BINUS UNIVERSITY INTERNATIONAL

Assignment Cover Letter (Group Work)

Student Information:	***Surname***	***Given Names***	***Student ID Number***
	1. Chandra	**Jason Christopher**	**2101725033**
	2. Vallenchee	**Jose**	**2101718482**
	3. Joyan	**Vincent**	**2101699672**

Course Code	**:COMP6340**	**Course Name**	**: Analysis of Algorithm**
Class Major	**:L3BC :Computer Science**	**Name of Lecturer(s)**	**: 1. Ms. Maria Seraphina**
Title of Assignment : (if any)			
Type of Assignment	**:Final Report Paper**		
Submission Pattern			
Due Date	**:15th January 2019**	**Submission Date**	**:15th January 2019**

1. Introduction

- Background

  Nowadays,Everyone saves most of their documents in a digital way. This is good because it means that they don't have to carry physical documents with them and this means that they can view their documents anywhere in the world. But, as we all know digital files will take up a lot of memory space.

  For example, in the 2017 Panama Papers leak, 13.4 million secret documents were leaked to the public. These documents took up to 2 terabytes of data therefore we believe that compressing these file could have saved up a lot of memory space.

  So, For our analysis of algorithms final project, we decided to compare the results between 2 lossless compression algorithms which can be used to compress text files.

- Problems

  While a single text file might be small in file size, we all know that if you have more files, more space will be used.

- Aims

  To efficiently compress text files without losing any data..

- Benefits

  To reduce amount of space used in electronics for text files.

- Features
  - Allows the user to choose between 2 lossless algorithms
  - Allows the user to choose to compress or decompress.
  - Create output file after compression and decompression.

2. Related Work

7zip, winrar, winzip are a few programs that execute lossless compression

3. Implementation

- Formal description of the problem

  As more people use text files, we wanted to reduce the amount of space that text files take up by finding the most efficient lossless compression algorithm.

- Design of algorithm

In this project, we implemented 2 algorithms:

- **Lempel-Ziv (LZ-77)**
- Used to losslessly compress and decompress text files
- **Lempel-Ziv-Welch**
- Used to losslessly compress and decompress text files

- Proof of correctness

Compare the output file from the compression and the output file from decompression with the original text file.

Compression Time Complexity in milliseconds

N (Total Words)	LZ-77	Lempel-Ziv-Welch
500	660	10
1000	1200	20
10000	12520	120

Compression Space Complexity in bytes

N (Total Words)	LZ-77	Lempel-Ziv-Welch
500	28646	41141
1000	54786	41154
10000	546972	327873

Decompression Time Complexity in milliseconds

N (Total Words)	LZ-77	Lempel-Ziv-Welch
500	10	10
1000	10	10
10000	140	50

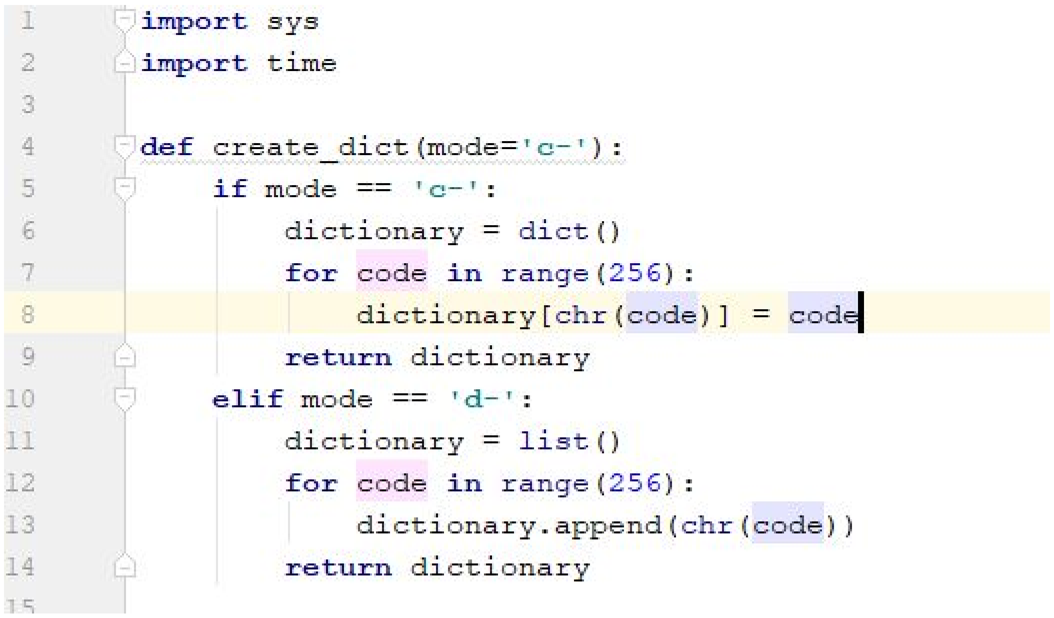
Decompression Space Complexity in bytes

N (Total Words)	LZ-77	Lempel-Ziv-Welch
500	28982	7382
1000	55798	11907
10000	530824	55542

4. Evaluation

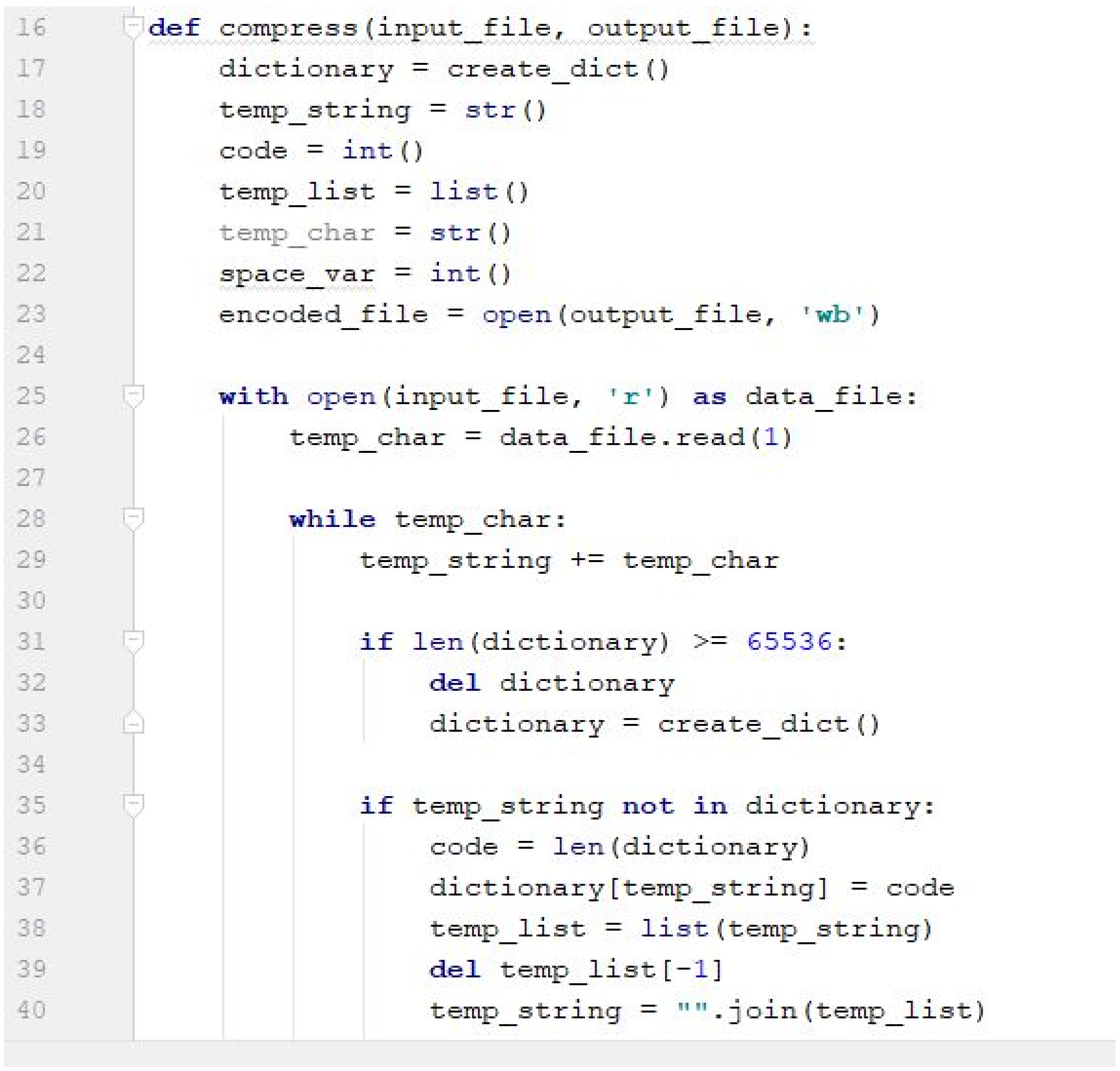
- Theoretical analysis of the algorithms
- Lempel-Ziv (LZ-77)
- Lempel-Ziv-Welch
- Implementation Details

➢ Lempel-Ziv-Welch analysis:



```python
import sys
import time

def create_dict(mode='c-'):
    if mode == 'c-':
        dictionary = dict()
        for code in range(256):
            dictionary[chr(code)] = code
        return dictionary
    elif mode == 'd-':
        dictionary = list()
        for code in range(256):
            dictionary.append(chr(code))
        return dictionary
```

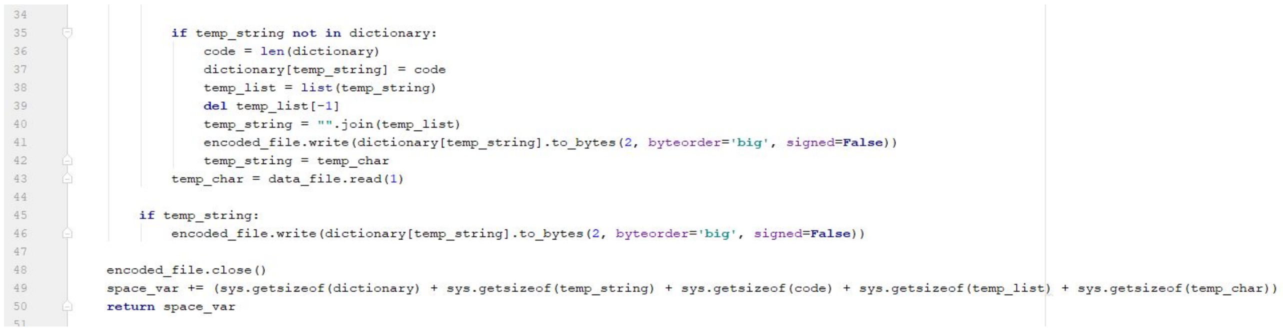
Creating a function to make a dictionary and appending 255 ASCII characters inside it.



```
def compress(input_file, output_file):
    dictionary = create_dict()
    temp_string = str()
    code = int()
    temp_list = list()
    temp_char = str()
    space_var = int()
    encoded_file = open(output_file, 'wb')

    with open(input_file, 'r') as data_file:
        temp_char = data_file.read(1)

        while temp_char:
            temp_string += temp_char

            if len(dictionary) >= 65536:
                del dictionary
                dictionary = create_dict()

            if temp_string not in dictionary:
                code = len(dictionary)
                dictionary[temp_string] = code
                temp_list = list(temp_string)
                del temp_list[-1]
                temp_string = "".join(temp_list)
```

Creating the compress function to declare variables (even though it is not needed just to prevent confusion), then open the file and reads it.

