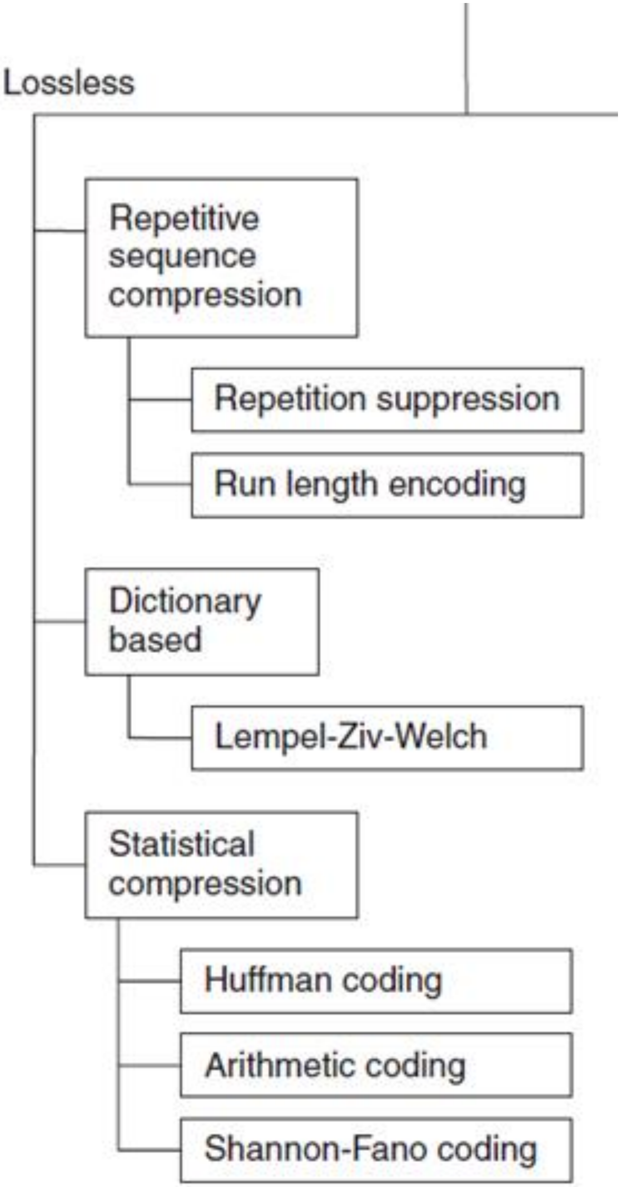
CMSC5714
Multimedia Technology

Dictionary based Lossless Compression and their Applications

Lossless Compression

- Coding Schemes
  - Repetitive Sequence
  - Statistical Compression/ Entropy Encoding
  - **Dictionary based**
- Compression commonly employ **redundancy** in the data



Dictionary based Compression

- Both compression and decompression uses a "dictionary"
- Here, the term "dictionary" is a mapping between
  - **a symbol or a string of symbols**
  - **an index**.
- Very like the Index section at the back of a book

Basic Consideration for Dictionary based Compression

- Split the input into classes:
  - frequently occurring, and
  - infrequently occurring
- Keep a list , or ***DICTIONARY***, of frequently occurring patterns
- Encode these repeated patterns with a shorter codeword/index

Dictionary Techniques

- The size of the dictionary must be much smaller that the number of all possible patterns
- Useful with sources that generate a relatively small number of patterns, such as text sources and computer commands

Dictionary based Compression

Two major categories

- Static dictionary compression
- Adaptive dictionary compression
  - LZ77
  - LZ78
  - LZW

Static Dictionary Coding

- The dictionary is permanent (or allowing addition, but not deletion)
- Application-specific, or data specific
- Example:

Digram Coding for text compression

be, th, ie, ch, sh, ar, or, en,…..

Static Dictionary Coding

Let the source alphabet
A={a,b,....z,., ,,!,?, :, ;} of size 32 (26+some other punctuations)

- For a 4-character word, there are $32^4=2^{20}$ patterns
- If Fixed–length coding is used in the dictionary, the original coding needs 20 bits/word

Example of static dictionary

- Put 256 $=2^8$ most frequently occurring patterns into a dictionary
- If a pattern is in the dictionary
  - (1-bit flag)+(8-bit index)=9 bits
- else
  - (1-bit flag)+(20-bit code)= 21 bits

Example of static dictionary

- $L = 9p+21(1-p)$

  $= 21-12p$ bits/word ,

  where p is the probability of a pattern in a dictionary
- We will have $L < 20$, if $p>0.0833$
  - p is to be skewed to get high compression

Problem of Static Dictionary

- However, a static dictionary may not work well for some inputs

- You may consider to use Oxford English dictionary which contains about 159,000 entries
- As a result, we have to use

  log2 (159,000) = 18 bits

  to encode words in any text

Problem of Static Dictionary

- It only be applicable to English
- Moreover, some words may not have high occurrences
- Also, the static dictionary must be available before the decoding
  - Sending of static dictionary before the decoding is necessary

Adaptive Dictionary Coding

Adaptive Dictionary Coding

- For static dictionary, everyone needs a dictionary to decode back to English. Only works for English text or a particular language
  - Too many bits per word
  - Synchronize the change of dictionary between sender and receiver
- A wise solution is to find a way to build the dictionary adaptively
- LZ family of algorithm
  - due to Lempel and Ziv in 1977/8. Quite a few variations on LZ.
  - Terry Welch improvement (1984), Patented LZW Algorithm

Lempel-Ziv Algorithms

LZ77 (Sliding Window)

- Variants: LZSS (Lempel-Ziv-Storer-Szymanski)
- Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (Dictionary Based)

- Variants: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Applications: `compress`, GIF, CCITT (modems), ARC, PAK

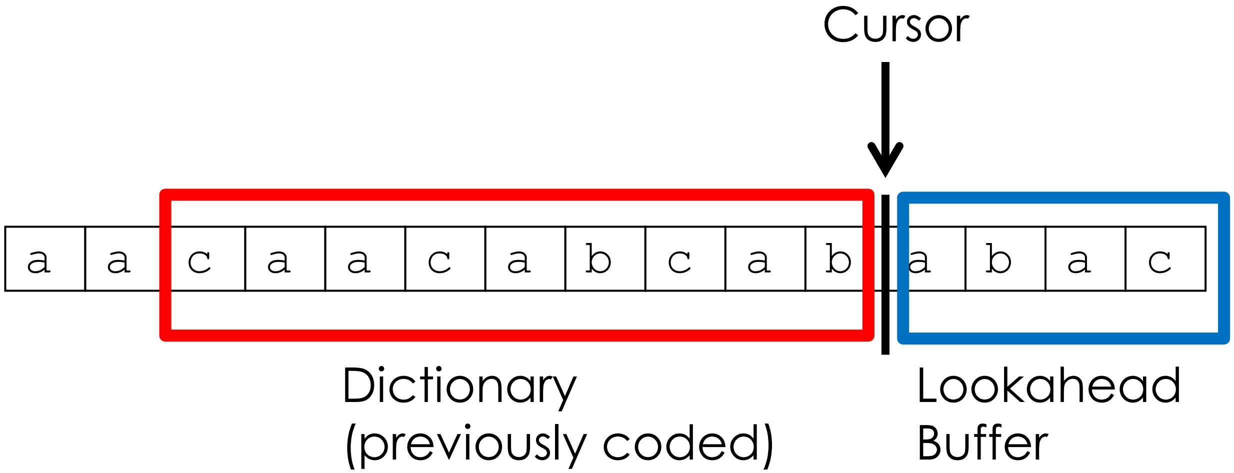
Traditionally LZ77 was better but slower, but the gzip version is almost as fast as any LZ78.

LZ77: Basic Idea

- Adapt to the characteristics of the source.
- The dictionary is a portion of the previously encoded sequence.
- Start with an empty dictionary.
- Add entries as they are found in the input stream.

LZ77 Algorithm

- Also referred as Sliding Window Lempel-Ziv
- Dictionary and buffer “windows” are fixed length and slide with the cursor



Cursor

a	a	c	a	a	c	a	b	c	a	b	a	b	a	c

Dictionary
(previously coded)

Lookahead
Buffer

LZ77 Algorithm

In each step:

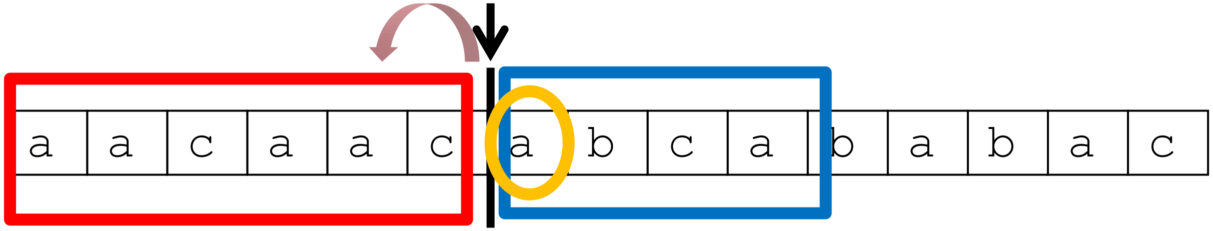
- Output (p,l,c)

  p = offset position of the longest match in the dictionary
  l = length of longest match
  c = next char in buffer beyond longest match
- Advance window by l + 1

- Loop until whole sequence is processed

LZ77 Algorithm

- An example:
  - Dictionary buffer of size 6, look-ahead buffer of size 6
  - A match (“a”) is found inside the lookahead from the Dictionary buffer
  - So the output: <2,1,b>



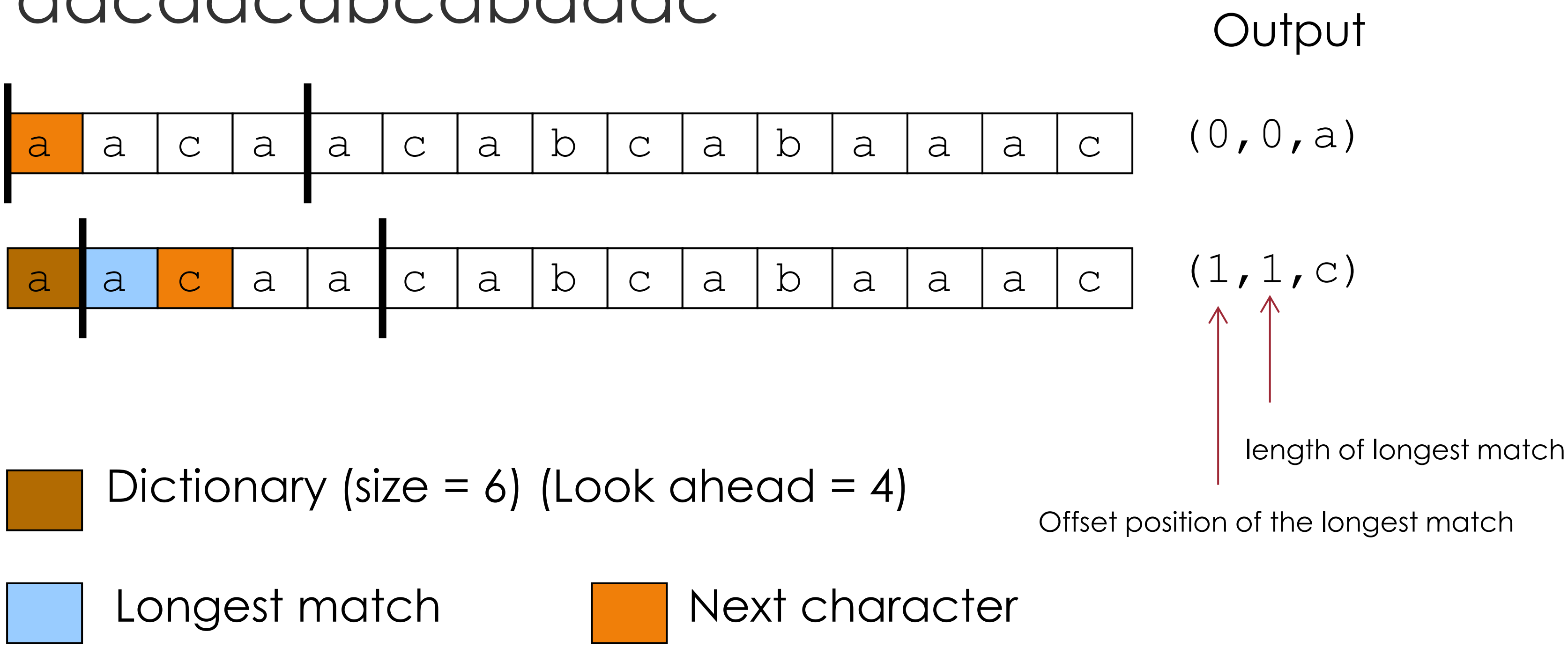
a	a	c	a	a	c	a	b	c	a	b	a	b	a	c

Dictionary
Buffer/ Window

Lookahead
Buffer

LZ77: Example

- We have a sequence aacaacabcabaaac



Output

a	a	c	a	a	c	a	b	c	a	b	a	a	a	c	(0,0,a)
a	a	c	a	a	c	a	b	c	a	b	a	a	a	c	(1,1,c)

length of longest match

Offset position of the longest match

🟫 Dictionary (size = 6) (Look ahead = 4)

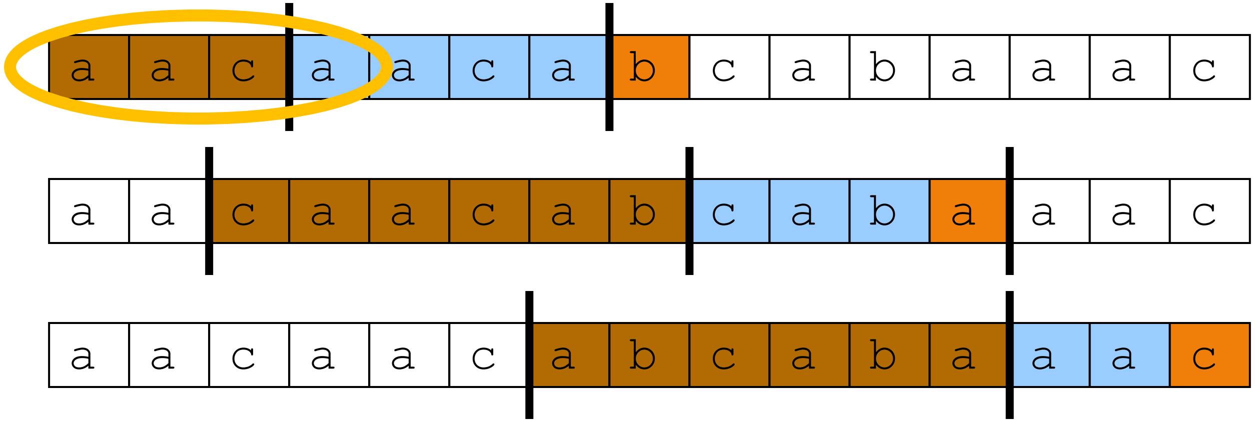
🟦 Longest match

🟧 Next character

LZ77: Example

We can encode even out of the dictionary window !!

Output



a a c a a c a b c a b a a a c (3,4,b)

a a c a a c a b c a b a a a c (3,3,a)

a a c a a c a b c a b a a a c (1,2,c)

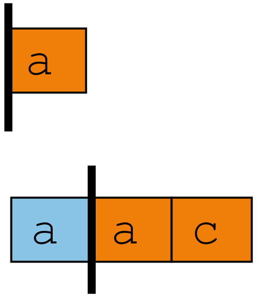
LZ77 Decoding

- Decoder keeps same dictionary window as encoder
- For each message it looks it up in the dictionary and inserts a copy

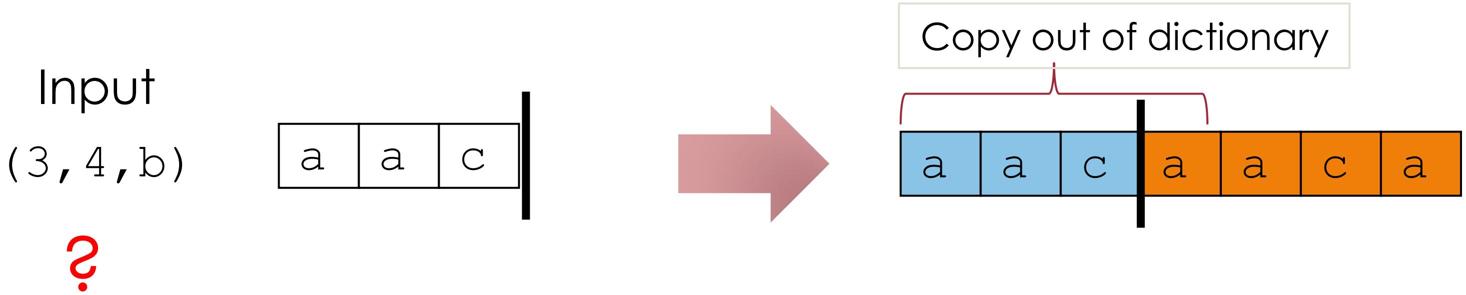
Input

`(0,0,a)`

`(1,1,c)`



LZ77 Decoding



Input

`(3,4,b)`

?

a	a	c

Copy out of dictionary

a	a	c	a	a	c	a

What if l > p? (only part of the message is in the dictionary.)

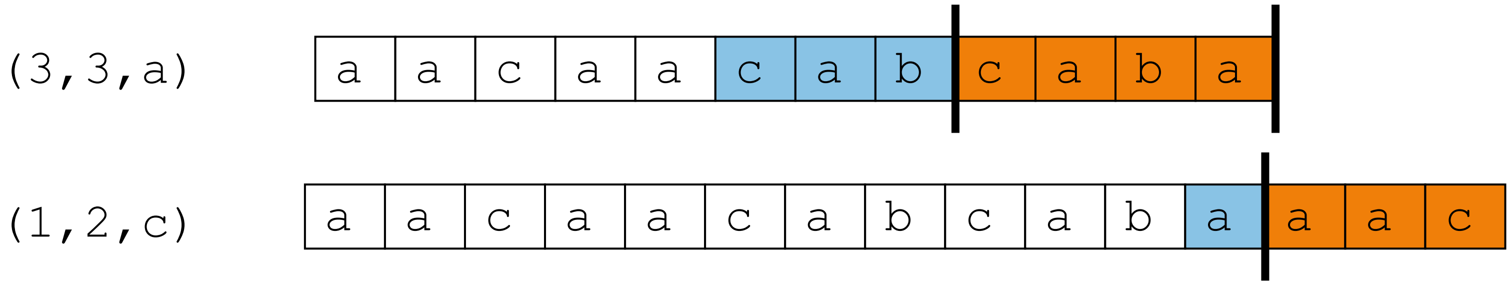
- E.g. dict = `aac`, codeword = `(3,4,a)`
- Simply copy out of the dictionary

```
for (i = 0; i < length; i++)
  out[cursor+i] = out[cursor-offset+i]
```

- Out =aacaaca

LZ77 Decoding

Input



(3,3,a) a a c a a c a b c a b a

(1,2,c) a a c a a c a b c a b a a a c

LZ77 : Optimization

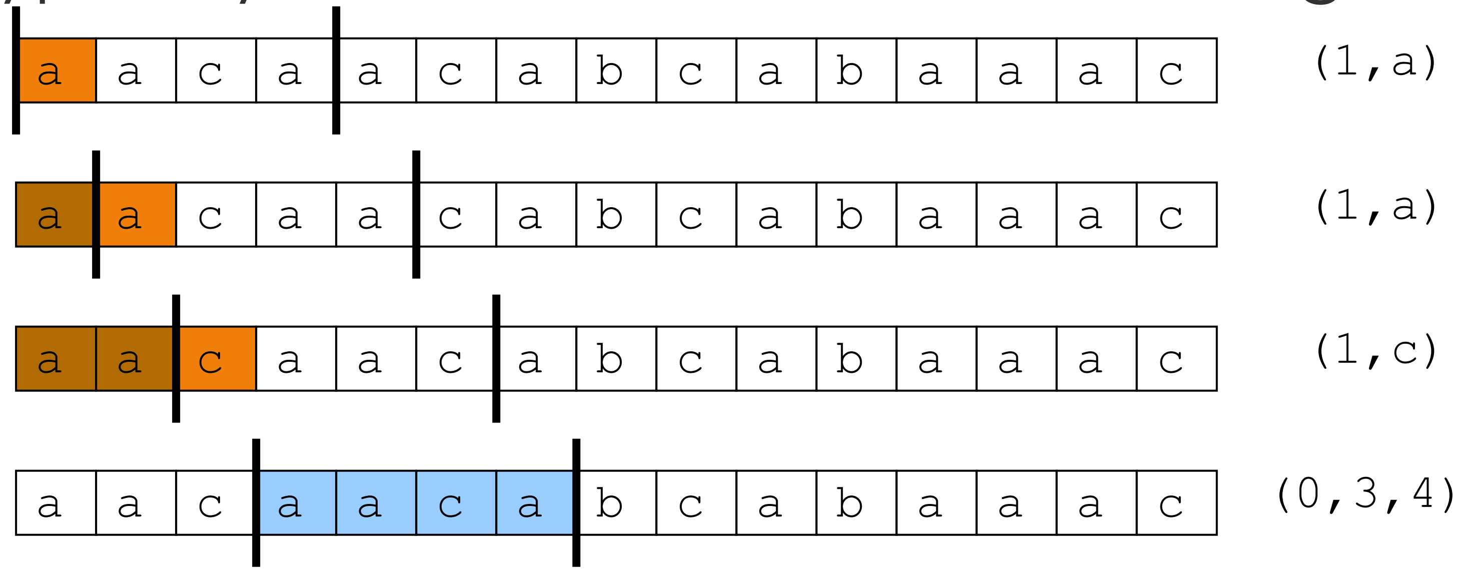
- LZSS : an improved version used by gzip
- Improvements
  - Encode two fields instead of three
  - Use a flag bit to indicate whether what follows is the codeword for a new symbol.
    - 0- for single characters
    - 1-for triples

LZ77: Optimizations

LZSS: Output one of the following formats

`(0, position, length)` or `(1,char)`

Typically use the second format if length < 3.



a a c a a c a b c a b a a a c (1,a)

a a c a a c a b c a b a a a c (1,a)

a a c a a c a b c a b a a a c (1,c)

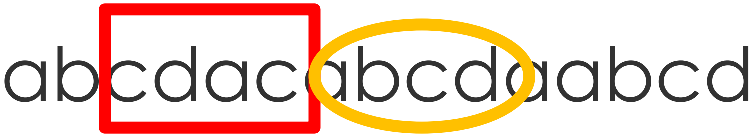
a a c a a c a b c a b a a a c (0,3,4)

LZ77: Optimizations (cont.)

- Huffman code the positions, lengths and chars
- Non greedy: possibly use shorter match so that next match is better
- Use hash table to store dictionary.
  - Hash is based on strings of length 3.
  - Find the longest match within the correct hash bucket.
  - Limit on length of search.
  - Store within bucket in order of position

Problem of LZ77

- Consider to encode the following sequence with a dictionary window of size 4



abcdacabcdaabcd

- We can see the pattern "abcd" repeats!
- But we can not encode it, as it is already out of the window

Analysis of LZ77

- LZ77 assumes patterns in the input stream occur close together
- Any pattern that recurs over a period longer than the search buffer size will not be captured
- The selection of window size is therefore critical for LZ77
  - But there is no optimal number of these sizes
  - The larger, the better!
  - In practice, DEFLATE uses 32k window size, DEFLATE64 uses 64k!

Analysis of LZ77

- When increasing L (or S), longer matches would be possible, thus compression efficiency increases
- But search for longer matches would reduce the speed.
- When increasing the length of buffers, compression efficiency drops
- A better compression method would save frequently occurring patterns in the dictionary separately

LZ78

- Instead of using a sliding window for dictionary
- We build a dictionary separately instead of based on the immediate previous sequence

LZ78: Algorithm

Basic algorithm:

- Keep dictionary of words with integer ***id*** for each entry
- Coding loop
  - find the longest match ***S*** in the dictionary
  - Output a pair ***(id,c)***, *where* ***id*** is the index of the match S into the dictionary, and **c** is the next character immediately after the match
    - id will be 0 if no match
  - Add a new phrase ***Sc*** to the dictionary

LZ78: Algorithm

Given that, we have built dictionary on the right, and there we are continue to encode:

…cbaac…

- The longest match is “cba”
- So our output pair ***(id,c)****will be*

(4,a)

- Then, we add a new entry “cbaa” to dictionary *and make its index as 5*

The Dictionary

Phase	Index
a	1
ca	2
cb	3
cba	4

+

cbaa	5

LZ78: Coding Example

- E.g. we have the sequence aabaacabcabcb

Output

Dict.

a	a	b	a	a	c	a	b	c	a	b	c	b

(0,a)

Phase	Index
a	1

a	a	b	a	a	c	a	b	c	a	b	c	b

(1,b)

Phase	Index
a	1
ab	2

LZ78: Example

a	a	b	a	a	c	a	b	c	a	b	c	b

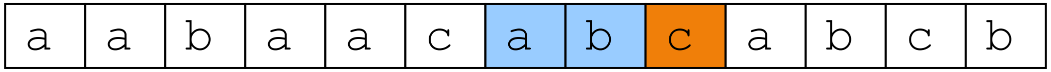
(1,a)

Phase	Index
a	**1**
ab	2
aa	3

a	a	b	a	a	c	a	b	c	a	b	c	b

(0,c)

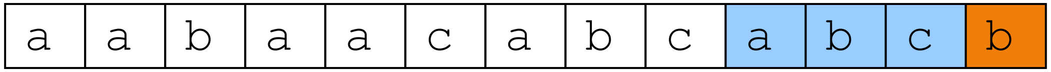
Phase	Index
a	1
ab	2
aa	3
c	4



a a b a a c a b c a b c b

(2,c)

Phase	Index
a	1
ab	**2**
aa	3
c	4
abc	5



a a b a a c a b c a b c b

(5,b)

Phase	Index
a	1
ab	2
aa	3
c	4
abc	**5**
abcd	6

LZ78 : Observation

- The dictionary is built on-the-fly during the encoding
- When the dictionary was developing, longer phase will be included
- Use of a single index to represent a longer phase provide a more compact representation, so as to provide compression

LZ78: Decoding

- The decoding of LZ78 is very similar to the encoding process
  - We build the dictionary on-the-fly
- Decoding loop
  - Decode the input pair ***(id,c)*** *by finding the phrase* ***S*** *mapped by the index* ***id*** *in the dictionary*
    - If id is 0 means a null phrase
  - Append **c** immediately after the phrase
  - Add the new phrase ***Sc*** to the dictionary

LZ78: Decoding Example

To decode the compressed message:
(0,a) (1,b) (1,a) (0,c) (2,c) (5,b)

Dict.

Input

(0,a)

a												

Phase	Index
a	1

(1,b)

a	a	b										

Phase	Index
a	**1**
ab	2

LZ78: Decoding Example

(1,a)

a	a	b	a	a								

Phase	Index
a	**1**
ab	2
aa	3

(0,c)

a	a	b	a	a	c							

Phase	Index
a	1
ab	2
aa	3
c	4

LZ78: Decoding Example

(2,c)

a	a	b	a	a	c	a	b	c				

Phase	Index
a	1
ab	**2**
aa	3
c	4
abc	5

(5,b)

a	a	b	a	a	c	a	b	c	a	b	c	b

Phase	Index
a	1
ab	2
aa	3
c	4
abc	**5**
abcd	6

LZ78 : Observation from decoding

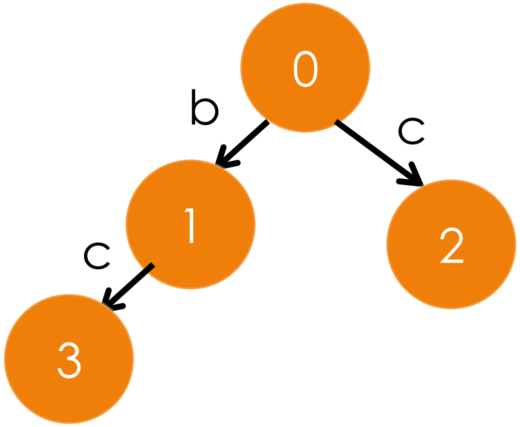
- The dictionary is built on-the-fly during the decoding
- No dictionary is sent explicitly, it is embedded in the encoded sequence
- The decoding process is relatively faster than the encoding
  - NO largest phrase matching is necessary
  - It is advantageous for many applications

LZ78: Optimization

- To improve efficiency, the dictionary can be represented in a tree with numbered nodes
  - Each new phrase to be included is formed by an existing phrase + one more character
  - Many repetition appears
- The coding will start with a single node tree, which represents empty string

LZ78: Optimization

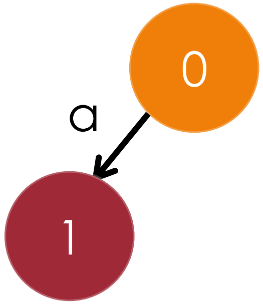
- The edge of a branch will represent a character in the phrase
- The node id represents the index
- For example, the tree on right
  - the "b" character being indexed by 1
  - the "c" character being indexed by 2
  - The "bc" phrase is having index = 3



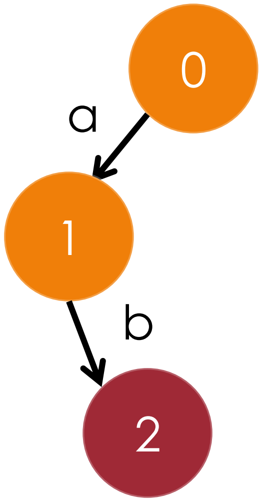
LZ78: Tree representation of dictionary

- Using the above example

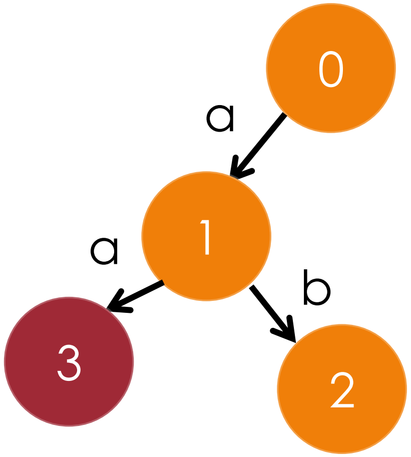
Phase	Index
a	1



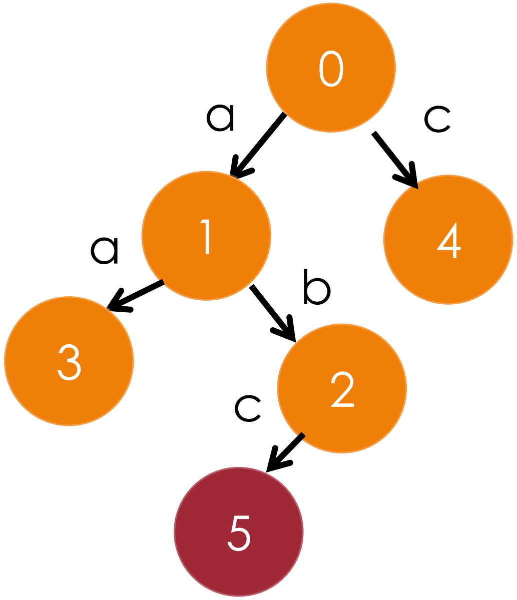
Phase	Index
a	1
ab	2



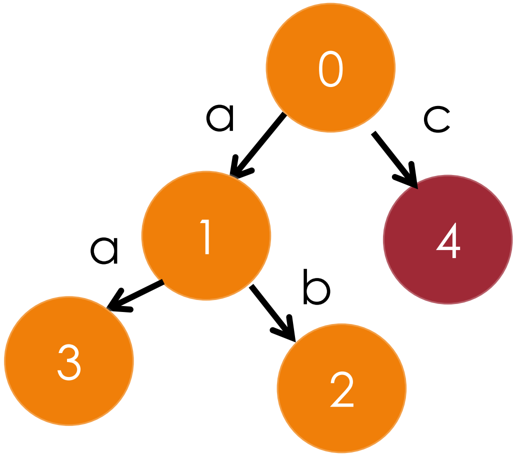
Phase	Index
a	1
ab	2
aa	**3**



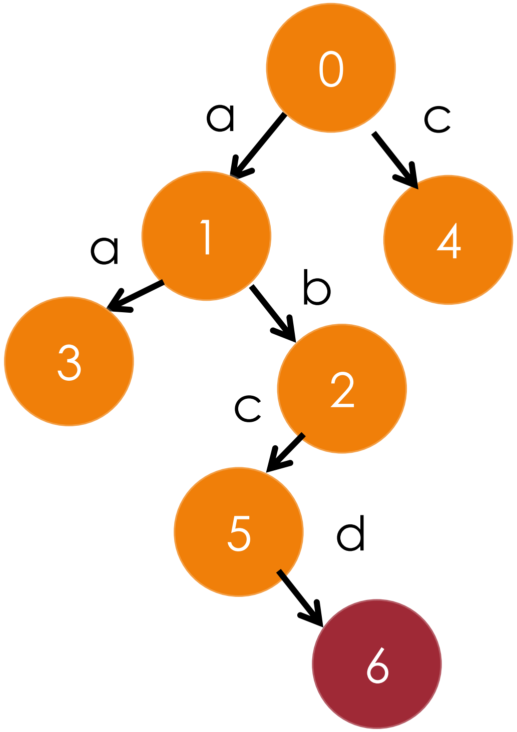
Phase	Index
a	1
ab	2
aa	3
c	4
abc	**5**



Phase	Index
a	1
ab	2
aa	3
c	**4**

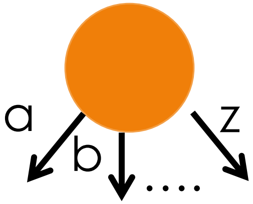


Phase	Index
a	1
ab	2
aa	3
c	4
abc	5
abcd	**6**



LZ78: Tree representation of dictionary

- The number of possible branches in the tree is equal to the number of possible character
- If we only have 26 English alphabet, the possible branches will be 26
- If we use 8-bit ASCII, the number of possible branches will be $2^8 = 256$



LZW (Lempel-Ziv-Welch)

- An improved version to LZ78
  - Used in GIF files, Adobe PDF file and Unix compress
  - Patented algorithm, although expired in 2003/2004
- Major issue in LZ78 is the inclusion of character in the output (id, **c**)
  - An overhead for phrase with only one character

LZW: Algorithm

- Can we do better to use only id in each output?

- Can we embed the character set in the dictionary at the beginning already?

LZW: Algorithm

- If we are using ASCII, the dictionary is first initialized with byte values being the first 256 entries (e.g. a = 112)

Phase	Index
\0	0
…	…
a	112
b	113
…	…
	255

LZW: Algorithm

- So, we can output the index even in the first few steps
- In each step, similar to LZ78, we add entries to dictionary, based on the seen patterns
- Every time, no matter how long is the pattern, we find a mapping to index from the dictionary

Phase	Index
\0	0
…	…
a	112
b	113
…	…
	255
aa	256

LZW: The Pseudo Code

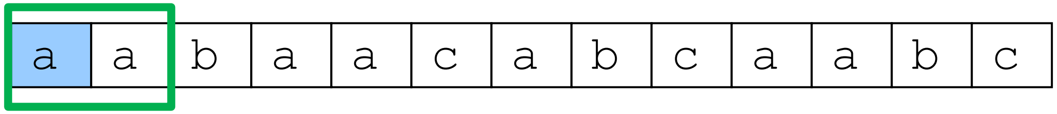
```
Initialize table with single character strings
P = first input character
WHILE not end of input stream
    C = next input character
    IF P + C is in the string table
      P = P + C
    ELSE
      output the code for P
            add P + C to the string table
       P = C
     END WHILE
output code for P
```

LZW: Encoding Example

- input sequence :aabaacabcaabc

Output

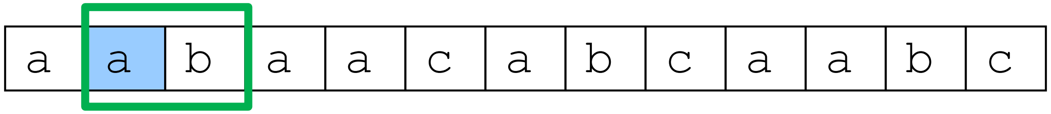
Dict.



a	a	b	a	a	c	a	b	c	a	a	b	c

112

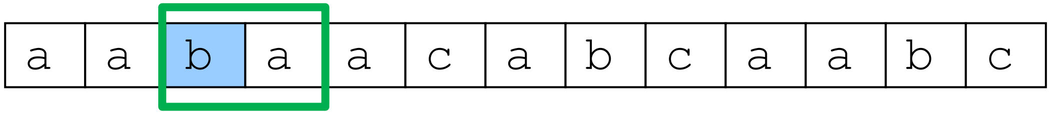
Phase	Index
aa	256



a	a	b	a	a	c	a	b	c	a	a	b	c

112

Phase	Index
aa	256
ab	257

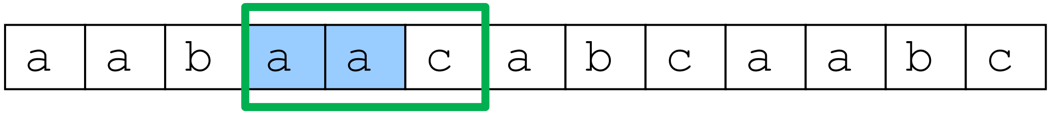


a	a	b	a	a	c	a	b	c	a	a	b	c

113

Phase	Index
aa	256
ab	257
ba	258

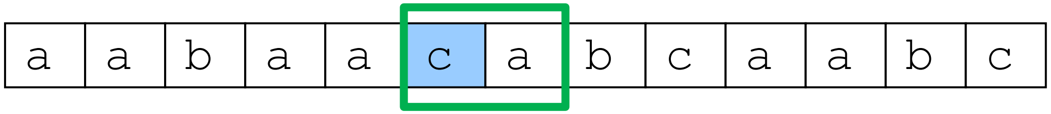
LZW: Encoding Example



a	a	b	a	a	c	a	b	c	a	a	b	c

256

Phase	Index
aa	**256**
ab	257
ba	258
aac	259

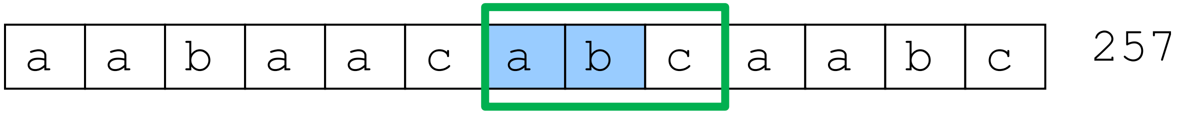


a	a	b	a	a	c	a	b	c	a	a	b	c

114

Phase	Index
aa	256
ab	257
ba	258
aac	259
ca	260

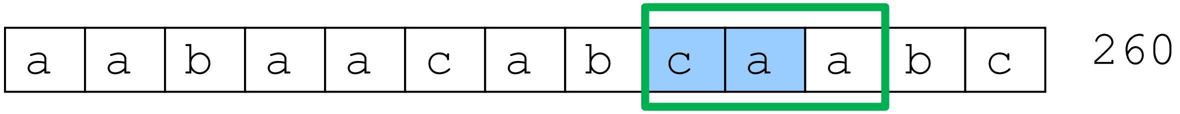
LZW: Encoding Example



a a b a a c a b c a a b c

257

Phase	Index
aa	256
ab	**257**
ba	258
aac	259
ca	260
abc	261



a a b a a c a b c a a b c

260

Phase	Index
aa	256
ab	257
ba	258
aac	259
ca	**260**
abc	261
caa	262

LZW: Encoding Example

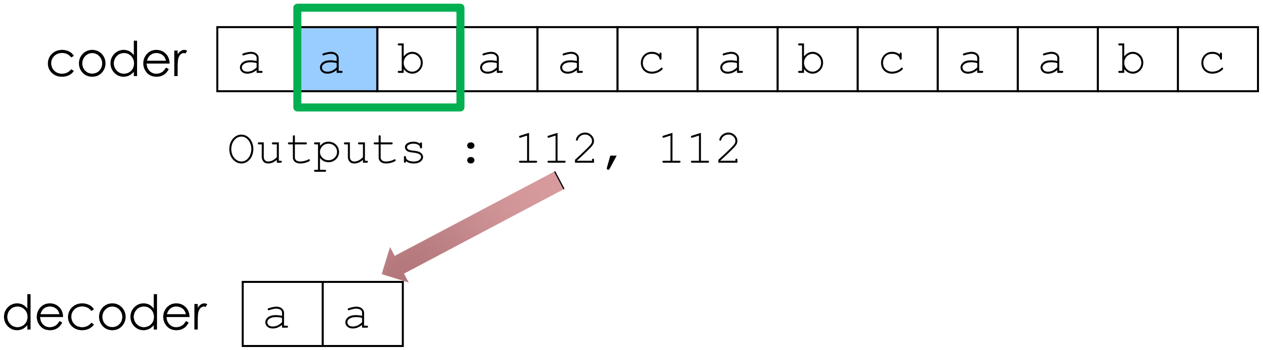
a	a	b	a	a	c	a	b	c	a	a	b	c

261

Phase	Index
aa	256
ab	257
ba	258
aac	259
ca	260
abc	**261**
caa	262

LZW: Decoding

- In the decoding, the building of dictionary will be one step behind the coder
  - since it does not know the followed character **c**
- In the above example,



coder

a	a	b	a	a	c	a	b	c	a	a	b	c

```
Outputs : 112, 112
```

decoder

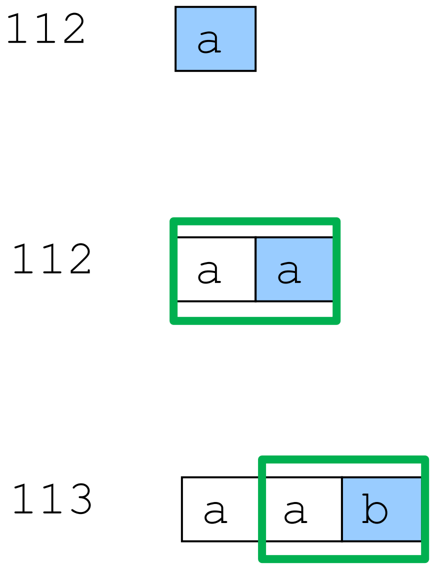
a	a

Phase	Index
aa	256
ab	257

Phase	Index
aa	256

LZW: Decoding Example

Input



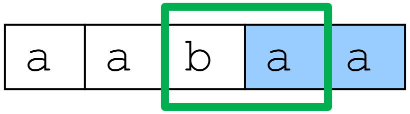
Dict

Phase	Index
aa	256

Phase	Index
aa	256
ab	257

LZW: Decoding Example

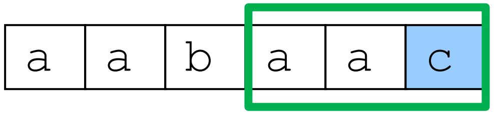
256



a a b a a

Phase	Index
aa	**256**
ab	257
ba	258

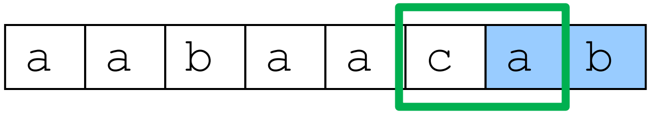
114



a a b a a c

Phase	Index
aa	256
ab	257
ba	258
aac	259

257



a a b a a c a b

Phase	Index
aa	256
ab	**257**
ba	258
aac	259
ca	260

LZW: Decoding Example

Phase	Index
aa	256
ab	257
ba	258
aac	259
ca	**260**
abc	261

260

a	a	b	a	a	c	a	b	c	a

261

a	a	b	a	a	c	a	b	c	a	a	b	c

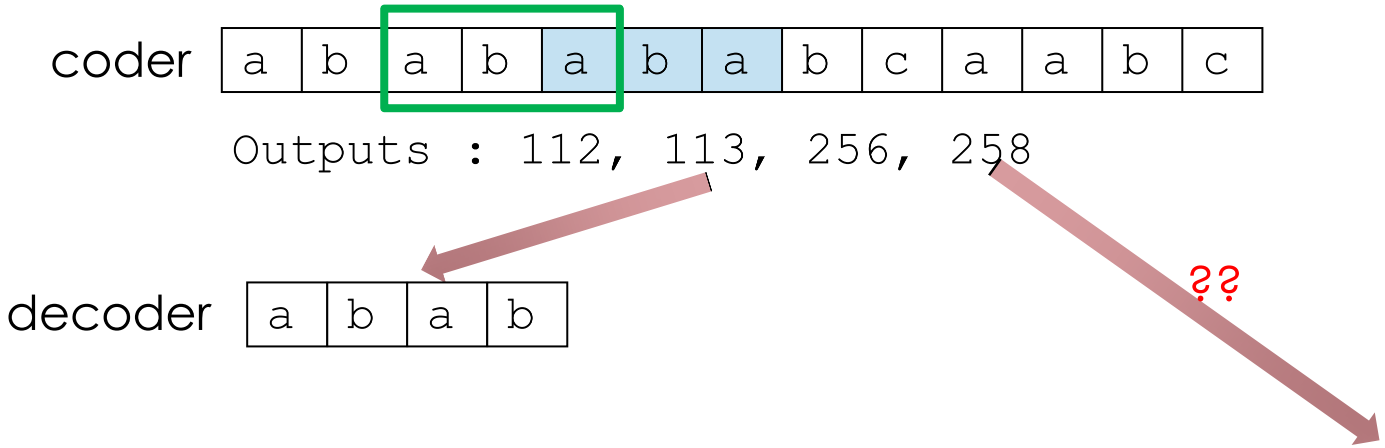
Phase	Index
aa	256
ab	257
ba	258
aac	259
ca	260
abc	**261**
caa	262

Issue in Decoding

- As mentioned, the decoder build the dictionary one entry behind the coder
- So, it is possible for the decoder receive an output code that is not in the dictionary yet !!
- This problem will only occur when for strings of the form ***SSc*** where the first character of S = **c**

Issue in Decoding

- For example, abababa....



a	b	a	b	a	b	a	b	c	a	a	b	c

```
Outputs : 112, 113, 256, 258
```

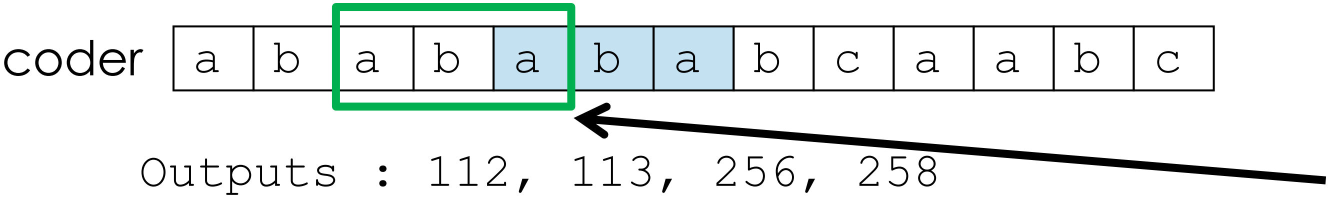
a	b	a	b

Phase	Index
ab	256
ba	257
aba	258

Phase	Index
ab	256
ba	257

Issue in Decoding

- This missing entry must have the pattern formed by [the last decoded phase + unknown char x]
  - i.e. “ab” + x
- Remember, this missing entry should immediate follow the last phase!
- So, x = first char in last phase
  - i.e. x = “a”



coder

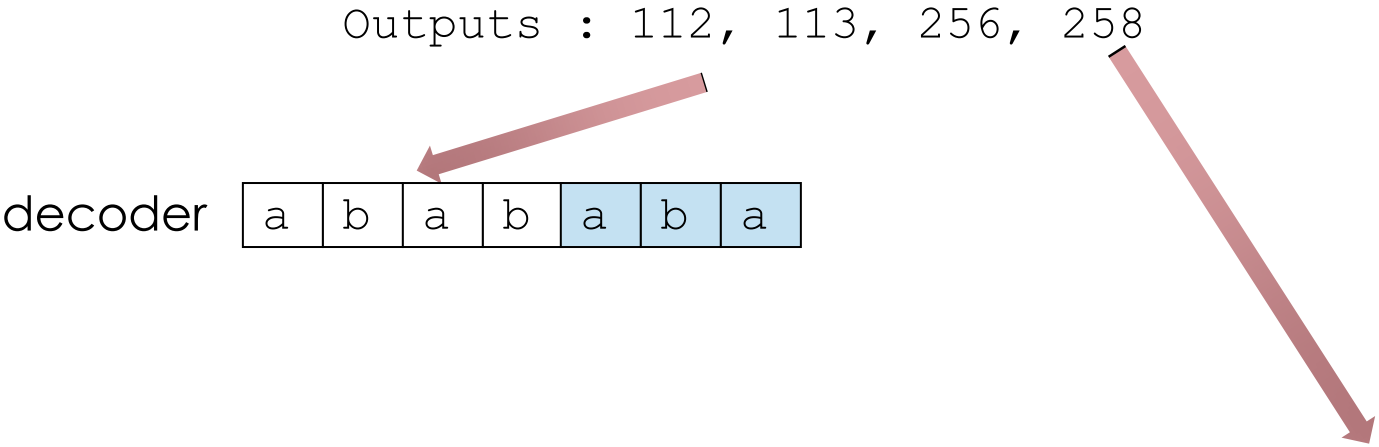
a	b	a	b	a	b	a	b	c	a	a	b	c

```
Outputs : 112, 113, 256, 258
```

Phase	Index
ab	256
ba	257
aba	258

Issue in Decoding

- So, when decoder sees an unknown codeword, it add a new entry to dictionary with:
- [the last phase + the first char in last phase]



```
Outputs : 112, 113, 256, 258
```

decoder							
	a	b	a	b	a	b	a

Phase	Index
ab	256
ba	257
aba	258

The Dictionary in LZW and LZ78

- Both algorithms can use a tree structure to encode the dictionary phases
- Accelerate the each phase's matching process to log (L)
  - where L is the average phase length

- What happens when the dictionary gets too large?

The Dictionary in LZW and LZ78

When dictionary too large:

- Throw the dictionary away when it reaches a certain size (used in GIF)
- Throw the dictionary away when it is no longer effective at compressing (used in unix `compress`)
- Throw the least-recently-used (LRU) entry away when it reaches a certain size (used in BTLZ, the British Telecom standard)

Lempel-Ziv Algorithms Summary

Both LZ77 and LZ78 and their variants keep a "dictionary" of recent strings that have been seen.

The differences are:

- How the dictionary is stored
- How it is extended
- How it is indexed
- How elements are removed

Lempel-Ziv Algorithms Summary (II)

Adapt well to changes in the file (e.g. a Tar file with many file types within it).

Initial algorithms did not use probability coding and perform very poorly in terms of compression (e.g. 4.5 bits/char for English)

More modern versions (e.g. gzip) do use probability coding as "second pass" and compress much better.

Are becoming outdated, but ideas are used in many of the newer algorithms.

Lempel-Ziv Algorithms Summary (III)

- LZW uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together, e.g., words in English text.
- The LZW encoder and decoder build up the same dictionary **dynamically** while receiving the data.
- LZW places longer and longer repeated entries into a dictionary, and then emits the code for an element, rather than the string itself, if the element has already been placed in the dictionary

LZW Algorithm

- Codes 0-255 in the code table are always assigned to represent single bytes from the input file.
- When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks.
- Compression is achieved by using codes 256 through 4095 to represent sequences of bytes.
- As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table.
- Decoding is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents.

Comparison of LZ Algorithms

	LZ77	LZ78	LZW
Length of Codeword	Variable	Variable	Fixed
Output codeword	Codeword + last character	Codeword + last character	codeword
Need to build dictionary	No (sliding window)	Yes	Yes
Dictionary Entries	Not applicable	Decoding has same entries as encoding	Encoding is one entry ahead decoding
Sending dictionary explicitly	No	No	No

Applications

- Dictionary based compression is widely used in many existing file format/ standard
- Zip file format
  - Support a range of compression method
  - DEFLATE is the default data compression algorithm
    - uses a combination of the LZ77 algorithm and Huffman coding
- Rar file format
  - Proprietary format from win.rar GmbH
  - Version 3 of RAR is based on LZSS algorithm

Applications

Image file formats include

- GIF (Graphics Interchange Format)
  - LZW
- PNG (Portable Network Graphics)
  - LZ77 (DEFLATE)
  - Huffman coding
- Both of them emerge when Web applications become popular
  - Compress to save bandwidth

DEFLATE

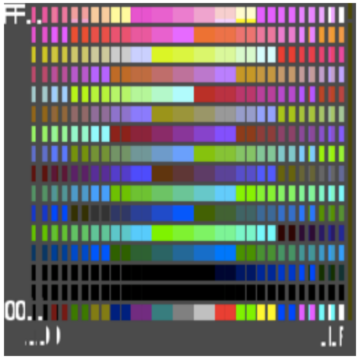
- Defined by Phil Katz of his PKZIP archiving tool
- A patent algorithm widely used practically
  - zlib, gzip
  - PNG, TIFF
- Include two major stages
  - LZ77
  - Huffman coding

DEFLATE

- First stage of compression is based on LZ77
  - An encoded match to an earlier string consists of a length (3–258 bytes) (i.e. the look-ahead buffer size), and
  - a distance (1–32,768 bytes) (i.e. the dictionary window size)
- A lengthy sequence is divided into blocks
- Second stage is applying Huffman coding
  - Reduce bits used by codewords (result from LZ77, e.g. the distance)

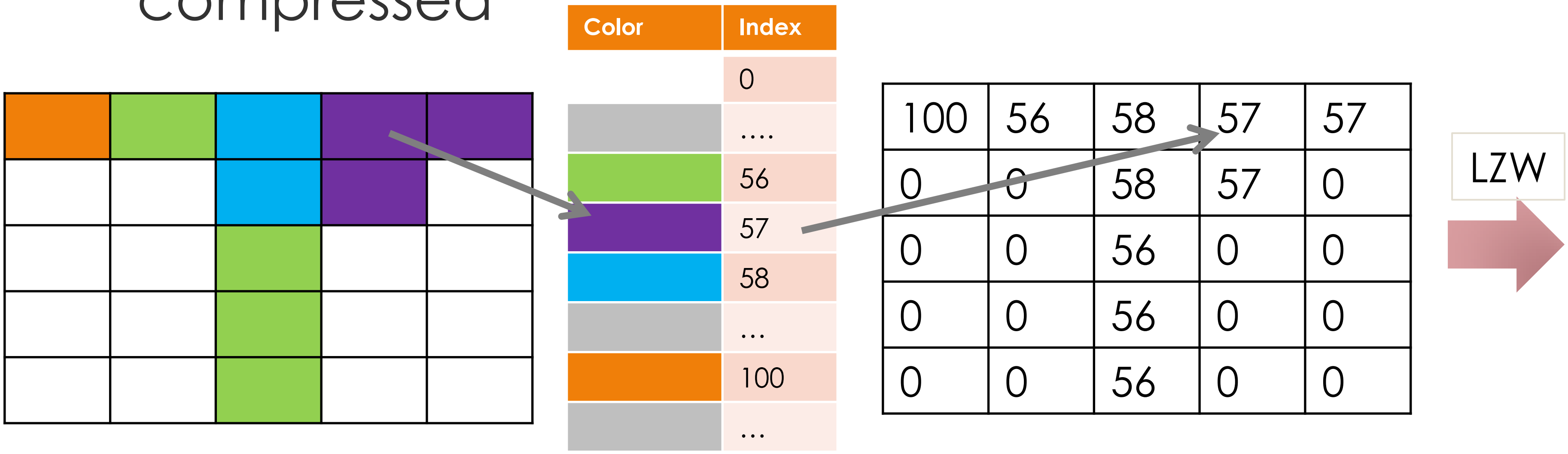
GIF (Graphics Interchange Format)

- GIF was devised by UNISYS and Compuserve.
- It supports 8bit (256) color images only
  - Supports 256 entries palette-based colors
  - Each image can have its own color table.



GIF (Graphics Interchange Format)

- Color indices are stored in raster order and LZW compressed



Color	Index
	0
▨	....
🟩	56
🟪	57
🟦	58
▨	...
🟧	100
▨	...

100	56	58	57	57
0	0	58	57	0
0	0	56	0	0
0	0	56	0	0
0	0	56	0	0

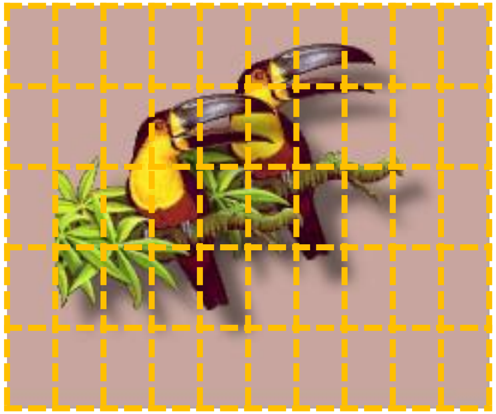
- Other features includes transparency layer, simple animation functions, interlaced coding and decoding.

PNG (Portable Network Graphics)

- PNG is a lossless image compressing method based on LZ77
  - supports three main image types: true color, grayscale and palette-based ("8-bit")
- The compression process is similar to GIF
- To allow interlaced display, the encoding order of the pixels are designed as follow.

PNG (cont)

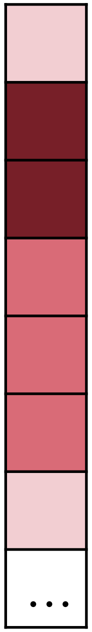
- The scanning patternis in a 8x8 block.
- The whole image is partitioned into 8x8 blocks and scanned based on the pattern in each block.



Order of scanning in block

1	6	4	6	2	6	4	6
7	7	7	7	7	7	7	7
5	6	5	6	5	6	5	6
7	7	7	7	7	7	7	7
3	6	4	6	3	6	4	6
7	7	7	7	7	7	7	7
5	6	5	6	5	6	5	6
7	7	7	7	7	7	7	7

Serialized values



Summary

- Adaptive Dictionary based Compression builds the dictionary on-the-fly with the input data
- The LZ family is one of the most popular dictionary based compression methods
  - LZ77 (Sliding Window)
  - LZ78
  - LZW