Step 1 - Convert hex to base64

> The string:
>
> `49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f697`
> 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.

In [ ]:
```
from base64 import b64decode, b64encode

def hextob64(string):
    output = b64encode(bytes.fromhex(string)).decode(); # convert string to by
    return output;
## function definition

string = '49276d206b696c6c696e6720796f757220627261696e206c696b65206120706f6973
b64 = hextob64(string)
if(b64 == 'SSdtIGtpbGxpbmcgeW91ciBicmFpbiBsaWtlIGEgcG9pc29ub3VzIG11c2hyb29t'):
    print("They match")
```

```
They match
```

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`

In [ ]:
```
from base64 import b16encode

str1 = '1c0111001f010100061a024b53535009181c'
str2 = '686974207468652062756c6c277320657965'
```

```
def fixedXOR(string1, string2):
    decode1 = bytes.fromhex(string1) # convert hex string to bytes
    decode2 = bytes.fromhex(string2) # convert hex string to bytes
    decode_output = bytes(a ^ b for a,b in zip(decode1, decode2)) # run throug
    result = b16encode(decode_output).decode() # fix formatting and go back to
    return result

test = fixedXOR(str1, str2)
if(test.lower() == '746865206b696420646f6e277420706c6179'): # need a lower for
    print("They match")
```

```
They match
```

Step 3 - Single-byte XOR cipher

> The hex encoded string:
>
> 1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b373
> ... 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!

To begin, I decided to look for a scoring system. There were a lot on the internet that involved using ETAOIN SHRDLU and that was the most common one. I found another one that used a scoring system that scored a string based on how English it was. It gave +2 points for each occurrence of the 6 most common English characters (and space), +1 points for each occurrence of the next 6 most common English characters, +1 points for each occurrence of 20 most common English Bigrams, +1 points for each occurrence of 20 most common English Trigrams, +1 points for each occurrence of 298 most common English words. Another approach I saw was a fitting quotient analysis using Letter Frequency which is essentially an expansion of the first 12 most commonly used letters (aka ETAOIN SHRDLU). I ended up going with the fitting quotient since I thought it was cool and efficient and the scoring system was a very long implementation and I thought I could achieve great efficiency with the expanded ETAOIN SHRDLU without having to implement a lengthy

scoring system. I implemented the fitting quotient and then I created a decrypt function. I began by taking the string and turning into bytes from hex. I then brute forced since I only had to iterate through 256 possible values since it was a simple byte XOR. I would then XOR all the bytes in the string by all 256 values. Through each iteration of the for loop, I would calculate the score using the fitting quotient and keep the highest score throughout all the iterations. I then returned the key and the message.

...

In [ ]:
```
########################## scoring system 1 ################################
from collections import Counter
occurance_english = {
    'a': 8.2389258,    'b': 1.5051398,    'c': 2.8065007,    'd': 4.2904556,
    'e': 12.813865,    'f': 2.2476217,    'g': 2.0327458,    'h': 6.1476691,
    'i': 6.1476691,    'j': 0.1543474,    'k': 0.7787989,    'l': 4.0604477,
    'm': 2.4271893,    'n': 6.8084376,    'o': 7.5731132,    'p': 1.9459884,
    'q': 0.0958366,    'r': 6.0397268,    's': 6.3827211,    't': 9.1357551,
    'u': 2.7822893,    'v': 0.9866131,    'w': 2.3807842,    'x': 0.1513210,
    'y': 1.9913847,    'z': 0.0746517
}
dist_english = list(occurance_english.values())
def computing_fit_quotient(text: bytes) -> float:
    counter = Counter(text)
    dist_text = [
        (counter.get(ord(ch), 0) * 100) / len(text)
        for ch in occurance_english
    ]
    return sum([abs(a - b) for a,b in zip(dist_english, dist_text)]) / len(dis
###########################################################################

########################## decryption ######################################
def decrypt(string):
    true_message, key, minscore = None, None, None
    str1b = bytes.fromhex(string)
    for k in range(256):
        message = bytes([b ^ k for b in str1b])
        score = computing_fit_quotient(message)
        if(minscore is None or score < minscore):
            true_message, key, minscore = message, k, score
    return true_message, k
###########################################################################

string = '1b37373331363f78151b7f2b783431333d78397828372d363c78373e783a393b3736

message, k = decrypt(string)

print("Key is:", k)
print("Phrase is: ", message)
```

```
Key is: 255
Phrase is:  b"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!

To begin, I transferred the 04.txt file into an array so I can iterate through each line. This code is essentially similar to Part 3. I iterate through each line and I use the decrypt function from part 3 to generate the message and the key for each line. I would then use the message found from each line and calculate a score using the fitting quotient and check if it is lower than the current minimum score. I would then iterate through all the 60 character lines and keep the message and the key. I would then return the key and message after all the iterations.)
...

In [ ]:
```
with open('04.txt') as file:
    lines = [line.rstrip('\n') for line in file]
min_score, real_message, key = None, None, None
for line in lines:
    message, k = decrypt(line)
    score = computing_fit_quotient(message)
    if(min_score is None or score < min_score):
        min_score, real_message, key = score, message, k
print("Key is: ", key)
print("The message is: ", real_message)
```