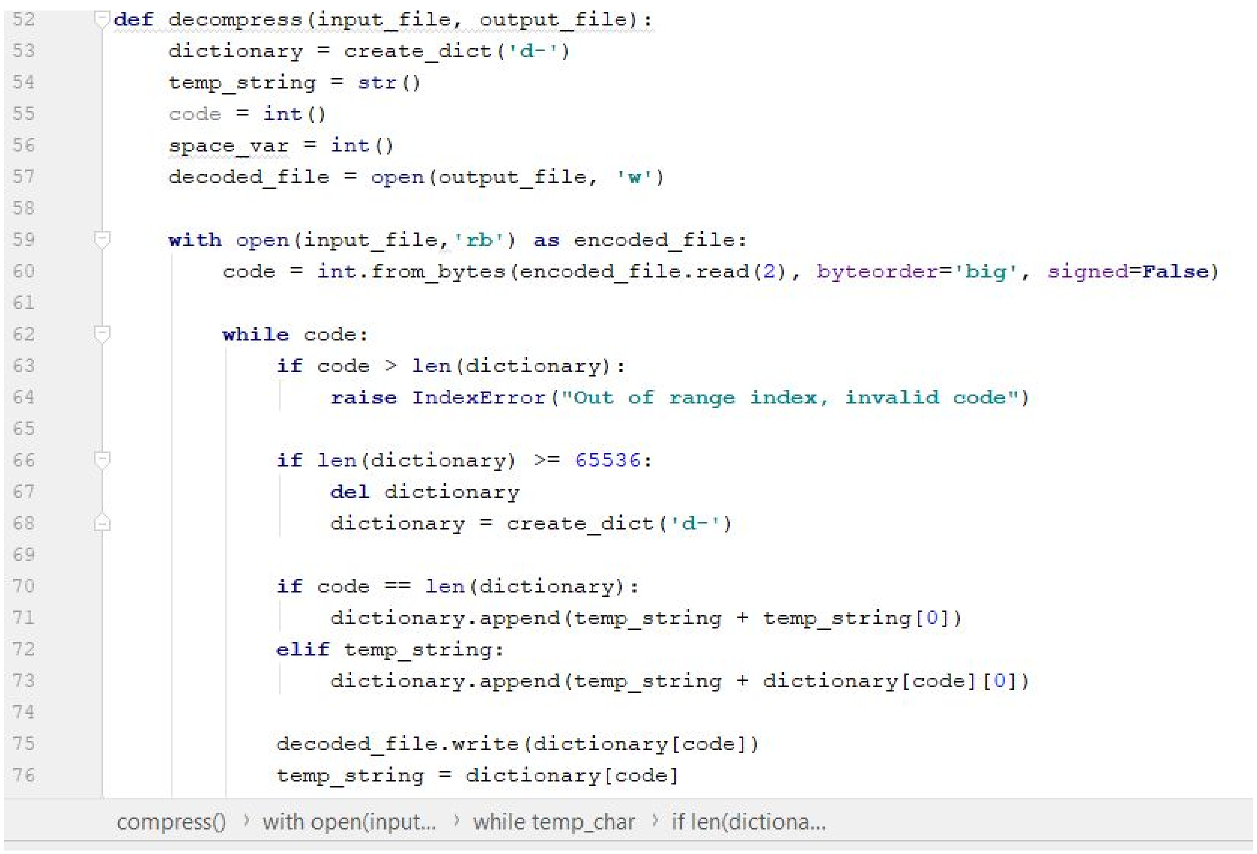
Using while loop to read the character one by one until and add it to the string variable. Length of the dictionary is checked first to make sure it doesn’t reach 65536 entries. 65536 is the max value of the string’s code because it uses 2 bytes / 16 bit only. There is also if condition to check if the string is not in the dictionary.



```python
            if temp_string not in dictionary:
                code = len(dictionary)
                dictionary[temp_string] = code
                temp_list = list(temp_string)
                del temp_list[-1]
                temp_string = "".join(temp_list)
                encoded_file.write(dictionary[temp_string].to_bytes(2, byteorder='big', signed=False))
                temp_string = temp_char
            temp_char = data_file.read(1)

        if temp_string:
            encoded_file.write(dictionary[temp_string].to_bytes(2, byteorder='big', signed=False))

    encoded_file.close()
    space_var += (sys.getsizeof(dictionary) + sys.getsizeof(temp_string) + sys.getsizeof(code) + sys.getsizeof(temp_list) + sys.getsizeof(temp_char))
    return space_var
```

If it’s not, it will append the string and its code to the dictionary. Then remove the last character from the string first and after that, write it to the file in bytes. The next string will then use the last read character and read a new one from the source file

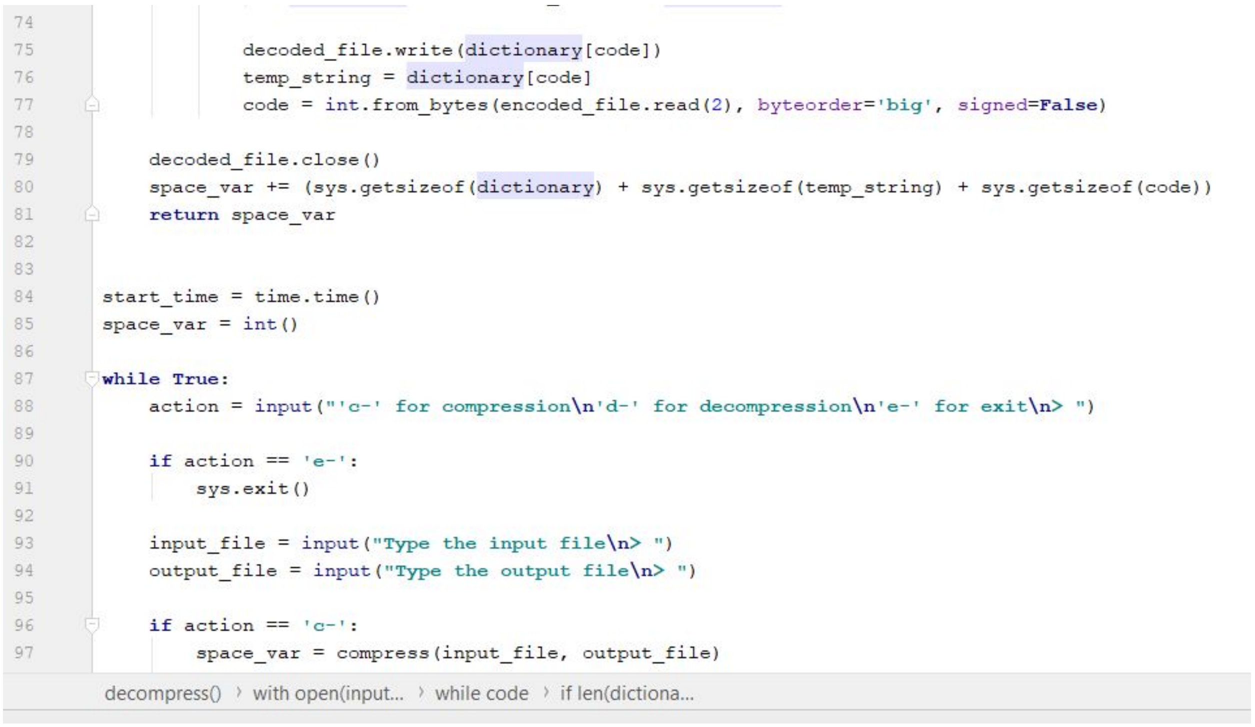
If there is any string left after the source file’s content has been read all, it will be written to the file in bytes. After the file is closed, the space used is calculated and returned.



```python
def decompress(input_file, output_file):
    dictionary = create_dict('d-')
    temp_string = str()
    code = int()
    space_var = int()
    decoded_file = open(output_file, 'w')

    with open(input_file,'rb') as encoded_file:
        code = int.from_bytes(encoded_file.read(2), byteorder='big', signed=False)

        while code:
            if code > len(dictionary):
                raise IndexError("Out of range index, invalid code")

            if len(dictionary) >= 65536:
                del dictionary
                dictionary = create_dict('d-')

            if code == len(dictionary):
                dictionary.append(temp_string + temp_string[0])
            elif temp_string:
                dictionary.append(temp_string + dictionary[code][0])

            decoded_file.write(dictionary[code])
            temp_string = dictionary[code]
```

compress() › with open(input... › while temp_char › if len(dictiona...

