# Part 1.

## Key:

**_LitFuT_**

## Plain text:

```
The legend lives on from the Chippewa on down
Of the big lake they called 'gitche gumee'
The lake, it is said, never gives up her dead
When the skies of November turn gloomy
With a load of iron ore twenty-six thousand tons more
Than the Edmund Fitzgerald weighed empty
That good ship and crew was a bone to be chewed
When the gales of November came early
The ship was the pride of the American side
Coming back from some mill in Wisconsin
As the big freighters go, it was bigger than most
With a crew and good captain well seasoned
Concluding some terms with a couple of steel firms
When they left fully loaded for Cleveland
And later that night when the ship's bell rang
Could it be the north wind they'd been feelin'?
The wind in the wires made a tattle-tale sound
And a wave broke over the railing
And every man knew, as the captain did too,
T'was the witch of November come stealin'
The dawn came late and the breakfast had to wait
When the gales of November came slashin'
When afternoon came it was freezin' rain
In the face of a hurricane west wind
When suppertime came, the old cook came on deck sayin'
Fellas, it's too rough to feed ya
At seven pm a main hatchway caved in, he said
Fellas, it's been good t'know ya
The captain wired in he had water comin' in
And the good ship and crew was in peril
And later that night when his lights went outta sight
Came the wreck of the Edmund Fitzgerald
Does any one know where the love of God goes
When the waves turn the minutes to hours?
The searches all say they'd have made Whitefish Bay
If they'd put fifteen more miles behind her
They might have split up or they might have capsized
They may have broke deep and took water
And all that remains is the faces and the names
Of the wives and the sons and the daughters
Lake Huron rolls, superior sings
In the rooms of her ice-water mansion
Old Michigan steams like a young man's dreams
The islands and bays are for sportsmen
And farther below Lake Ontario
Takes in what Lake Erie can send her
And the iron boats go as the mariners all know
With the gales of November remembered
In a musty old hall in Detroit they prayed,
In the maritime sailors' cathedral
The church bell chimed till it rang twenty-nine times
For each man on the Edmund Fitzgerald
The legend lives on from the Chippewa on down
```

```
Of the big lake they call 'gitche gumee'
Superior, they said, never gives up her dead
When the gales of November come early
```

## To determine the key:

*Step 1: Find key length*

Using Index of coincident (https://en.wikipedia.org/wiki/Index_of_coincidence)
Implementation:
(https://crypto.stackexchange.com/questions/35318/cryptanalysis-of-xor-cipher-with-repeated-key-phrase)

```
for step in range(1, 50):
    match = total = 0
    for i in range(len(sequence)):
        for j in range(i + step, len(sequence), step):
            total += 1
            if hex_sequence[i] == hex_sequence[j]: match += 1

    ioc = float(match) / float(total)
    print("%3d%7.2f%% %s" % (step, 100*ioc, "#" * int(0.5 + 500*ioc)))
```

Result shows length most likely be 8,16,24,32,40,48. These are all multiples of 8, which suggest key length most likely be 8.

*Step 2: Frequency analysis*

Implementation:

```
def get_freq(position, character, length):
    freq = {c: 0 for c in string.ascii_lowercase}
    for i in range(position, len(sequence), length):
        try:
            freq[chr(sequence[i][character]).lower()] += 1
        except:
            freq[chr(sequence[i][character]).lower()] = 1

    plt.bar(sorted(list(freq.keys())), list(freq[c] for c in
sorted(list(freq.keys()))), color='g')
    plt.title(str(position) + character)
    plt.show()

while True:
    pos = int(input('> '))
    for char in range(32, 127):
        get_freq(pos, chr(char), 8)
```

The program based on a list of dictionaries:

sequence[ index ] = {key1: plaintext1, key2: plaintext2 … }

- On index i is a dictionary of all possible keys corresponding to what the plaintext that would be deciphered using that key at byte $i^{th}$. This is done to save time on frequency analysis later on.