```
Key is:  255
The message is:  b'Now that the party is jumping\n'
```

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:

```
0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a262263242
a282b2f20430a652e2c652a3124333a653e2b2027630c692b20283165286326302e
```

In [ ]:
```
input = "Burning 'em, if you ain't quick and nimble\nI go crazy when I hear a
key = 'ICE'

def repeatXOR(string, key_string):
    input = bytes(string, 'utf-8') # convert the input string into a bytes
    key = bytes(key_string, 'utf-8') # do the same thing for the key
    bruh = [] # empty array
    for i in range(len(input)):
        bruh.append(input[i] ^ key[i % len(key)]) # iterate through all of the
                                                  # with the i mod length so mak
    return bytes(bruh) # put it back as hex string

output = repeatXOR(input, key)
bruh1 = b16encode(bytes(output)).decode()

if(bruh1.lower() == '0b3637272a2b2e63622c2e69692a23693a2a3c6324202d623d63343c2a
    print('They match!') # ensure that our result matches the expected output
```

```
They match!
```

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)

...

In [ ]:
```
from itertools import combinations, zip_longest

with open('06.txt') as file:
    message = b64decode(file.read())



def hammingDistance(input1, input2):
    distance = 0
    #input1 = bytes(string1, 'utf-8')
    #input2 = bytes(string2, 'utf-8')
    for(byte1, byte2) in zip(input1, input2):
        distance += bin(byte1 ^ byte2).count('1')
    return distance

input1 = bytes("this is a test", 'utf-8')
input2 = bytes("wokka wokka!!!", 'utf-8')
if(hammingDistance(input1, input2) == 37):
   print("It is 37!")

def findLength():
    minscore = len(message)
    for KEYSIZE in range(2,40):
        KEYSIZE_CHUNK = [message[i:i+KEYSIZE] for i in range(0, len(message), K
        subchunk = KEYSIZE_CHUNK[:4]
        average_score = (sum(hammingDistance(a,b) for a,b in combinations(subch
        if average_score < minscore:
            minscore = average_score
            key_length = KEYSIZE
    return key_length

key_size = findLength()
print("Key size is:", key_size)

def XOR_decode_bytes(encoded_array):
    last_score = 0
    greatest_score = 0
    for n in range(256): # checks for every possible value for XOR key
        xord_str = [byte ^ n for byte in encoded_array]
```

```
        xord_ascii = ('').join([chr(b) for b in xord_str])
        last_score = computing_fit_quotient(xord_ascii)
        if (last_score > greatest_score):
            greatest_score = last_score
            key = n
    return key


def decrypt1(str1b):
    true_message, key, minscore = None, None, None
    #str1b = bytes.fromhex(string)
    for k in range(256):
        message = bytes([b ^ k for b in str1b])
        score = computing_fit_quotient(message)
        if(minscore is None or score < minscore):
            true_message, key, minscore = message, k, score
    return k

def findKey():
    blocks = [message[j:j + key_size] for j in range(0, len(message), key_size
    key = []
    xorBlocks = [list(filter(None, k)) for k in zip_longest(*blocks)]
    for block in xorBlocks:
        key_n  = decrypt1(block)
        key.append(key_n)
    ascii = ''.join([chr(c) for c in key])
    return ascii

def repeatXOR1(input, key_string):
    #input = bytes(string, 'utf-8') # convert the input string into a bytes
    key = bytes(key_string, 'utf-8') # do the same thing for the key
    bruh = [] # empty array
    for i in range(len(input)):
        bruh.append(input[i] ^ key[i % len(key)]) # iterate through all of the
                                                  # with the i mod length so make
    return bytes(bruh) # put it back as hex string
```

```
It is 37!
Key size is: 29
```

In [ ]: