Modular arithmetic

Modular square root soluions:

```
def legendre_symbol(a, p):
    return pow(a, (p - 1) // 2, p)

def tonelli_shanks(n, p):
    if legendre_symbol(n, p) != 1:
        return None  # No solution exists

    # Factorize p-1 as 2^s * q
    q = p - 1
    s = 0
    while q % 2 == 0:
        q //= 2
        s += 1

    # Find a non-residue (i.e., a number whose Legendre symbol is -1)
    z = 2
    while legendre_symbol(z, p) != p - 1:
        z += 1

    # Initialize variables
    m = s
    c = pow(z, q, p)
    t = pow(n, q, p)
    r = pow(n, (q + 1) // 2, p)

    while t != 1:
        # Find the smallest i such that t^(2^i) = 1
        i = 0
        tmp = t
        while tmp != 1:
            tmp = (tmp * tmp) % p
            i += 1

        # Update variables
        b = pow(c, 2 ** (m - i - 1), p)
        m = i
        c = (b * b) % p
        t = (t * b * b) % p
        r = (r * b) % p

    return r
```

```python
def modular_square_root(a, p):
    sqrt_a_mod_p = tonelli_shanks(a, p)
    if sqrt_a_mod_p is not None:
        # There are two square roots modulo p, return the smaller one
        return min(sqrt_a_mod_p, p - sqrt_a_mod_p)
    else:
        return None

# Example usage
a =
84799946583167721519416165100971270875545412748124351120094257785954953597
00244470400642403747058566807127814165396640215844192327900454116257979487
43201676932997076704673509124989867808806163479655955670495984642413182041
60484365013876172117701242927933080792141531799776244404386169585750583611
93975686620046439877308339989295604537867493683872778843921771307305602776
39878697835386623166145337605677197206977639899901376958893619485934494126
82231841972313688870606092128755075189361720607022095571244304771374218471
30682601666968691651447236917018634902407704797328509461854842432015009878
011354022108661461024768

p =
30531851861994333252675935111487950694414332763909083514133769861350960895
07650468726136981573574254942878913830084308208655005908283514145452661816
06341099691954863220157759430300604495570900648119401394317352091859964547
39163555910726493597222646855506445602953689527405362207926990442391705014
60477703868588052753748984535910155244229280439847264235660930481068073155
65420023015478466351014559957325840713559030108567186807323373691284986552
55277003643669031694516851390505923416710601212618443109844041514942401969
62915897545707902690630432874903999726296030120915817592005189062094706393
6347307238412281568760161

result = modular_square_root(a, p)
print(f"The modular square root of {a} modulo {p} is: {result}")
```

OUTPUT:

```
23623393076830486383277732985804892989321375055205003883382710520537347478623
51779647314176817953359071871560041125289919247146074907151612762640868199621
18655952206833803260099131188222401602122267224313936218046123264673246584884
04254582579308878565833796009677617385967828778513184893556798228131551230457
05285112099448146426755110160002515592418850432103641815811071548456284263507
80558944507365756538185052136796967569976075531078462357707644003774768176030
24349249321136400617387776011946222441927580241808539162444272540654419625572
82572849162772740798989647948645207349737457445440405057156897508368531939120
```

Chinese reminder theorem code as well the output:

```python
def chinese_remainder_theorem(moduli, remainders):
    """
    Chinese Remainder Theorem implementation
    :param moduli: List of pairwise coprime moduli
    :param remainders: List of remainders corresponding to the moduli
    :return: The solution modulo the product of moduli
    """
    def modinv(a, m):
        """
        Modular multiplicative inverse
        :param a: Integer
        :param m: Modulus
        :return: Modular inverse of a modulo m
        """
        m0, x0, x1 = m, 0, 1
        while a > 1:
            q = a // m
            m, a = a % m, m
            x0, x1 = x1 - q * x0, x0
        return x1 + m0 if x1 < 0 else x1

    product = 1
    for m in moduli:
        product *= m

    result = 0
    for mi, ai in zip(moduli, remainders):
        bi = product // mi
        result += ai * modinv(bi, mi) * bi

    return result % product

# Example usage
moduli = [5, 11, 17]
remainders = [2, 3, 5]

result = chinese_remainder_theorem(moduli, remainders)
print("The solution a is:", result)
```

```
The solution a is: 872
```

