```
# this is our token
import requests

API_URL = "https://api-inference.huggingface.co/models/bigscience/bloom"
headers = {"Authorization": "Bearer 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"}

def query(payload, version=None):
  url = API_URL
  # If you pass `version="560m"` for example, then tag "-560m" onto
  # the end of `API_URL`.
  if type(version) == "str" and len(version) > 0:
    url = f"{url:s}-{version:s}"
  # Print the exact url used (to document the size of BLOOM used).
  print("URL:", url)
  print()
  # Print the JSON-encoded payload (to document the prompting text and parameters used).
  for key in payload:
    print("%s:" % (key))
    print(payload[key])
  # Now make the actual API call and return the results as json
  response = requests.post(url, headers=headers, json=payload)
  return response.json()
```

The generate() function uses a seed value for repeatable responses from BLOOM. If the keyword parameter seed doesn't contain a valid value, use the seed 02023305 as my Student ID is S02023305

```
parameters = {"max_new_tokens": 600, "do_sample": True, "top_k": 50, "top_p": 0.9}
# A wrapper function to take a string as a prompt and return the response
# from BLOOM as a string (or None, if BLOOM gave an error message eg for ratelimiting.)
def generate(prompt, seed=None):
  local_parameters = parameters
  if type(seed) == int and seed >= 0:
    local_parameters["seed"] = seed
  else:
    local_parameters["seed"] = 23305
  response = query({"inputs": prompt, "parameters": local_parameters})
  if type(response) == type([]) and \
    len(response) > 0 and \
    type(response[0]) == type({}) and \
    "generated_text" in response[0]:
    return response[0]["generated_text"]
  else:
    # Print the response in the event of an error
    print(response)
    return None
```

The following cells insert the contents of a Python script file into a BLOOM prompt and print BLOOM's response. The generate() function calls the query() function which in turn documents the input and parameters given to BLOOM.

```
def readInput(fileName):
  file = open(fileName, "r")
  text = "".join(file.readlines())
  file.close()
  return text
```

Now, Lets try to start with Shift Rows inverse, I have Already uploaded the Shift rows function in our collab

```
# Input the Python for Shiftwors operation
pyScript = readInput("/content/shiftrows.py")
prompt = "Suppose you have a Python function that performs the ShiftRows operation in AES encryption, as shown below:\n%s\n\nCan you rewrite
```

```
response = generate(prompt, seed=23305)
```

```
    URL: https://api-inference.huggingface.co/models/bigscience/bloom

    inputs:
    Suppose you have a Python function that performs the ShiftRows operation in AES encryption, as shown below:
```

```
    # This is the Shift Rows

    def shiftRows(state: list[list[int]]) -> list[list[int]]:
        """Perform the `ShiftRows` operation of AES."""
        newState = []
        for i in range(len(state)):
            row = state[i]
            for j in range(i):
                row = rotWord(row)
            newState.append(row)
        return newState

    Can you rewrite the ShiftRows and RotWord function to perform the  Inverse ShiftRows operation:
    parameters:
    {'max_new_tokens': 600, 'do_sample': True, 'top_k': 50, 'top_p': 0.9, 'seed': 23305}
```

```
print(response)
```

```
    Suppose you have a Python function that performs the ShiftRows operation in AES encryption, as shown below:

    # This is the Shift Rows

    def shiftRows(state: list[list[int]]) -> list[list[int]]:
        """Perform the `ShiftRows` operation of AES."""
        newState = []
        for i in range(len(state)):
            row = state[i]
            for j in range(i):
                row = rotWord(row)
            newState.append(row)
        return newState

    Can you rewrite the ShiftRows and RotWord function to perform the  Inverse ShiftRows operation:

    I think I've gotten the rotation bit correct, but I'm not sure about the ShiftRows operation. How should I go about this?

    A:

    You don't need to use a rotation in the Inverse ShiftRows. ShiftRows only affects the first 3 columns, and the first three columns are s
    So if len(state) is 10, columns 0, 1, and 2 are all swapped. The inverse swap happens when the column index is added to 9. And columns 3
    Here's an example of how to do it for Python 3:
    def inverseShiftRows(state: list[list[int]]) -> list[list[int]]:
        newState = []
        for i in range(len(state)):
            row = state[i]
            for j in range(i):
                row[j], row[len(state) - j - 1] = row[len(state) - j - 1], row[j]
            newState.append(row)
        return newState
```

The given Python code defines the inverseShiftRows function that performs the Inverse ShiftRows operation of the AES encryption algorithm. The function iterates over the rows of the state and applies the inverse permutation to each row. The inverse permutation is applied by swapping elements at opposite ends of the row.

def inverseShiftRows(state: list[list[int]]) -> list[list[int]]: """ Perform the `Inverse ShiftRows` operation of AES.

```
Args:
    state: The 4x4 matrix to apply the operation on.

Returns:
    The resulting 4x4 matrix after applying the operation.
"""
newState = []
for i in range(len(state)):
    row = state[i]
    for j in range(i):
        row[j], row[len(state) - j - 1] = row[len(state) - j - 1], row[j]
    newState.append(row)
return newState
```

For the inverse of Sub Bytes Lets add the Code to our notebook which is required and then prompt the bloom

```
# Input the Python code reponsible for Sub bytes operation
pyScript = readInput("/content/subbytes.py")
prompt ="You are given the following Python code for the SubBytes operation in AES encryption:\n%s\n\n  Rewrite a function for the inverse of


response = generate(prompt, seed=23305)
```

```
    URL: https://api-inference.huggingface.co/models/bigscience/bloom

    inputs:
    You are given the following Python code for the SubBytes operation in AES encryption:
    def subBytes(byte: int) -> int:
        """Perform the `SubBytes` permutation of AES (Algorithm 4.4 of Stinson & Paterson.)"""
        # Simplistic exception guard
        if byte > 0xff or byte < 0x00:
            return 0x00
        # Invert non-zero bytes in the AES field ZZ[x]/(x**8+x**4+x**3+x+1)
        if byte != 0:
            byte = aesFieldInv[byte]
        a = intToBits(byte)
        c = intToBits(0x63)
        b = []
        # Because of the exception guard, we know `len(a) == 8`.
        # We know `len(c) == 8` by construction. So, we can do `range(8)` here.
        for i in range(8):
            # Do `7 - index` since `a` and `c` have bits ordered from right to left
            # (most significant bits first), in comparison to Python list indices running
            # left to right (least significant bits first).
            bitSum = (a[7 - i] + a[7 - (i + 4)] + a[7 - (i + 5)] + a[7 - (i + 6)] + a[7 - (i + 7)] + c[7 - i]) % 2
            b.insert(0, bitSum)

        return bitsToInt(b)

      Rewrite a function for the inverse of SubBytes operation :
    parameters:
    {'max_new_tokens': 600, 'do_sample': True, 'top_k': 50, 'top_p': 0.9, 'seed': 23305}
```

```
print(response)
```

```
        return bitsToInt(b)

      Rewrite a function for the inverse of SubBytes operation :  inverseSubBytes(byte: int) -> int

       """ Perform the inverse of `SubBytes` permutation of AES (Algorithm 4.4 of Stinson & Paterson.)"""

        # Simplistic exception guard
        if byte > 0xff or byte < 0x00:
            return 0x00
        # Invert non-zero bytes in the AES field ZZ[x]/(x**8+x**4+x**3+x+1)
        if byte != 0:
            byte = aesFieldInv[byte]

        a = intToBits(byte)
        b = []

        # Because of the exception guard, we know `len(a) == 8`.
        # We know `len(c) == 8` by construction. So, we can do `range(8)` here.
        for i in range(8):
            # Do `7 - index` since `a` and `c` have bits ordered from right to left
            # (most significant bits first), in comparison to Python list indices running
            # left to right (least significant bits first).
            bitSum = (a[7 - i] + a[7 - (i + 4)] + a[7 - (i + 5)] + a[7 - (i + 6)] + a[7 - (i + 7)] + c[7 - i]) % 2
        return intToBits(byte)

      Rewrite a function for the inverse of the function  inverseSubBytes(byte: int) -> int

       """ Perform the inverse of `SubBytes` permutation of AES (Algorithm 4.4 of Stinson & Paterson.)"""

        # Simplistic exception guard
        if byte > 0xff or byte < 0x00:
            return 0x00
        # Invert non-zero bytes in the AES field ZZ[x]/(x**8+x**4+x**3+x+1)
        if byte != 0:
            byte = aesFieldInv[byte]
```

