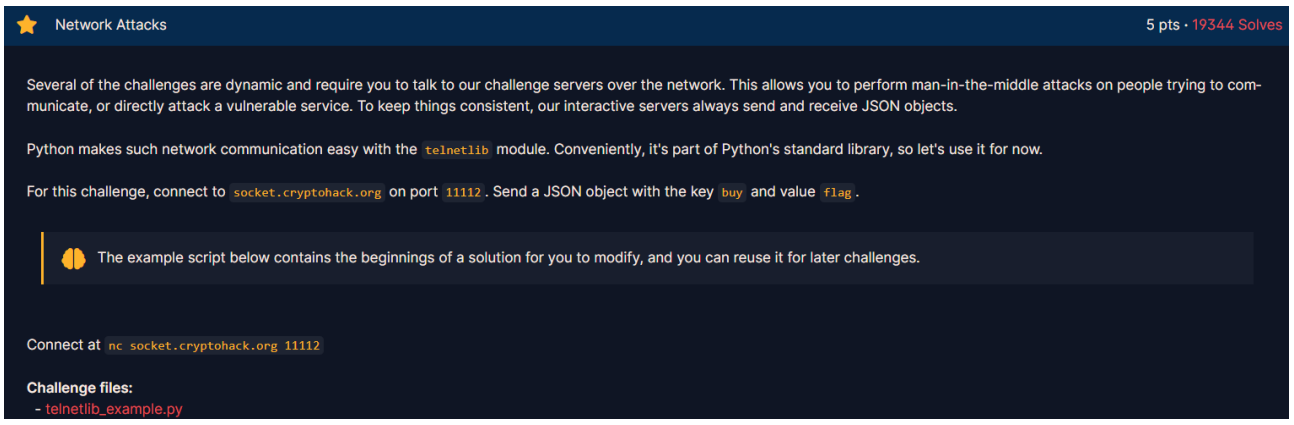
```python
def mod_inverse(a, m):
    # Calculate the modular inverse of a mod m using the extended Euclidean algorithm
    if gcd(a, m) != 1:
        raise ValueError("a and m are not coprime")
    x, y = extended_gcd(a, m)
    return x % m
```

```
1   mod_inverse(3, 13)
```

```
9
```

This code use some math like Extended Eucidean algorithm, you can find the explain in the internet… Any way, the flag is '9'.

16.



Challenge file code:

```python
#!/usr/bin/env python3

import telnetlib
import json

HOST = "socket.cryptohack.org"
PORT = 11112

tn = telnetlib.Telnet(HOST, PORT)


def readline():
    return tn.read_until(b"\n")

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

def json_send(hsh):
    request = json.dumps(hsh).encode()
    tn.write(request)


print(readline())
print(readline())
print(readline())
print(readline())


request = {
    "buy": "clothes"
}
json_send(request)

response = json_recv()

print(response)
```

When run:

```
b"Welcome to netcat's flag shop!\n"
b'What would you like to buy?\n'
b"I only speak JSON, I hope that's ok.\n"
b'\n'
{'error': 'Sorry! All we have to sell are flags.'}
```

Correct the code: change “clothes” to “flag” because:

Send a JSON object with the key `buy` and value `flag`.

```python
import telnetlib
import json

HOST = "socket.cryptohack.org"
PORT = 11112

tn = telnetlib.Telnet(HOST, PORT)


def readline():
    return tn.read_until(b"\n")

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

def json_send(hsh):
    request = json.dumps(hsh).encode()
    tn.write(request)
```

```python
print(readline())
print(readline())
print(readline())
print(readline())


request = {
    "buy": "flag"
}
json_send(request)

response = json_recv()

print(response)
```

```
b"Welcome to netcat's flag shop!\n"
b'What would you like to buy?\n'
b"I only speak JSON, I hope that's ok.\n"
b'\n'
{'flag': 'crypto{sh0pp1ng_f0r_fl4g5}'}
```

⇨ The flag: crypto{sh0pping_f0r_fl4g5}.

17.

Use a simple loop to find the quadratic residue **x** and all roots **a** respectively:

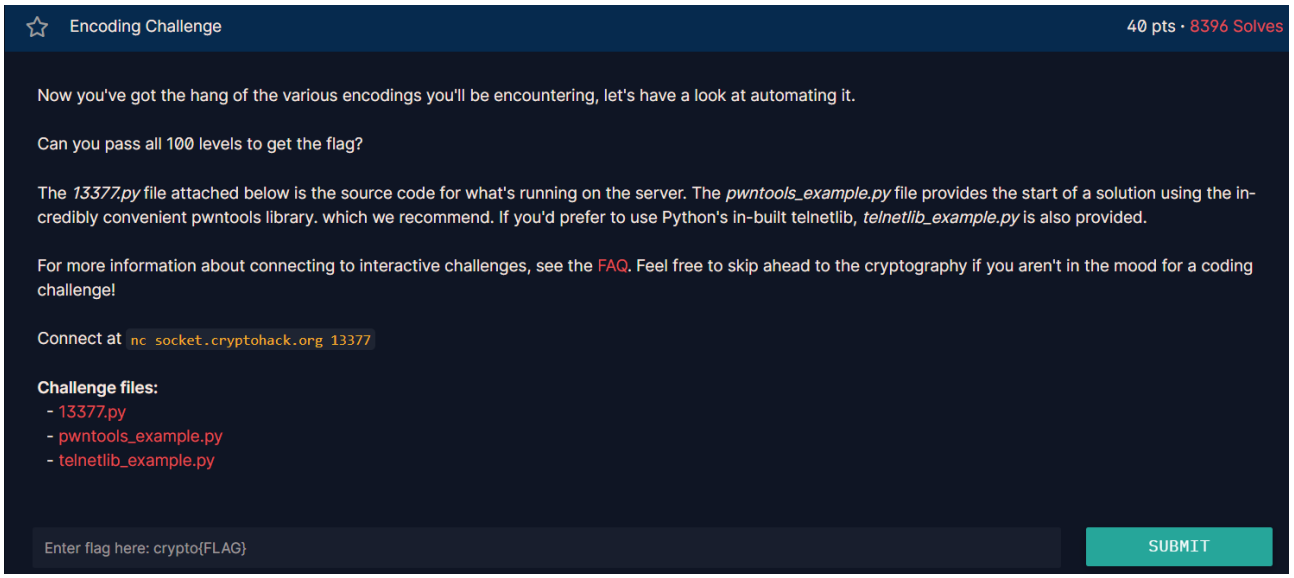
```python
p = 29
ints = [14, 6, 11]

for i in range(p):
    q = (i*i) % p
    if q in ints:
        print(i, q)
```

```
8 6
21 6
```

Look at the result, we can see the flag is '8', because it's the lower root of quadratic residue 6

⇨ Flag: '8'

18.



13377.py code: there is what's running on the server:

```python
#!/usr/bin/env python3

from Crypto.Util.number import bytes_to_long, long_to_bytes
from utils import listener # this is cryptohack's server-side module and not
part of python
import base64
import codecs
import random

FLAG = "crypto{????????????????????}"
ENCODINGS = [
    "base64",
    "hex",
    "rot13",
    "bigint",
    "utf-8",
]
with open('/usr/share/dict/words') as f:
    WORDS = [line.strip().replace("'", "") for line in f.readlines()]


class Challenge():
    def __init__(self):
        self.challenge_words = ""
        self.stage = 0

    def create_level(self):
        self.stage += 1
        self.challenge_words = "_".join(random.choices(WORDS, k=3))
        encoding = random.choice(ENCODINGS)

        if encoding == "base64":
            encoded = base64.b64encode(self.challenge_words.encode()).decode()
            # wow so encode
        elif encoding == "hex":
            encoded = self.challenge_words.encode().hex()
        elif encoding == "rot13":
            encoded = codecs.encode(self.challenge_words, 'rot_13')
```

```
    def create_level(self):
        self.stage += 1
        self.challenge_words = "_".join(random.choices(WORDS, k=3))
        encoding = random.choice(ENCODINGS)

        if encoding == "base64":
            encoded = base64.b64encode(self.challenge_words.encode()).decode()
            # wow so encode
        elif encoding == "hex":
            encoded = self.challenge_words.encode().hex()
        elif encoding == "rot13":
            encoded = codecs.encode(self.challenge_words, 'rot_13')
        elif encoding == "bigint":
            encoded = hex(bytes_to_long(self.challenge_words.encode()))
        elif encoding == "utf-8":
            encoded = [ord(b) for b in self.challenge_words]

        return {"type": encoding, "encoded": encoded}

    #
    # This challenge function is called on your input, which must be JSON
    # encoded
    #
    def challenge(self, your_input):
        if self.stage == 0:
            return self.create_level()
        elif self.stage == 100:
            self.exit = True
            return {"flag": FLAG}

        if self.challenge_words == your_input["decoded"]:
            return self.create_level()

        return {"error": "Decoding fail"}


listener.start_server(port=13377)
```

We can see at line 9 there is a flag: crypto{??????????????????}

I have submitted that flag but it's not the flag need to find, the flag may be (or should be) hidden in this code.

Continue, because I want to avoid installing many libraries on my computer which I may not use later, so I use tebnet code to start:

```
1   import telnetlib
2   import json
3
4   HOST = "socket.cryptohack.org"
5   PORT = 13377
6
7   tn = telnetlib.Telnet(HOST, PORT)
8
9   def readline():
10      return tn.read_until(b"\n")
11
12  def json_recv():
13      line = readline()
14      return json.loads(line.decode())
15
16  def json_send(hsh):
17      request = json.dumps(hsh).encode()
18      tn.write(request)
```

```
19
20  received = json_recv()
21
22  print("Received type: ")
23  print(received["type"])
24  print("Received encoded value: ")
25  print(received["encoded"])
26
27  to_send = {
28      "decoded": "changeme"
29  }
30  json_send(to_send)
31
32  json_recv()
33
```

Here is my code after addition and correction:

```python
import telnetlib
import json
import base64
import binascii
import codecs
import sys

HOST = "socket.cryptohack.org"
PORT = 13377

tn = telnetlib.Telnet(HOST, PORT)

def readline():
    return tn.read_until(b"\n")

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

def json_send(hsh):
    request = json.dumps(hsh).encode()
    tn.write(request)

def decode(t, data):
    if t == 'base64':
        return base64.b64decode(data).decode('utf-8')
    elif t == 'hex':
        return binascii.unhexlify(data).decode('utf-8')
    elif t == 'bigint':
        return binascii.unhexlify(data.replace('0x', '')).decode('utf-8')
    elif t == 'rot13':
        return codecs.encode(data, 'rot_13')
    elif t == 'utf-8':
        s = ""
        for c in data:
            s += chr(c)
        return s

while True:

    received = json_recv()

    if "flag" in received:
        print("FLAG: %s" % received["flag"])
        sys.exit(0)

    to_send = {
        "decoded": decode(received["type"], received["encoded"])
    }
    json_send(to_send)
```

```
FLAG: crypto{3nc0d3_d3c0d3_3nc0d3}
An exception has occurred, use %tb to see the full traceback.

SystemExit: 0
```

⇨ The flag: crypto{3nc0d3_d3c0d3_3nc0d3}

HẾT