Step 1 - Convert hex to base64

> The string:
>
> ```
> 49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e6f7573206d757368726f6f6d
> ```
>
> Should produce:
>
> ```
> SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
> ```
>
> So go ahead and make that happen. You'll need to use this code for the rest of the exercises.
>
> ### Comment
>
> Always operate on raw bytes, never on encoded strings. Only use hex and base64 for pretty-printing.

3]:

```python
import base64

# The function converts a hex number using the base64 library
# Function converts hex to raw bytes, to base64 encoring, and then to a utf-8 string format.

def convert_hex_to_base(s):
    # convert hex to raw bytes
    s_bytes = bytes.fromhex(s)
    # encode raw bytes to base64
    base = base64.b64encode(s_bytes)
    # decode base64 to string
    return base.decode('utf-8')


stringg = "49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f69736f6e6f7573206d757368726f6f6d"
print(convert_hex_to_base(stringg))
if convert_hex_to_base(stringg) == "SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t":
    print("Correct!")
```

```
SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t
Correct!
```

Step 2 - Fixed XOR

> Write a function that takes two equal-length buffers and produces their XOR combination.
>
> If your function works properly, then when you feed it the string:
>
> ```
> 1c0111001f010100061a024b53535009181c
> ```
>
> ... after hex decoding, and when XOR'd (bitwise) against:
>
> ```
> 686974207468652062756c6c277320657965
> ```
>
> ... should produce:
>
> ```
> 746865206b696420646f6e277420706c6179
> ```

[219]:

```
# This function takes two hex strings as inputs and performs a bitwise XOR on the corresponding bytes of both inputs
# The function returns a new hex string that represents the XOR result of the two input strings

def xor(hex1, hex2):

    # convert hex to raw bytes for both buffers
    h1_bytes = bytes.fromhex(hex1)
    h2_bytes = bytes.fromhex(hex2)

    # get byte array of each xor'd bit using zip function that compares both buffer's bit by bit
    byte_array = []
    for b1, b2 in zip(h1_bytes,h2_bytes):
        byte_array.append(b1 ^ b2)

    # creates a byte object
    xor_bytes = bytes(byte_array)

    # convert raw bytes to hex
    return xor_bytes.hex()


res = xor('1c0111001f010100061a024b53535009181c', '686974207468652062756c6c277320657965')
print(res)
if res == "746865206b696420646f6e277420706c6179":
    print("Correct!")
```

```
746865206b696420646f6e277420706c6179
Correct!
```

Step 3 - Single-byte XOR cipher

> The hex encoded string:
>
> ```
> 1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b3736
> ```
>
> ... has been XOR'd against a single character. Find the key (which is one byte) and decrypt the message. The message is a meaningful sentence in English!
>
> You should write a code to find the key and decrypt the message. Don't do it manually!
>
> ### Comment
>
> There are several mini steps to achieve this! First, you need a strategy for searching in the key space. Second, you need a test/scoring mechanism to check whether the decrypted message is meaningful or not (i.e., detecting garbage vs. the correct output). You can read more about "*Caesar*" cipher to get some ideas and more background!

Description

A brief description of your approach. Don't just put the code. First explain what you did and WHY you did it!

(your description)
...

```
!pip install textdistance
```

```
Collecting textdistance
  Downloading textdistance-4.5.0-py3-none-any.whl (31 kB)
Installing collected packages: textdistance
Successfully installed textdistance-4.5.0
WARNING: You are using pip version 20.1.1; however, version 23.1.2 is available.
You should consider upgrading via the '/Users/yash/Desktop/ec engr 209as/CA1/venv/bin/python3.11 -m pip install --upg
rade pip' command.
```