Adrien's signs:

```
from sympy.ntheory import legendre_symbol
p = 1007621497415251
c = [67594220461269, 501237540280788, 718316769824518, 296304224247167,
48290626940198, 30829701196032, 521453693392074, 840985324383794,
770420008897119, 745131486581197, 729163531979577, 334563813238599,
289746215495432, 538664937794468, 894085795317163, 983410189487558,
863330928724430, 996272871140947, 352175210511707, 306237700811584,
631393408838583, 589243747914057, 538776819034934, 365364592128161,
454970171810424, 986711310037393, 657756453404881, 388329936724352,
90991447679370, 714742162831112, 62293519842555, 653941126489711,
448552658212336, 970169071154259, 339472870407614, 406225588145372,
205721593331090, 926225022409823, 904451547059845, 789074084078342,
886420071481685, 796827329208633, 433047156347276, 21271315846750,
719248860593631, 534059295222748, 879864647580512, 918055794962142,
635545050939893, 319549343320339, 93008646178282, 926080110625306,
385476640825005, 483740420173050, 866208659796189, 883359067574584,
913405110264883, 898864873510337, 208598541987988, 23412800024088,
911541450703474, 57446699305445, 513296484586451, 180356843554043,
756391301483653, 823695939808936, 452898981558365, 383286682802447,
381394258915860, 385482809649632, 357950424436020, 212891024562585,
906036654538589, 706766032862393, 500658491083279, 134746243085697,
240386541491998, 850341345692155, 826490944132718, 329513332018620,
41046816597282, 396581286424992, 488863267297267, 92023040998362,
529684488438507, 925328511390026, 524897846090435, 413156582909097,
840524616502482, 325719016994120, 402494835113608, 145033960690364,
43932113323388, 683561775499473, 434510534220939, 92584300328516,
763767269974656, 289837041593468, 11468527450938, 628247946152943,
8844724571683, 813851806959975, 72001988637120, 875394575395153,
70667866716476, 75304931994100, 226809172374264, 767059176444181,
45462007920789, 472607315695803, 325973946551448, 64200767729194,
534886246409921, 950408390792175, 492288777130394, 226746605380806,
944479111810431, 776057001143579, 658971626589122, 231918349590349,
699710172246548, 122457405264610, 643115611310737, 999072890586878,
203230862786955, 348112034218733, 240143417330886, 927148962961842,
661569511006072, 190334725550806, 763365444730995, 516228913786395,
846501182194443, 741210200995504, 511935604454925, 687689993302203,
631038090127480, 961606522916414, 138550017953034, 932105540686829,
215285284639233, 772628158955819, 496858298527292, 730971468815108,
896733219370353, 967083685727881, 607660822695530, 650953466617730,
```

133773994258132, 623283311953090, 436380836970128, 237114930094468,
115451711811481, 674593269112948, 140400921371770, 659335660634071,
536749311958781, 854645598266824, 303305169095255, 91430489108219,
573739385205188, 400604977158702, 728593782212529, 807432219147040,
893541884126828, 183964371201281, 422680633277230, 218817645778789,
313025293025224, 657253930848472, 747562211812373, 83456701182914,
470417289614736, 641146659305859, 468130225316006, 46960547227850,
875638267674897, 662661765336441, 186533085001285, 743250648436106,
451414956181714, 527954145201673, 922589993405001, 242119479617901,
865476357142231, 988987578447349, 430198555146088, 477890180119931,
844464003254807, 503374203275928, 775374254241792, 346653210679737,
789242808338116, 48503976498612, 604300186163323, 475930096252359,
860836853339514, 994513691290102, 591343659366796, 944852018048514,
82396968629164, 152776642436549, 916070996204621, 305574094667054,
981194179562189, 126174175810273, 55636640522694, 44670495393401,
74724541586529, 988608465654705, 870533906709633, 374564052429787,
486493568142979, 469485372072295, 221153171135022, 289713227465073,
952450431038075, 107298466441025, 938262809228861, 253919870663003,
835790485199226, 655456538877798, 595464842927075, 191621819564547]
print(bytes.fromhex(hex(int(''.join(['1' if legendre_symbol(i,p)==1 else '0' for i in c]), 2))[2:]).decode())