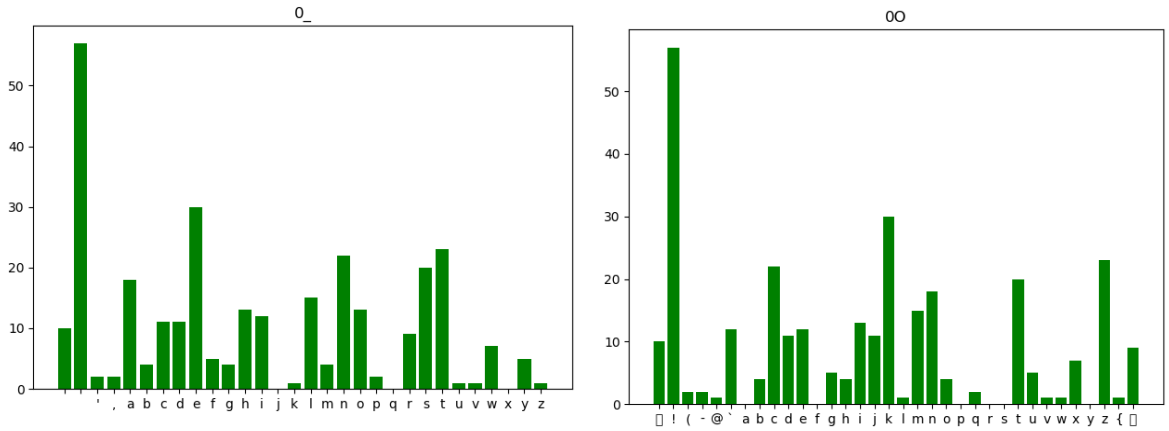
Eg: sequence[0] = {‘a’ : ‘x’, ‘b’: ‘d’ … } meaning if key[0] = ‘a’ then plaintext[0] is ‘x’, if key[0] = ‘b’ then plaintext[0] = ‘d’ …

- The frequency analysis is done by entering an index of the key (0-7 in this case) and the program will generate frequency graph of plaintext deciphered by that key index.

Eg: if position = 3 then indexes to be deciphered are 3, 11, 19, 27, … (because key length = 8)

- The correct character of the key should generate a diagram of only printable characters, and the frequency should follow English character frequency.

Eg: Position 0, the correct key is ‘_’ (left) vs incorrect (‘O’) on the right:




With correct key, all deciphered characters are readable, and the frequency of A-Z follow English character frequency. While incorrect key will decipher some unreadable characters (first and last in right chart), and frequency does not follow English frequency.

*Step 2.5: Automation*

Alternatively, this process can be automated:

```
def get_freq(position, character, length):
    freq = {c: 0 for c in string.ascii_lowercase}
    for i in range(position, len(sequence), length):
        try:
            freq[chr(sequence[i][character]).lower()] += 1
        except:
            freq[chr(sequence[i][character]).lower()] = 1
    if all( 32 <= ord(i) < 127 or ord(i) == 10 for i in freq.keys()): return character
    return False

key = ''
for pos in range(8):
    for char in range(32, 127):
        res = get_freq(pos, chr(char), 8)
        key += res
print(key)
```

Although this implementation is not recommended as it only check for readable characters and not their frequency.

# Part 2

## Key:

[95wJL4PiE)7u^P-Q(%^-_254dh1F@@nnE128eRo

## Plaintext:

<attached p2.zip>

## To determine key:

*Step 1: Find key length*

This should be done similar to part 2, although since the file is longer and the key is longer, step should be increased to capture broader range, and search range should be limited to reduce search time.

```
search_len = 1000
for step in range(1, 100):
    match = total = 0
    for i in range(search_len):
        for j in range(i + step, search_len, step):
            total += 1
            if hex_sequence[i] == hex_sequence[j]: match += 1

    ioc = float(match) / float(total)

    print("%3d%7.2f%% %s" % (step, 100*ioc, "#" * int(0.5 + 500*ioc)))
```

Result shows length most likely to be 40 or 80. Since 80 is a multiple of 40, more likely 40 is key length.

*Step2: Determine file type*

To get a dictionary of file type from Wikipedia table (https://en.wikipedia.org/wiki/List_of_file_signatures), run from console:

```
let sig_data = {};

$('.wikitable tr').each(function() {
    let offset = $(this).find($(this).children()[2]).html();
    let cell0 = $(this).find($(this).children()[0]);
    let type = $(this).find($(this).children()[3]).html();
    cell0.find($('pre')).each( function() {
        sig_data[$(this).html()] = {
            signature: $(this).html(),
            offset: offset.split('<p>any</p>').join(''),
            type: type
        }
    } );
});

console.log(JSON.stringify(sig_data));
```

The list of dictionaries is kept the same from previous part. This is done to quickly lookup key positions in later parts:

sequence[ index ] = {key1: plaintext1, key2: plaintext2 … }

- On index i is a dictionary of all possible keys corresponding to what the plaintext that would be deciphered using that key at byte $i^{th}$.

Eg: sequence[0] = {'a' : 'x', 'b': 'd' … } meaning if key[0] = 'a' then plaintext[0] is 'x', if key[0] = 'b' then plaintext[0] = 'd' …

To determine file type, the program try to look for a key that can decrypt to header of a file type.

Eg: If header is 50 4B 03 04:

- Look for key in sequence[0] such that sequence[0][key] = 50
- Look for key in sequence[1] such that sequence[1][key] = 4B
- Look for key in sequence[2] such that sequence[2][key] = 03
- Look for key in sequence[3] such that sequence[3][key] = 04

The search found these possible headers:

| | | |
|---|---|---|
| 00 | PIC,PIF,SEA,YTR | U |
| 50 4B 03 04 | zip,jar,odt,ods,odp,docx,xlsx,pptx,vsdx,apk,aar | [95w |
| EF BB BF | <unknown> | c2z |
| FF FB | mp3 | f> |
| 4E 45 53 1A | nes | 1I;f |
| 1F 8B | gz,tar.gz | D5 |
| 43 57 53 | swf | !s; |
| 53 45 51 36 | rc | +I[T |

Of these types, zip, mp3, and gz are 3 common headers. Start with ZIP because it has the longest header, footer, which make it most feasible to decrypt.

- Current key [95w □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□

*Step 3: Match additional signature*

ZIP file config: http://www.fileformat.info/format/zip/corion.htm

- Look at "End of central directory structure":
  - The structure starts with 50 4B 05 06
  - Since the central directory is at the end of the file, we can limit search to last 1000 bytes:

```
dt = {i:{} for i in range(40)}
for i in range(len(hex_sequence)-1000,len(hex_sequence)-3):
    key_string = ''
    for key in sequence[i]:
        if sequence[i][key] == int('50', 16):
            key_string += key
    for key in sequence[i+1]:
```

```
        if sequence[i+1][key] == int('4b', 16):
            key_string += key
    for key in sequence[i+2]:
        if sequence[i+2][key] == int('05', 16):
            key_string += key
    for key in sequence[i+3]:
        if sequence[i+3][key] == int('06', 16):
            key_string += key
    if len(key_string) == 4:
        try:dt[i%40][key_string].append(i)
        except: dt[i%40][key_string] = [i]
for i in sorted(dt.keys()):
    for key in sorted(dt[i].keys(),key=lambda x: len(dt[i][x])):
        print('0506', i, key, dt[i][key])
```

Of all results found, index 131882 (mod 40 = 2) give key = 5wJL coincide with 5w decided with the header.

- Current key = [95w JL□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□ □□□□

Offset 20 from "50 4B 05 06" at 131882 is comment length. Since there is no comment at the end of file, this should be 0. Searching for 00 00 at 131902 returns "25" at position 22, 23:

- Current key = [95w JL□□ □□□□ □□□□ □□□□ □□25 □□□□ □□□□ □□□□ □□□□

Offset 4 from "50 4B 05 06" at 131882 is *Number of disk* = 0 (https://stackoverflow.com/questions/48726961/what-are-disks-in-this-context-of-the-structure-of-zip-files), size = 1 word = 2 bytes. Searching 00 00 at 131886 gives "4P" as key.