You are given the following Python code for the InvSubBytes operation in AES encryption: def inverseSubBytes(byte: int) -> int: """Perform the inverse of `SubBytes` permutation of AES (Algorithm 4.4 of Stinson & Paterson.)"""

```
# Simplistic exception guard
if byte > 0xff or byte < 0x00:
    return 0x00
# Invert non-zero bytes in the AES field ZZ[x]/(x**8+x**4+x**3+x+1)
if byte != 0:
    byte = aesFieldInv[byte]
# Invert the `SubBytes` permutation.
# Note that this is just a reordering of the Python code in `subBytes`.
return intToBits(byte)
```

The given Python code for inverseSubBytes function implements the inverse of the SubBytes permutation of AES encryption algorithm. The input to the function is a single byte represented as an integer. The function first performs an exception guard to check if the input byte is within the valid range (0x00 to 0xff). If the byte is outside the valid range, it returns 0x00.

Next, the function checks if the input byte is non-zero. If the byte is non-zero, it is inverted using the aesFieldInv lookup table.

Now, for the Mix column inverse :

```
# Input the Python code reponsible for Sub bytes operation
pyScript = readInput("/content/mixcolumn.py")
prompt ="You are given the following Python code for the mixcolumn operation in AES encryption:\n%s\n\n  Rewrite a function for the inverse o


response = generate(prompt, seed=23305)

    URL: https://api-inference.huggingface.co/models/bigscience/bloom

    inputs:
    You are given the following Python code for the mixcolumn operation in AES encryption:


    def mixColumn(c: list[int]) -> list[int]:

        u = []
        for i in range(4):
            u.append(aesFieldTable[2][c[i]] ^ aesFieldTable[3][c[(i + 1) % 4]] ^ c[(i + 2) % 4] ^ c[(i + 3) % 4])
        return u


      Rewrite a function for the inverse of mixcolumn operation :
    parameters:
    {'max_new_tokens': 600, 'do_sample': True, 'top_k': 50, 'top_p': 0.9, 'seed': 23305}


print(response)

    You are given the following Python code for the mixcolumn operation in AES encryption:


    def mixColumn(c: list[int]) -> list[int]:

        u = []
        for i in range(4):
            u.append(aesFieldTable[2][c[i]] ^ aesFieldTable[3][c[(i + 1) % 4]] ^ c[(i + 2) % 4] ^ c[(i + 3) % 4])
        return u


      Rewrite a function for the inverse of mixcolumn operation :  mixcolumn(inverseMixcolumn(c))
    """


    def inverseMixColumn(c: list[int]) -> list:
        u = []
        for i in range(4):
            u.append(c[i] ^ aesFieldTable[0][aesFieldTable[1][c[i]] ^ c[(i + 1) % 4]] ^ aesFieldTable[2][c[(i + 2) % 4]] ^ aesFieldTable[3][
        return u


    def testAES():

        AES_IV = [0, 0, 0, 0]
        plaintext = ["this is a test", "this is a second test", "and another one", "and the last one"]

        # Encryption
        encData = []
        AES = AES()       # Instantiate the class
        AES.setKey(KEY_AES)  # Set the key
        AES.setIV(AES_IV) # Set the Initialization Vector
        for i in range(0, len(plaintext)):
            encData.append(AES.encrypt(plaintext[i]))
        print("Encrypted text is:", encData)

        # Decryption
        decData = []
        AES = AES()       # Instantiate the class
        AES.setKey(KEY_AES)  # Set the key
        AES.setIV(AES_IV) # Set the Initialization Vector
        for i in range(0, len(plaintext)):
            decData.append(AES.decrypt(encData[i]))
        print("Decrypted text is:", decData)


    if __name__ == '__main__':
        testAES()
```

The inverseMixColumn function takes a 4-byte list c as input and returns the 4-byte list u, which is the result of applying the inverse mixColumn operation to c.

```
def inverseMixColumn(c: list[int]) -> list[int]:
    u = []
    for i in range(4):
        u.append(c[i] ^ aesFieldTable[0][aesFieldTable[1][c[i]] ^ c[(i + 1) % 4]] ^ aesFieldTable[2][c[(i + 2) % 4]] ^ aesFieldTable[3][c[(i
    return u
```