In [220]:
```python
import textdistance

# Strategy for searching:
    # Iterate through the bytes and xor each character with each byte string
    # Check if the xor-ed output is similar to english by iterating word by word and giving a score to each word
    # if the score is better than previously obtained scores, update the score, text, and key object to give us
    # the best possible chance for finding the encrypted text
# Test/Scoring Mechanism:
    # Use textdistance library to attach a score using a list of common english words in sentences
    # use best-score as mechanism to score the similarity of words from the cipertext and real english words

# I did this because I knew that I had to determine to what level is the text english, so I searched for a library
# that could help me make that determiniation. I learned of the Jaro Winkler algorithm that determines
# the distance of text to other words. Using that, I could make a scoring mechanism.
# Then all I needed to do was iterate through the bytes and find the decrypted message

def is_english_text(text):
    # asked chatGPT for a list of common english words
    common_english_words = ["the", "be", "to", "of", "and", "in", "that", "have", "it", "for", "not", "on", "with", "he

    jw = textdistance.JaroWinkler()
    score = 0

    # iterate through each word in the cipher text
    for word in text.lower().split():
        # iterate through the list of common english words and compare the similarity of the words
        for en_word in common_english_words:
            sim = jw.similarity(word, en_word)
            # add similarity score to score
            score += sim

    return score

def find_key_and_decrypt(hex_string):
    best_key = None
    best_text = None
    best_score = 0

    for i in range(128):
        # xor string with current character
        key = bytes(i for _ in range(len(hex_string)))
        decrypted_text = xor(hex_string, key.hex())

        # convert hex back to text
        decrypted_text = bytes.fromhex(decrypted_text).decode('utf-8', errors='ignore')

        # check if the text is english and give it a score
        score = is_english_text(decrypted_text)

        # Update the best key abd decrypted text if the current score is higher
        if score > best_score:
            best_score = score
            best_decrypted_text = decrypted_text
            best_key = chr(i)

    return best_key, best_score, best_decrypted_text

hex_string = "1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b3736"
key, score, decrypted_message = find_key_and_decrypt(hex_string)
print("Key:", key)
print("Decrypted message:", decrypted_message)
```

```
Key: X
Decrypted message: Cooking MC's like a pound of bacon
```

Step 4 - Detect single-character XOR

> One of the 60-character strings in this file has been encrypted by single-character XOR (each line is one string).
>
> Find it.
>
> ### Comment
>
> You should use your code in Step 3 to test each line. One line should output a meaningful message. Remeber that you don't know the key either but you can find it for each line (if any).

Description

A brief description of your approach. Don't just put the code. First explain what you did and WHY you did it!

(your description)
...

[222]:
```python
# This function utilizes the find_key_and_decrypt function for each line in the file
# Then I use the associated scores to determine what line gives the best decrypted text
# Once that is done, then I can return the text to get the decrypted message
def detect_single_character_xor(lines):

    best_key = None
    best_score = float('-inf')
    best_text = None

    # iterate through each of the lines
    for line in lines:
        hex_string = line.strip()
        # call the find_key_and_decrypt function to get the associated key, score, and decrypted text
        key, score, text = find_key_and_decrypt(hex_string)
        if text and score > best_score:
            best_score = score
            best_key = key
            best_text = text

    return best_key, best_text

with open('data/04.txt') as f:
    lines = f.readlines()

key, text = detect_single_character_xor(lines)
print("Key:", key)
print("Decrypted message:", text)
```

```
Key: 5
Decrypted message: Now that the party is jumping
```

Step 5 - Implement repeating-key XOR

> Here is the opening stanza of an important work of the English language:
>
> ```
> Burning 'em, if you ain't quick and nimble
> I go crazy when I hear a cymbal
> ```
>
> Encrypt it, under the key "ICE", using repeating-key XOR.
>
> In repeating-key XOR, you'll sequentially apply each byte of the key; the first byte of plaintext will be XOR'd against I, the next C, the next E, then I again for the 4th byte, and so on.
>
> It should come out to:
>
> ```
> 0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324272765272
> a282b2f20430a652e2c652a3124333a653e2b2027630c692b20283165286326302e27282f
> ```

[226]:
```python
# The repeating_key_xor function takes a text and a key as input and performs the XOR operation
# between the text and the repeating key. The de parameter controls whether the operation is
# for encryption (False) or decryption (True) for the final function which was giving a type error when
# calling this function. The function returns the result as a hexadecimal string.

def repeating_key_xor(text, key, de=False):

    encrypted_bytes = []
    for i in range(len(text)):
        if de:
            # if decrypting, convert byte from text and key to hex
            t_byte = format(text[i], '02x')
            k_byte = format(key[i % len(key)], '02x')
        else:
            # if encrypting, convert character from text and key to hex
            t_byte = text[i].encode('utf-8').hex()
            k_byte = key[i % len(key)].encode('utf-8').hex()

        # xor the two hex strings and append the result to the encrypted bytes
        encrypted_bytes.append(xor(t_byte, k_byte))

    # join the string
    return ''.join(encrypted_bytes)

text = "Burning 'em, if you ain't quick and nimble\nI go crazy when I hear a cymbal"
expected = '0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324272765272a282b2f20430a652e2c652a3124333a6
encrypted_string = repeating_key_xor(text, 'ICE')
print(encrypted_string)

if expected == encrypted_string:
    print('Correct')
```

```
0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a26226324272765272a282b2f20430a652e2c652a3124333a653e2b20276
30c692b20283165286326302e27282f
Correct
```

Step 6 (Main Step) - Break repeating-key XOR

There's a file here. It's been base64'd after being encrypted with repeating-key XOR.

Decrypt it.

Here's how:

- Let KEYSIZE be the guessed length of the key; try values from 2 to (say) 40.
- Write a function to compute the edit distance/Hamming distance between two strings. The Hamming distance is just the number of differing bits. The distance between:

`"this is a test"` and `"wokka wokka!!!"` is 37. Make sure your code agrees before you proceed.

- For each KEYSIZE, take the first KEYSIZE worth of bytes, and the second KEYSIZE worth of bytes, and find the edit distance between them. Normalize this result by dividing by KEYSIZE.
- The KEYSIZE with the smallest normalized edit distance is probably the key. You could proceed perhaps with the smallest 2-3 KEYSIZE values. Or take 4 KEYSIZE blocks instead of 2 and average the distances.
- Now that you probably know the KEYSIZE: break the ciphertext into blocks of KEYSIZE length.
- Now transpose the blocks: make a block that is the first byte of every block, and a block that is the second byte of every block, and so on.
- Solve each block as if it was single-character XOR. You already have code to do this. For each block, the single-byte XOR key that produces the best looking histogram is the repeating-key XOR key byte for that block. Put them together and you have the key.

Description

A brief description of your approach. Don't just put the code. First explain what you did and WHY you did it!

(your description)
...

[*]:
```python
# your code with comments
from itertools import combinations

# a lot of the "why" of what I did comes from the hint -- tried following it to the best to my degree
# Created the hamming distance function and used weightage of different byte values to create an average
# hamming distance for any given block size. Using that average size, we can create the most likely
# key size and utilize that to decrypt the text using the functions we utilized before.

def hamming_distance(s1, s2):
    count = 0
    # gets hamming distance by xor-ing the bits respectively and iterating the counter by 1 for each binary
    # difference
    for b1, b2 in zip(s1, s2):
        xor_result = b1 ^ b2
        count += bin(xor_result).count('1')
    return count

def get_weights():
    # create a dictionary of weights for all possible byte values
    weights = {'0': 0}
    for i in range(1, 256):
        weights[hex(i)[2:]] = bin(i).count('1')
    return weights

# normalize the average hamming distance for a given block size
def get_blocks(text, size):
    total_distance = 0
    # divide text into 4 equal sized chunks
    chunks = [text[i*size:(i+1)*size] for i in range(4)]

    # calculate hamming distance between all pairs of chunks and sum it up
    for i, s1 in enumerate(chunks[:-1]):
        for s2 in chunks[i+1:]:
            total_distance += hamming_distance(s1, s2)
    # calculate average hamming distance and normalize by diving the block size
    avg = total_distance / 4
    return avg/size

# find most liekly key size using the average normalized hamming distance
def find_keysize(text, num_guesses=4):
    scores = []
    for size in range(2, 41):
        score = get_blocks(text, size)
        scores.append((score, size))

    # sort scores and return top num_guesses key sizes
    scores.sort()
    return scores[:num_guesses]

# decrypt text with repeating key xor
def decrypt_repeating_key_xor(text, keysize):
    chunks = []
    # divide text into chunks with each chunk containing every keysize
    for i in range(keysize):
        chunk = text[i::keysize]
        chunks.append(chunk)

    decrypted = []
    combined_score = 0

    # find best key and decryption score for each chunk
    for chunk in chunks:
        guess = find_key_and_decrypt(chunk.hex())
        decrypted.append(guess)
        combined_score += guess[1]

    # combine keys from decrypted chunks to form the entire key
    key = b''
    for guess in decrypted:
        key += bytes([ord(guess[0])])

    return combined_score, key

def decrypt_cipher_with_repeating_key_xor():

    with open('data/06.txt', 'r') as f:
        f_contents = f.read().strip()

    # decode contents
    text = base64.b64decode(f_contents)
    # find likely key size
    keysizes = find_keysize(text)
    # decrtot text using each key and store results as candidates
    poss = []
    for _, guess in keysizes:
        decrypted = decrypt_repeating_key_xor(text, guess)
        poss.append(decrypted)
    # sort candidates by score
    poss.sort()
    best_poss = poss[0]
    best_key = poss[1]

    return repeating_key_xor(text, best_key, True).decode('ascii')

print(decrypt_cipher_with_repeating_key_xor())
```

NuSRZ [TB FI wN)tO]DHNy, DA_ bNCNIIYuG WOE IU] sTdXTBRLII E ENNXsE FLCUVYB[R@HIeT dY WH o IMKI 6bID SG y NhSI]N]O] @ YJ
wE SHJT 69~HIGI EY ZX 7}{EUOKI S6 i JIB F N n F:ESW LN@X N ~'+Q LL NOEIL^Oh^[LILQ U6~'x DOG HIGL^Fu^HK AA^ uNt eR *} L^T KX_:
BH AL@M
[HUOLBn IS AA^ Q } NeH SF TB ONC!+ vU C6?c_LF UD E PRD~ 6eHR]NSH{E GFNL_ ^ b ~ SHH GE JNY t SKAV +6=b i C N UOX uE
Y ^ V]x N`UTTL RDN*hS X:DZE MV IY rTc KE]NNNNNBDFI jG]Z TGHNI ~Td_S i
yT eNKS diIDio_ MHT SG y I~Y A UGM R 7!CDI KL]DHNx dYUE NTI L@DX :ES SIB_U] sT ,ZKY STGNAOCF{@Y O
E] VY 8Td uM SMNNs@JNDt sNWRS6 ,xBL@ [N dtIBON=X MHK_OE xSB,eHU
y nSCY NCI^NPSH_JxGE BA[+o w iHSI SRCNA TMDuE dOB_ S6 h TAPNSR
gUI SMNNcSFGR:lPVB%wNJ xSNmRCN UI] N IYcBRD T@ RUqT `SIG d{JGNTIUR^}C WOE]IY b NkNHO_ T HHTCYY TOA] 6 y NuSR [G
F ZDCIc HKBysq y i *#=NC[IOXu TBUD VHW 6 NKYUMH hJ I -n_{YHOBD XX ~ Nn]TEZNRO_ IOLhDIMC%nIY sS,RH] SV[N RE:FUMB s QN| x @
E] SHNDNPS xLBUKNW@H uXNE JA TAB STLr_ ONKJ @w xUD#7US R@WMN~ QF OAYD x NE SHF]N_NTIFI rSV MF]IHN~ i NT diI TDWOu\R FN
K MYrT i ^O\ CJ + :BR JY NHQ 7T7cI W
*%cNIId N{YFKLSHNF@TI :rs AA^ uNu ,HBLENSR dyNRO xDXZS]DH I,THT NII NSN t yJR TMJVM bTd_S DFR E J\O:JRG D@T HNt N]C#IyG^E S
UN:GEQNC\ CY y ,HH`-
Q JD BSOoYY -lJN H sT eHDH [TBOTE=] AI[O 6 iL BH
tH GH[RD
OTNN*rHHuFY RP YMSsT bX DFR E TL^{YY -y@O K x ,^FT] _ F
jtRHJJE @R a ~Y *#7US HNRZCn HKFT s K eT i]L ,U_ NYNR Y XFFD MSSqTd_S CF _ D BKNhR^LCY [OXNek SH@ UDG -7x{R
ZD:!pB^KUVRNw F SHDx{R @O MIU sTcE N QC TD _Dc [L *_V@ENb x AUG C F SHDlu KKNTJ CS :T c PH@ _ I Y
NxTFTOH@ LRNBy TOI[_ ^ oT-cQBF h MD RE6 LJE UO dF u SHH @^KX P^iB_ -HN bS sT b LL KNNHDFO {PB^[R@HNp gE M\SE HHSX xDE ^O
Z R] 6 TAPNSR dpMFD nC]W FZTJEN{ e_ a SR_ E KR^~NN IOX +l w