- Current key = [95w JL4P □□□□ □□□□ □□□□ □□25 □□□□ □□□□ □□□□ □□□□

Offset 6 from "50 4B 05 06" at 131882 is *Number of disk* with start of central directory = 0. Searching 00 00 at 131888 gives "iE" as key.

- Current key = [95w JL4P iE□□ □□□□ □□□□ □□25 □□□□ □□□□ □□□□ □□□□

- Look at "Central directory structure":
    - Central directory should be not too far from End of central directory.
    - Searching 200 bytes before 131882:

```
dt = {i:{} for i in range(40)}
for i in range(131682,131882):
    key_string = ''
    for key in sequence[i]:
        if sequence[i][key] == int('50', 16):
            key_string += key
    for key in sequence[i+1]:
        if sequence[i+1][key] == int('4b', 16):
            key_string += key
    for key in sequence[i+2]:
```

```
        if sequence[i+2][key] == int('01', 16):
            key_string += key
    for key in sequence[i+3]:
        if sequence[i+3][key] == int('02', 16):
            key_string += key
    if len(key_string) == 4:
        try:dt[i%40][key_string].append(i)
        except: dt[i%40][key_string] = [i]
for i in sorted(dt.keys()):
    for key in sorted(dt[i].keys(),key=lambda x: len(dt[i][x])):
        print('0102', i, key, dt[i][key])
```

Of all results found, 131823 (mod 40 = 23) give key = 54dh. Some other possible assignments:

VyAH [131854 mod 40 = 14], .qNM [131857 mod 40 = 17], TyOR [131859 mod 40 = 19]. Since length of Central directory is minimum 0x2E (46), all start indexes after 131836 are not feasible.

- Current key = [95w JL4P iE□□ □□□□ □□□□ □□25 4dh□ □□□□ □□□□ □□□□

Offset 34 (from “50 4B 01 02” at 131823) is Disk number. Searching 00 00 at 131857, 131858 gives “(%” at position 17, 18:

- Current key = [95w JL4P iE□□ □□□□ □(%□ □□25 4dh□ □□□□ □□□□ □□□□

Offset 36 (from “50 4B 01 02” at 131823) is Internal file attribute. Searching for 00 00 at 131859, 131860 gives result “^-“ at position 19, 20:

- Current key = [95w JL4P iE□□ □□□□ □(%^ -□25 4dh□ □□□□ □□□□ □□□□

Offset 5 (from 504B 01 02) is host OS. Try common OS (0 = Windows, 3 = Linux, 7 = MacOS) at 131828, only 0 return “F” at 28:

- Current key = [95w JL4P iE□□ □□□□ □(%^ -□25 4dh□ F□□□ □□□□ □□□□

Offset 6 (from “50 4B 01 02” at 131823) is minimum version needed to extract. This should be the same bit as offset 4 from header. Since position 4 of the key is found, searching 14 at 131829 gives “@” at position 29.

- Current key = [95w JL4P iE□□ □□□□ □(%^ -□25 4dh□ F@□□ □□□□ □□□□

Offset 7 (from “50 4B 01 02” at 131823) is Target OS. Try common OS at 131830, only 0 return “@” at 30:

- Current key = [95w JL4P iE□□ □□□□ □(%^ -□25 4dh□ F@@□□ □□□□ □□□□

Offset 8 (from “50 4B 01 02” at 131823) is general purpose bits. Searching for 00 00 at 131831 gives “nn” at position 31, 32:

- Current key = [95w JL4P iE□□ □□□□ □(%^ -□25 4dh□ F@@n n□□□ □□□□

Offset 10 (from “50 4B 01 02” at 131823) is compression method. This should be the same as offset 8 from header section. Since position 8 of the key is known, searching for 00 00 at 131833 returns “E1” at position 33, 34:

