A DETAILED ANALYSIS OF COMPRESSION ACHIEVED BY DIFFERENT ALGORITHMS

Progress Report - 4

Prepared by : Rachit Mishra(201201092)
Dhrumil Shah (201201117)

This is a study we are doing of different algorithms namely LZ77(a variant of LZW), huffman encoding, Run length encoding and Arithmetic encoding compression algorithms in order to present a comparative analysis of performance of different algorithms. As of now, we are in development phase.

We are ready with input files(ranging from over 42k characters to 7k characters). At the end of this study a graphical statistical analysis of the comparison of compression algorithms would be given. That's our aim.

As of now, we are working upon arithmetic and huffman algorithms and the details are given below.

Arithmetic Compression

: A form of entropy coding(a type of lossless coding) used in lossless data compression and lossy data compression. Frequently seen symbols are encoded with fewer bits and lesser seen symbols are encoded with more bits. In this way compression is achieved.

- Takes a message input which is composed of symbols (8-bit characters generally) and converts it to a floating point number generally >=0 or <=1.
- Entire output file is one long number.
- As each character in the file is encoded, a few bits will be added to the encoded message.
- Probability of occurrence of a character in the message is specified and is really important.

- While encoding, it's presumed that each symbol has it's own number line b//w 0 and 1.

- Some probability determining models are static and some aren't, they update after each character is processed.

- If we consider a string, say 'WXYZ' and say we have an encoder which can encode an alphabet of 100 different characters. We start with uppercase letters then move to lower case letters.

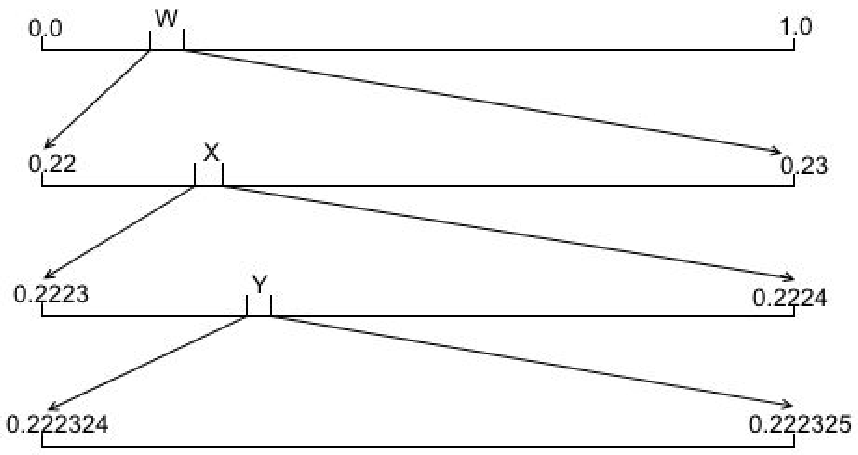
- A has 0 t0 0.01, B has 0.01 to 0.02. At any instance, it is a half closed interval.

- W is the 1st character in this string. It's in the interval [0.22, .23). At this moment high is 1.0 and LOW is 0.0

- When we move to X, high is set to 0.23 and low is set to 0.22

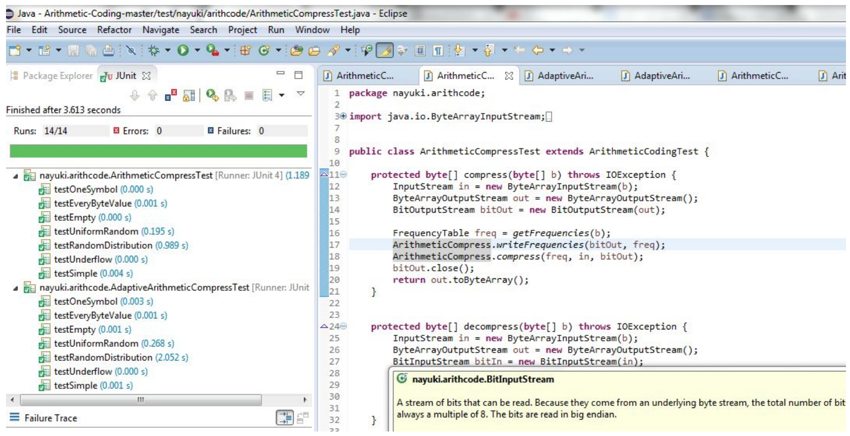
- X lies within W, Y within X, so on as can be seen...

- In this way, the gap b/w the high and low keeps on decreasing till we get the most precise interval and thus achieve compression.



We are working on the Arithmetic coding as of now and we would be using the results to compare with the compression comparison parameters with other algorithms namely LZW, LZ77, Run length encoding and Huffman encoding which are discussed further.

At the left, a panel of all the tests we are running can be seen for both adaptive and normal Arithmetic encoding techniques.



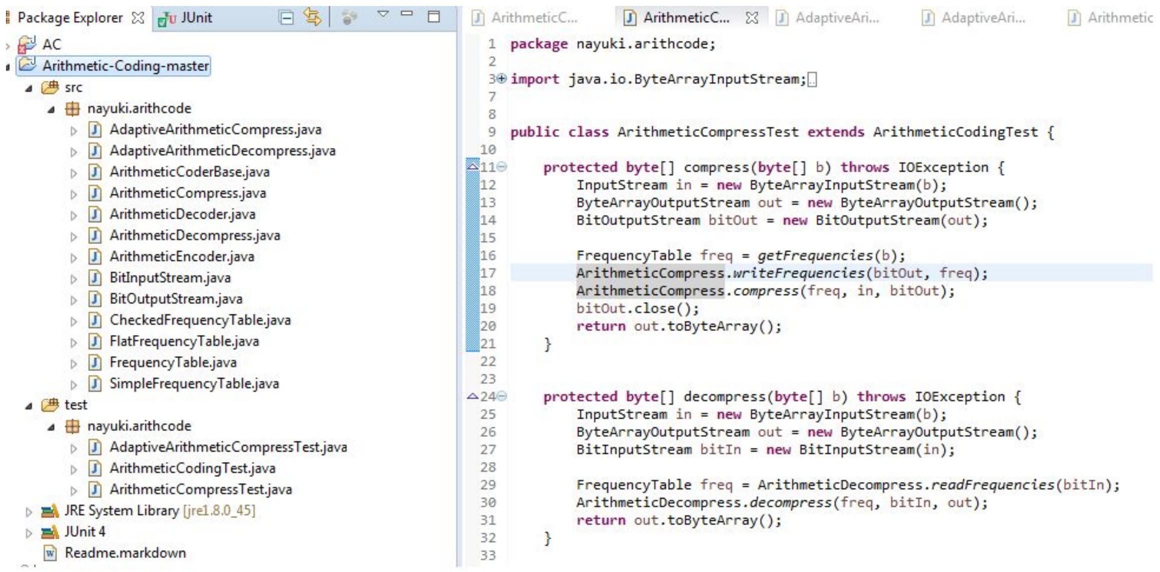
Java - Arithmetic-Coding-master/test/nayuki/arithcode/ArithmeticCompressTest.java - Eclipse

```
Finished after 3.613 seconds
Runs: 14/14     Errors: 0     Failures: 0

nayuki.arithcode.ArithmeticCompressTest [Runner: JUnit 4] (1.189
    testOneSymbol (0.000 s)
    testEveryByteValue (0.001 s)
    testEmpty (0.000 s)
    testUniformRandom (0.195 s)
    testRandomDistribution (0.989 s)
    testUnderflow (0.000 s)
    testSimple (0.004 s)
nayuki.arithcode.AdaptiveArithmeticCompressTest [Runner: JUnit
    testOneSymbol (0.003 s)
    testEveryByteValue (0.001 s)
    testEmpty (0.001 s)
    testUniformRandom (0.268 s)
    testRandomDistribution (2.052 s)
    testUnderflow (0.000 s)
    testSimple (0.001 s)
Failure Trace
```

```java
package nayuki.arithcode;

import java.io.ByteArrayInputStream;


public class ArithmeticCompressTest extends ArithmeticCodingTest {

    protected byte[] compress(byte[] b) throws IOException {
        InputStream in = new ByteArrayInputStream(b);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        BitOutputStream bitOut = new BitOutputStream(out);

        FrequencyTable freq = getFrequencies(b);
        ArithmeticCompress.writeFrequencies(bitOut, freq);
        ArithmeticCompress.compress(freq, in, bitOut);
        bitOut.close();
        return out.toByteArray();
    }


    protected byte[] decompress(byte[] b) throws IOException {
        InputStream in = new ByteArrayInputStream(b);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        BitInputStream bitIn = new BitInputStream(in);
```

nayuki.arithcode.BitInputStream

A stream of bits that can be read. Because they come from an underlying byte stream, the total number of bit always a multiple of 8. The bits are read in big endian.



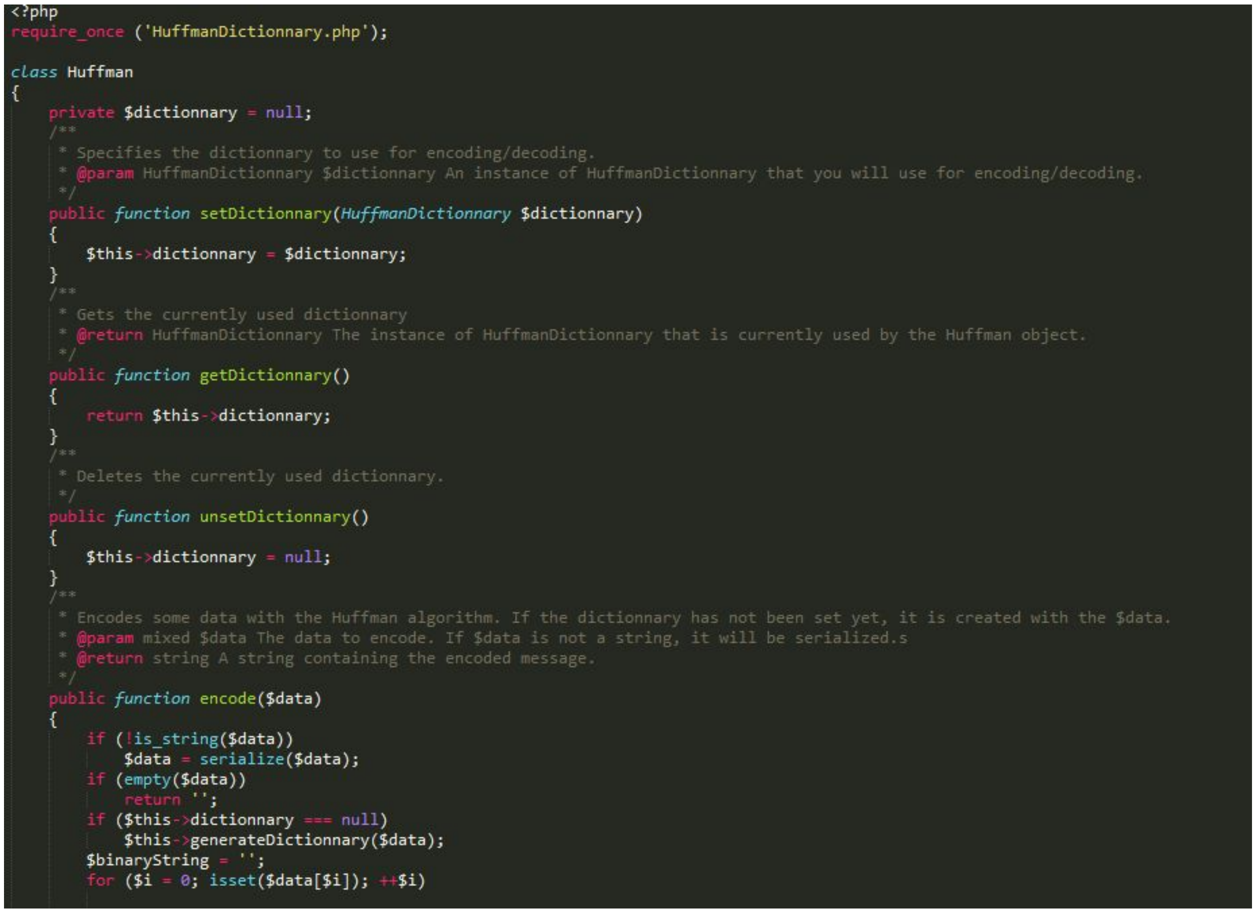
```
Package Explorer  JUnit
AC
Arithmetic-Coding-master
    src
        nayuki.arithcode
            AdaptiveArithmeticCompress.java
            AdaptiveArithmeticDecompress.java
            ArithmeticCoderBase.java
            ArithmeticCompress.java
            ArithmeticDecoder.java
            ArithmeticDecompress.java
            ArithmeticEncoder.java
            BitInputStream.java
            BitOutputStream.java
            CheckedFrequencyTable.java
            FlatFrequencyTable.java
            FrequencyTable.java
            SimpleFrequencyTable.java
    test
        nayuki.arithcode
            AdaptiveArithmeticCompressTest.java
            ArithmeticCodingTest.java
            ArithmeticCompressTest.java
    JRE System Library [jre1.8.0_45]
    JUnit 4
    Readme.markdown
```

```java
package nayuki.arithcode;

import java.io.ByteArrayInputStream;


public class ArithmeticCompressTest extends ArithmeticCodingTest {

    protected byte[] compress(byte[] b) throws IOException {
        InputStream in = new ByteArrayInputStream(b);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        BitOutputStream bitOut = new BitOutputStream(out);

        FrequencyTable freq = getFrequencies(b);
        ArithmeticCompress.writeFrequencies(bitOut, freq);
        ArithmeticCompress.compress(freq, in, bitOut);
        bitOut.close();
        return out.toByteArray();
    }


    protected byte[] decompress(byte[] b) throws IOException {
        InputStream in = new ByteArrayInputStream(b);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        BitInputStream bitIn = new BitInputStream(in);

        FrequencyTable freq = ArithmeticDecompress.readFrequencies(bitIn);
        ArithmeticDecompress.decompress(freq, bitIn, out);
        return out.toByteArray();
    }
```

HUFFMAN ENCODING : Huffman scheme uses a table of frequency of occurrence for each symbol (or character) in the input.

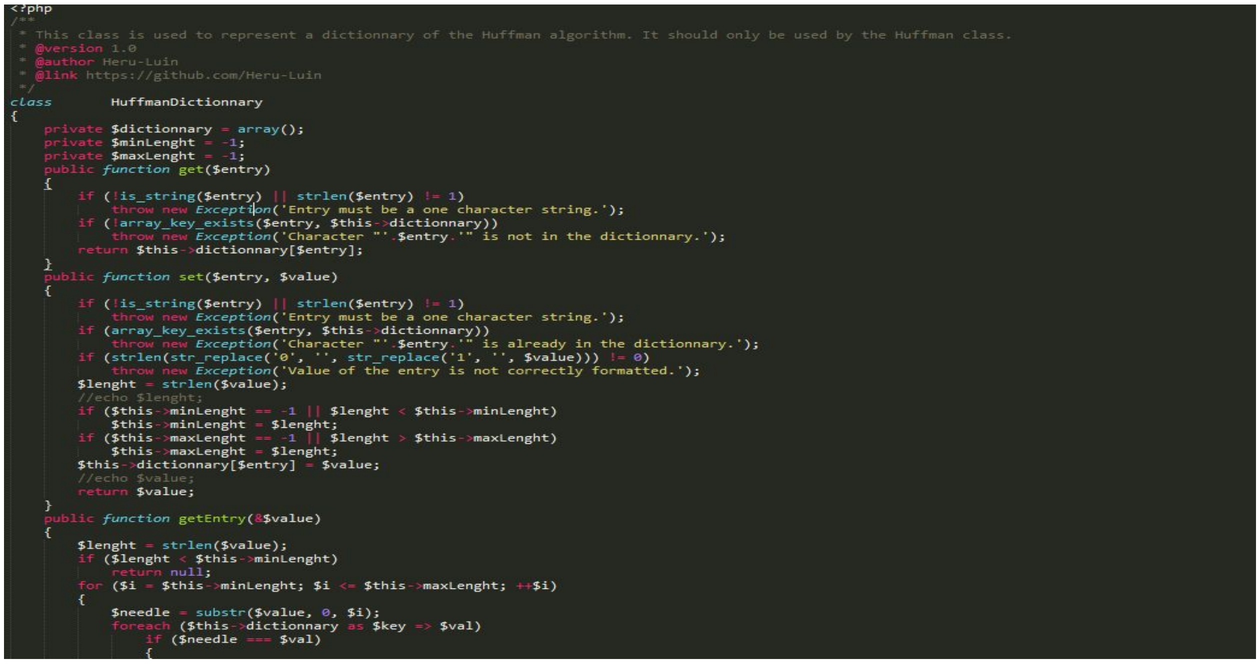
- This table may be derived from the input itself or from data which is representative of the input. For instance, the frequency of occurrence of letters in normal English might be derived from processing a large number of text documents and then used for encoding all text documents.

- We then need to assign a variable-length bit string to each character that unambiguously represents that character. This means that the encoding for each character must have a unique prefix. If the characters to be encoded are arranged in a binary tree



```php
<?php
require_once ('HuffmanDictionnary.php');

class Huffman
{
    private $dictionnary = null;
    /**
     * Specifies the dictionnary to use for encoding/decoding.
     * @param HuffmanDictionnary $dictionnary An instance of HuffmanDictionnary that you will use for encoding/decoding.
     */
    public function setDictionnary(HuffmanDictionnary $dictionnary)
    {
        $this->dictionnary = $dictionnary;
    }
    /**
     * Gets the currently used dictionnary
     * @return HuffmanDictionnary The instance of HuffmanDictionnary that is currently used by the Huffman object.
     */
    public function getDictionnary()
    {
        return $this->dictionnary;
    }
    /**
     * Deletes the currently used dictionnary.
     */
    public function unsetDictionnary()
    {
        $this->dictionnary = null;
    }
    /**
     * Encodes some data with the Huffman algorithm. If the dictionnary has not been set yet, it is created with the $data.
     * @param mixed $data The data to encode. If $data is not a string, it will be serialized.s
     * @return string A string containing the encoded message.
     */
    public function encode($data)
    {
        if (!is_string($data))
            $data = serialize($data);
        if (empty($data))
            return '';
        if ($this->dictionnary === null)
            $this->generateDictionnary($data);
        $binaryString = '';
        for ($i = 0; isset($data[$i]); ++$i)
```

HUFFMAN DICTIONARY



```php
<?php
/**
 * This class is used to represent a dictionnary of the Huffman algorithm. It should only be used by the Huffman class.
 * @version 1.0
 * @author Heru-Luin
 * @link https://github.com/Heru-Luin
 */
class       HuffmanDictionnary
{
    private $dictionnary = array();
    private $minLenght = -1;
    private $maxLenght = -1;
    public function get($entry)
    {
        if (!is_string($entry) || strlen($entry) != 1)
            throw new Exception('Entry must be a one character string.');
        if (!array_key_exists($entry, $this->dictionnary))
            throw new Exception('Character "'.$entry.'" is not in the dictionnary.');
        return $this->dictionnary[$entry];
    }
    public function set($entry, $value)
    {
        if (!is_string($entry) || strlen($entry) != 1)
            throw new Exception('Entry must be a one character string.');
        if (array_key_exists($entry, $this->dictionnary))
            throw new Exception('Character "'.$entry.'" is already in the dictionnary.');
        if (strlen(str_replace('0', '', str_replace('1', '', $value))) != 0)
            throw new Exception('Value of the entry is not correctly formatted.');
        $lenght = strlen($value);
        //echo $lenght;
        if ($this->minLenght == -1 || $lenght < $this->minLenght)
            $this->minLenght = $lenght;
        if ($this->maxLenght == -1 || $lenght > $this->maxLenght)
            $this->maxLenght = $lenght;
        $this->dictionnary[$entry] = $value;
        //echo $value;
        return $value;
    }
    public function getEntry(&$value)
    {
        $lenght = strlen($value);
        if ($lenght < $this->minLenght)
            return null;
        for ($i = $this->minLenght; $i <= $this->maxLenght; ++$i)
        {
            $needle = substr($value, 0, $i);
            foreach ($this->dictionnary as $key => $val)
                if ($needle === $val)
                {
```

We are in development phase when as we work upon huffman encoding. The outputs from these would be used to compare with other compression algorithms based on different parameters.

(Refer : https://www.cs.auckland.ac.nz/software/AlgAnim/huffman.html)

Some backgound on the compression algorithms we are using :

- LZ77 and LZ78 are lossless data compression algorithms : Forms basis of **DEFLATE** used in PNG files) **and GIF**

- Lossless Data Compression algorithms : Original data is completely reconstructed from the compressed data. Lossy data compression allows compression of only an approximation of the original data.(although improves compression rates)

- LDC algos used in ZIP file formats. PNG and GIF use lossless compressions. 3D graphics(OpenCTM), Cryptography.

❖ Both LZ77 and LZ78 are **DICTIONARY CODERS.(Substitution coders) :** A class of LDC algorithms.

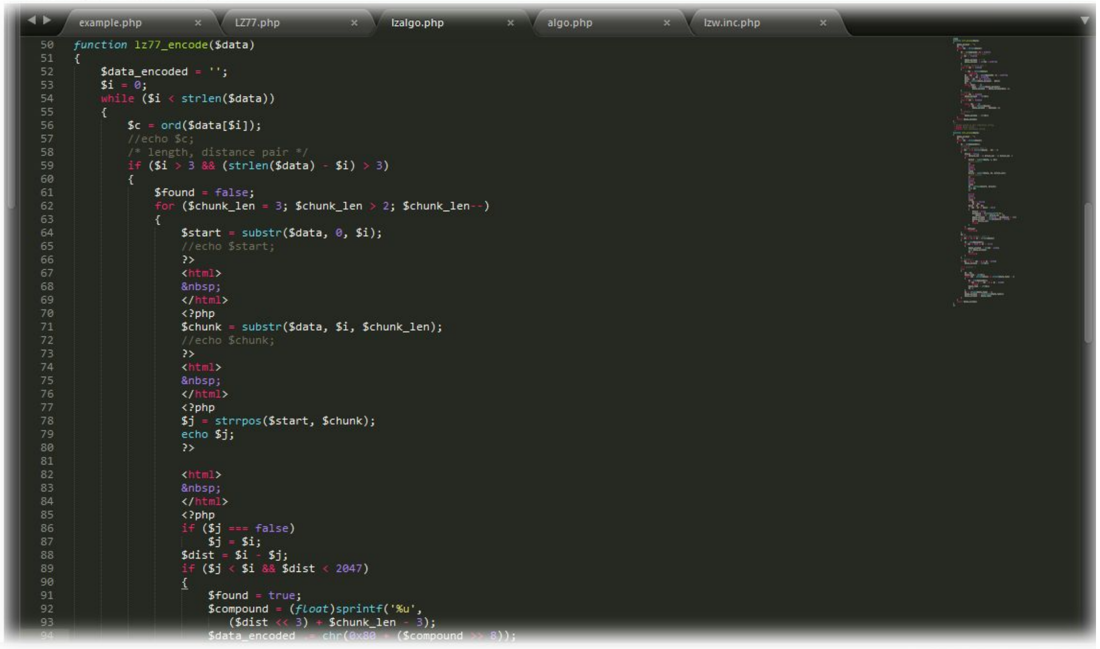
- ❖ Look for matches between data set to be compressed and a set of strings(contained in a data structure maintained by the encoder) called dictionary.

- ❖ When a match is found, a reference to the string's position in the data structure is given.

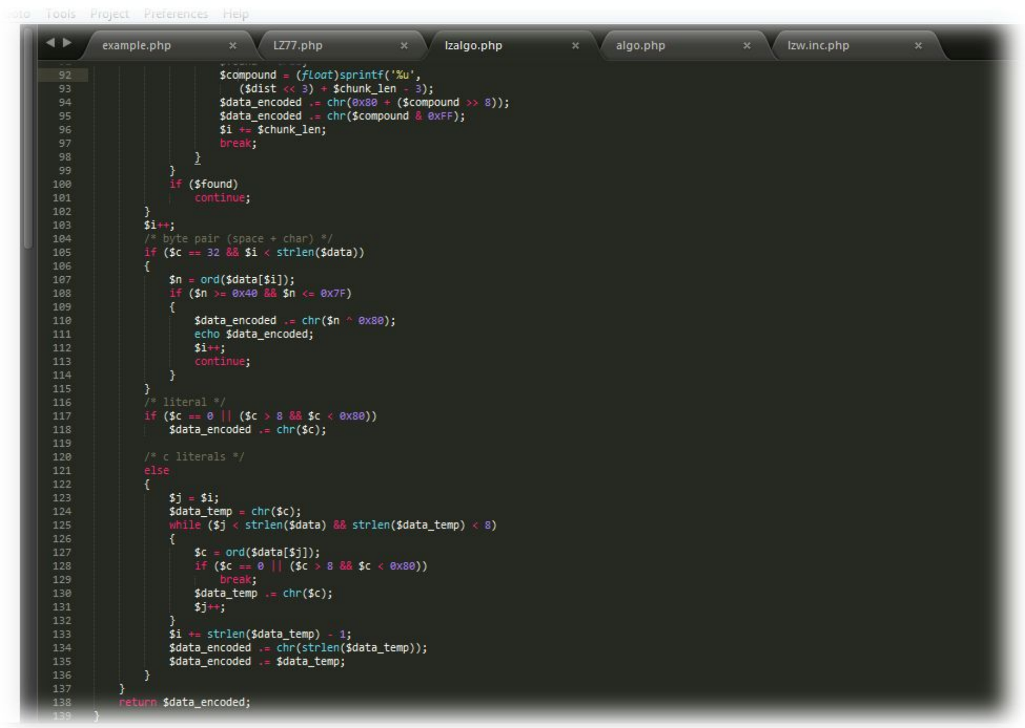
- ❖ In LZ77 compression MUST start at the beginning of the input itself.

- ❖ Typically every character is stored with 8 bits(up to 256 unique symbols for this data)..... can be extended to 9 to 12 bits per character. New unique symbols made of combinations of symbols which occurred previously in the string.

Below 2 screenshots depicts the code for compressing strings via LZ77 algorithm.

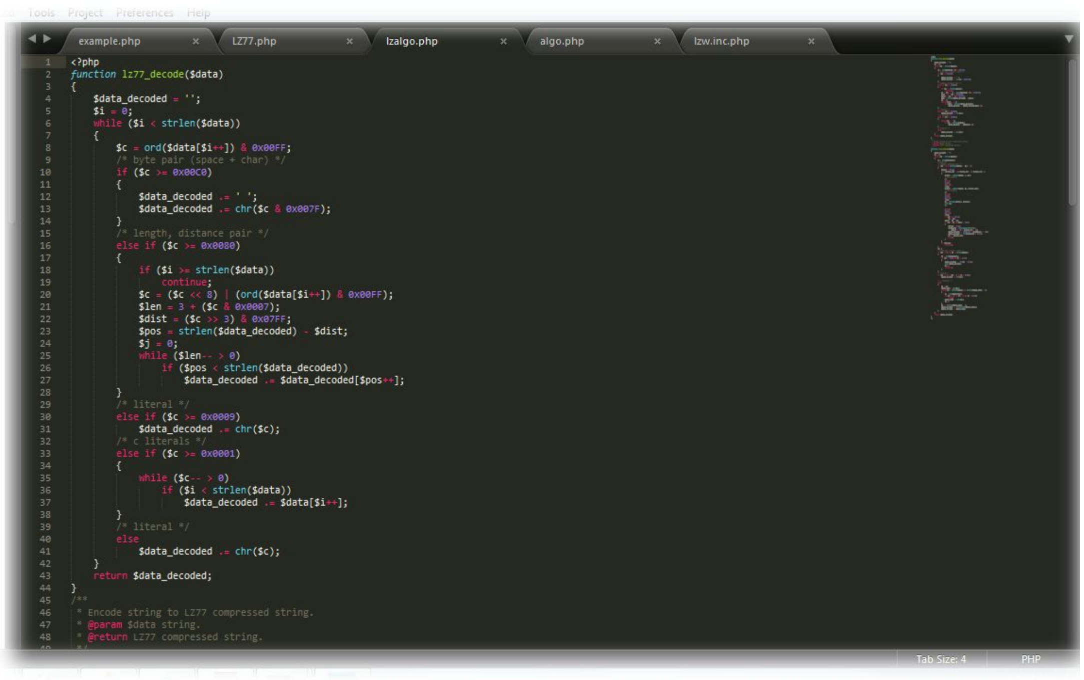


```php
function lz77_encode($data)
{
    $data_encoded = '';
    $i = 0;
    while ($i < strlen($data))
    {
        $c = ord($data[$i]);
        //echo $c;
        /* length, distance pair */
        if ($i > 3 && (strlen($data) - $i) > 3)
        {
            $found = false;
            for ($chunk_len = 3; $chunk_len > 2; $chunk_len--)
            {
                $start = substr($data, 0, $i);
                //echo $start;
                ?>
                <html>
                &nbsp;
                </html>
                <?php
                $chunk = substr($data, $i, $chunk_len);
                //echo $chunk;
                ?>
                <html>
                &nbsp;
                </html>
                <?php
                $j = strrpos($start, $chunk);
                echo $j;
                ?>

                <html>
                &nbsp;
                </html>
                <?php
                if ($j === false)
                    $j = $i;
                $dist = $i - $j;
                if ($j < $i && $dist < 2047)
                {
                    $found = true;
                    $compound = (float)sprintf('%u',
                        ($dist << 3) + $chunk_len - 3);
                    $data_encoded .= chr(0x80 + ($compound >> 8));
```



```php
                    $compound = (float)sprintf('%u',
                        ($dist << 3) + $chunk_len - 3);
                    $data_encoded .= chr(0x80 + ($compound >> 8));
                    $data_encoded .= chr($compound & 0xFF);
                    $i += $chunk_len;
                    break;
                }
            }
            if ($found)
                continue;
        }
        $i++;
        /* byte pair (space + char) */
        if ($c == 32 && $i < strlen($data))
        {
            $n = ord($data[$i]);
            if ($n >= 0x40 && $n <= 0x7F)
            {
                $data_encoded .= chr($n ^ 0x80);
                echo $data_encoded;
                $i++;
                continue;
            }
        }
        /* literal */
        if ($c == 0 || ($c > 8 && $c < 0x80))
            $data_encoded .= chr($c);

        /* c literals */
        else
        {
            $j = $i;
            $data_temp = chr($c);
            while ($j < strlen($data) && strlen($data_temp) < 8)
            {
                $c = ord($data[$j]);
                if ($c == 0 || ($c > 8 && $c < 0x80))
                    break;
                $data_temp .= chr($c);
                $j++;
            }
            $i += strlen($data_temp) - 1;
            $data_encoded .= chr(strlen($data_temp));
            $data_encoded .= $data_temp;
        }
    }
    return $data_encoded;
}
```

The one screenshot shown below describes the method for decompressing strings via LZ77 algorithm.



```php
<?php
function lz77_decode($data)
{
    $data_decoded = '';
    $i = 0;
    while ($i < strlen($data))
    {
        $c = ord($data[$i++]) & 0x00FF;
        /* byte pair (space + char) */
        if ($c >= 0x00C0)
        {
            $data_decoded .= ' ';
            $data_decoded .= chr($c & 0x007F);
        }
        /* length, distance pair */
        else if ($c >= 0x0080)
        {
            if ($i >= strlen($data))
                continue;
            $c = ($c << 8) | (ord($data[$i++]) & 0x00FF);
            $len = 3 + ($c & 0x0007);
            $dist = ($c >> 3) & 0x07FF;
            $pos = strlen($data_decoded) - $dist;
            $j = 0;
            while ($len-- > 0)
                if ($pos < strlen($data_decoded))
                    $data_decoded .= $data_decoded[$pos++];
        }
        /* literal */
        else if ($c >= 0x0009)
            $data_decoded .= chr($c);
        /* c literals */
        else if ($c >= 0x0001)
        {
            while ($c-- > 0)
                if ($i < strlen($data))
                    $data_decoded .= $data[$i++];
        }
        /* literal */
        else
            $data_decoded .= chr($c);
    }
    return $data_decoded;
}
/**
 * Encode string to LZ77 compressed string.
 * @param $data string.
 * @return LZ77 compressed string.
```