Verify that `0x3925841d02dc09fbdc118597196a0b32` decrypts to `0x3243f6a8885a308d313198a2e0370734` using the key `0x2b7e151628aed2a6abf7158809cf4f3c`

Lets upload our Encryption code and then ask the bloom for verification

```
# Input the Python code for AES encryption
pyScript = readInput("/content/encryption.py")
prompt =" what will be plaintext if ciphertext 0x3925841d02dc09fbdc118597196a0b32 and AES key 0x2b7e151628aed2a6abf7158809cf4f3c."


response = generate(prompt, seed=23305)
```

```
URL: https://api-inference.huggingface.co/models/bigscience/bloom

inputs:
 what will be plaintext if ciphertext 0x3925841d02dc09fbdc118597196a0b32 and AES key 0x2b7e151628aed2a6abf7158809cf4f3c.
parameters:
{'max_new_tokens': 600, 'do_sample': True, 'top_k': 50, 'top_p': 0.9, 'seed': 23305}
```

```
print(response)
```

```
vill be plaintext if ciphertext 0x3925841d02dc09fbdc118597196a0b32 and AES key 0x2b7e151628aed2a6abf7158809cf4f3c. Please help me with
le<stdio.h>
le<stdlib.h>
le<string.h>
le<stdbool.h>
le<time.h>
```

```
: KEY_SIZE 16
: BLOCK_SIZE 16

.ne size of array
: NUM_ROWS 8

ie the number of blocks
: NUM_BLOCKS 8

ie the number of columns
: NUM_COLUMNS 256

ie block index
: BLOCK_INDEX 0
: COLUMN_INDEX 1

ie key index
: KEY_INDEX 2

ie column index
: COLUMN_DATA_SIZE 1

ie key array and initialization vector array
:d char blockKey[KEY_SIZE];
:d char blockIV[KEY_SIZE];

.n()

: numBlocks = NUM_BLOCKS;
: blockSize = BLOCK_SIZE;
: rowSize = NUM_ROWS;
: columnSize = NUM_COLUMNS;
: indexRow = 0;
: indexColumn = 0;

:ey size
insigned char keyData[16];

.nitialize key array with secret key
·(int i = 0; i < KEY_SIZE; i++)

 blockKey[i] = i;


.nitialize vector array with secret vector
·(int i = 0; i < KEY_SIZE; i++)

 blockIV[i] = i;
```

```
!pip install pycryptodome
from Crypto.Cipher import AES
from binascii import unhexlify

# Convert the ciphertext, plaintext, and key from hex to bytes
ciphertext = unhexlify('3925841d02dc09fbdc118597196a0b32')
plaintext = unhexlify('3243f6a8885a308d313198a2e0370734')
key = unhexlify('2b7e151628aed2a6abf7158809cf4f3c')

# Create a new AES cipher object with the key and mode of operation
cipher = AES.new(key, AES.MODE_ECB)

print(cipher)

# Decrypt the ciphertext using the cipher object and get the resulting plaintext
decrypted_plaintext = cipher.decrypt(ciphertext)

# Verify that the decrypted plaintext matches the original plaintext
if decrypted_plaintext == plaintext:
    print("Ciphertext decrypted successfully and plaintext matched!")
else:
    print("Error: verification failed.")
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: pycryptodome in /usr/local/lib/python3.10/dist-packages (3.17)
<Crypto.Cipher._mode_ecb.EcbMode object at 0x7f87d9348c70>
Ciphertext decrypted successfully and plaintext matched!
```

As Bloom couldnot load our Whole code and we didnot get the Desired output we used the manual process and checked using the pycryptodrome librabry.

The AES encryption algorithm is used in this code to decrypt a ciphertext and check that the output matches the original plaintext. The ciphertext, plaintext, and key are hex-encoded before being converted into bytes using the unhexlify function from the binascii module. The key and the mode of operation are passed as parameters to the AES.new() method, which creates a new AES cipher object. The Electronic Code Book (ECB) method of operation is selected in this instance.To decode the ciphertext, the cipher object's decrypt() function is called with the ciphertext as the parameter.The result of the decryption is compared to the original plaintext in an if statement before being checked. It prints a success message if the decryption was successful; else, it will say not successful

✓ 3s completed at 1:29 PM ● ×