Solution:

```
crypto{p4tterns_1n_re5idu3s}
```

Modular Binomials:

from math import gcd

n =
1490556225784271405793272412957500282540539350265086976711594260640860034338032786625898240244799256498846658830517427167465784435245454395884
7568190372446723549627752274442789184236490768272313187410077124234699854724907039770193680822495470532218905083459730998003622926152590597710
2131279521410560295161167852295046451798300379372220222915717389736039206649291504364636323056646879032449728800620283010857494346881599057680
5204120751314937021231394311766591480237915861335904995768856388539197215121867654597211849496924744048976343135967977042293944171078357566867
969367843566954178149021773161922447015246776807 3

e1 =
12886657667389660800780796462970504910193928992888518978200029826975978624718627799215564700096007849924866627154987365059524315097631111242449314835868137
e2 =
121105866739917884157803551396355790579209268648871103083432292560468682421794454448977901713513025751886071170815801214882535402157816255980480211616756 97
c1 =
1401072941870322823435246588304127061111373588983875343329547849576340905613673415561215693467398834488262954120498590965043381920529893987783
7314145082403528055884752079219150739849992921393509593620449489882380176216648401057401569934043087087362272303101549800941212057354903559653
3732991534307538820352333543047832759823329957667784994255295700080080294013256683011441889704809755652159539539850782813955459021022457558626
6362118743867759662810996706641899385163254313735304171272191929152176726267814011518873599444794916661610118280682074192829288264223423845020
747291423259674775526132509822596826892658099305 1
c2 =
1438699713863797886074827898694509864850714286458411112420258036510379316581166698766485121023000937526739895797949406688029641801334500697765
4742303441030008490816239306394492168516278328851513359596253775965916326353050138738183351643338294802012193721879700283088378587949921991198
2319568714298058477677161378173136123048337339186578874804687244097535223693251385020594082412321556338064967523505622847947153218352269911475
4765115528781248586279493569524161267625537448013272294068214039572508 9

32944535643448938483103620538729376078997661521031043673281384893766660
8611803196199865435145094486231635966885932646519

```
q1 = pow(c1, e2, n)
q2 = pow(c2, e1, n)
d = pow(5, e1 * e2, n) * q1 - pow(2, e1 * e2, n) * q2
q = gcd(d, n)
p = n // q
print("crypto{%d,%d}" % (p,q))
```

Solutions:

```
crypto{1122740001692584863902620644419912006085563761274089527015149626443409
21899196091557519382763356534106376906489445103255177593594898966250176773605
43276598389710504779561947065915705709377140730916834567054141877242780714803
92074899008100137836739579840062691206521340076892724845178053983902773080017
19431273,13276058780636530197147915707203144838013576579446678745694878673116
80958779568752952826615654882421907315932826636947289149459672531730473243539
81530949360031535707374701705328450856944598803228299967009004598984671293494
37559940876413974321746501277037672887654795885202542553929841075113278263281
7947101601}
```

Keyed Permutations:
Solution:
crypto{bijection}

Resisting Brute-force:
crypto{biclique}

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):
    for i in range(4):
```

```
        for j in range(4):
            print(chr(s[i][j]^k[i][j]), end="")

print(add_round_key(state, round_key))
```

Solution: 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):
    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))
```

flag: crypto{l1n34rly}

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]

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

# learned from http://cs.ucsb.edu/~koc/cs178/projects/JT/aes.c
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)


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)
print(bytes(sum(state, [])))
```

Solution: crypto{d1ffUs3R}

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}

Mode of operation starter:

```
import requests

# request encrypted flag
r =
requests.get('http://aes.cryptohack.org/block_cipher_starter/encrypt_flag/
')
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_!}

Password as Keys:

```
from Crypto.Cipher import AES
import codecs
import random
import hashlib

def hex_to_string(hex):
    if hex[:2] == '0x':
        hex = hex[2:]
    string_value = bytes.fromhex(hex).decode('utf-8')
    return string_value

# /usr/share/dict/words from
#
https://gist.githubusercontent.com/wchargin/8927565/raw/d9783627c731268fb2935
a731a618aa8e95cf465/words
with open('C:/Users/Aayush/Documents/file.txt') as f:
    words = [w.strip() for w in f.readlines()]
keyword = random.choice(words)

KEY = hashlib.md5(keyword.encode()).digest()
FLAG = '73616468616264207163717771646277 6b7164647177646e6b736e646b717769'


# @chal.route('/passwords_as_keys/decrypt/<ciphertext>/<password_hash>/')
def decrypt(ciphertext, password_hash):
    ciphertext = bytes.fromhex(ciphertext)
    key = bytes.fromhex(password_hash)

    cipher = AES.new(key, AES.MODE_ECB)
    try:
        decrypted = cipher.decrypt(ciphertext)
    except ValueError as e:
        return {"error": str(e)}

    return {"plaintext": decrypted.hex()}


# @chal.route('/passwords_as_keys/encrypt_flag/')
def encrypt_flag():
    cipher = AES.new(KEY, AES.MODE_ECB)
    encrypted = cipher.encrypt(FLAG.encode())

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

print(encrypt_flag())
final="gahnta"
for word in words:
    opr =
decrypt('c92b7734070205bdf6c0087a751466ec13ae15e6f1bcdd3f3a535ec0f4bbae66',ha
shlib.md5(word.encode()).digest().hex())
    try:
```

```
        final = hex_to_string(opr['plaintext'])
    except:
        xx=1

print(final)
{'ciphertext':
'c7710e2a035ae4f9a8d61de21363bf84c343c6ad2ceb06ccd4d760e14f6f87c51b47b911193f
afb72864104b52626af50ba978944ffe55780de011bf9bfa690e'}

crypto{k3y5__r__n07__p455w0rdz?}
```

ECB Oracle:

```
def encrypt(plaintext):
    plaintext = bytes.fromhex(plaintext)

    padded = pad(plaintext + FLAG.encode(), 16)
    cipher = AES.new(KEY, AES.MODE_ECB)

    try:
        encrypted = cipher.encrypt(padded)
    except ValueError as e:
        return {"error": str(e)}

    return {"ciphertext": encrypted.hex()}
print('  ', end='')
print_blk(encrypt(bytes.hex(b'1'*15)), 32)

for i in range(ord('a'),ord('z')):
    print(chr(i), '', end='')
    print_blk(encrypt(bytes.hex(b'1'*15+int.to_bytes(i, 1, 'little'))), 32)
def bruteforce():
    flag = ''
    total = 32 - 1
    alphabet =
'_'+'@'+'}'+string.digits+string.ascii_lowercase+string.ascii_uppercase

    while True:
        payload = '1' * (total-len(flag))
        expected = encrypt(payload.encode().hex())
        print('E', '', end='')
        print_blk(expected, 32)

        for c in alphabet:
            res = encrypt(bytes.hex((payload + flag + c).encode()))
            print(c, '', end='')
            print_blk(res, 32)
            if res[32:64] == expected[32:64]:
                flag += c
                print(flag)
                break
            time.sleep(1)

        if flag.endswith('}'): break
```

```
    print(flag)
crypto{p3n6u1n5_h473_3cb}
```

ECB CBC WTF:

CODE:

```
from pwn import *
import json
import base64
import binascii
import codecs
import sys

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

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)

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{3cb_5uck5_4v01d_17_!!!!!}

Flipping Cookie:

```
def check_admin(cookie, iv):
    cookie = bytes.fromhex(cookie)
    iv = bytes.fromhex(iv)


    try:
        cipher = AES.new(KEY, AES.MODE_CBC, iv)
        decrypted = cipher.decrypt(cookie)
        unpadded = unpad(decrypted, 16)
    except ValueError as e:
        return {"error": str(e)}


    if b"admin=True" in unpadded.split(b";"):
        return {"flag": FLAG}
    else:
```

```
        return {"error": "Only admin can read the flag"}


def get_cookie():
    expires_at = (datetime.today() + timedelta(days=1)).strftime("%s")
    cookie = f"admin=False;expiry={expires_at}".encode()

    iv = os.urandom(16)
    padded = pad(cookie, 16)
    cipher = AES.new(KEY, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(padded)
    ciphertext = iv.hex() + encrypted.hex()

    return {"cookie": ciphertext}
p1  = c0 ^ d(c1)
=> d(c1) = p1 ^ c0
p1' = c0' ^ d(c1)
    = c0' ^ p1 ^ c0
cipher ^ iv = plain
cipher = plain ^ iv
fake = cipher ^ iv'
=> iv' = fake ^ cipher = fake ^ plain ^ iv
def flip(cookie, plain):
    start = plain.find(b'admin=False')
    cookie = bytes.fromhex(cookie)
    iv = [0xff]*16
    cipher_fake = list(cookie)
    fake = b';admin=True;'
    for i in range(len(fake)):
        cipher_fake[16+i] = plain[16+i] ^ cookie[16+i] ^ fake[i]
        iv[start+i] = plain[start+i] ^ cookie[start+i] ^ fake[i]
```

```
    cipher_fake = bytes(cipher_fake).hex()
    iv = bytes(iv).hex()
    return cipher_fake, iv
expires_at = (datetime.today() + timedelta(days=1)).strftime("%s")
plain = f"admin=False;expiry={expires_at}".encode()
cookie = request_cookie()
cookie, iv = flip(cookie, plain)
print(request_check_admin(cookie, iv))
```

flag: crypto{4u7h3n71c4710n_15_3553n714l}

Diffie Hellman Starter 1

```
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

SALTY:

salty

```
from Crypto.Util.number import inverse, long_to_bytes n =
1105817957159585662066003921613602125796696373914370977036851542370173515704647677253241820511999019203182112904047772597289236149172112915625558647530051793261018904276698198346420079244068624 82343
```

61448876825695161608628704472503441280217631227308132219586604609859530626178178827657092046784017200453087 3767

e = 1 ct =
449812307182121836042747859257931454426554650252645540460282513111
64494127485

print(long_to_bytes(ct))

crypto{saltstack_fell_for_this!}

Curves and logs:

Code:

```
import math
import hashlib
from Crypto.Util import number


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

"""
A  QA = nA*G
B  QB = nB*G
A  S = nA*QB
B  S = nB*QA
"""
# E: Y2 = X3 + 497 X + 1768, p: 9739, G: (1804,5368)
a = 497
b = 1768
p = 9739
```

```
# QA = (815, 3190), with your secret integer nB = 1829.
nB = 1829
QA = (815, 3190)
S = scalar_mul(QA, nB, a, p)
print(S)
sha1 = hashlib.sha1()
sha1.update(str(S[0]).encode())
print(sha1.hexdigest())
```

Flag: `crypto{80e5212754a824d3a4aed185ace4f9cac0f908bf}`

Bean Counter:

Code:

```
class StepUpCounter(object):
    def __init__(self, value=os.urandom(16), step_up=False):
        self.value = value.hex()
        self.step = 1
        self.stup = step_up

    def increment(self):
        if self.stup:
            self.newIV = hex(int(self.value, 16) + self.step)
        else:
            self.newIV = hex(int(self.value, 16) - self.stup)
        self.value = self.newIV[2:len(self.newIV)]
        return bytes.fromhex(self.value.zfill(32))

    def __repr__(self):
        self.increment()
        return self.value

def encrypt():
    cipher = AES.new(KEY, AES.MODE_ECB)
```

```
    ctr = StepUpCounter()

    out = []
    with open("challenge_files/bean_flag.png", 'rb') as f:
        block = f.read(16)
        while block:
            keystream = cipher.encrypt(ctr.increment())
            xored = [a^b for a, b in zip(block, keystream)]
            out.append(bytes(xored).hex())
            block = f.read(16)

    return {"encrypted": ''.join(out)}
```

Flag: *crypto{hex_bytes_beans}.*

Stream of Consciousness:

Code:

```
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()
```

Flag: **crypto{k3y57r34m_r3u53_15_f474l}**

RSA Starter 1:

Code :

```
result = pow(101, 17, 22663)
print(result)
```

Output: 19906

RSA Starter 2:

```
message = 12
exponent = 65537
p = 17
q = 23

modulus = p * q
ciphertext = pow(message, exponent, modulus)

print(ciphertext)
```

OUTPUT:

19906

RSA STARTER 2:

Code:
```
message = 12
exponent = 65537
p = 17
q = 23

modulus = p * q
ciphertext = pow(message, exponent, modulus)
```

print(ciphertext)

OUTPUT: 301

Ron was wrong whit is right:

```
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

Modular Binomials:

```
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)
```

FLAG: 702

SIZE AND BASICS:

```
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)
```

FLAG: 9.0

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)
```

FLAG: 7410790865146821

```
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
a = 1
b = 4
```

```
E = EllipticCurve(GF(p), [a,b])

G =
E(4319096045221802357578789921402301493892663179265163804468016860098960906920
0,
20971936269255296908588589778128791635639992476076894152303569022736123671173)

P_A =
E.lift_x(ZZ(87360200456784002948566700858113190957688355783112995047798140117594
305287669))

P_B =
E.lift_x(ZZ(60828963734991266240293432937501384601375317744734503412352176994976
02895121))


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))
```

crypto{50m3_p30pl3_d0n7_7h1nk_IV_15_1mp0r74n7_?}

lazy cbc

```
@chal.route('/lazy_cbc/encrypt/<plaintext>/')
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"}
```