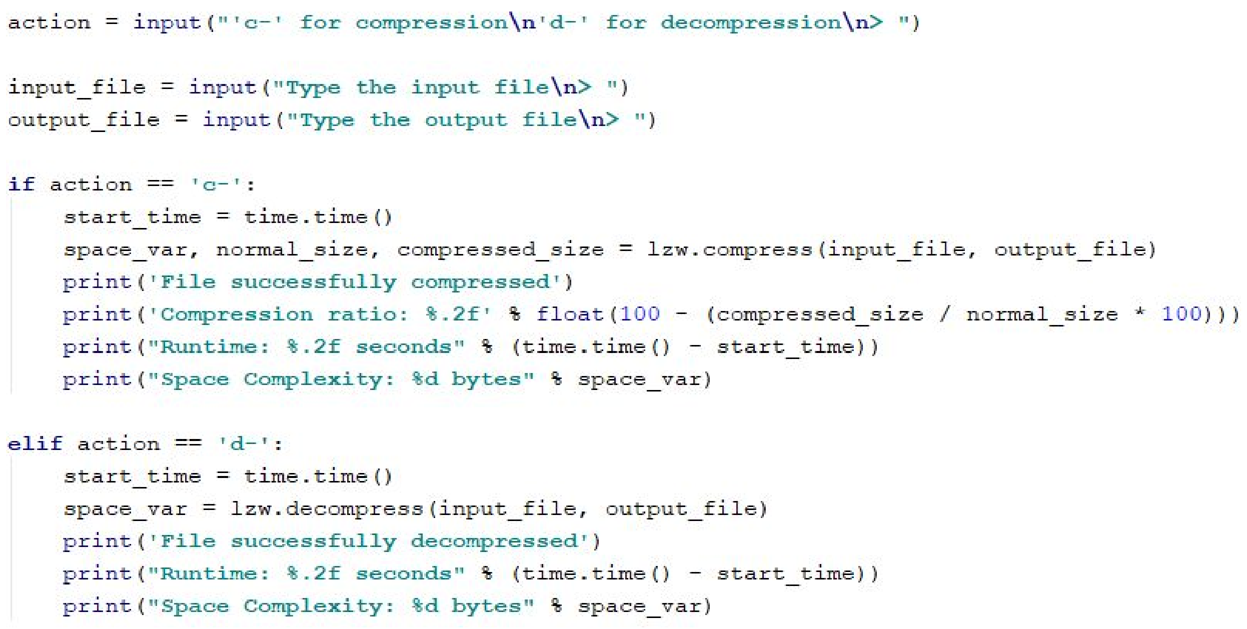
It works first by creating the dictionary with 256 ascii pre-initialized first, same as the compress function. It will then read the bytes, 2 at each time one by one. It will then convert the bytes to int. If the code (int) is bigger than the current length of the dictionary. That means, the dictionary is corrupted. That is because the dictionary is built incrementally (relative with the dictionary's length) along with reading the file. If the code is the same as the dictionary's length (not yet exists), it will be added to the dictionary (the string and its first character) firsthand. If the code is below the current dictionary's length (already exists), the string combined with the first character of the string which the code refers to will be appended to the dictionary.



```python

                decoded_file.write(dictionary[code])
                temp_string = dictionary[code]
                code = int.from_bytes(encoded_file.read(2), byteorder='big', signed=False)

        decoded_file.close()
        space_var += (sys.getsizeof(dictionary) + sys.getsizeof(temp_string) + sys.getsizeof(code))
        return space_var


start_time = time.time()
space_var = int()

while True:
    action = input("'c-' for compression\n'd-' for decompression\n'e-' for exit\n> ")

    if action == 'e-':
        sys.exit()

    input_file = input("Type the input file\n> ")
    output_file = input("Type the output file\n> ")

    if action == 'c-':
        space_var = compress(input_file, output_file)
```

decompress() › with open(input... › while code › if len(dictiona...

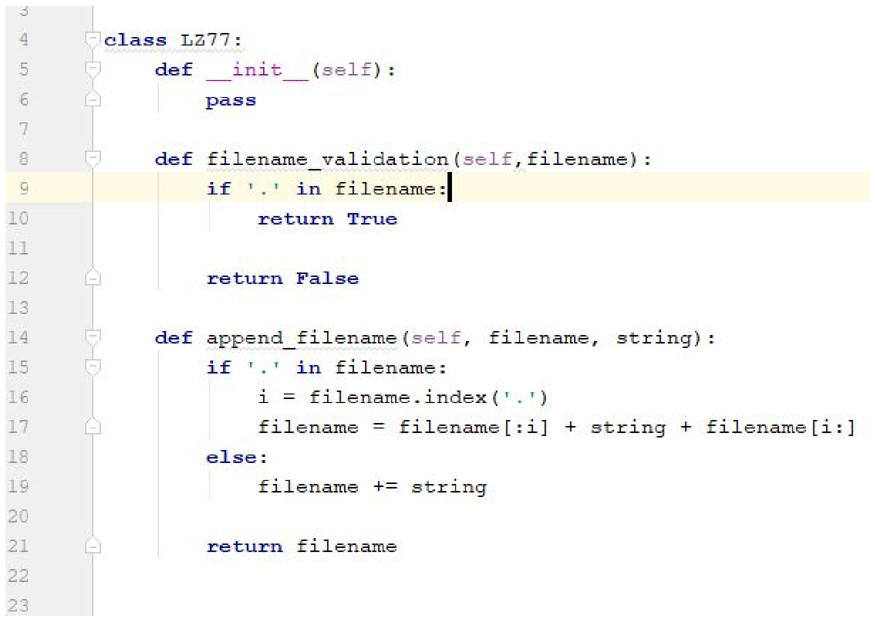
After that, it will write the string that the code refers to to the new file and use it for the next string. Last, it will calculate all the spaces that are used and return it.



```python
action = input("'c-' for compression\n'd-' for decompression\n> ")

input_file = input("Type the input file\n> ")
output_file = input("Type the output file\n> ")

if action == 'c-':
    start_time = time.time()
    space_var, normal_size, compressed_size = lzw.compress(input_file, output_file)
    print('File successfully compressed')
    print('Compression ratio: %.2f' % float(100 - (compressed_size / normal_size * 100)))
    print("Runtime: %.2f seconds" % (time.time() - start_time))
    print("Space Complexity: %d bytes" % space_var)

elif action == 'd-':
    start_time = time.time()
    space_var = lzw.decompress(input_file, output_file)
    print('File successfully decompressed')
    print("Runtime: %.2f seconds" % (time.time() - start_time))
    print("Space Complexity: %d bytes" % space_var)
```

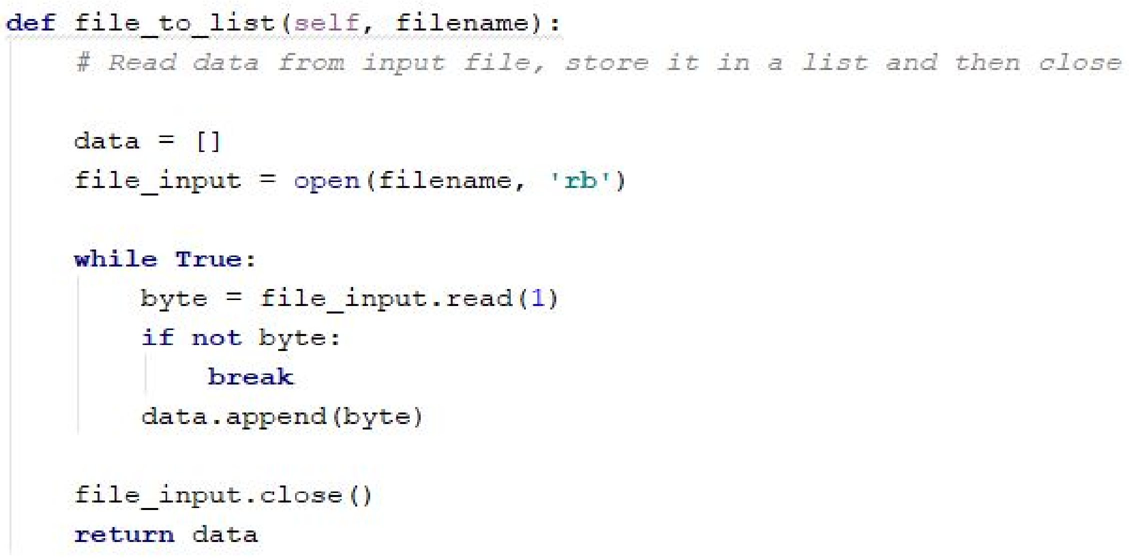
Creating a while loop for the input and output file when the file is compressed and decompressed, also printing out the runtime, compression ratio and space complexity.

➢ Lempel-Ziv (LZ-77):



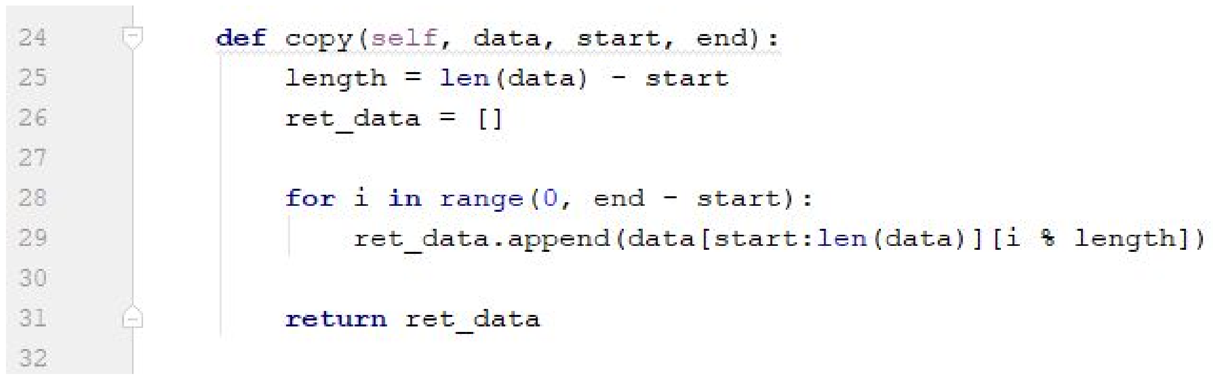
```python
class LZ77:
    def __init__(self):
        pass

    def filename_validation(self,filename):
        if '.' in filename:
            return True

        return False

    def append_filename(self, filename, string):
        if '.' in filename:
            i = filename.index('.')
            filename = filename[:i] + string + filename[i:]
        else:
            filename += string

        return filename
```

Filename validation checks if the file name has a "." file extension. If it does, it returns true, if it doesn;t it returns false, while append_filename is used to rewrite the output file.