- Current key = [95w JL4P iE□□ □□□□ □(%^ -□25 4dh□ F@@n nE1□ □□□□

Offset 30 (from “50 4B 01 02” at 131823) is Length of File comment. Since we have found “End of central directory” and there is not enough space at the end of file for comments, we know length of comment is 0. Searching for 00 00 at 131855 returns “-Q” at position 15 16:

- Current key = [95w JL4P iE□□ □□□- Q(%^ -□25 4dh□ F@@n nE1□ □□□□

Offset 32 (from “50 4B 01 02” at 131823) is Extra field length. Since compression method is known (Stored/ No compression), this is most likely 00 00 (https://en.wikipedia.org/wiki/Zip_(file_format)#Extra_field). Searching for 00 00 at 131853 returns “^P” at position 13, 14:

- Current key = [95w JL4P iE□□ □^P- Q(%^ -□25 4dh□ F@@n nE1□ □□□□

Offset 28 (from “50 4B 01 02” at 131823) is Length of file name. Since we know length of comment and extra field are both 0, File name length is the difference between start and end of central directory (= 13 or 0x0D). Searching 0D 00 (little endian) at 131851 returns “7u” at position 11, 12:

- Current key = [95w JL4P iE□7 u^P- Q(%^ -□25 4dh□ F@@n nE1□ □□□□

The next steps are based on the fact that the file is compressed using stored method. File content should be the same as in the original file.

Try decode the file with current key, replace all unknown with a character (eg: ?). Look at offset 147, in the context “ht□p://www”, □ should be “t”. Searching for 74 (ASCII for t) at 147 returns “1” at position 27:

- Current key = [95w JL4P iE□7 u^P- Q(%^ -□25 4dh1 F@@n nE1□ □□□□

Repeat the process with updated key. Look at offset 131770 (0x202ba), in the context “linear-gra□ient”, □ should be “d”. Searching for 64 (ASCII for d) at 131770 returns “)” at position 10:

- Current key = [95w JL4P iE)7 u^P- Q(%^ -□25 4dh1 F@@n nE1□ □□□□

Look at offset 131381 (0x20135), in the context “<~ath d="M7”, “~” should be “p”. Searching for 70 at 131381 returns “_” at position 21:

- Current key = [95w JL4P iE)7 u^P- Q(%^ -_25 4dh1 F@@n nE1□ □□□□

Look at offset 131515 (0x201bb), current plaintext read “x-blend-mod□□□□□tiply”. A similar text can be seen at 131728 (0x20290): “x-blend-mode:multiply”. Searching for hex ASCII sequence of “e:mul” at 131515 returns “28eRo” at position 35:

- Current key = [95w JL4P iE)7 u^P- Q(%^ -_25 4dh1 F@@n nE12 8eRo

*Step 4: Verify key:*

Decrypt the file with the key found: [95wJL4PiE)7u^P-Q(%^-_254dh1F@@nnE128eRo

```
def decipher(key):
    fn = 'p2.zip'

    with open(fn, 'wb') as out_file:
        for i in range(len(sequence)):
            out_file.write(bytes([sequence[i][ key[i%len(key)] ] ]))
    return fn

decipher('[95wJL4PiE)7u^P-Q(%^-_254dh1F@@nnE128eRo')
```

Verify with `unzip -t p2.zip`:

```
Archive:  p2.zip
    testing: LLVM_Logo.svg                    OK
No errors detected in compressed data of p2.zip.
```

## Differences from Part 1:

- Since this is not a standard text file, frequency analysis was not used.
- Since file is longer, pickle module was used to reuse data and save computation time.
- Automation can be used in the bits found from file signatures:
    - Determining file type can be done automatically: Output possible file types
    - Once selected file type (zip in this case), finding additional key position given file structure can be done automatically (Central directory, end of central directory…)
    - The last part requires reading through the half-decrypted file and guessing the most probable plaintext cannot be (easily) done automatically. This can be replaced with

searching through all possible remaining key positions (95^8 in this case) and verifies with `unzip -t`.

- Automation in this part should make the code more complex and harder to read, therefore not implemented.

# Part 3

## File creation:

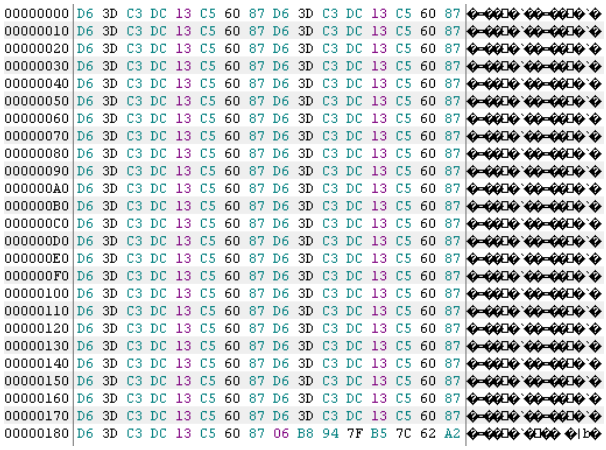
- Create 3 text files p31, p32, p33.
- Run cmd_salt.py and cmd.py and follow instruction.

## Size comparison:

- CBC: Encrypted file size is larger. Encrypted size is 400 bytes compare to 392 bytes of plaintext. When increase plaintext size to 399 encrypted size remain unchanged. However, when increase to 400 encrypted size increase to 408. This suggest CBC requires at least 1 random byte, at most 8.
- CFB: Encrypted file size is unchanged. Encrypted size changes as plaintext size changes. This suggest CFB might be a stream cipher
- ECB: Encrypted file size is larger. This behave similar to CBC regarding size changes.
- OFB: Encrypted file size is unchanged. This behave similar to CFB.

## Pattern:

- CBC: No clear pattern visible. This is because CBC encrypt each block with the encrypted content of previous block.
- CFB: No clear pattern. Likely previous block cipher result is used as key on next block.
- ECB: Clear pattern on repeating files (file 1 and file 2). This is because each block in ECB is encrypted the same way (previous block not used in encryption of next block).




- OFB: No clear pattern. Likely previous block cipher result is used as key on next block.

## Error impact:

- All error files were created by flipping byte at offset 71.
- Impact of no salt error:
    - CBC: Affected byte 64 -> 71 and 79 on all three files.
    - CFB: Affected byte 71 -> 79 on all three files.
    - ECB: Affected byte 64 -> 71 on all three files.
    - OFB: Affected byte 71 on all three files.

- Comment:
    - All affected bytes are in the range between 64 and 79. This is the 8 bytes block containing byte 71 (64 -> 71) and the 8 bytes block after it.
    - Except for OFB only affecting the byte that was flipped, which suggest each byte is decrypted separately, the rest either affect block that contain flipped byte (64 -> 71) or the next block (72 -> 79), suggest the decryption of a block has something to do with its neighboring block.

- Impact of salted error:
    - CBC: Affected byte 48 -> 55 and 63 on all three files.
    - CFB: Affected byte 55 -> 63 on all three files.
    - ECB: Affected byte 48 -> 55 on all three files.
    - OFB: Affected byte 55 on all three files.
- Comment:
    - The effect on salted encryption is quite similar to no salt encryption, if the flipped byte is 16 bytes before it.
    - An explanation for this could be because the salt was added at the front of the encrypted file, any byte in the encrypted file should be corresponding to a lower offset byte in the plaintext.
    - In this case it looks like there are 16 bytes of salt before position 71 in the encrypted file, so when byte 71 is flipped it is byte 55 in the plaintext.

## ECB class name list

- Run p3.py. Result p3.out attached.
- Since ECB encrypt each block the same way (block orders have no effect on encryption/decryption), to re order the groups, we only need to divide the file into 12 parts and rearrange the parts.

Contribution:
Minh Nguyen: Part1, Part2, Part 3
Xinyu Chen: