# Multimedia Systems

## Part 20

Mahdi Vasighi
www.iasbs.ac.ir/~vasighi

Department of Computer Science and Information Technology,
Institute for Advanced Studies in Basic Sciences, Zanjan, Iran

# Arithmetic Coding

- A widely used entropy coder.
- Variable length source coding technique
- Only problem is its speed due possibly complex computations due to large symbol tables.
- Good compression ratio (better than Huffman coding), entropy around the Shannon ideal value.
- Here we describe basic approach of Arithmetic Coding.

# Arithmetic Coding

The idea behind arithmetic coding is:

encode the entire message into a single real number, n, ($0.0 \leq n < 1.0$).

- Consider a probability line segment, [0. . . 1),
- Assign to every symbol a **range** in this interval
- Range is **proportional to probability** with position at cumulative probability.

Once we have defined the ranges and the probability line:

- Start to encode symbols.
- Every symbol defines where the output real number lands within the range.

# Arithmetic Coding

Assume we have the following string: **BACA**

- A occurs with probability 0.5.
- B and C with probabilities 0.25.

Start by assigning each symbol to the probability range [0. . . 1).

Sort symbols highest probability first:

| Symbol | Range |
|---|---|
| **A** | **[0.0, 0.5)** |
| **B** | **[0.5. 0.75)** |
| **C** | **[0.75, 1.0)** |

The first symbol in our example stream is B

# Arithmetic Coding

The first symbol in our example stream is B [0.5. 0.75)

- Subdivide the range for the first symbol

For the second symbol (range = 0.25, low = 0.5, high = 0.75)

| Symbol | Range |
|---|---|
| BA | [0.5, 0.625) |
| BB | [0.625. 0.6875) |
| BC | [0.6875, 0.75) |

reapply the subdivision of our scale again to get for our third symbol:(range = 0.125, low = 0.5, high = 0.625):

| Symbol | Range |
|---|---|
| BAA | [0.5, 0.5625) |
| BAB | [0.5625. 0.59375) |
| BAC | [0.59375, 0.625) |

# Arithmetic Coding

Subdivide again:

(range = 0.03125, low = 0.59375, high = 0.625):

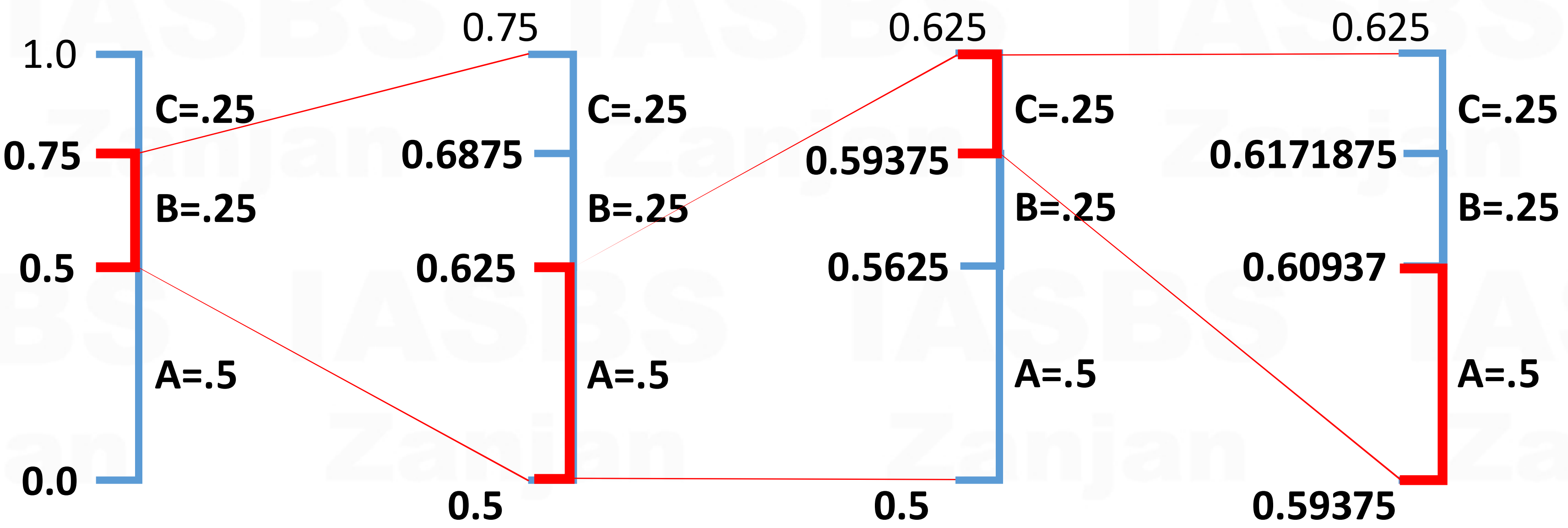
| Symbol | Range |
|---|---|
| **BACA** | **[0.59375, 0.60937)** |
| **BACB** | **[0.60937. 0.6171875)** |
| **BACC** | **[0.6171875, 0.625)** |

So the (unique) output code for BACA is any number in the range:

[0.59375, 0.60937)

This number is referred to as a **tag**.

# Arithmetic Coding




| Sym | Range |
|---|---|
| A | [0.0, 0.5) |
| B | [0.5. 0.75) |
| C | [0.75, 1.0) |

| Sym | Range |
|---|---|
| BA | [0.5, 0.625) |
| BB | [0.625. 0.6875) |
| BC | [0.6875, 0.75) |

| Sym | Range |
|---|---|
| BAA | [0.5, 0.5625) |
| BAB | [0.5625. 0.59375) |
| BAC | [0.59375, 0.625) |

| Sym | Range |
|---|---|
| BACA | [0.59375, 0.60937) |
| BACB | [0.60937. 0.6171875) |
| BACC | [0.6171875, 0.625) |

# Arithmetic Coding

Suppose the alphabet is [A, B,C, D, E, F, $] with known probability distribution.

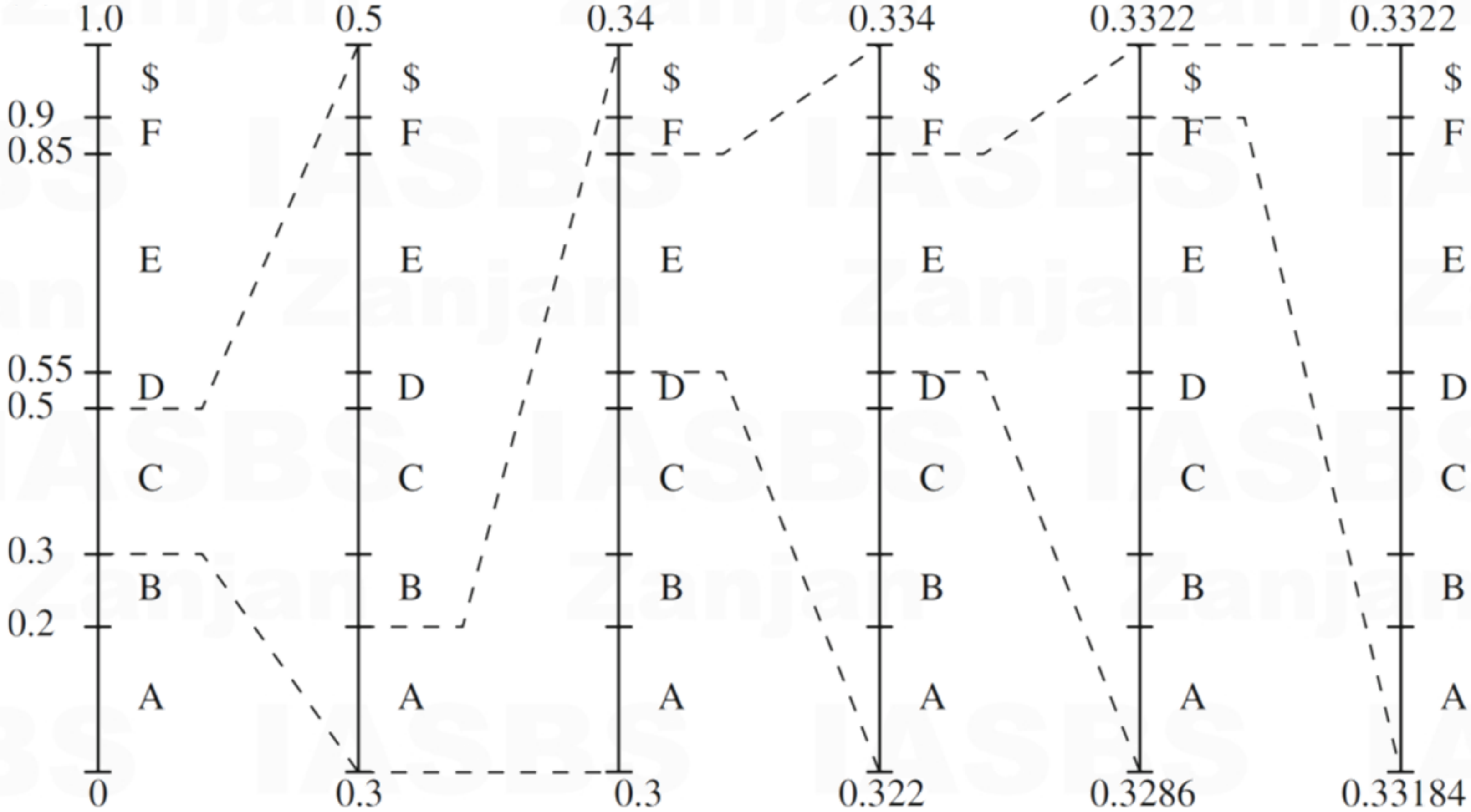
**$** is a special symbol used to terminate the message

We want to encode a string of symbols **CAEE$**

| Symbol | Probability | Range | Range_low | Range_high |
|---|---|---|---|---|
| A | 0.2 | [0,0.2) | 0 | 0.2 |
| B | 0.1 | [0.2,0.3) | 0.2 | 0.3 |
| C | 0.2 | [0.3,0.5) | 0.3 | 0.5 |
| D | 0.05 | [0.5,0.55) | 0.5 | 0.55 |
| E | 0.3 | [0.55,0.85) | 0.55 | 0.85 |
| F | 0.05 | [0.85,0.9) | 0.85 | 0.9 |
| $ | 0.1 | [0.9,1.0) | 0.9 | 1.0 |

# Arithmetic Coding

Suppose the alphabet is [A, B,C, D, E, F, $] with known probability distribution. We want to encode a string of symbols **CAEE$**




**low= low + range × Range_low(sym);**
**high= low + range × Range_high(sym);**

# Arithmetic Coding

```
BEGIN
   low = 0.0; high = 1.0; range = 1.0;
   initialize symbol;
   while (symbol $= terminator)
      { get (symbol);
         low = low + range * Range_low(symbol);
         high = low + range * Range_high(symbol);
         range = high - low; }
   output a code so that low <= code < high;
END
```

| Symbol | low | high | range |
|---|---|---|---|
| | 0 | 1.0 | 1.0 |
| C | 0.3 | 0.5 | 0.2 |
| A | 0.30 | 0.34 | 0.04 |
| E | 0.322 | 0.334 | 0.012 |
| E | 0.3286 | 0.3322 | 0.0036 |
| $ | 0.33184 | 0.33220 | 0.00036 |

**[0.33184, 0.33220)**

range = $P_C \times P_A \times P_E \times P_E \times P_\$$
= 0.2 × 0.2 × 0.3 × 0.3 × 0.1
= 0.00036

# Arithmetic Coding

| Binary fractional | Decimal |
|---|---|
| 0.1 | 0.5 |
| 0.01 | 0.25 |
| 0.001 | 0.125 |
| 0.0001 | 0.0625 |
| 0.00001 | 0.0313 |
| 0.000001 | 0.0156 |
| 0.0000001 | 0.0078 |
| 0.00000001 | 0.0039 |

0.01010101

treat the whole message as one unit

$$2^{-2} + 2^{-4} + 2^{-6} + 2^{-8}$$
$$= 0.33203125$$

| Symbol | low | high | range |
|---|---|---|---|
| | 0 | 1.0 | 1.0 |
| C | 0.3 | 0.5 | 0.2 |
| A | 0.30 | 0.34 | 0.04 |
| E | 0.322 | 0.334 | 0.012 |
| E | 0.3286 | 0.3322 | 0.0036 |
| $ | 0.33184 | 0.33220 | 0.00036 |

**[0.33184, 0.33220)**

**range = $P_C \times P_A \times P_E \times P_E \times P_\$$**
**= 0.2 × 0.2 × 0.3 × 0.3 × 0.1**
**= 0.00036**

# Arithmetic Coding

In the worst case, the shortest codeword in arithmetic coding will require *k* bits to encode a sequence of symbols:

$$k = log_2 \frac{1}{range} = log_2 \frac{1}{\prod_i P_i}$$

Arithmetic coding achieves better performance than Huffman coding but it has some limitations:

- long sequences of symbols: a very small range. It requires very high-precision numbers
- The encoder will not produce any output codeword until the entire sequence is entered.

# Binary Arithmetic Coding

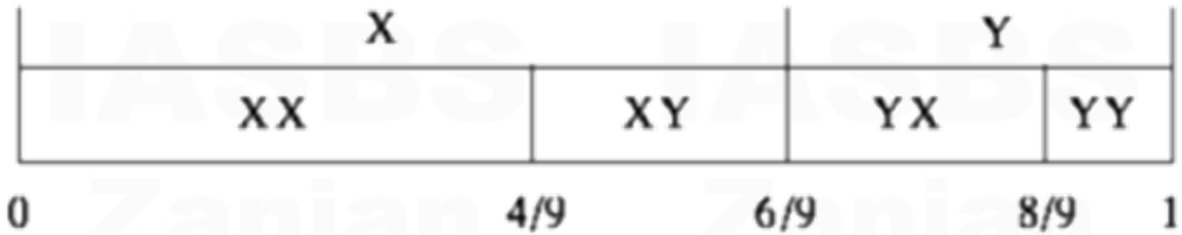
Binary Arithmetic Coding deals with two symbols only, 0 and 1 and uses binary fractions.

Idea: Suppose alphabet was X, Y and consider stream:

**XXY**

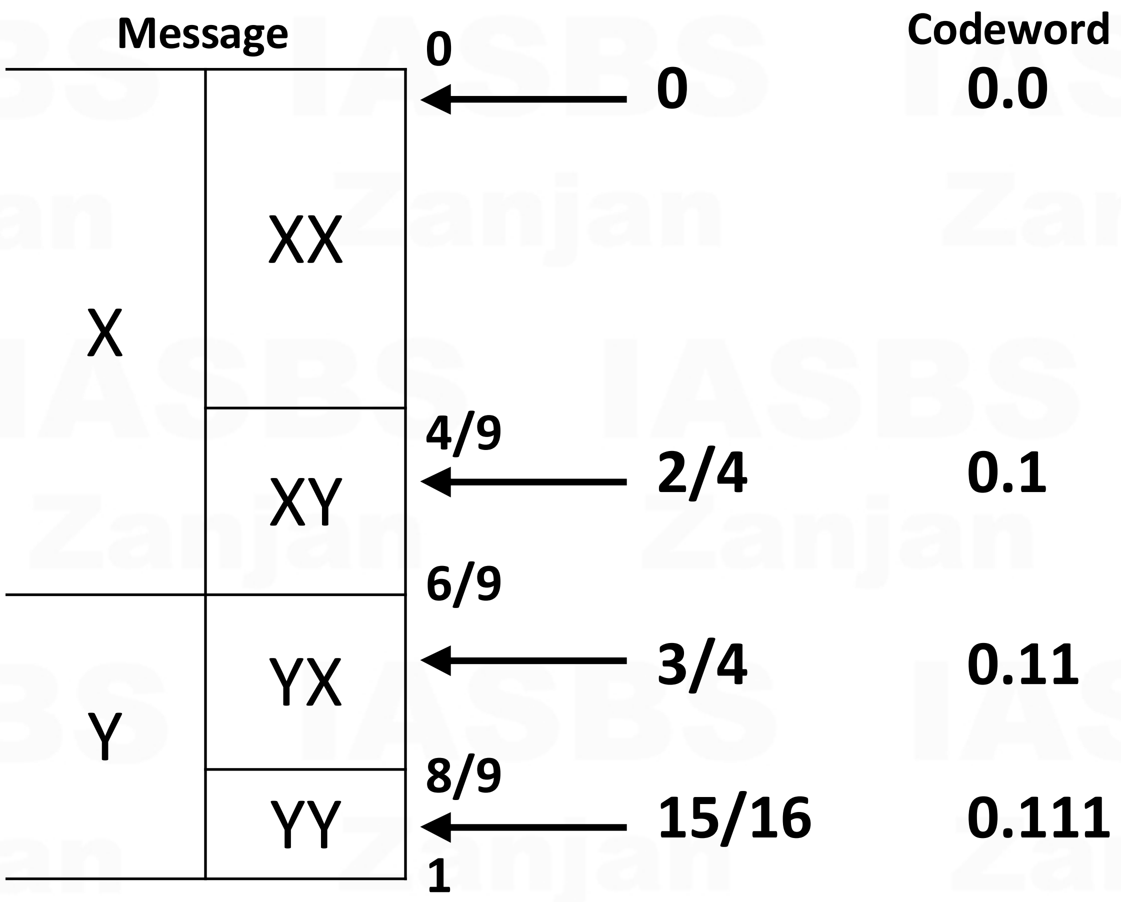
Therefore: $P(X) = 2/3$, $P(Y) = 1/3$

For encoding length 2 messages, we can map all possible messages to intervals in the range [0. . . 1):

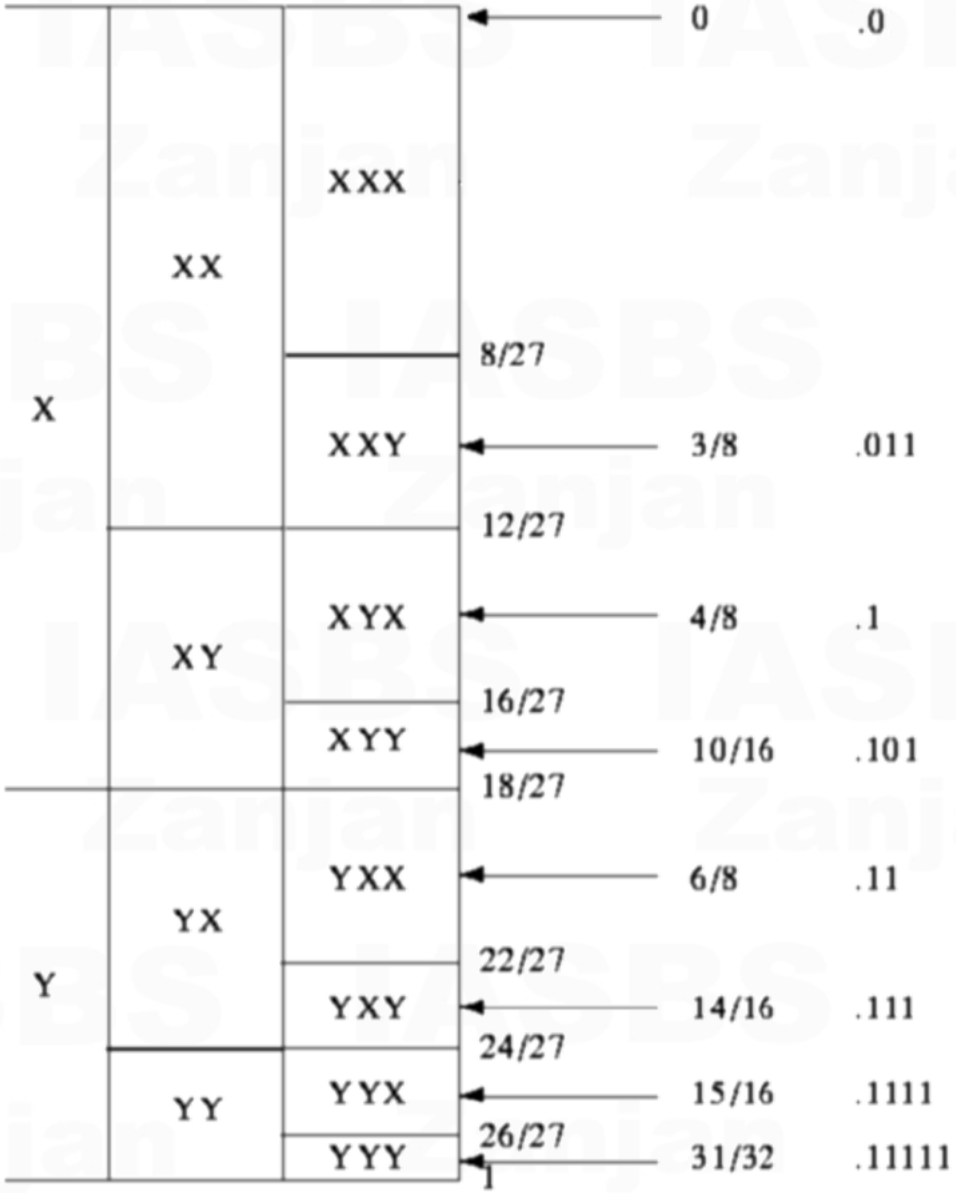


# Binary Arithmetic Coding

To encode message, just send enough bits of a binary fraction that uniquely specifies the interval.



# Binary Arithmetic Coding




Similarly, we can map all possible length 3 messages to intervals range [0. . . 1)

$-\log_2 p$ bits to represent interval of size p.

$-\text{Log}_2(1/27)=4.7549 \cong 5$

# Lempel-Ziv-Welch (LZW) Algorithm

- A very common compression technique.
- Used in GIF files (LZW), Adobe PDF file (LZW),
- Patented: LZW Patent expired in 2003/2004.

## Basic idea/Example

Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries.

$$\lceil log_2 159000 \rceil = 18 \; bits$$

Why not just transmit each word as an 18 bit number?

# Lempel-Ziv-Welch (LZW) Algorithm

**Problem**

- Too many bits per word
- Everyone needs a dictionary to decode back to English.
- Only works for English text.

**Solution**

- Find a way to build the dictionary adaptively.
- Original methods (LZ) due to Lempel and Ziv in 1977.
- Terry Welch improvement (1984), Patented LZW Algorithm
    - **LZW** idea is that only the initial dictionary needs to be transmitted to enable decoding:
    - The decoder is able to build the rest of the table from the encoded sequence.

# Lempel-Ziv-Welch (LZW) Algorithm

```
BEGIN
   s = next input character;
   while not EOF
      { c = next input character;
      if s + c exists in the dictionary
         s = s + c;
      else
         { output the code for s;
         add string s + c to the dictionary
         with a new code;
         s = c; }
      }
   output the code for s;
END
```

# Lempel-Ziv-Welch (LZW) Algorithm

An example of a stream containing only two alphabets:

**BABAABAAA**

Let us start with a very simple dictionary (string table)

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| **output code** | **representing** | **index** | **string** |
| | | **0** | **A** |
| | | **1** | **B** |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Lempel-Ziv-Welch (LZW) Algorithm

BABAABAAA

$s = B \rightarrow A$
$c = A$

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| output code | representing | index | string |
| | | 0 | A |
| | | 1 | B |
| 1 | B | 2 | BA |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Lempel-Ziv-Welch (LZW) Algorithm

BABAABAAA

$s = A \rightarrow B$
$c = B$

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| output code | representing | index | string |
| | | 0 | A |
| | | 1 | B |
| 1 | B | 2 | BA |
| 0 | A | 3 | AB |
| | | | |
| | | | |
| | | | |
| | | | |

# Lempel-Ziv-Welch (LZW) Algorithm

BABAABAAA

$s = BA \rightarrow A$
$c = A$

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| output code | representing | index | string |
| | | 0 | A |
| | | 1 | B |
| 1 | B | 2 | BA |
| 0 | A | 3 | AB |
| 2 | BA | 4 | BAA |
| | | | |
| | | | |
| | | | |

# Lempel-Ziv-Welch (LZW) Algorithm

BABAABAAA

$s = AB \rightarrow A$
$c = A$

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| output code | representing | index | string |
| | | 0 | A |
| | | 1 | B |
| 1 | B | 2 | BA |
| 0 | A | 3 | AB |
| 2 | BA | 4 | BAA |
| 3 | AB | 5 | ABA |
| | | | |
| | | | |

# Lempel-Ziv-Welch (LZW) Algorithm

BABAABAAA

$s = A \rightarrow A$
$c = A$

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| output code | representing | index | string |
| | | 0 | A |
| | | 1 | B |
| 1 | B | 2 | BA |
| 0 | A | 3 | AB |
| 2 | BA | 4 | BAA |
| 3 | AB | 5 | ABA |
| 0 | A | 6 | AA |
| | | | |

# Lempel-Ziv-Welch (LZW) Algorithm

BABAABAAA

s = AA
c = empty

| OUTPUT | | STRING TABLE | |
|---|---|---|---|
| output code | representing | index | string |
| | | 0 | A |
| | | 1 | B |
| 1 | B | 2 | BA |
| 0 | A | 3 | AB |
| 2 | BA | 4 | BAA |
| 3 | AB | 5 | ABA |
| 0 | A | 6 | AA |
| 6 | AA | | |

# Lempel-Ziv-Welch (LZW) Algorithm

The LZW decompressor creates the same string table during decompression. decompress the output sequence of previous example:

**1 0 2 3 0**

| ENCODER OUTPUT | STRING TABLE | |
|---|---|---|
| string | codeword | string |
| B | | |
| A | 2 | BA |
| | | |
| | | |
| | | |

# Lempel-Ziv-Welch (LZW) Algorithm

The LZW decompressor creates the same string table during decompression. decompress the output sequence of previous example:

**1 0 2 3 0**

| ENCODER OUTPUT | STRING TABLE | |
|---|---|---|
| **string** | **codeword** | **string** |
| **B** | | |
| **A** | **2** | **BA** |
| BA | 3 | AB |
| | | |
| | | |

# Lempel-Ziv-Welch (LZW) Algorithm

The LZW decompressor creates the same string table during decompression. decompress the output sequence of previous example:

**1 0 2 3 0**

| ENCODER OUTPUT | STRING TABLE | |
|---|---|---|
| **string** | **codeword** | **string** |
| **B** | | |
| **A** | **2** | **BA** |
| BA | 3 | AB |
| AB | 4 | BAA |
| | | |

# Lempel-Ziv-Welch (LZW) Algorithm

The LZW decompressor creates the same string table during decompression. decompress the output sequence of previous example:

**1 0 2 3 0**

| ENCODER OUTPUT | STRING TABLE | |
|---|---|---|
| **string** | **codeword** | **string** |
| **B** | | |
| **A** | **2** | **BA** |
| BA | 3 | AB |
| AB | 4 | BAA |
| A | 5 | ABA |