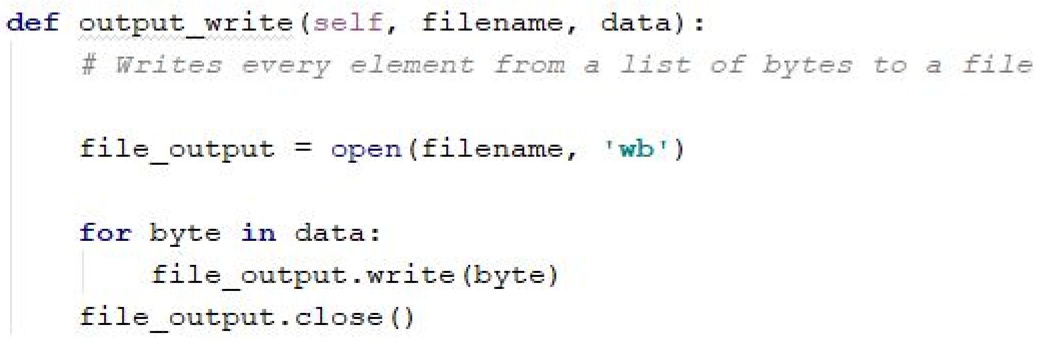


```python
def file_to_list(self, filename):
    # Read data from input file, store it in a list and then close

    data = []
    file_input = open(filename, 'rb')

    while True:
        byte = file_input.read(1)
        if not byte:
            break
        data.append(byte)

    file_input.close()
    return data
```

Reads data from input file, checks if it is byte and closes it. First,we open the file as read in binary mode, then we used a while loop to read the input file one byte at a time and checks if it is in the byte data type. If it isn't the code breaks. If it is, the byte from the file is stored into the list. After the whole file has been read, the file is closed and the list containing the byte data is returned.

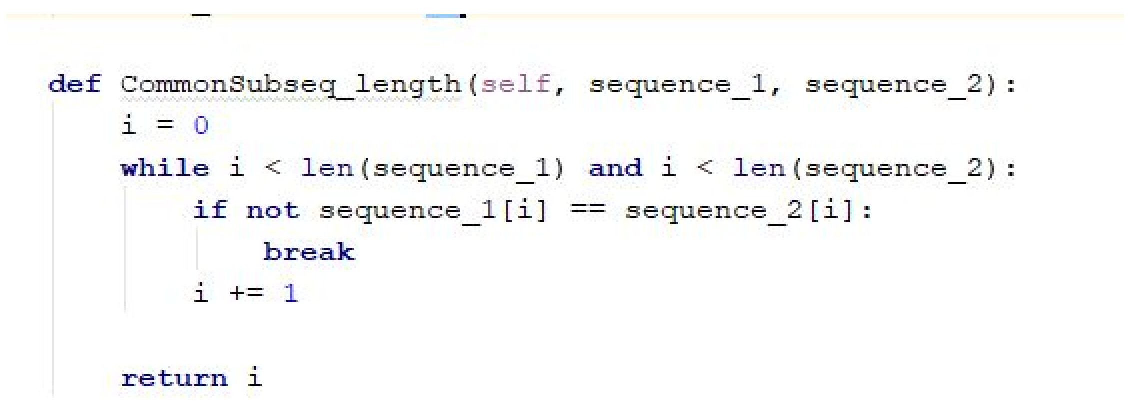


```python
    def copy(self, data, start, end):
        length = len(data) - start
        ret_data = []

        for i in range(0, end - start):
            ret_data.append(data[start:len(data)][i % length])

        return ret_data

```

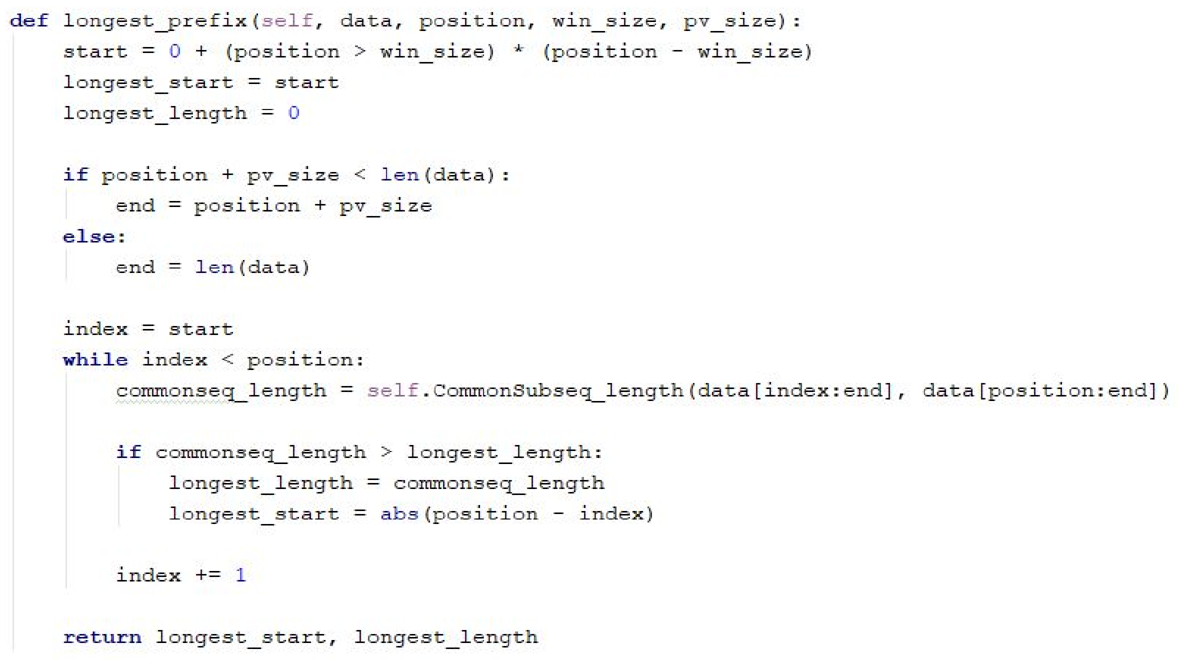


```python
def output_write(self, filename, data):
    # Writes every element from a list of bytes to a file

    file_output = open(filename, 'wb')

    for byte in data:
        file_output.write(byte)
    file_output.close()
```

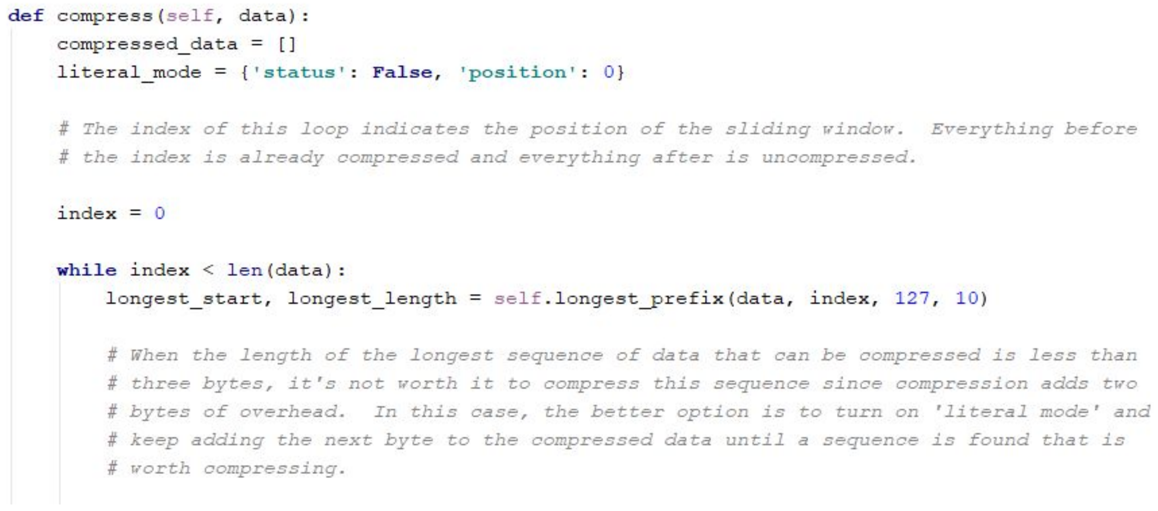
Used to write every element from a list of bytes to a file. First, it opens the output file as wb which means that it is writing to a file in binary mode, then we used a for loop to write every element from the list into the output file. After that,we close the output file.



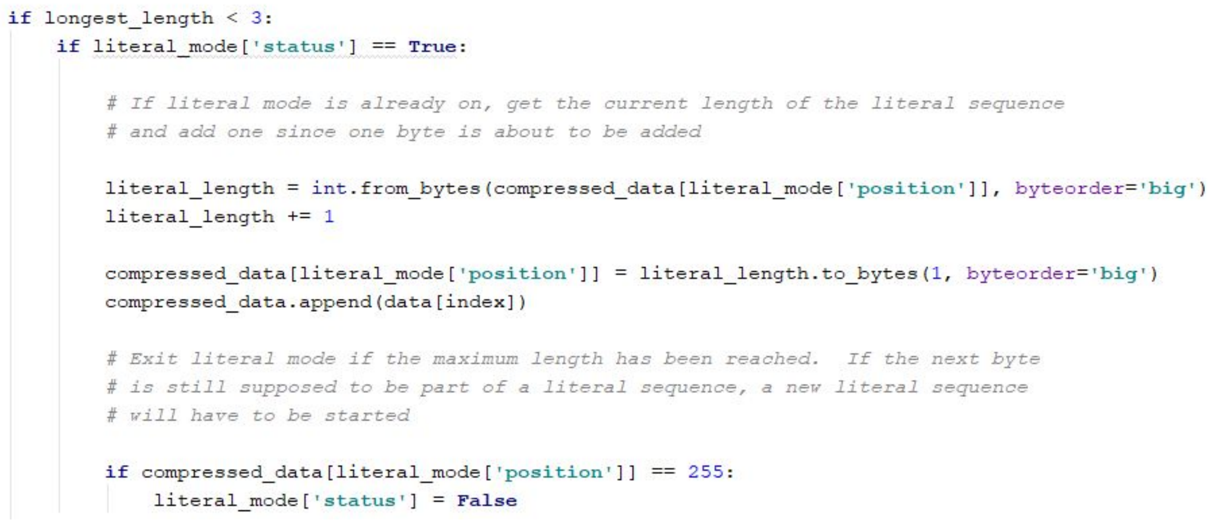
```
def CommonSubseq_length(self, sequence_1, sequence_2):
    i = 0
    while i < len(sequence_1) and i < len(sequence_2):
        if not sequence_1[i] == sequence_2[i]:
            break
        i += 1

    return i
```



