Question 1:

Plaintext is:

A Polar Explorer

All the huskies are eaten. There is no space
left in the diary, And the beads of quick
words scatter over his spouse's sepia-shaded face
adding the date in question like a mole to her lovely cheek.
Next, the snapshot of his sister. He doesn't spare his kin:
what's been reached is the highest possible latitude!
And, like the silk stocking of a burlesque half-nude
queen, it climbs up his thigh: gangrene.

Joseph Brodsky

Key is: 2brodsky

The technique for determining the key was exploiting the known vulnerability in viginere ciphers in which bytes will repeat. We take the lowest common factor of all the repeating bytes found, and that will be our key length.

Our strategy for part 1 was based on the assumption that the entirety of the file will decrypt to human-readable ASCII with the key. Therefore if a character yields a single non-readable decryption in the entire file, we can throw it out as a potential key character in that position. This left an incredibly small amount of possible keys fitting that criteria (there were 3). We simply present those three keys to a human and have the human decide which plaintext fits his worldview better.

Compressing a file would have caused my algorithm to fail. My code assumes the plaintext is uncompressed, readable ASCII. While there may be a compression algorithm that stores all the resultant file in printable ASCII, I do not know of it. Gunzip and regular zip both yield some characters not within readable-ASCII, so my program would find no possible keys.

QUESTION 2:

The key is Large_Hadron_Collider_at_CERN_map. The plaintext is a PDF showing exactly what the key says it does. To view this, you'll have to run our software.

The technique used to decrypt this file is for the most part, identical. Based on the knowledge that it was a common file format, I assumed there would be a binary blob in the middle, but it would be preceded by a header and a trailer. The header should be mostly readable ASCII for common file formats. Since this was a PDF, my assumption was correct. I simply look solely at about 500 characters from the beginning of the file and see which key characters decrypt the most cipher bytes into printable ASCII (this is expressed as a decimal, successful decryptions divided by total attempts). The key characters with the highest percentage of successful decryptions were my most likely potential keys.

I then made the assumption that the key would make sense, as in words. Humans are really good at looking at a string of characters and putting them together in words, so since there were about three character options for each position in the key, I simply display them five positions at a time, and let a human CHOOSE what looks like words. They can then try the decryption with that key.

This obviously is not a programmatic way of assuring the correct key was found but I think humans are pretty cool and don't need no programs to do their jobs for them.

IMPORTANCE OF KNOWN PLAINTEXT:

For both of my decryption programs, the structure of the plaintext must be known for my program's assumptions to hold. Part 1 required the knowledge that the plaintext was completely readable ASCII and part 2 required the knowledge that there was plaintext surrounding a binary blob of data. It would be possible to solve without known plaintext structure, but ultimately more difficult.

QUESTION 3:
Encryption key: group4

1) Inspect the contents of the .enc files.
a) Sizes
Plaintext: 80 bytes
ECB and CBC: 88 bytes
CFB and OFB: 80 bytes
CFB and OFB are the same length of the plaintext because they do not use padding. They do not need to be
padded, even if the blocks are not a multiple of 8 bytes.
ECB and CBC are longer because they have an extra block of padding at the end. While in this case the cipher
files did not need to be padded because they are both multiple of 8 bytes, the padding is most likely created
so when decrypting, it is known all blocks have been recieved.

b) Patterns
ECB: Identical ciphertext blocks. This is because ecb divides the plaintext into 8 byte blocks and encrypts each separately. But each block is encrypted with the same key, so trying to encrypt blocks
of plaintext that are the same produces identical ciphertext blocks.

CBC: No patterns or similarities to ciphertext encrypted with other forms of DES.

CFB and OFB: No patterns in individual ciphertext files. However the first 8 bytes of both the files had
the same ciphertext. This is because they are both initially encrypted in the same way, but the next 8 bytes are
XORed at different points during encryption.

2) Impact of "error" on decryption.
ECB: Entire 3rd block decrypted is decrypted incorrectly. Only this block is affected because each block is
decrypted separately.

CBC: The entire 3rd block was decrypted incorrectly. The 27th character was also decrypted incorrectly.
The 3rd block is corrupted because the error is in this block, so during decryption with the key
the bytes are rearranged incorrectly. The 27th byte gets corrupted as well because the 4th block
is XORed with the 3rd block during decrpytion. However, because the 3rd block has not been decrypted yet, only the
19th byte has the error. This is why only the 27th byte is affect, instead of the entire 4th block.

CFB: The 4th block was decrypted incorrectly as well as the 19th character. To decrypt the 3rd block, it's cipher text
is XORed with the 2nd block that has been decrypted with the key. Since it is only XORed and not decrypted with the key
yet, only the 19th byte is corrupted. The entire 4th block however comes out corrupted because the 3rd block cipher text
is decypted with the key and then XORed with the 4th block cipher text. Decrypting the 3rd block with the key rearranges
the bytes in the block, so this corrupted block is being XORed with block 4.

OFB: Only the 19th character was incorrect. This is because each block is only XORed with the initialization vector, which
is decrypted with the key before each XOR. Because it is only XORed and not decrypted with the key, only the 19th byte is
corrupted.

3) Ciphertext3 observations.
The length of the file is 288 bytes, which is not a multiple of 40. There is an extra block (8 bytes) of padding added to the
length. The actual 280 bytes of the plaintext is divisible by 40 since 32 bytes on each line are reserved for the name and

the last 8 are for the salary. This means there are 7 employees in the file. We figured this out by 280 divided by 40.

SOURCES:

PHP.net helped me understand that manipulating binary bytes as their string representation of 1's and 0's was altogether easier than bit shifting.

Moneer told me that viginere ciphers have repeating bytes.

All group members contributed equally.

Mad props to my cat for sticking with me through the tough times. My girlfriend also made me supper when I was hungry and working late into the night.

WORKLOAD:

I think the workload was high. Probably too high. I have a very small course load this semester and had a lot of fun doing it, but it took my several hours of throwing things at a wall before something stuck. I worked well over the lab hours. I think more guidance would have taken away the fun, but I think without the guidance the workload is simply too high for this assignment.

I don't envy your position in dealing with this information.