CSC 321 Lab3 Write-up
Hank Tsai
Partner: Phu Lam

Task I: Implement Diffie Hellman Key Exchange
What I did for task 1 was I used/generated two prime numbers, and used the formula for diffie hellman formula to calculate the shared private key for Alice and Bob. The interesting thing I found out was that if we started with a very large prime number p and g, the computed shared secret key is extremely big as well, which means it is safer.

How hard would it be for an adversary to solve the Diffie Hellman Problem (DHP) given these parameters? What strategy might the adversary take?
Since we are using very small prime numbers, it is relatively easy to solve the Diffie Hellman Problem by brute forcing it to find the factor.

Would the same strategy used for the tiny parameters work here? Why or why not?
It would be difficult because there are so many combinations of Bob's and Alice's private key, and they are both large numbers. Calculating the shared secret is nearly impossible to brute force. Hence, it works here.

```
import random
from Crypto.Hash import SHA256

from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

def part1():
    p = 37
    g = 5

    x = random.randint(1, p - 2)
    y = random.randint(1, p - 2)
    print("p = " + str(p))
    print("g = " + str(g))
    print("x = " + str(x))
    print("y = " + str(y))

    A = pow(g, x) % p
    B = pow(g, y) % p
    print("\n")
    print("A = " + str(A))
    print("B = " + str(B))
```

```
    s1 = pow(A, y) % p
    s2 = pow(B, x) % p
    print("\n")
    print("s1 = " + str(s1))
    print("s2 = " + str(s2))

    k = SHA256.new()
    k.update(bytes(s1))
    truncate_k = bytes(k.hexdigest()[0:16], encoding='utf8')
    print("The key is " + truncate_k.decode('utf8'))

    iv = get_random_bytes(16)
    alice_message = pad(bytes("Hi Bob!", encoding='utf8'), AES.block_size)
    bob_message = pad(bytes("Hi Alice!", encoding='utf8'), AES.block_size)
    print("\n")
    print(unpad(alice_message, AES.block_size))
    print(unpad(bob_message, AES.block_size))

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    alice_ciphertext = cipher.encrypt(alice_message)
    print("\n")
    print("Alice's ciphertext: ", end="")
    print(alice_ciphertext)

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    bob_ciphertext = cipher.encrypt(bob_message)
    print("Bob's ciphertext: ", end="")
    print(bob_ciphertext)

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    bob_message = cipher.decrypt(bob_ciphertext)
    print("\n")
    print("Bob's message: ", end="")
    print(unpad(bob_message, AES.block_size))

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    alice_message = cipher.decrypt(alice_ciphertext)
    print("Alice's message: ", end="")
    print(unpad(alice_message, AES.block_size))

def part2():
    p =
"B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073
E28675A23D189838EF1E2EE652C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90
```

```
C4BD7098488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0A151AF5F0DC8B4BD
45BF37DF365C1A65E68CFDA76D4DA708DF1FB2BC2E4A4371"
    g =
"A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA
1E5C41564B777E690F5504F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92E
E1909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28AD662A4D18E73AFA32D779
D5918D08BC8858F4DCEF97C2A24855E6EEB22B3B2E5"
    p = int.from_bytes(bytes(p, encoding="utf8"), "big")
    g = int.from_bytes(bytes(g, encoding="utf8"), "big")

    x = random.randint(1, p - 2)
    y = random.randint(1, p - 2)