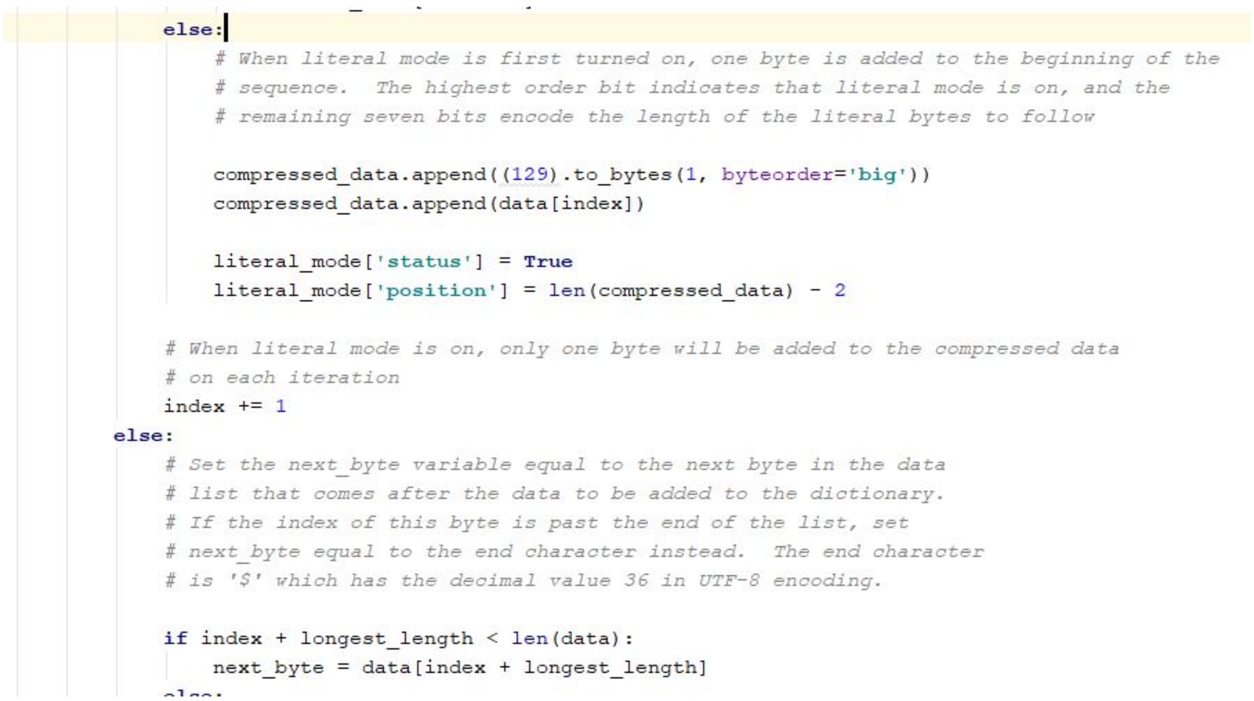
```
def longest_prefix(self, data, position, win_size, pv_size):
    start = 0 + (position > win_size) * (position - win_size)
    longest_start = start
    longest_length = 0

    if position + pv_size < len(data):
        end = position + pv_size
    else:
        end = len(data)

    index = start
    while index < position:
        commonseq_length = self.CommonSubseq_length(data[index:end], data[position:end])

        if commonseq_length > longest_length:
            longest_length = commonseq_length
            longest_start = abs(position - index)

        index += 1

    return longest_start, longest_length
```



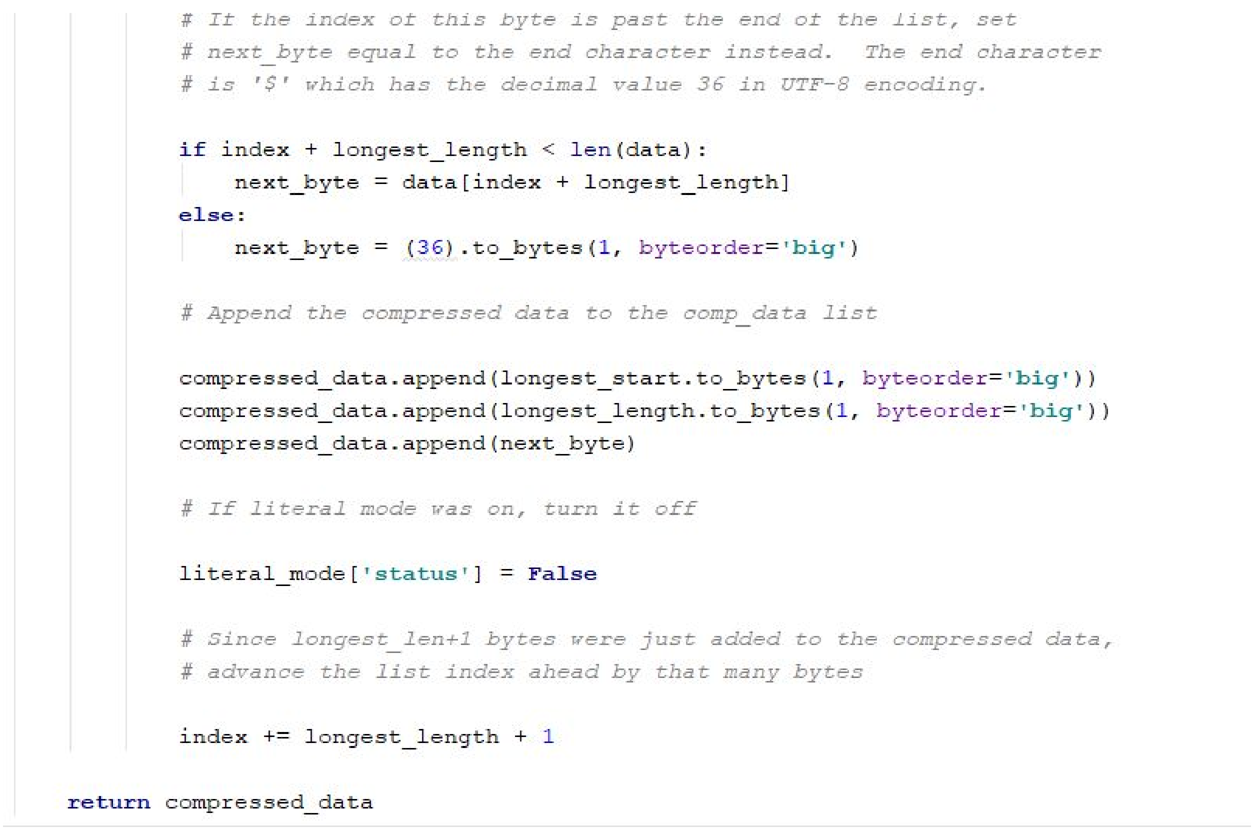
```python
def compress(self, data):
    compressed_data = []
    literal_mode = {'status': False, 'position': 0}

    # The index of this loop indicates the position of the sliding window.  Everything before
    # the index is already compressed and everything after is uncompressed.

    index = 0

    while index < len(data):
        longest_start, longest_length = self.longest_prefix(data, index, 127, 10)

        # When the length of the longest sequence of data that can be compressed is less than
        # three bytes, it's not worth it to compress this sequence since compression adds two
        # bytes of overhead.  In this case, the better option is to turn on 'literal mode' and
        # keep adding the next byte to the compressed data until a sequence is found that is
        # worth compressing.
```



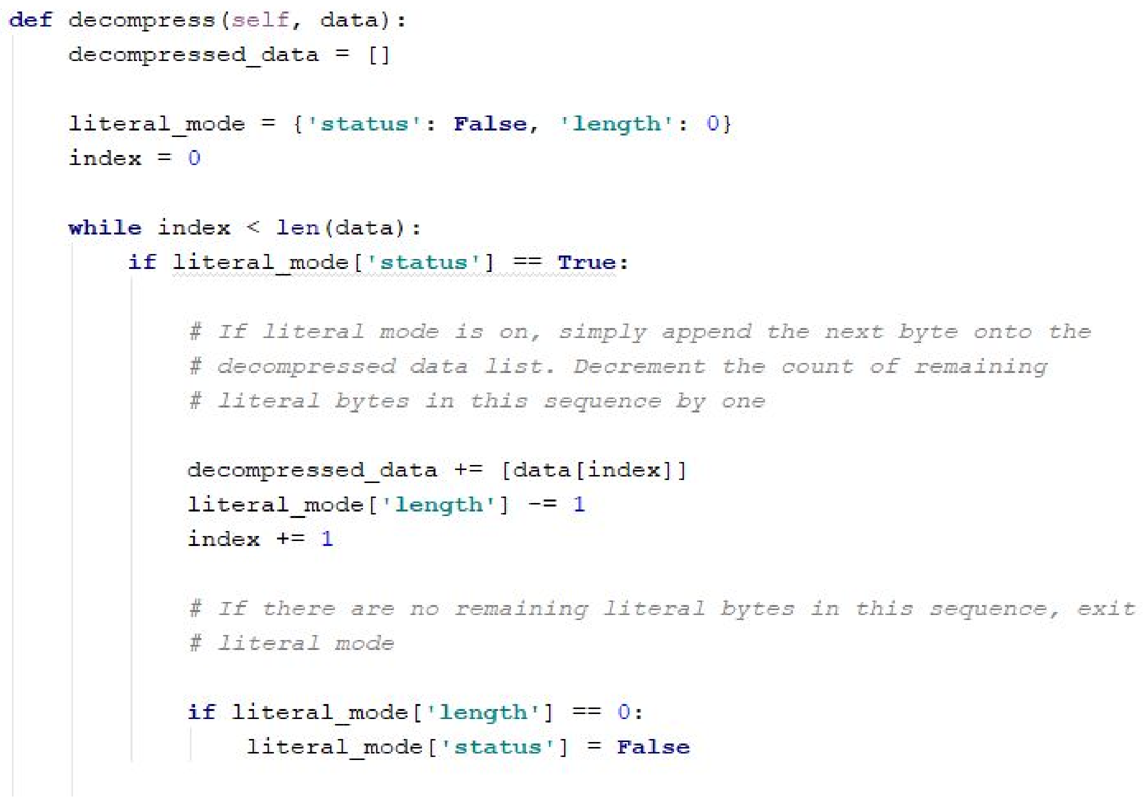
```python
if longest_length < 3:
    if literal_mode['status'] == True:

        # If literal mode is already on, get the current length of the literal sequence
        # and add one since one byte is about to be added

        literal_length = int.from_bytes(compressed_data[literal_mode['position']], byteorder='big')
        literal_length += 1

        compressed_data[literal_mode['position']] = literal_length.to_bytes(1, byteorder='big')
        compressed_data.append(data[index])

        # Exit literal mode if the maximum length has been reached.  If the next byte
        # is still supposed to be part of a literal sequence, a new literal sequence
        # will have to be started

        if compressed_data[literal_mode['position']] == 255:
            literal_mode['status'] = False
```



