Basic_Number_Theory_Without_Images

August 24, 2022

0.1 Elementary Number Theory

[1]:
```
def binaryPow(a, b, mod):
    res = 1
    while b > 0:
        if b & 1:
            res = (res * a) % mod
        a = (a * a) % mod
        b >>= 1
    return res
print(binaryPow(2, 300, 1000000000000))
```

```
706183397376
```

[2]:
```
print(binaryPow(103232,1002032003, 2e17) * binaryPow(1032,100000034, 2e17) %␣
↪2e17)
```

```
1.349003109770199e+17
```

[3]:
```
import math
def leastCommonMultiple(a, b):
    return a / math.gcd(a, b) * b
```

[4]:
```
print(leastCommonMultiple(100, 20))
```

```
100.0
```

0.2 Extended GCD

- Extended Euclidean Algorithm
  - $a * x + b * y = gcd(a, b)$
  - Going backwards: let us assume we found the coefficients (x_1, y_1) for $(b, a * mod * b)$
    - $b * x_1 + (a * mod * b) * y_1 = g$
    - $a * x + b * y = g$
    - $a * mod * b = a - \frac{a}{b} * b$
    - After rearranging: $g = a * y_1 + b(x_1 - y_1 * \frac{a}{b})$
    - $\begin{cases} x = y_1 \\ y = x_1 - y_1 * \frac{a}{b} \end{cases}$

[5]:
```python
def extendedGCD(a, b):
    x,y, u,v = 0,1, 1,0

    while a:
        q, r = b // a, b % a
        m, n = x - u * q, y - v * q
        b,a, x,y, u,v = a,r, u,v, m,n

    gcd = b
    return gcd, x, y
```

0.3 Modular Arithmetic

- $(\frac{a}{b})\%c = ((a\%c) * (b^{-1}\%c))\%c$
- b^{-1} is the multiplicative modulo inverse of b
- Modular exponentiation -> $x^n = x * x * x ... * x \ (n \ times)$
  - Break down with Binary Exponentiation -> Time Complexity $O(log \ N)$
    * If n is even, replace x^n with $(x^2)^{\frac{n}{2}}$
    * If n is odd, replace x^n with $x * x^{n-1}$. $n-1$ becomes even
    * Use modularExponentiation for faster runtimes -> Time Complexity and Space Complexity $O(log \ N)$
- Modular multiplicative inverse
  - If $A, B \ = 1$ then B is $\frac{1}{A} \ or \ A^{-1}$
  - If $(A, B) \ \% \ M = 1$ then B is said to be modular multiplicative inverse of A under modulo M if it satisfies the following equation: $A, B \ = \ 1 * (mod \ M)$ where B is in the range [1, M-1]
  - An inverse exists only when A and M are coprime: $GCD(A, M) \ = \ 1$
  - Two approaches
    * A and M are coprime, $Ax \ + \ My \ = 1$. In the extended Euclidean algorithm, x is the modular multiplicative inverse of A under modulo M. Therefore, the answer is x.
    * Used only when M is prime, $A^{M-1} \ = \ 1(mod \ M)$. Mulitply both sides by A^{-1} and rearrange: $A^{-1} \ = A^{M-2}(mod \ M)$

[6]:
```python
# Binary Exponentiation

def binaryExponentiationR(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return binaryExponentiationR(x * x, n / 2)
    else:
        return x * binaryExponentiationR(x * x,  (n-1)/2)

def binaryExponentiationI(x, n):
    result = 1
```

```
    while n:
        if n % 2 == 1:
            result *= x
        x *= x
        n /= 2

    return result

def modularExponentiation(x, n, m):
    result = 1

    while n:
        if n % 2 == 1:
            result = (result * x) % m
        x = (x * x) % m
        n /= 2

    return result

# Mod Inverse when A and M are coprime
def modInverseCoPrime(a, m):
    d, x, y = extendedGCD(a, m)

    return (x % m + m) % m

# Mod Inverse only when M is prime
def modInverseM_Prime(a, m):
    return modularExponentiation(a, m-2, m)

print(modInverseCoPrime(5, 12))
print(modInverseM_Prime(5, 11))

print(modInverseCoPrime(3, 13))
```

```
5
5
9
```

0.4 Primes

- Composite numbers are also numbers that are greater than 1 but they have at least one more divisor other than 1 and itself.
- Determine if prime using Sieve of Eratosthenes for fast factorization -> $O(N * loglogN)$, factorization in $\sqrt{N}$
- Fermat's Little Theorem
  - If p is a prime number, then for any integer a, the number $a^p - p$ is an integer multiple of p
  - $a^p = a \ mod \ p$ -> $a^p \% p = a$

  - If a is not divisible by p then $a^{p-1} - 1$ is an integer multiple of p: $a^{p-1} = 1\ mod\ p$ -> $a^{p-1} \% p = 1$

- Pingala's Exponentiation Algo: begin with the desired exponent (e.g. e=90), and carry out a series of steps: replace e by e/2 if e is even, and replace e by (e−1)/2 if e is odd. Repeat this until the exponent is decreased to zero.
- Miller-Rabin test. We carry out Pingala's exponentiation algorithm to compute $b^{p-1}\ mod\ p$. If we find a violation of ROO along the way, then the test number p is not prime. And if, at the end, the computation does not yield 1, we have found a Fermat's Little Theorem (FLT) violation, and the test number p is not prime.
  - Conditional probability: $Prob(tests\ prime \mid is\ composite) < \frac{1}{4^{\#witnesses}}$

[7]:
```
def fermat_theorem_mod(base, exp, prime):
    if exp == prime:
        return base
    elif exp + 1 == prime and base % prime != 0:
        return 1
    else:
        return -1;

print(fermat_theorem_mod(273246787654, 65536, 65537))
```

```
1
```

[8]:
```
def Pingala(e):
    current_number = e
    while current_number > 0:
        if current_number%2 == 0:
            current_number = current_number // 2
            print("Exponent {} BIT 0".format(current_number))
        if current_number%2 == 1:
            current_number = (current_number - 1) // 2
            print("Exponent {} BIT 1".format(current_number))

Pingala(90)
```

```
Exponent 45 BIT 0
Exponent 22 BIT 1
Exponent 11 BIT 0
Exponent 5 BIT 1
Exponent 2 BIT 1
Exponent 1 BIT 0
Exponent 0 BIT 1
```

[9]:
```
def pow_Pingala(base,exponent):
    result = 1
    bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary␣
 ↪expansion of exponent
```

```
    for bit in bitstring: # Iterates through the "letters" of the string.  Here␣
↪the letters are '0' or '1'.
        if bit == '0':
            result = result*result
        if bit == '1':
            result = result*result * base
    return result

pow_Pingala(5, 90)
```

[9]: 807793566946316088741610050849573099185363389551639556884765625

[10]:
```
def powmod_Pingala(base,exponent,modulus):
    result = 1
    bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary␣
↪expansion of exponent
    for bit in bitstring: # Iterates through the "letters" of the string.  Here␣
↪the letters are '0' or '1'.
        if bit == '0':
            result = (result*result) % modulus
        if bit == '1':
            result = (result*result * base) % modulus
    return result

powmod_Pingala(5, 90, 91)
```

[10]: 64

[11]:
```
def Miller_Rabin(p, base):
    '''
    Tests whether p is prime, using the given base.
    The result False implies that p is definitely not prime.
    The result True implies that p **might** be prime.
    It is not a perfect test!
    '''
    result = 1
    exponent = p-1
    modulus = p
    bitstring = bin(exponent)[2:] # Chop off the '0b' part of the binary␣
↪expansion of exponent
    for bit in bitstring: # Iterates through the "letters" of the string.  Here␣
↪the letters are '0' or '1'.
        sq_result = result*result % modulus  # We need to compute this in any␣
↪case.
        if sq_result == 1:
            if (result != 1) and (result != exponent):  # Note that exponent is␣
↪congruent to -1, mod p.
```

```
                return False  # a ROO violation occurred, so p is not prime
        if bit == '0':
            result = sq_result
        if bit == '1':
            result = (sq_result * base) % modulus
    if result != 1:
        return False  # a FLT violation occurred, so p is not prime.

    return True  # If we made it this far, no violation occurred and p might be prime.
```

[12]:

```
# Let's see how many witnesses observe the nonprimality of 41041.
for witness in range(2,20):
    MR = Miller_Rabin(41041, witness) #
    if MR:
        print("{} is a bad witness.".format(witness))
    else:
        print("{} detects that 41041 is not prime.".format(witness))
```

```
2 detects that 41041 is not prime.
3 detects that 41041 is not prime.
4 detects that 41041 is not prime.
5 detects that 41041 is not prime.
6 detects that 41041 is not prime.
7 detects that 41041 is not prime.
8 detects that 41041 is not prime.
9 detects that 41041 is not prime.
10 detects that 41041 is not prime.
11 detects that 41041 is not prime.
12 detects that 41041 is not prime.
13 detects that 41041 is not prime.
14 detects that 41041 is not prime.
15 detects that 41041 is not prime.
16 is a bad witness.
17 detects that 41041 is not prime.
18 detects that 41041 is not prime.
19 detects that 41041 is not prime.
```

[13]:

```
def sieve(N):
    isPrime = [True if idx >= 2 else False for idx in range(N+1)]

    idx = 2

    while idx ** 2 <= N:
        if isPrime[idx]:
            idx_j = idx ** 2
```

```
            while idx_j <= N:
                isPrime[idx_j] = False
                idx_j += idx

        idx += 1

    return isPrime

print(sieve(10))
```

[False, False, True, True, False, True, False, True, False, False, False]

[14]:
```
def factorize(n):
    res = []
    idx = 2

    while idx ** 2 <= n:
        while (n % idx == 0):
            res.append(idx)
            n //= idx
        idx += 1

    if n != 1:
        res.append(n)

    return res

print(factorize(50))
```

[2, 5, 5]

[15]:
```
import math as mt

MAXN = 100001

# stores smallest prime factor for
# every number
spf = [0 for i in range(MAXN)]

# Calculating SPF (Smallest Prime Factor)
# for every number till MAXN.
# Time Complexity : O(nloglogn)
def sieve():
    spf[1] = 1
    for i in range(2, MAXN):

        # marking smallest prime factor
```

```
        # for every number to be itself.
        spf[i] = i

    # separately marking spf for
    # every even number as 2
    for i in range(4, MAXN, 2):
        spf[i] = 2

    for i in range(3, mt.ceil(mt.sqrt(MAXN))):

        # checking if i is prime
        if (spf[i] == i):

            # marking SPF for all numbers
            # divisible by i
            for j in range(i * i, MAXN, i):

                # marking spf[j] if it is
                # not previously marked
                if (spf[j] == j):
                    spf[j] = i

# A O(log n) function returning prime
# factorization by dividing by smallest
# prime factor at every step
def getFactorization(x):
    ret = list()
    while (x != 1):
        ret.append(spf[x])
        x = x // spf[x]

    return ret

# Driver code

# precalculating Smallest Prime Factor
sieve()
x = 50
print("prime factorization for", x, ": ",
                                end = "")

# calling getFactorization function
p = getFactorization(x)

for i in range(len(p)):
    print(p[i], end = " ")
```

```
prime factorization for 50 : 2 5 5
```

0.5 Euler's Theorem and Modular Roots

- If m is a positive integer and $GCD(a, m) = 1$ then $a^{\phi(m)} = 1 \ mod \ m$
  - $\phi(m)$ denotes the totient of m, which is the number of elements of {1,...,m} which are coprime to m
- The totient is a multiplicative function, meaning that if $GCD(a, b) = 1$ then $\phi(ab) = \phi(a)\phi(b)$. Therefore, the totient of number can be found quickly from the totient of the prime powers within its decomposition
- Totient of a prime power is: $\phi(p^e) = p^e - p^{e-1} = p^{e-1}(p - 1)$

[16]:
```python
def GCD(a,b):
    while b:    # Recall that != means "not equal to".
        a, b = b, a % b
    return abs(a)

def totient(m):
    tot = 0 # The running total.
    j = 0
    while j < m:  # We go up to m, because the totient of 1 is 1 by convention.
        j = j + 1  # Last step of while loop:  j = m-1, and then j = j+1, so j␣
↪= m.
        if GCD(j,m) == 1:
            tot = tot + 1
    return tot

print(totient(17))  # The totient of a prime p should be p-1.
print(totient(1000))
print(17**totient(1000) % 1000) # Let's demonstrate Euler's theorem.  Note␣
↪GCD(17,1000) = 1.
```

```
16
400
1
```

[17]:
```python
from math import sqrt  # We'll want to use the square root.

def smallest_factor(n):
    '''
    Gives the smallest prime factor of n.
    '''
    if n < 2:
        return None # No prime factors!

    test_factor = 2 # The smallest possible prime factor.
    max_factor = sqrt(n) # we don't have to search past sqrt(n).
```

```python
    while test_factor <= max_factor:
        if n%test_factor == 0:
            return test_factor
        test_factor = test_factor + 1  # This could be sped up.

    return n  # If we didn't find a factor up to sqrt(n), n itself is prime!

def decompose(N):
    '''
    Gives the unique prime decomposition of a positive integer N,
    as a dictionary with primes as keys and exponents as values.
    '''
    current_number = N  # We'll divide out factors from current_number until we get 1.
    decomp = {} # An empty dictionary to start.
    while current_number > 1:
        p = smallest_factor(current_number) # The smallest prime factor of the current number.
        if p in decomp.keys():  # Is p already in the list of keys?
            decomp[p] = decomp[p] + 1 # Increase the exponent (value with key p) by 1.
        else:  # "else" here means "if p is not in decomp.keys()".
            decomp[p] = 1  # Creates a new entry in the dictionary, with key p and value 1.
        current_number = current_number // p # Factor out p.
    return decomp

def mult_function(f_pp):
    '''
    When a function f_pp(p,e) of two arguments is input,
    this outputs a multiplicative function obtained from f_pp
    via prime decomposition.
    '''
    def f(n):
        D = decompose(n)
        result = 1
        for p in D:
            result = result * f_pp(p, D[p])
        return result

    return f

def totient_pp(p,e):
    return (p**(e-1)) * (p-1)

totient = mult_function(totient_pp)
totient(143564423)
```

[17]: 142968480

0.6 Euler's Totient Function

- Euler's Totient function Φ (n) for an input n is the count of numbers in {1, 2, 3, ..., n-1} that are relatively prime to n, i.e., the numbers whose GCD (Greatest Common Divisor) with n is 1
- $\phi(n) = n * \Pi_{p\ prime\ p|n}(1 - \frac{1}{p})$
- Observations
  - The sum of all values of Totient Function of all divisors of N is equal to N
  - The value of Totient function for a certain prime P will always be $P - 1$ as the number P will always have a GCD of 1 with all numbers less than or equal to it except itself.
  - For 2 number A and B, if $GCD(A, B)\ ==\ 1$ then $Totient(A) * Totient(B) = Totient(A * B)$
- If f, then e is a multiplicative inverse of f modulo $\phi(m)$. It exists only if $GCD(e, \phi(m)) = 1$.
  - $ex = 1\ mod\ m$ is equivalent to solving the linear Diophantine equation $ex + my = 1$

[18]:
```python
def phi(n):

    # Initialize result as n
    result = n;

    # Consider all prime factors
    # of n and subtract their
    # multiples from result
    p = 2;
    while(p * p <= n):

        # Check if p is a
        # prime factor.
        if (n % p == 0):

            # If yes, then
            # update n and result
            while (n % p == 0):
                n = int(n / p);
            result -= int(result / p);
        p += 1;

    # If n has a prime factor
    # greater than sqrt(n)
    # (There can be at-most
    # one such prime factor)
    if (n > 1):
        result -= int(result / n);
    return result;
```

```
# Driver Code
for n in range(1, 11):
    print("phi(",n,") =", phi(n));
print(f"phi({1023}) = {phi(1023)}")
```

```
phi( 1 ) = 1
phi( 2 ) = 1
phi( 3 ) = 2
phi( 4 ) = 2
phi( 5 ) = 4
phi( 6 ) = 2
phi( 7 ) = 6
phi( 8 ) = 4
phi( 9 ) = 6
phi( 10 ) = 4
phi(1023) = 600
```

[19]:
```
def mult_inverse(a,m):
    '''
    Finds the multiplicative inverse of a, mod m.
    If GCD(a,m) = 1, this is returned via its natural representative.
    Otherwise, None is returned.
    '''
    u = a # We use u instead of dividend.
    v = m # We use v instead of divisor.
    u_hops, u_skips = 1,0 # u is built from one hop (a) and no skips.
    v_hops, v_skips = 0,1 # v is built from no hops and one skip (b).
    while v != 0:   # We could just write while v:
        q = u // v  # q stands for quotient.
        r = u % v  # r stands for remainder.  So u = q(v) + r.

        r_hops = u_hops - q * v_hops  # Tally hops
        r_skips = u_skips - q * v_skips  # Tally skips

        u,v = v,r  # The new dividend,divisor is the old divisor,remainder.
        u_hops, v_hops = v_hops, r_hops # The new u_hops, v_hops is the old␣
↪v_hops, r_hops
        u_skips, v_skips = v_skips, r_skips # The new u_skips, v_skips is the␣
↪old v_skips, r_skips

    g = u # The variable g now describes the GCD of a and b.
    if g == 1:
        return u_hops % m
    else:  # When GCD(a,m) is not 1...
        return None

print(mult_inverse(3,40))  # 3 times what is congruent to 1, mod 40?
```

```python
print(mult_inverse(7, 57))
```

```
27
49
```

0.7 Chinese Remainder Theorem

- Let $p = p_1 * p_2 * ... * p_k$ where p_i are pairwise relatively prime
  - $a = a_1(mod\ p_1)$
  - $a = a_2(mod\ p_2)$ -...
  - $a = a_k(mod\ p_k)$
- Then the given set of congruence equations always has one and exactly one solution modulo p
  - Then all the systems of equations is equivalent to $x = a(mod\ p)$
- We seek a representation on the form -> called the mixed radix representation of a
  - $a = x_1 + x_2 * p_1 + x_3 * p_1 * p_2 + ... + x_k * p_1 * ... * p_(k - 1)$
- Garner's algorithm computes the coefficients of $x_1, ..., x_k$

[20]:
```python
def inv(a, m) :

    m0 = m
    x0 = 0
    x1 = 1

    if (m == 1) :
        return 0

    # Apply extended Euclid Algorithm
    while (a > 1) :
        # q is quotient
        q = a // m

        t = m

        # m is remainder now, process
        # same as euclid's algo
        m = a % m
        a = t

        t = x0

        x0 = x1 - q * x0

        x1 = t

    # Make x1 positive
    if (x1 < 0) :
        x1 = x1 + m0
```

```python
    return x1

# k is size of num[] and rem[].
# Returns the smallest
# number x such that:
# x % num[0] = rem[0],
# x % num[1] = rem[1],
# ..................
# x % num[k-2] = rem[k-1]
# Assumption: Numbers in num[]
# are pairwise coprime
# (gcd for every pair is 1)
def findMinX(num, rem, k) :

    # Compute product of all numbers
    prod = 1
    for i in range(0, k) :
        prod = prod * num[i]

    # Initialize result
    result = 0

    # Apply above formula
    for i in range(0,k):
        pp = prod // num[i]
        result = result + rem[i] * inv(pp, num[i]) * pp


    return result % prod

# Driver method
num = [3, 4, 5]
rem = [2, 3, 1]
k = len(num)
print( "x is " , findMinX(num, rem, k))
```

```
x is  11
```

1 Cryptography On Crypto Hack

- Grew up in Flagstaff Arizona
  - Cryptography is in my roots, Flagstaff is where the NSA recruited the Navajo during WWII
  - Go through the history of the Wind Talkers.
  - This book is in honor of those heros
  - https://www.npr.org/2022/07/31/1114766110/samuel-sandoval-one-of-the-last-

remaining-navajo-code-talkers-has-died-at-age-98

1.1 What?

- This book is an introduction to Crypto Hack in order to learn more about cryptography.

1.2 Why?

- Cryptographic security and cryptanalysis is used extensively in capture the flag competitions.
- Crypto Hack is usually contracted by Hack The Box (HTB) in order to provide these crypto puzzles.
- Examining Crypto Hack should enable us to be better at cyber security and penetration testing.

1.3 Dragons Beware

- Some of the algorithms below were stolen without attribution.
- Fort Meade/John Hopkins spoke this month on that very same issue.
- According to NSA, there is a systemic issue impacting our cyber-security community: the theft and unauthorized use of algorithms by corporate entities. Entities who themselves may be part of the community. Ultimately because we do not live in an ideal world.
  - https://www.blackhat.com/us-22/briefings/schedule/#dj-vu-uncovering-stolen-algorithms-in-commercial-products-27673
- SO, since I am not making money off of this... take this reference as a way to study for Number Theory

1.3.1 Registering For Crypto Hack

- Later we will use a dictionary to be able to tell which sentence is likely English/Code Page of interest.

[21]:
```python
import functools

def checkTransmission(func):

    @functools.wraps(func)
    def CaesarCipher(*args, **kwargs):
        return func([x.lower() if x.isalpha() else " " for x in args[0]])

    return CaesarCipher
```

[22]:
```python
@checkTransmission
def decryptCaesarCipher(message):
    results = []

    for idx in range(26):
        mapper = {chr(97+tmpIdx): chr(((tmpIdx + idx) % 26) + 97)  for tmpIdx␣
 ↪in range(26)}
```

```
        results.append("".join([mapper[message[x]] if message[x].isalpha() else␣
↪" " for x in range(len(message))]))

    print(results)
```

[23]:
```
print(decryptCaesarCipher([x.lower() if x.isalpha() else " " for x in "GTMCQDC␣
↪RVZLO LZQFHM CDRHFM"]))
```

```
['gtmcqdc rvzlo lzqfhm cdrhfm', 'hundred swamp margin design', 'ivoesfe txbnq
nbshjo eftjho', 'jwpftgf uycor octikp fgukip', 'kxqguhg vzdps pdujlq ghvljq',
'lyrhvih waeqt qevkmr hiwmkr', 'mzsiwji xbfru rfwlns ijxnls', 'natjxkj ycgsv
sgxmot jkyomt', 'obukylk zdhtw thynpu klzpnu', 'pcvlzml aeiux uizoqv lmaqov',
'qdwmanm bfjvy vjaprw mnbrpw', 'rexnbon cgkwz wkbqsx nocsqx', 'sfyocpo dhlxa
xlcrty opdtry', 'tgzpdqp eimyb ymdsuz pqeusz', 'uhaqerq fjnzc znetva qrfvta',
'vibrfsr gkoad aofuwb rsgwub', 'wjcsgts hlpbe bpgvxc sthxvc', 'xkdthut imqcf
cqhwyd tuiywd', 'yleuivu jnrdg drixze uvjzxe', 'zmfvjwv koseh esjyaf vwkayf',
'angwkxw lptfi ftkzbg wxlbzg', 'bohxlyx mqugj gulach xymcah', 'cpiymzy nrvhk
hvmbdi yzndbi', 'dqjznaz oswil iwncej zaoecj', 'erkaoba ptxjm jxodfk abpfdk',
'fslbpcb quykn kypegl bcqgel']
None
```

1.4 Introduction To Crypto Hack

1.4.1 ASCII Conversion

[24]:
```
def convert2ASCII(arr):
    return "".join([chr(x) for x in arr])
```

[25]:
```
convert2ASCII([99, 114, 121, 112, 116, 111, 123, 65, 83, 67, 73, 73, 95, 112,␣
↪114, 49, 110, 116, 52, 98, 108, 51, 125])
```

[25]: 'crypto{ASCII_pr1nt4bl3}'

1.4.2 Bytes To ASCII

- Hard way
- Easy way: bytes.fromhex()

[26]:
```
# Hard way
hexString =␣
↪"63727970746f7b596f755f77696c6c5f62655f776f726b696e675f776974685f6865785f737472696e67735f61
for idx in range(0, len(hexString), 2):
    print(chr(int(hexString[idx], 16) * 16 + int(hexString[idx+1], 16)), end =␣
↪"")
print("\nNext")
# Easy Way
```

```
print(bytes.
 ↪fromhex("63727970746f7b596f755f77696c6c5f62655f776f726b696e675f776974685f6865785f737472696e
 ↪decode())
```

```
crypto{You_will_be_working_with_hex_strings_a_lot}
Next
crypto{You_will_be_working_with_hex_strings_a_lot}
```

1.4.3 Hex To Base64

- Hard way
- Easy way: base64.b64encode()
- Linux way: xxd -p -r hexString.txt | base64

[27]:
```
# Hard way
hexString = "72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf"
binaryString = ""
mapping = {idx: chr(65 + idx) for idx in range(26)}
for idx in range(26):
    mapping[26+idx] = chr(97 + idx)
for idx in range(10):
    mapping[52+idx] = idx
mapping[62] = "+"
mapping[63] = "/"

for idx in range(0, len(hexString), 2):
    tmp = bin(int(hexString[idx], 16) * 16 + int(hexString[idx+1], 16))[2:]
    while len(tmp) < 8:
        tmp = "0" + tmp
    binaryString += tmp

for idx in range(0, len(binaryString), 6):
    print(mapping[int(binaryString[idx:idx+6], 2)], end = "")

print()

# Easy way
import base64

print(base64.b64encode(bytes.fromhex(hexString)).decode())

# Linux way
!touch hexString.txt
!echo "72bca9b68fc16ac7beeb8f849dca1d8a783e8acf9679bf9269f7bf" > hexString.txt
!xxd -p -r hexString.txt | base64
```

```
crypto/Base+64+Encoding+is+Web+Safe/
crypto/Base+64+Encoding+is+Web+Safe/
```

crypto/Base+64+Encoding+is+Web+Safe/

1.4.4 Big Integers To Bytes

- long_to_bytes()
- Reverse: bytes_to_long()

[28]:
```
# !pip3 install pycryptodome
from Crypto.Util.number import *

print(long_to_bytes(11515195063862318899931685488813747395775516287289682636499965282714637259
 ↪decode())
```

crypto{3nc0d1n6_4ll_7h3_w4y_d0wn}

1.4.5 XOR Starter

[29]:
```
xorString = "".join([chr(ord(letter)^13) for letter in 'label'])
print("crypto{" + xorString + "}")
```

crypto{aloha}

1.4.6 XOR Properties

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

[30]:
```
# Information given
KEY1 = bytes.fromhex("a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313")
KEY2_KEY1 = bytes.
 ↪fromhex("37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e")
KEY2_KEY3 = bytes.
 ↪fromhex("c1545756687e7573db23aa1c3452a098b71a7fbf0fddddde5fc1")
FLAG_KEY1_KEY3_KEY2 = bytes.
 ↪fromhex("04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf")

KEY1 = [ord(x) for x in KEY1.decode(encoding= 'unicode_escape')]
KEY2_KEY1 = [ord(x) for x in KEY2_KEY1.decode(encoding= 'unicode_escape')]
KEY2_KEY3 = [ord(x) for x in KEY2_KEY3.decode(encoding= 'unicode_escape')]
FLAG_KEY1_KEY3_KEY2 = [ord(x) for x in FLAG_KEY1_KEY3_KEY2.decode(encoding=␣
 ↪'unicode_escape')]

# All are the same length
print([len(x) for x in [KEY1, KEY2_KEY1, KEY2_KEY3, FLAG_KEY1_KEY3_KEY2]])

def xorPairs(arrX, arrY):
    for idx in range(len(arrX)):
```

```
        arrX[idx] ^= arrY[idx]

    return arrX

# Result
KEY2 = xorPairs(KEY2_KEY1, KEY1)
KEY3 = xorPairs(KEY2_KEY3, KEY2)
FLAG = xorPairs(FLAG_KEY1_KEY3_KEY2, KEY2)
FLAG = xorPairs(FLAG, KEY3)
FLAG = xorPairs(FLAG, KEY1)

print("".join(chr(x) for x in FLAG))
```

```
[26, 26, 26, 26]
crypto{x0r_i5_ass0c1at1v3}
```

1.4.7 Favorite Byte

- No broken (British) English here

[31]:
```
import re
pattern = "^crypto{.*}$"
flag = [ord(x) for x in bytes.
 ↪fromhex("73626960647f6b206821204f21254f7d694f7624662065622127234f726927756d").
 ↪decode()]

for num in range(256):
    result = "".join([chr(x^num) for x in flag])
    if re.search(pattern, result):
        print(result)
```

```
crypto{0x10_15_my_f4v0ur173_by7e}
```

1.4.8 You Either Know, XOR You Don't

- Computer being tapped here without legal right. Wiretapping Act of 1968. Means you have the right to listen, not to manipulate user systems.
- FBI has labeled everyone at NSA as Anonymous. We have been at war since Project Carnivore. Support the homeland and keep out these terrorists.
- Finish this later!
- Long Live the Resistance: NSA, CIA, Pentagon, American People.

[32]:
```
encrypted_msg =␣
 ↪"0e0b213f26041e480b26217f27342e175d0e070a3c5b103e2526217f27342e175d0e077e263451150104"
encrypted_msg = bytes.fromhex(encrypted_msg)

flag_format = b"crypto{"
key = [o1 ^ o2
```

```
        for (o1, o2) in zip(encrypted_msg, flag_format)] + [ord("y")]

flag = []
key_len = len(key)
for i in range(len(encrypted_msg)):
    flag.append(
        encrypted_msg[i] ^ key[i % key_len]
    )
flag = "".join(chr(o) for o in flag)

print("Flag:")
print(flag)
```

```
Flag:
crypto{1f_y0u_Kn0w_En0uGH_y0u_Kn0w_1t_4ll}
```

1.5 Modular Arithmetic

1.5.1 Greatest Common Divisor

- Naive way
- Smart way

[33]:

```
a = 66528
b = 52920

# Naive way
smallest = min(a, b)

largest = max(a, b)
divisors = [smallest]

for i in range(1, smallest//2 + 1):
    if smallest % i == 0:
        divisors.append(i)

for num in sorted(divisors, reverse = True):
    if largest % num == 0:
        print(num)
        break

# Smart way
def gcd(a, b):
    while b:
        a %= b
        tmp = a
        a = b
        b = tmp
```

```
    return a

print(gcd(a, b))
```

```
1512
1512
```

1.5.2 Extended GCD

- Extended Euclidean Algorithm
  - $a * x + b * y = gcd(a, b)$
  - Going backwards: let us assume we found the coefficients (x_1, y_1) for $(b, a * mod * b)$
    * $b * x_1 + (a * mod * b) * y_1 = g$
    * $a * x + b * y = g$
    * $a * mod * b = a - \frac{a}{b} * b$
    * After rearranging: $g = a * y_1 + b(x_1 - y_1 * \frac{a}{b})$
    * $\begin{cases} x = y_1 \\ y = x_1 - y_1 * \frac{a}{b} \end{cases}$

[34]:
```
def extendedGCD(a, b):
    x,y, u,v = 0,1, 1,0

    while a:
        q, r = b // a, b % a
        m, n = x - u * q, y - v * q
        b,a, x,y, u,v = a,r, u,v, m,n

    gcd = b
    return gcd, x, y
```

[35]:
```
p = 26513
q = 32321

larger = max(p, q)
smaller = min(p, q)

def captureAllExtendedEuclidean(large, small):
    steps = []
    while large and small != 0:
        q = large // small
        r = large - (small * q)
        steps.append((large, q, small, r))
        large = small
        small = r
    return steps

for line in captureAllExtendedEuclidean(larger, smaller):
```

```
    print(f"{line[0]} - {line[1]}({line[2]}) = {line[3]}")

gcd_num, u, v = extendedGCD(p, q)

print("Equation")
print(f"{gcd_num} = {u}({p}) + {v}({q})")
```

```
32321 - 1(26513) = 5808
26513 - 4(5808) = 3281
5808 - 1(3281) = 2527
3281 - 1(2527) = 754
2527 - 3(754) = 265
754 - 2(265) = 224
265 - 1(224) = 41
224 - 5(41) = 19
41 - 2(19) = 3
19 - 6(3) = 1
3 - 3(1) = 0
Equation
1 = 10245(26513) + -8404(32321)
```

1.5.3 Modular Arithmetic I

- $ a = b * mod * m$
  - If m | a (if m divides a) -> $\frac{a}{m} = 0 * mod * m$

[36]:
```
def getB(a, m):
    if a // m <= 0:
        return False

    while a > (m * 2):
        a //= m

    return a % m

a1, m1 = 11, 6
a2, m2 = 8146798528947, 17

print(min(getB(a1,m1), getB(a2,m2)))
```

```
4
```

1.5.4 Modular Arithmetic II

[37]:
```
def fermat_theorem_mod(base, exp, prime):
    if exp == prime:
        return base
    elif exp + 1 == prime and base % prime != 0:
```

```
        return 1
    else:
        return -1;

print(fermat_theorem_mod(273246787654, 65536, 65537))
```

```
1
```

1.6 Modular Inverting

[38]:
```
# Mod Inverse when A and M are coprime
def modInverseCoPrime(a, m):
    d, x, y = extendedGCD(a, m)

    return (x % m + m) % m

print(modInverseCoPrime(3, 13))
```

```
9
```

1.7 Quadratic Residues

- We say that an integer x is a Quadratic Residue if there exists an a such that $a^2 = x\ mod\ p$. If there is no such solution, then the integer is a Quadratic Non-Residue
- If $a^2 = x$ then $(-a)^2 = x$. So if x is a quadratic residue in some finite field, then there are always two solutions for a

[39]:
```
p = 29
ints = [14, 6, 11]

qr = [a for a in range(p) if pow(a,2,p) in ints]
print(f"flag {min(qr)}")
```

```
flag 8
```

1.8 Legendre Symbol (Study)

- Allows us to make a single calculation to determine whether an integer is a quadratic residue (assuming modulo is prime)
- Property of quadratic (non-)residues
  - $(Quadratic\ Residue)^2 = Quadratic\ Residue$ -> $1^2 = 1$
  - $Quadratic\ Residue * Quadratic\ Nonresidue = Quadratic\ Nonresidue$ -> $1 * -1 = -1$
  - $Quadratic\ Nonresidue * Quadratic\ Nonresidue = Quadratic\ Residue$ -> $-1 * -1 = 1$
- Legendre's Symbol: $\frac{a}{p} = a^{\frac{p-1}{2}}\ mod\ p$ obeys
  - $\frac{a}{p} = 1$ if a is a quadratic residue and $a \neq 0\ mod\ p$
  - $\frac{a}{p} = 0$ if $a = 0\ mod\ p$
  - $\frac{a}{p} = -1$ if a is a $(quadratic\ nonresidue)\ mod\ p$
  - Which means given any integer a, calculating pow(a,(p-1)/2,p) is enough to determine if a is a quadratic residue

- Calculating the square root modulo a prime: use Tonelli-Shanks
  - All primes that aren't 2 are of the form $p = 1 \ mod \ 4$ or $p = 3 \ mod \ 4$
  - We still need the case $p = 1 \ mod \ 4$
  - Tonelli-Shanks doesn't work for composite (non-prime) moduli. Finding square roots modulo composites is computationally equivalent to integer factorization.

[40]:
```
p =␣
 ↪10152403517453989048540857567108526178875896518906016448438569080146616735666703667793299888

ints =␣
 ↪[25081841204695904475894082974192007718642931811040324543182130088804239047149283334700530б
 ↪45471765180330439060504647480621449634904192839383897212809808339619841633826534856109999027
 ↪17364140182001694956465593533200623738590196990236340894554145562517924989208719245429557764
 ↪14388109104985808487337749876058284426747816961971581447380608277949200244660381570568531122
 ↪43794993083107728210040904476507850953566435904117063581192391666620894286855627192334356155
 ↪85256449776780591202928235662805033201684571648990042997557084658000067050672130152734911911
 ↪50576597458517451578431293746926099486388286246142012476814190030935689430726042810458344822
 ↪96868738830341112368094632337476840272563704408573054404213766500407517251810212494515862176
 ↪48812616568466388006235496629433932343610618271286101200463156497070782441803136610630043907
 ↪18237936726367556664171427575475596460727369368246286138804284742124256700367133250078608537

quadr_res = [n for n in ints if pow(n, (p-1)//2, p)==1][0]
print(f"Square root: {pow(quadr_res, (p+1)//4, p)}")
```

```
Square root: 932917991253667068065456384757974305121049760661036102699380257099522470200610908048701861952859987276802009798538487185891267657425508559548052902535921442095521230621614585845750609394813682106886298620369588576047074683723842780497413691535061826602648761154282519834553442191941330331777004909816961415266
```

[41]:
```
def power(x, y, p) :

    res = 1 # Initialize result
    x = x % p # Update x if it is more
              # than or equal to p

    while (y > 0):

        # If y is odd, multiply x with result
        if (y & 1):
            res = (res * x) % p

        # y must be even now
        y = y >> 1 # y = y/2
        x = (x * x) % p

    return res
```

```python
# Returns true if square root of n under
# modulo p exists. Assumption: p is of the
# form 3*i + 4 where i >= 1
def squareRoot(n, p):

    if (p % 4 != 3) :
        print( "Invalid Input" )
        return


    # Try "+(n^((p + 1)/4))"
    n = n % p
    x = power(n, (p + 1) // 4, p)
    if ((x * x) % p == n):
        print( "Square root is ", x)
        return

    # Try "-(n ^ ((p + 1)/4))"
    x = p - x
    if ((x * x) % p == n):
        print( "Square root is ", x )
        return

    # If none of the above two work, then
    # square root doesn't exist
    print( "Square root doesn't exist " )
```

[42]:
```python
for num in ints:
    if pow(num, (p-1)//2, p) == 1:
        squareRoot(num, p)
```

```
Square root is  93291799125366706806545638475797430512104976066103610269938025709
9952247020061090804870186195285998727680200979853848718589126765742550855954805
2902535921442095521230621614585845750609394813682106886298620369588576047074683
7238427804974136915350618266026487611542825198345534421919413303317770049098169614
1526
```

1.9 Modular Square Root (Study)

[43]:
```python
n =␣
 ↪8479994658316772151941616510097127087554541274812435112009425778595495359700244470400642403
p =␣
 ↪3053185186199433325267593511148795069441433276390908351413376986135096089507650468726136981

def legendre(n, p):
    return pow(n, (p - 1) // 2, p) # 1 == quadratic residue
```

```python
def tonelli(n, p):
    assert legendre(n, p) == 1, "not a square (mod p)"
    q = p - 1
    s = 0
    while q % 2 == 0: #(p-1)%2
        q //= 2
        s += 1
    if s == 1:
        return pow(n, (p + 1) // 4, p)
    for z in range(2, p): #quadratic non-residue (z)
        if p - 1 == legendre(z, p):
            break
    c = pow(z, q, p) # c <- z^q
    r = pow(n, (q + 1) // 2, p) # r <- n^q
    t = pow(n, q, p) # t <- n^q
    m = s # m <-s
    t2 = 0
    while (t - 1) % p != 0: #t=0 or t=1
        t2 = (t * t) % p
        for i in range(1, m): #0<i<m t^2*i = 1
            if (t2 - 1) % p == 0:
                break
            t2 = (t2 * t2) % p
        b = pow(c, 1 << (m - i - 1), p) # b <- c^2m-i-1
        r = (r * b) % p # r<-rb
        c = (b * b) % p # c <-b^2
        t = (t * c) % p # t <-t*b^2
        m = i # m <- i
    return r

if __name__ == '__main__':
    ttest = [(n,p)]
    for n, p in ttest:
        r = tonelli(n, p)
        assert (r * r - n) % p == 0
        print("\t  roots : \n%d \n%d" % (r, p - r))
# r is first solution, second solution is -r mod p =
```

```
          roots :
23623393076830486383277732985804892989321375055205003883382710520537347478623517
```

roots :
23623393076830486383277732985804892989321375055205003883382710520537347478623517
79647314176817953359071871560041125289919247146074907151612762640868199621186559
52206833803260099131188222401602122267224313936218046123264673246584884042545825
79308878565833796009677617385967828778513184893556798228131551230457052851120994
48146426755110160002515592418850432103641815811071548456284263507805589445073657
56538185052136796967569976075531078462357707644003774768176030243492493211364006
17387776011946222441927580241808539162444272540654419625572825728491627727407989

```
89647948645207349737457445440405057156897508368531939120
28169512554311284614348161812907461395482195258388583125795498809297226147214152
90761405563891778919035691757825971779216730291300792798984176397729243448878263
59642536777433420387485673330740435892678962923730287247638080066977070703010353
39291758998923066001985927788808579330075671953036025191791621915640175242425390
39721267479733213280188288022350617720116886492048499354601728433882951201092207
50186895053816428870429809715820583438750781788369658959872713920819264583922833
54971823611423820865651283490761548053384731721391637064349021755899877224522161
311561209530712702153163501623531290150340903913036821041
```

[44]:
```python
def square_root(a, p):
    #Tonelli-Shanks algorithm
    if legendre_symbol(a, p) != 1:
        return 0
    elif a == 0:
        return 0
    elif p == 2:
        return 0
    elif p % 4 == 3:
        return pow(a, (p + 1) / 4, p)

    s = p - 1
    e = 0
    while s % 2 == 0:
        s //= 2
        e += 1

    n = 2
    while legendre_symbol(n, p) != -1:
        n += 1

    x = pow(a, (s + 1) // 2, p)
    b = pow(a, s, p)
    g = pow(n, s, p)
    r = e

    while True:
        t = b
        m = 0
        for m in range(r):
            if t == 1:
                break
            t = pow(t, 2, p)

        if m == 0:
            return x
```

```
        gs = pow(g, 2 ** (r - m - 1), p)
        g = (gs * gs) % p
        x = (x * gs) % p
        b = (b * g) % p
        r = m


def legendre_symbol(a, p):

    ls = pow(a, (p - 1) // 2, p)
    return -1 if ls == p - 1 else ls

print (square_root(n, p))
```

```
23623393076830486383277732985804892989321375055205003883382710520537347478623517
```

1.10 Chinese Remainder Theorem (Study)

[45]:
```
def inv(a, m) :

    m0 = m
    x0 = 0
    x1 = 1

    if (m == 1) :
        return 0

    # Apply extended Euclid Algorithm
    while (a > 1) :
        # q is quotient
        q = a // m

        t = m

        # m is remainder now, process
        # same as euclid's algo
        m = a % m
        a = t

        t = x0
```

```
        x0 = x1 - q * x0

        x1 = t

    # Make x1 positive
    if (x1 < 0) :
        x1 = x1 + m0

    return x1

# k is size of num[] and rem[].
# Returns the smallest
# number x such that:
# x % num[0] = rem[0],
# x % num[1] = rem[1],
# ..................
# x % num[k-2] = rem[k-1]
# Assumption: Numbers in num[]
# are pairwise coprime
# (gcd for every pair is 1)
def findMinX(num, rem, k) :

    # Compute product of all numbers
    prod = 1
    for i in range(0, k) :
        prod = prod * num[i]

    # Initialize result
    result = 0

    # Apply above formula
    for i in range(0,k):
        pp = prod // num[i]
        result = result + rem[i] * inv(pp, num[i]) * pp


    return result % prod

# Driver method
num = [5, 11, 17]
rem = [2, 3, 5]
k = len(num)
print( "x is " , findMinX(num, rem, k) % 935)
```

```
x is  872
```

[46]:
```
from functools import reduce

def chinese_remainder(n, a):
    sum = 0
    prod = reduce(lambda a, b: a*b, n)
    for n_i, a_i in zip(n,a):
        p = prod/n_i
        sum += a_i * mul_inv(p, n_i) * p
    return sum % prod

def mul_inv(a, b):
    b0 = b
    x0, x1 = 0,1
    if b == 1:
        return 1
    while a > 1:
        q = a // b
        a , b = b, a%b
        x0, x1 = x1 -q*x0, x0
    if x1 < 0:
        x1 += b0
    return x1

a = [2,3,5] # x = a mod x
n = [5,11,17] # x = x mod n


print(chinese_remainder(n,a))
```

```
872.0
```

1.11 Symmetric Cryptography

[47]:
```
# Needed for code
import numpy as np
```

1.12 Keyed Permuations

- One-to-one correspondence is a bijection

1.13 Resisting Bruteforce

- Best single-key attack against AES is the biclique attack

1.14 Structure Of AES

- AES-128 begins with a key schedule and then runs 10 rounds over a state. The starting state is just the plaintext block that we want to encrypt, represented as a 4x4 matrix of bytes.

Over the course of the 10 rounds, the state is repeatedly modified by a number of invertible transformations.

- 1. Key Expansion or Key Schedule
  * From the 128 bit key, 11 128 bit round keys are derived; one at each AddRoundKey step
- 2. Initial Key Addition
  * AddRoundKey -> bytes of first round key are XORd with bytes of state
- 3. Round -> this phase is looped 10 times
  * SubBytes -> each byte of state substituted for a different byte according to lookup table/S-box
  * ShiftRows -> last three rows of the state matrix are transposed-shifted over a column(s)
  * MixColumns -> matrix multiplication on columns of the state, combining the four bytes in each column. Skipped in final round.
  * AddRoundKey -> bytes of current round key are XORd with bytes of the state

[48]:
```python
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.  """
    arr = []

    for row in matrix:
        for num in row:
            arr.append(num)

    return arr

def bytes2matrixNP(text):
    return np.array([list(text[i:i+4]) for i in range(0, len(text), 4)],␣
 ↪dtype=np.uint8)

def matrix2bytesNP(matrix_np):
    return matrix_np.ravel()

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

encodedMatrixNP = bytes2matrixNP(("yell"*4).encode())
print(matrix2bytesNP(encodedMatrixNP))
print("".join([chr(x) for x in matrix2bytes(matrix)]))
```

```
[121 101 108 108 121 101 108 108 121 101 108 108 121 101 108 108]
crypto{inmatrix}
```

1.15 Round Keys

- KeyExpansion takes in our 16 byte key and produces 11 4x4 matrices called round keys.
- Initial key addition phase has a single AddRoundKey step: XORs current state with current round key
- AddRoundKey is what makes AES a "keyed permutation" rather than just a permutation

[49]:
```
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):
    #     return [[sss^kkk for sss, kkk in zip(ss, kk)] for ss, kk in zip(s, k)]
    if len(s) != len(k) or len(s[0]) != len(k[0]):
        return False
    for x in range(len(s)):
        for y in range(len(s[0])):
            s[x][y] ^= k[x][y]

    return s

def add_round_keyNP(s, k):
    return np.bitwise_xor(s, k)

print("".join([chr(x) for x in matrix2bytes(add_round_keyNP(state,
 ↪round_key))]))
print("".join([chr(x) for x in matrix2bytes(add_round_key(state, round_key))]))
```

```
crypto{r0undk3y}
crypto{r0undk3y}
```

1.16 Confusion Through Substitution

- First step of each AES round is SubBytes

- Takes each byte of the state matrix and substitutes it for a different byte in a preset 16x16 lookup table/S-box (Substitution)
    - "Confusion" means that the relationship between the ciphertext and the key should be as complex as possible. Given just a ciphertext, there should be no way to learn anything about the key.
    - If a cipher has poor confusion, it is possible to express a relationship between ciphertext, key, and plaintext as a linear function. For instance, in a Caesar cipher, ciphertext = plaintext + key
- S-boxes are aiming for high non-linearity
- The fast lookup in an S-box is a shortcut for performing a very nonlinear function on the input bytes. This function involves taking the modular inverse in the Galois field 2^8 and then applying an affine transformation which has been tweaked for maximum confusion.
- $f(x) = 5 * x^{fe} + 9 * x^{fb} + f * 9x^{fb} + 25 * x^{f7} + f * 4x^{ef} + x^{df} + b * 5x^{bf} + 8 * fx^{7f} + 63$
- To make the S-box, the function has been calculated on all input values from 0x00 to 0xff and the outputs put in the lookup table.

[50]:
```
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):
    answer = []
    for x in range(len(s)):
        for y in range(len(s[0])):
            answer.append(sbox[s[x][y]])
    return answer

def sub_bytesNP(s, sbox=s_box):
    return [sbox[x] for x in s.ravel()]

print("".join([chr(x) for x in sub_bytesNP(np.array(state, dtype=np.int8),␣
 ↪sbox=inv_s_box)]))
print("".join([chr(x) for x in sub_bytes(state, sbox=inv_s_box)]))
```

```
crypto{l1n34rly}
crypto{l1n34rly}
```

1.17 Diffusion Through Permutation

- Diffusion: relates to how every part of a cipher's input should spread to every part of the output
- Substitution on its own creates non-linearity, however it doesn't distribute it over the entire state. Without diffusion, the same byte in the same position would get the same transformations applied to it each round
- Avalanche Effect: An ideal amount of diffusion causes a change of one bit in the plaintext to lead to a change in statistically half the bits of the ciphertext
- ShiftRows and MixColumns work together to ensure every byte affects every other byte in the state within just two rounds
  - ShiftRows: keeps the first row of the state matrix the same. The second row is shifted over one column to the left, wrapping around. The third row is shifted two columns, the fourth row by three
  - MixColumn: performs Matrix multiplication in Rijndael's Galois field between the columns of the state matrix and a preset matrix. Each single byte of each column therefore affects all the bytes of the resulting column.

[51]:
```
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]
```

```python
def inv_shift_rowsNP(s):
    for i in range(s.shape[1]):
        s[:, i] = np.concatenate((s[-i:, i], s[:-i, i]), axis = 0)

# 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],
]

stateNP = np.array(state, dtype=np.uint8)

inv_mix_columns(state)
```

```
inv_mix_columns(stateNP)

inv_shift_rows(state)
inv_shift_rowsNP(stateNP)

print("".join([chr(x) for x in matrix2bytes(state)]))
print("".join([chr(x) for x in matrix2bytes(stateNP)]))
```

```
crypto{d1ffUs3R}
crypto{d1ffUs3R}
```

1.18 Bringing It All Together

- AddRoundKey seeds the key into this substitution-permutation network, making the cipher a keyed permutation
- Decryption involves performing the steps described in the "Structure of AES" challenge in reverse, applying the inverse operations. Note that the KeyExpansion still needs to be run first, and the round keys will be used in reverse order. AddRoundKey and its inverse are identical as XOR has the self-inverse property

[52]:
```
import binascii

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'

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 inv_sub_bytes(s, sbox=inv_s_box):
    for i in range(len(s)):
        for j in range(len(s[0])):
            s[i][j] = sbox[s[i][j]]

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

    # Convert ciphertext to state matrix
    ciphertext = bytes2matrix(ciphertext)
    add_round_key(ciphertext, round_keys[-1])

    # Initial add round key step
    inv_shift_rows(ciphertext)
    inv_sub_bytes(ciphertext)

    for i in range(N_ROUNDS - 1, 0, -1):
        add_round_key(ciphertext, round_keys[i])
        inv_mix_columns(ciphertext)
        inv_shift_rows(ciphertext)
```

```
        inv_sub_bytes(ciphertext)

    # Run final round (skips the InvMixColumns step)
    add_round_key(ciphertext, round_keys[0])

    # Convert state matrix to plaintext
    return matrix2bytes(ciphertext)

print("".join([chr(k) for k in decrypt(key, ciphertext)]))
```

crypto{MYAES128}

[54]:
```
# Work in Progress
def decryptNP(key, ciphertext):
    # binascii.hexlify(key).decode('utf-8')
    # binascii.hexlify(ciphertext).decode('utf-8')
    # Remember to start from the last round key and work backwards through them␣
 ↪when decrypting
    #print(expand_key(key))
    #round_keys = [binascii.hexlify(x).decode('utf-8') for x in expand_key(key)]
    round_keys = []
    keys = expand_key(key)

    for idx in range(len(keys)):
        if idx == 0:
            round_keys.append(keys[idx])
        else:
            #round_keys.append([list(binascii.hexlify(row)) for row in␣
 ↪keys[idx]])
            tmp = []
            for row in keys[idx]:
                row = binascii.hexlify(row)
                newRow = []
                for idx in range(0, len(row), 2):
                    newRow.append(row[idx] + row[idx+1])
                tmp.append(newRow)
            round_keys.append(tmp)

    ciphertext = bytes2matrixNP(ciphertext)

    add_round_keyNP(ciphertext, round_keys[-1])

    # Initial add round key step
    inv_shift_rowsNP(ciphertext)
    inv_sub_bytes(ciphertext)

    for i in range(N_ROUNDS - 1, 0, -1):
```

```
        add_round_keyNP(ciphertext, round_keys[i])
        inv_mix_columns(ciphertext)
        inv_shift_rowsNP(ciphertext)
        inv_sub_bytes(ciphertext)

    # Run final round (skips the InvMixColumns step)
    add_round_keyNP(ciphertext, round_keys[0])
    print(type(ciphertext))
    # Convert state matrix to plaintext
    return matrix2bytes(ciphertext)

#print("".join([chr(k) for k in decryptNP(key, ciphertext)]))
```

1.19 Public Key Encryption

- Maybe later
- Don't use in my projects much

1.20 Elliptic Curves

- Same thing