    A = pow(g, x, p)
    B = pow(g, y, p)
    print("\n")
    print("A = " + str(A))
    print("B = " + str(B))

    s1 = pow(A, y, p)
    s2 = pow(B, x, p)
    print("\n")
    print("s1 = " + str(s1))
    print("s2 = " + str(s2))

    k = SHA256.new(bytes(str(s1), encoding='utf8'))
    truncate_k = bytes(k.hexdigest()[0:16], encoding='utf8')
    print("The key is " + truncate_k.decode('utf8'))

    iv = get_random_bytes(16)
    alice_message = pad(bytes("Hi Bob!", encoding='utf8'), AES.block_size)
    bob_message = pad(bytes("Hi Alice!", encoding='utf8'), AES.block_size)
    print("\n")
    print(unpad(alice_message, AES.block_size))
    print(unpad(bob_message, AES.block_size))

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    alice_ciphertext = cipher.encrypt(alice_message)
    print("\n")
    print("Alice's ciphertext: ", end="")
    print(alice_ciphertext)

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    bob_ciphertext = cipher.encrypt(bob_message)
```

```
    print("Bob's ciphertext: ", end="")
    print(bob_ciphertext)

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    bob_message = cipher.decrypt(bob_ciphertext)
    print("\n")
    print("Bob's message: ", end="")
    print(unpad(bob_message, AES.block_size))

    cipher = AES.new(truncate_k, AES.MODE_CBC, iv)
    alice_message = cipher.decrypt(alice_ciphertext)
    print("Alice's message: ", end="")
    print(unpad(alice_message, AES.block_size))

if __name__ == "__main__":

    part1()
    part2()
```

```
Bob's message: b'Hi Alice!'
Alice's message: b'Hi Bob!'


A = 37088279708477652030909697787467738085452101082808990756154546832897392060276468673506316559729523899277767187205927551177851939254281148792648551583566840628270104282498053627092469723278279361678410832965933661570162820408611313052882894075874610720708070655462546329584232856343999691963895086426466590529039223705873522783606085303020101518008217445359647058062564672367303105584766555734081306733646390573 4
B = 58999159441883227287564737104400962136830021232772321346151787181084156322193885341015798454677599980269015066216578216596574209990083392883052860568808705940689964781490157876016601734569431991836393235056185422834040169163042211923670618489264437317965200593750069756020661892677060818330388725436484894743045593602485016908319260159274313103859372589750274668306135761400700292314494659687699817757109619238


s1 = 31612718777473459878254946476191760659350581692833084632048519673339992872909839872338321235161189181383484685628958244958033331613732177953841045855899426763537553313259267848408238525882796362750500564142780007570150420015522152412833528562047939863702398660160734659007465068714650366720381170014734737014377082501729983271612720771917735509391532404571010527192340745308046485408621550680692768573998680263 0
s2 = 31612718777473459878254946476191760659350581692833084632048519673339992872909839872338321235161189181383484685628958244958033331613732177953841045855899426763537553313259267848408238525882796362750500564142780007570150420015522152412833528562047939863702398660160734659007465068714650366720381170014734737014377082501729983271612720771917735509391532404571010527192340745308046485408621550680692768573998680263 0
The key is b2118803f0f0fa21


b'Hi Bob!'
b'Hi Alice!'


Alice's ciphertext: b'\xef\xcc\xb54S\xba3O_\xccB_nf\xd1\xd2'
Bob's ciphertext: b'\xba\xa8\xf4\xcf\x1e\xde\xe7\x0eG\xc1mr\x8e4\xbc]'


Bob's message: b'Hi Alice!'
Alice's message: b'Hi Bob!'
hanktsai@Hanks-MacBook-Air Lab3  %
```

Task II: Implement MITM Key Fixing & Negotiated Groups
What I did was to simulate a middleman that receives Alice and Bob's modulus arithmetic results, and use that to generate two shared malicious secret keys and send them back to Alice and Bob. So they use it to encrypt plaintexts and get decrypted and re-encrypted by the middleman.

Why were these attacks possible? What is necessary to prevent it?
Man in the middle attack works because the communication between Bob and Alice can be unsafe under circumstances. In the process of exchanging keys, their modulus result was attacked half way. The way to prevent it is to make sure the integrity of their key change is preserved by using some safer way such as SSL/HTTPS.

```
import random
from Crypto.Hash import SHA256

from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

def part1():
    p = 37
    g = 5

    # alice
    x_alice = random.randint(1, p-2)
    y_alice = pow(g, x_alice, p)  # sent to mal

    # mal
    x_mal_bob = random.randint(1, p-2)
    y_mal_bob = pow(g, x_mal_bob, p)  # sent to bob

    # bob
    x_bob = random.randint(1, p-2)
    y_bob = pow(g, x_bob) % p
    k_bob = pow(y_mal_bob, x_bob, p)  # sent to mal

    # mal
    x_mal_alice = random.randint(1, p-2)
    k_mal_bob = pow(y_bob, x_mal_bob) % p
    y_mal_alice = pow(g, x_mal_alice) % p  # sent to alice
    k_mal_alice = pow(y_alice, x_mal_alice, p)
```

```
    # alice
    k_alice = pow(y_mal_alice, x_alice, p)
    k = SHA256.new()
    k.update(bytes(k_alice))
    key_alice = bytes(k.hexdigest()[0:16], encoding='utf8')
    iv = get_random_bytes(16)
    cipher_alice = AES.new(key_alice, AES.MODE_CBC, iv)
    message_alice = pad(bytes("Hi Bob!", encoding='utf8'), AES.block_size)
    ciphertext_alice = cipher_alice.encrypt(message_alice)  # sent to mal

    # mal
    k = SHA256.new()
    k.update(bytes(k_mal_alice))
    key_mal_alice = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_mal_alice = AES.new(key_mal_alice, AES.MODE_CBC, iv)
    mal_decrypt_alice = cipher_mal_alice.decrypt(ciphertext_alice)
    print("Mal decrypts Alice's message: ", end="")
    print(unpad(mal_decrypt_alice, AES.block_size))  # mal decrypts alice's 's message

    k = SHA256.new()
    k.update(bytes(k_mal_bob))
    key_mal_bob = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_mal_bob = AES.new(key_mal_bob, AES.MODE_CBC, iv)
    ciphertext_mal_bob = cipher_mal_bob.encrypt(pad(bytes("Hi Bob!", encoding="utf8"),
AES.block_size))  # sent to bob

    # bob
    k = SHA256.new()
    k.update(bytes(k_bob))
    key_bob = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_bob = AES.new(key_bob, AES.MODE_CBC, iv)
    bob_decrypt_mal = cipher_bob.decrypt(ciphertext_mal_bob)
    print("Bob decrypts Mal's message: ", end="")
    print(unpad(bob_decrypt_mal, AES.block_size))  # bob decrypts alice's 's message

    k = SHA256.new()
    k.update(bytes(k_bob))
    key_bob = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_bob = AES.new(key_bob, AES.MODE_CBC, iv)
    message_bob = pad(bytes("Hi Alice!", encoding='utf8'), AES.block_size)
    ciphertext_bob = cipher_bob.encrypt(message_bob)  # sent to mal

    # mal
    k = SHA256.new()
```

```
    k.update(bytes(k_mal_bob))
    key_mal_bob_1 = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_mal_bob_1 = AES.new(key_mal_bob_1, AES.MODE_CBC, iv)
    mal_decrypt_bob = cipher_mal_bob_1.decrypt(ciphertext_bob)
    print("Mal decrypts Bob's message: ", end="")
    print(unpad(mal_decrypt_bob, AES.block_size))  # Mal decrypts Bob's 's message

    k = SHA256.new()
    k.update(bytes(k_mal_alice))
    key_mal_bob_1 = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_mal_alice_1 = AES.new(key_mal_bob_1, AES.MODE_CBC, iv)
    ciphertext_mal_alice_1 = cipher_mal_alice_1.encrypt(
        pad(bytes("Hi Alice!", encoding="utf8"), AES.block_size))  # sent to alice

    # alice
    k = SHA256.new()
    k.update(bytes(k_alice))
    key_alice_1 = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_alice_1 = AES.new(key_alice_1, AES.MODE_CBC, iv)
    alice_decrypt_mal = cipher_alice_1.decrypt(ciphertext_mal_alice_1)
    print("Alice decrypts Mal's message: ", end="")
    print(unpad(alice_decrypt_mal, AES.block_size))  # Mal decrypts Bob's 's message
    print("Done with tampering A and B\n\n")


def part2():
    p = 37
    g = 1

    # alice
    x_alice = random.randint(1, p-2)
    y_alice = pow(g, x_alice, p)  # sent to mal

    # bob
    x_bob = random.randint(1, p-2)
    y_bob = pow(g, x_bob, p)
    k_bob = pow(y_alice, x_bob, p)  # sent to mal

    # alice
    k_alice = pow(y_bob, x_alice, p)
    k = SHA256.new()
    k.update(bytes(k_alice))
    key_alice = bytes(k.hexdigest()[0:16], encoding='utf8')
    iv = get_random_bytes(16)
```

```
    cipher_alice = AES.new(key_alice, AES.MODE_CBC, iv)
    message_alice = pad(bytes("Hi Bob!", encoding='utf8'), AES.block_size)
    ciphertext_alice = cipher_alice.encrypt(message_alice)  # sent to mal
    print("Alice sends ciphertext: ", end="")
    print(ciphertext_alice)

    # mal
    k = SHA256.new()
    k.update(bytes(1))
    key_mal = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_mal = AES.new(key_mal, AES.MODE_CBC, iv)
    mal_decrypt_alice = cipher_mal.decrypt(ciphertext_alice)
    print("Mal decrypts Alice's message: ", end="")
    print(unpad(mal_decrypt_alice, AES.block_size))  # mal decrypts alice's 's message

    # bob
    k = SHA256.new()
    k.update(bytes(k_bob))
    key_bob = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_bob = AES.new(key_bob, AES.MODE_CBC, iv)
    message_bob = pad(bytes("Hi Alice!", encoding='utf8'), AES.block_size)
    ciphertext_bob = cipher_bob.encrypt(message_bob)  # sent to mal
    print("Bob sends ciphertext: ", end="")
    print(cipher_bob)

    # mal
    k = SHA256.new()
    k.update(bytes(1))
    key_mal = bytes(k.hexdigest()[0:16], encoding='utf8')
    cipher_mal = AES.new(key_mal, AES.MODE_CBC, iv)
    mal_decrypt_bob = cipher_mal.decrypt(ciphertext_bob)
    print("Mal decrypts Bob's message: ", end="")
    print(unpad(mal_decrypt_bob, AES.block_size))  # mal decrypts alice's 's message
    print("Done with setting p = 1")


if __name__ == "__main__":

    part1()
    part2()
```

```
hanktsai@Hanks-MacBook-Air Lab3  % /usr/bin/python3 "/Users/hanktsai/Documents/
Mal decrypts Alice's message: b'Hi Bob!'
Bob decrypts Mal's message: b'Hi Bob!'
Mal decrypts Bob's message: b'Hi Alice!'
Alice decrypts Mal's message: b'Hi Alice!'
Done with tampering A and B


Alice sends ciphertext: b'\xa9\xc1 \x13l\x95\x88\x92\x0f\xf2\xd7\xe31;\x19j'
Mal decrypts Alice's message: b'Hi Bob!'
Bob sends ciphertext: <Crypto.Cipher._mode_cbc.CbcMode object at 0x100f615e0>
Mal decrypts Bob's message: b'Hi Alice!'
Done with setting p = 1
```

Task III: Implement "textbook" RSA & MITM Key Fixing via Malleability

What I did was use the formula for RSA to solve those parameters we need and calculate d using extended euclidean algorithm. And for part two we used the formula described in the table, and had mallory modify the c prime.

While it's very common for many people to share an e (common values are 3,7, 216+1), it is very bad if two people share an RSA modulus n. Briefly describe why this is, and what the ramifications are.

If a person encrypts the same plaintext two different times with the same modulus and different e. The attacker can eavesdrop on the communication and decrypt the message.

Give another example of how RSA's malleability could be used to exploit a system (e.g. to cause confusion, disruption, or violate integrity)

Another example of RSA's malleability is the chosen ciphertext attack where the attacker chooses the ciphertexts C' and to view their corresponding decryptions.

```
import random
from Crypto.Util import number

# def gcd(a, b):
#     while b != 0:
#         a, b = b, a % b
#     return a


def modInverse(A, M):
    for X in range(1, M):
        if (((A % M) * (X % M)) % M == 1):
            return X
```

```
    return -1

def part1():
    p = number.getPrime(2018)
    q = number.getPrime(2018)

    n = p * q
    fn = (p-1) * (q-1)

    e = 65537
    # d = modInverse(e, fn)
    d = pow(e, -1, fn)

    public_key = (e, n)
    private_key = (d, n)
    print("public key: ", public_key)
    print("private key: ", private_key)

    message = random.randrange(100, 1000)
    print("message is: ", message)

    ciphertext = pow(message, e, n)
    print("Encrypted cipher: ", ciphertext)

    plaintext = pow(ciphertext, d, n)
    print("plaintext: ", plaintext, end="\n\n")

def part2():
    n = 14
    e = 5
    d = 11

    # bob
    s = 2
    c_bob = pow(s, e, n)  # sent
    print("Bob sent c ", c_bob)

    # mal
    random = 3
    c_mal = c_bob * pow(random, e, n)  # sent
    print("Mal sent c' ", c_mal)

    # ali
    s_ali = pow(c_mal, d, n)  # sent
```

```
    print("Alice sent back ", s_ali)

    # mal (decrypts)
    res = (s_ali * modInverse(random, n)) % n
    print("Mal decrypt original message as ", res)


if __name__ == "__main__":
    part1()
    part2()
```

```
897846586627390569843110694472147301178251626
81361)
message is:  961
Encripted cipher:  436454617945833506285503 2
411866465046842587265909512199970840038343 11
886023381582474999094912942579341757350773 37
346431499659520034916325690813005578173090 10
702002042338613724549340525031014991744684 97
730497579924817753865257475255195643959227 77
022178834489404477422893942124359787207227 66
342850372615407463842121390199637623402564 93
819384672349347858846515089099663783201330 29
014691060829740482504188463352692737710961 74
493066104286098482197021173943177319176202 42
5072368646182
plaintext:  961

Bob sent c  2
Mal sent c'  2
Alice sent back  6
Mal decrypt original message as  2
hanktsai@Hanks-MacBook-Air Lab3  %
```

Task IV: Exploring Pseudo-Randomness and Collision Resistance

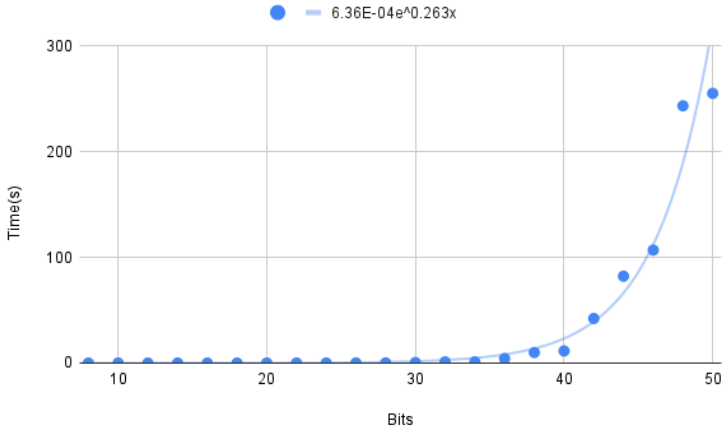
We used a dictionary to record the mapped String, and we kept doing that for different sizes of bits by going 2 bits up at a time until we found a collison. It was interesting to me that it took significantly longer than the previous one since it was 42 bits size.

What do you observe? How many of the bytes are different between the two digests?
Changing one bit in the string leads to a change in every single digest

What is the maximum number of files you would ever need to hash to find a collision on an n-bit digest? Given the Birthday Bound, what is the expected number of hashes before a collision on an n-bit digest? Is this what you observed? Given the data you've collected, speculate on how long it might take to find a collision on the full 256-bit digest.

With n-bit, we expect to $2^n + 1$ 2^n+1 to guarantee a collision; And with the Birthday Bound, we would expect a collision around $2^{\sqrt{n}}$ number of hashes. Yes, According to the data, we expect a collision of around 50%. Put x = 250 into the equation,

$y = 6.36E - 04e^{0.253x} = 2.2820192e + 25$ seconds



Given an 8-bit digest, would you be able to break the one-way property (i.e. can you find any pre-image)? Do you think this would be easier or harder (i.e. more or less work) than finding a collision? Why or why not?

With an 8 bit digest, it would be much easier to find a collision than finding any pre-image. Since, there are only 8 bits. We can guarantee to find a collision with 257 hashes.

```
from Crypto.Hash import SHA256
import time
# from bitstring import BitArray

def part1():
    h = SHA256.new(b'hello')
    print("hello ", h.hexdigest())
    h = SHA256.new(b'Hello')
    print("Hello ", h.hexdigest())
    h = SHA256.new(b'hEllo')
    print("hEllo ", h.hexdigest())
    h = SHA256.new(b'heLlo')
    print("heLlo ", h.hexdigest())
```

```
    h = SHA256.new(b'helLo')
    print("helLo ", h.hexdigest())
    h = SHA256.new(b'hellO')
    print("hellO ", h.hexdigest())


def part2Truncate(input, size):
    h = SHA256.new(input)
    return h.hexdigest()[0:size + 1]


def find_collision(size):
    hash_map = dict()
    i = 0
    flag = True
    while flag:
        h = SHA256.new(bytes(str(i), encoding="utf8"))
        hash_result = h.hexdigest()
        hash_result = bin(int(hash_result, base=16))[0:size + 1]


        if hash_result in hash_map:
            print("String", i, "\thashes to\t", hash_result)
            print("String", hash_map[hash_result], "\thashes to\t", hash_result)
            print("\n")
            flag = False

        hash_map[hash_result] = i

        i += 1

if __name__ == "__main__":
    # part1()
    xandy = dict()

    for i in range(8, 50, 2):
        st = time.time()
        find_collision(i)
        et = time.time()
        elapsed_time = et - st
        print("it takes ", elapsed_time, " seconds", "for", i, "bits")
        xandy[i] = elapsed_time

    print(xandy)
```

```
hanktsai@Hanks-MacBook-Air Lab3  % /usr/bin/python3 "/Users/hanktsa
String 12       hashes to       0b1101011
String 1        hashes to       0b1101011


it takes  0.0003559589385986328  seconds for 8 bits
String 29       hashes to       0b110101000
String 2        hashes to       0b110101000


it takes  0.0005319118499755859  seconds for 10 bits
String 54       hashes to       0b10111111001
String 26       hashes to       0b10111111001


it takes  0.0008628368377685547  seconds for 12 bits
String 84       hashes to       0b1000100110010
String 43       hashes to       0b1000100110010


it takes  0.0013120174407958984  seconds for 14 bits
String 140      hashes to       0b110110111010111
String 136      hashes to       0b110110111010111


it takes  0.0021581649780273438  seconds for 16 bits
String 522      hashes to       0b10101001001101000
String 452      hashes to       0b10101001001101000


it takes  0.00793600082397461  seconds for 18 bits
String 1597     hashes to       0b1110101101010111111
String 1549     hashes to       0b1110101101010111111


it takes  0.023616790771484375  seconds for 20 bits
String 1849     hashes to       0b111100111111110001001
String 1376     hashes to       0b111100111111110001001


it takes  0.027798175811767578  seconds for 22 bits
```

```
it takes  6.471225023269653  seconds for 38 bits
String 480034   hashes to       0b10111101011010111011111011101000100010
String 342678   hashes to       0b10111101011010111011111011101000100010


it takes  7.917487144470215  seconds for 40 bits
String 1506679  hashes to       0b1110001011111111011011001011111001101011 0
String 1014958  hashes to       0b1110001011111111011011001011111001101011 0


it takes  25.662642002105713  seconds for 42 bits
String 3290784  hashes to       0b101101111011111011011100110001110111011 1000
String 2309439  hashes to       0b101101111011111011011100110001110111011 1000


it takes  56.92756676673889  seconds for 44 bits
String 4600132  hashes to       0b11111111101000001110101110000001110001101 0010
String 4265222  hashes to       0b11111111101000001110101110000001110001101 0010


it takes  79.57947182655334  seconds for 46 bits
String 10452899         hashes to       0b1000010010110110110110011111110001100011011 1100
String 5263026  hashes to       0b1000010010110110110110011111110001100011011 1100
```