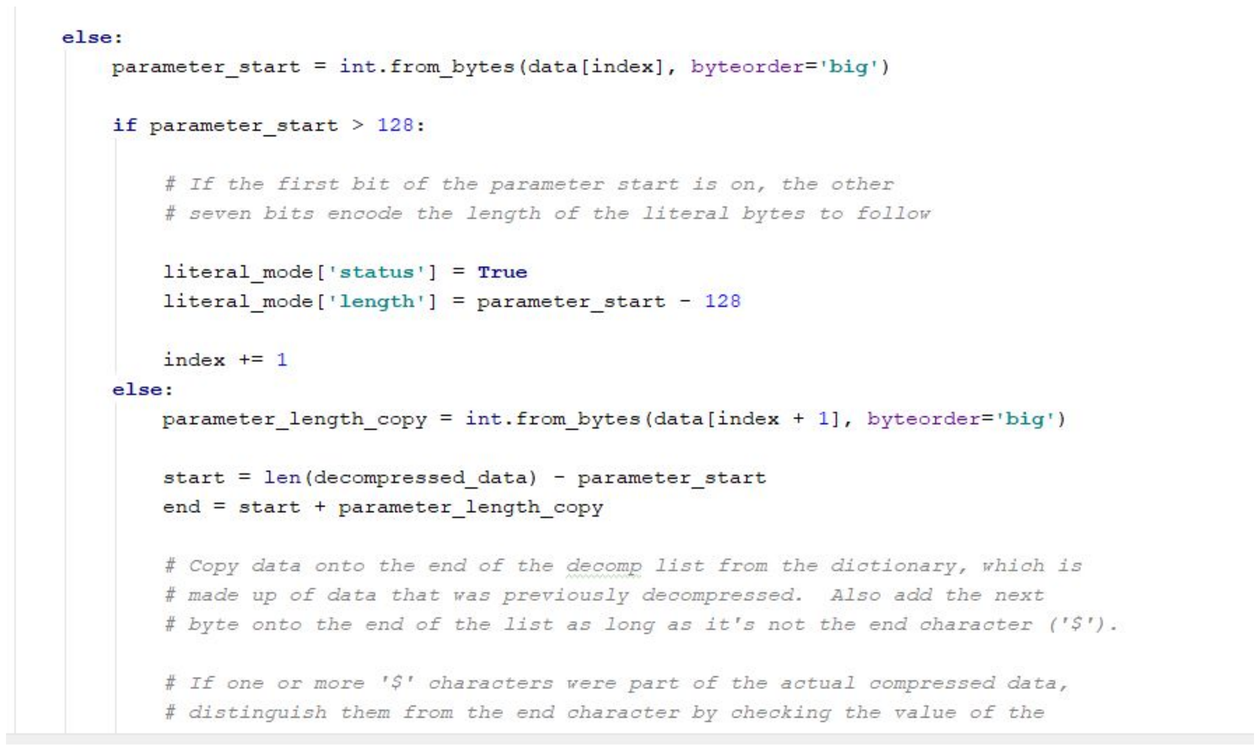
```
            else:
                # When literal mode is first turned on, one byte is added to the beginning of the
                # sequence.  The highest order bit indicates that literal mode is on, and the
                # remaining seven bits encode the length of the literal bytes to follow

                compressed_data.append((129).to_bytes(1, byteorder='big'))
                compressed_data.append(data[index])

                literal_mode['status'] = True
                literal_mode['position'] = len(compressed_data) - 2

            # When literal mode is on, only one byte will be added to the compressed data
            # on each iteration
            index += 1
        else:
            # Set the next_byte variable equal to the next byte in the data
            # list that comes after the data to be added to the dictionary.
            # If the index of this byte is past the end of the list, set
            # next_byte equal to the end character instead.  The end character
            # is '$' which has the decimal value 36 in UTF-8 encoding.

            if index + longest_length < len(data):
                next_byte = data[index + longest_length]
```



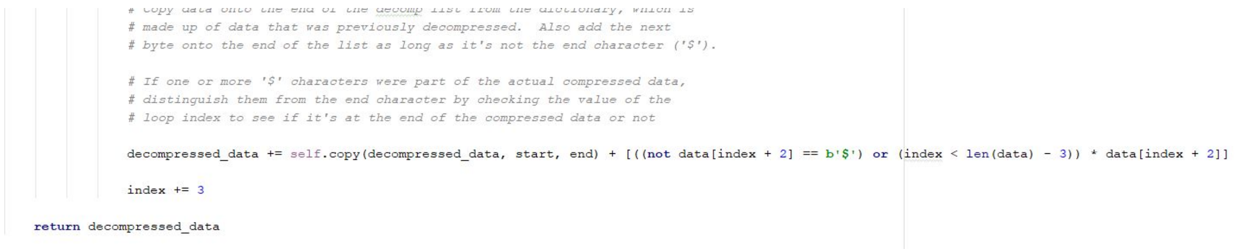
```
            # If the index of this byte is past the end of the list, set
            # next_byte equal to the end character instead.  The end character
            # is '$' which has the decimal value 36 in UTF-8 encoding.

            if index + longest_length < len(data):
                next_byte = data[index + longest_length]
            else:
                next_byte = (36).to_bytes(1, byteorder='big')

            # Append the compressed data to the comp_data list

            compressed_data.append(longest_start.to_bytes(1, byteorder='big'))
            compressed_data.append(longest_length.to_bytes(1, byteorder='big'))
            compressed_data.append(next_byte)

            # If literal mode was on, turn it off

            literal_mode['status'] = False

            # Since longest_len+1 bytes were just added to the compressed data,
            # advance the list index ahead by that many bytes

            index += longest_length + 1

    return compressed_data
```



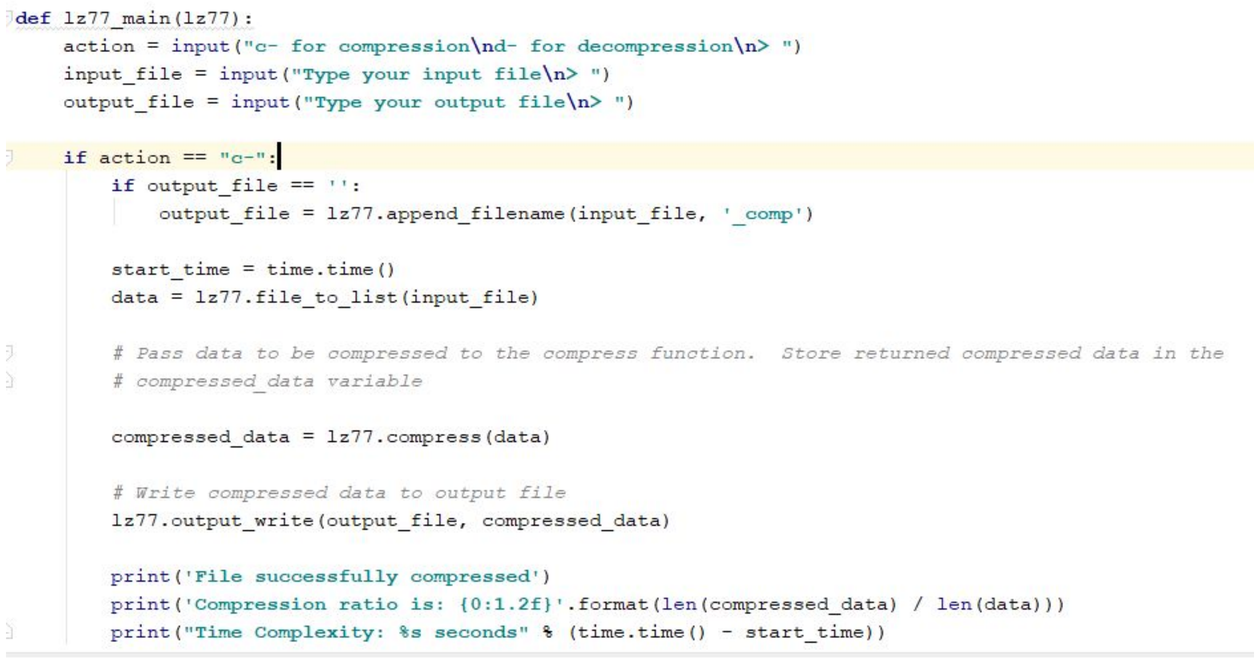
```python
def decompress(self, data):
    decompressed_data = []

    literal_mode = {'status': False, 'length': 0}
    index = 0

    while index < len(data):
        if literal_mode['status'] == True:

            # If literal mode is on, simply append the next byte onto the
            # decompressed data list. Decrement the count of remaining
            # literal bytes in this sequence by one

            decompressed_data += [data[index]]
            literal_mode['length'] -= 1
            index += 1

            # If there are no remaining literal bytes in this sequence, exit
            # literal mode

            if literal_mode['length'] == 0:
                literal_mode['status'] = False
```



```python
        else:
            parameter_start = int.from_bytes(data[index], byteorder='big')

            if parameter_start > 128:

                # If the first bit of the parameter start is on, the other
                # seven bits encode the length of the literal bytes to follow

                literal_mode['status'] = True
                literal_mode['length'] = parameter_start - 128

                index += 1
            else:
                parameter_length_copy = int.from_bytes(data[index + 1], byteorder='big')

                start = len(decompressed_data) - parameter_start
                end = start + parameter_length_copy

                # Copy data onto the end of the decomp list from the dictionary, which is
                # made up of data that was previously decompressed.  Also add the next
                # byte onto the end of the list as long as it's not the end character ('$').

                # If one or more '$' characters were part of the actual compressed data,
                # distinguish them from the end character by checking the value of the
```

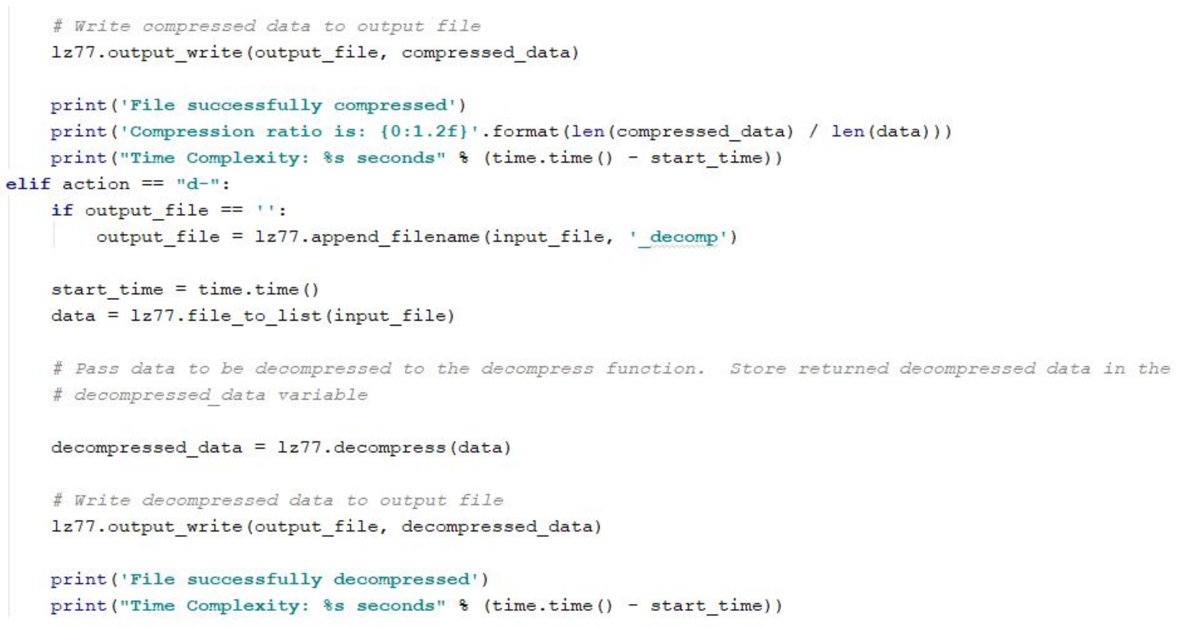


```
            # copy data onto the end of the decomp list from the dictionary, which is
            # made up of data that was previously decompressed.  Also add the next
            # byte onto the end of the list as long as it's not the end character ('$').

            # If one or more '$' characters were part of the actual compressed data,
            # distinguish them from the end character by checking the value of the
            # loop index to see if it's at the end of the compressed data or not

            decompressed_data += self.copy(decompressed_data, start, end) + [((not data[index + 2] == b'$') or (index < len(data) - 3)) * data[index + 2]]

            index += 3

    return decompressed_data
```



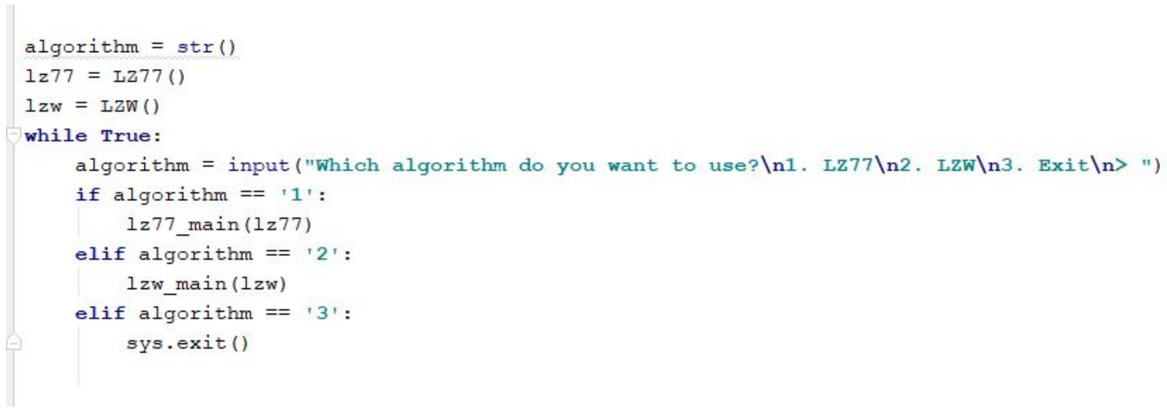
```
def lz77_main(lz77):
    action = input("c- for compression\nd- for decompression\n> ")
    input_file = input("Type your input file\n> ")
    output_file = input("Type your output file\n> ")

    if action == "c-":
        if output_file == '':
            output_file = lz77.append_filename(input_file, '_comp')

        start_time = time.time()
        data = lz77.file_to_list(input_file)

        # Pass data to be compressed to the compress function.  Store returned compressed data in the
        # compressed_data variable

        compressed_data = lz77.compress(data)

        # Write compressed data to output file
        lz77.output_write(output_file, compressed_data)

        print('File successfully compressed')
        print('Compression ratio is: {0:1.2f}'.format(len(compressed_data) / len(data)))
        print("Time Complexity: %s seconds" % (time.time() - start_time))
```

The main of the lz77, here the user will have to type the input and output file. Then the file will be appended inside the lz77 function and therefore will be compressed, and the time complexity along with the compression ratio will be printed out.



```
    # Write compressed data to output file
    lz77.output_write(output_file, compressed_data)

    print('File successfully compressed')
    print('Compression ratio is: {0:1.2f}'.format(len(compressed_data) / len(data)))
    print("Time Complexity: %s seconds" % (time.time() - start_time))
elif action == "d-":
    if output_file == '':
        output_file = lz77.append_filename(input_file, '_decomp')

    start_time = time.time()
    data = lz77.file_to_list(input_file)

    # Pass data to be decompressed to the decompress function.  Store returned decompressed data in the
    # decompressed_data variable

    decompressed_data = lz77.decompress(data)

    # Write decompressed data to output file
    lz77.output_write(output_file, decompressed_data)

    print('File successfully decompressed')
    print("Time Complexity: %s seconds" % (time.time() - start_time))
```

Else if function is used to produce the decompressed file along with its time complexity.



```
algorithm = str()
lz77 = LZ77()
lzw = LZW()
while True:
    algorithm = input("Which algorithm do you want to use?\n1. LZ77\n2. LZW\n3. Exit\n> ")
    if algorithm == '1':
        lz77_main(lz77)
    elif algorithm == '2':
        lzw_main(lzw)
    elif algorithm == '3':
        sys.exit()
```

The lz77 and lzw function are called here inside the while loop where the user will have to choose to run either the compressor or exit.

~ Explanations of each algorithms:

1. Lempel-Ziv (LZ-77) :

    - LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a *length-distance pair*, which is equivalent to the statement "each of the next *length* characters is equal to the characters exactly *distance* characters behind it in the uncompressed stream".

- LZ77 maintains a *sliding window* during compression (the encoder must keep track of some amount of the most recent data, such as the last 2 kB, 4 kB, or 32 kB. The structure in which this data is held is called a *sliding window*, which is why LZ77 is sometimes called *sliding-window compression*).

2. Lempel-Ziv-Welch :

- Encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095 are created in a dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence for which there is no code yet in the dictionary. The code for the sequence (without that character) is added to the output, and a new code (for the sequence with that character) is added to the dictionary.

- Encoding :
    - 1. Initialize the dictionary to contain all strings of length one.
    - 2. Find the longest string W in the dictionary that matches the current input.
    - 3. Emit the dictionary index for W to output and remove W from the input.
    - 4. Add W followed by the next symbol in the input to the dictionary.
    - 5. Go to step 2.

- Decoding :
    - The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. In order to rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the concatenation of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded (If the next value is unknown to the decoder, then it must be the value that will be added to the dictionary this iteration, and so its first character must be the same as the first character of the current string being sent to decoded

output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.

- In this way the decoder builds up a dictionary which is identical to that used by the encoder, and uses it to decode subsequent input values. Thus the full dictionary does not need to be sent with the encoded data; just the initial dictionary containing the single-character strings is sufficient (and is typically defined beforehand within the encoder and decoder rather than being explicitly sent with the encoded data).

5. Discussion :

- We chose to use the Lempel-Ziv (LZ-77) because of its efficiency in data compression which is very suitable for our situation.

- We chose to use the Lempel-Ziv-Welch because it is a universal lossless data compression algorithm which is simple to implement and has the potential for very high throughput in hardware implementations.

- As our purpose is to compare 2 algorithms, these 2 algorithms is very suitable to be compared as the Lempel-Ziv-Welch is the improved implementation of LZ-78 (which is pretty similar to LZ-77 but the differences are LZ-78 is faster than LZ-77 but doesn't always achieve as high a compression ratio as LZ-77. The biggest advantage LZ-78 has over the LZ-77 algorithm is the reduced number of string comparisons in each encoding step. The only similarity between LZ-77 & LZ-78 is that they have fast decompression but slow compression).

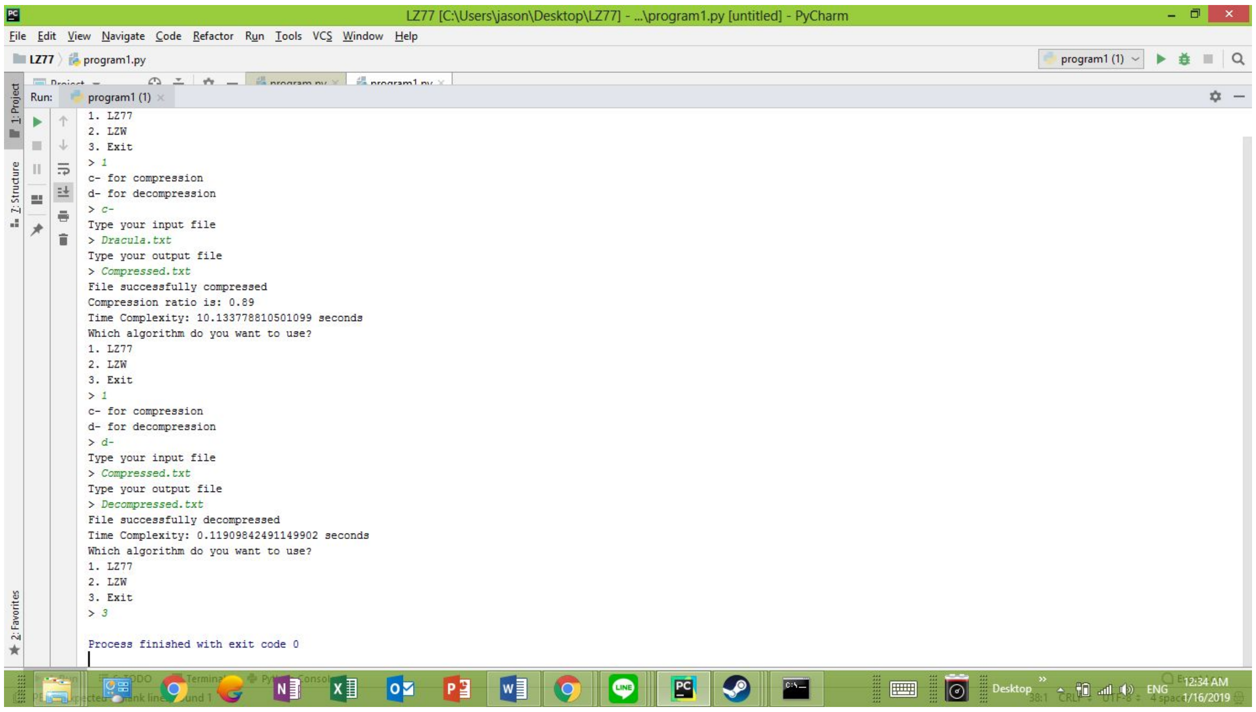
- We will be comparing these 2 codes in terms of their time & space complexity and their better usage on which conditions.

6. Conclusion & Recommendation :

★ As expected, the Lempel-Ziv-Welch algorithm is better at both compression and decompression in terms of both time complexity and space complexity in almost every situation. So, it is more recommended to use the Lempel-Ziv-Welch algorithm compared to the LZ-77.

7. Program's Manual :

Step 1 :

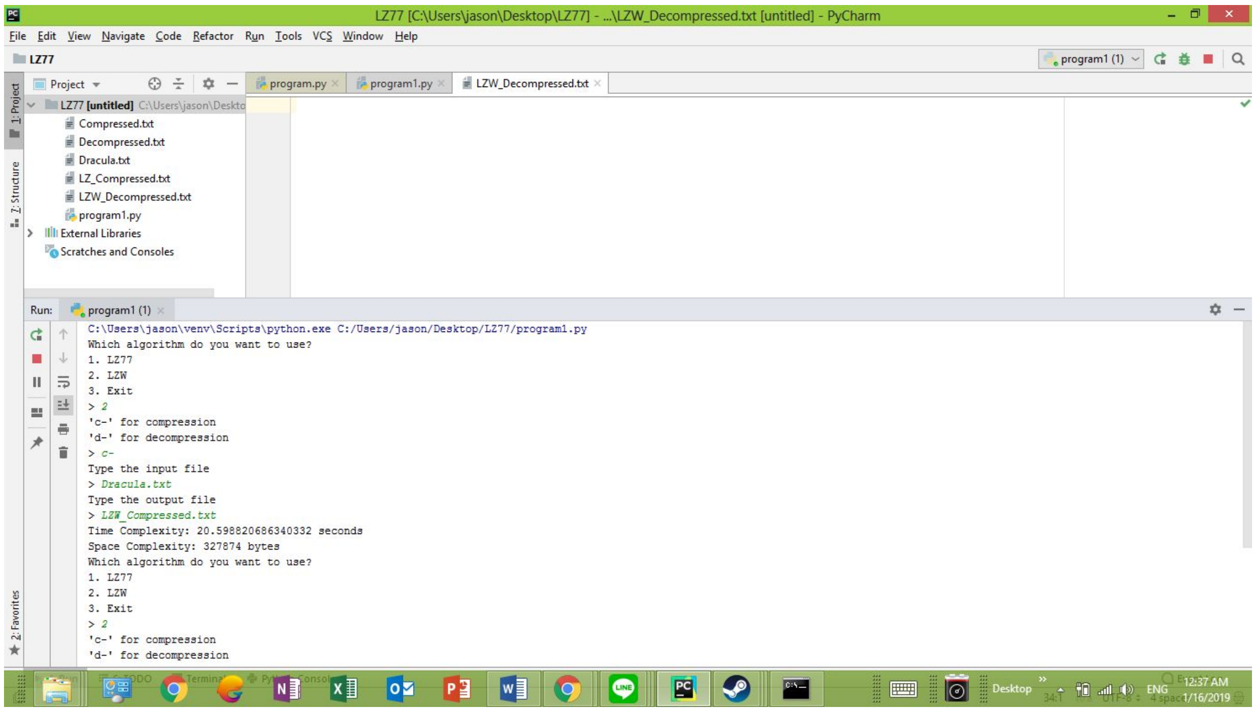


LZ77 [C:\Users\jason\Desktop\LZ77] - ...\program1.py [untitled] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
LZ77 program1.py
program1 (1)
Run: program1 (1)

```
1. LZ77
2. LZW
3. Exit
> 1
c- for compression
d- for decompression
> c-
Type your input file
> Dracula.txt
Type your output file
> Compressed.txt
File successfully compressed
Compression ratio is: 0.89
Time Complexity: 10.133778810501099 seconds
Which algorithm do you want to use?
1. LZ77
2. LZW
3. Exit
> 1
c- for compression
d- for decompression
> d-
Type your input file
> Compressed.txt
Type your output file
> Decompressed.txt
File successfully decompressed
Time Complexity: 0.11909842491149902 seconds
Which algorithm do you want to use?
1. LZ77
2. LZW
3. Exit
> 3

Process finished with exit code 0
```

There are 3 choices to be made : 1) , 2) & 3).
If the user press 1, then the encoding for Lempel-Ziv (LZ-77) will run.
If the user press 2, then the encoding for Lempel-Ziv-Welch will run.
If the user press 3, then the system will stop running.

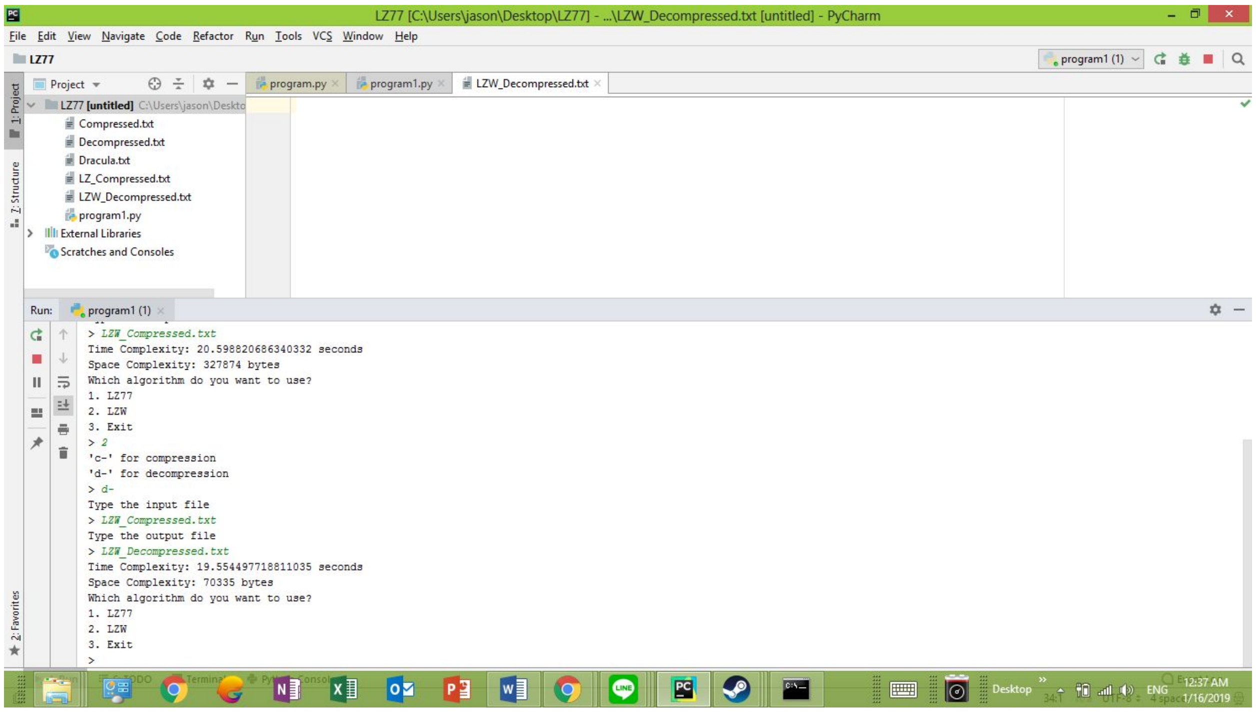
Step 2 :



```
C:\Users\jason\venv\Scripts\python.exe C:/Users/jason/Desktop/LZ77/program1.py
Which algorithm do you want to use?
1. LZ77
2. LZW
3. Exit
> 2
'c-' for compression
'd-' for decompression
> c-
Type the input file
> Dracula.txt
Type the output file
> LZW_Compressed.txt
Time Complexity: 20.598820686340332 seconds
Space Complexity: 327874 bytes
Which algorithm do you want to use?
1. LZ77
2. LZW
3. Exit
> 2
'c-' for compression
'd-' for decompression
```

If the user press either 1) or 2) then the user will have to type either “c-” or “d-”.
In which “c-” is for compression and “d-” is for decompression.

Step 3 :



```
> LZW_Compressed.txt
Time Complexity: 20.598820686340332 seconds
Space Complexity: 327874 bytes
Which algorithm do you want to use?
1. LZ77
2. LZW
3. Exit
> 2
'c-' for compression
'd-' for decompression
> d-
Type the input file
> LZW_Compressed.txt
Type the output file
> LZW_Decompressed.txt
Time Complexity: 19.554497718811035 seconds
Space Complexity: 70335 bytes
Which algorithm do you want to use?
1. LZ77
2. LZW
3. Exit
>
```

The user will have to type in the input and output file to be compressed/decompressed in order to run the program.

After the program is done running, it will shot the time and space complexity. Then the program will go back to **<u>Step 1</u>** where the user has to make the first choices again.

<u>Notes</u> : The text file must be in the same folder as the program in order to run.