Lempel-Ziv Data Compression Algorithms

Guy Blelloch, CMU

Goal: compression of strings (whereas Huffman Coding just compresses character codes).

- Keep a “dictionary” of recent strings that have been seen.
- Gives much better compression than Huffman Coding
    - Adapts well to changes in the file.

Lempel-Ziv Algorithms

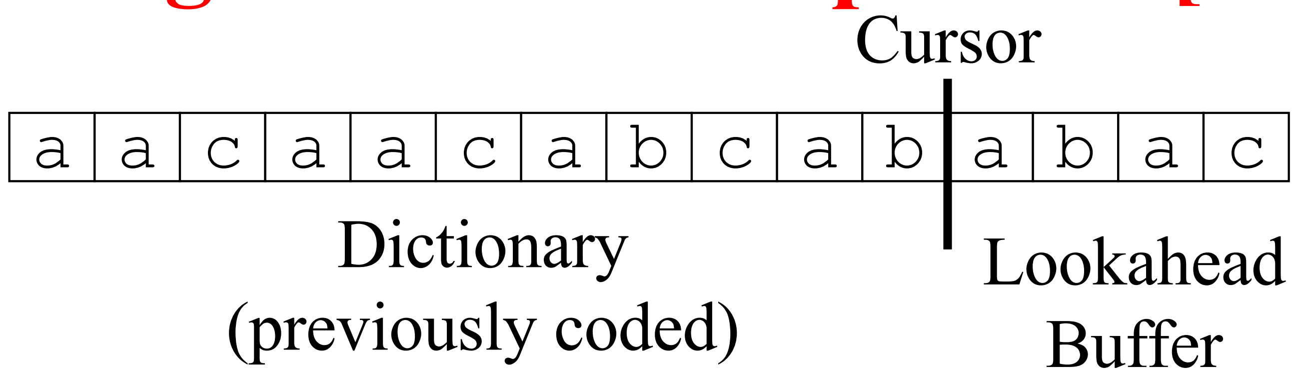
Sliding Window Lempel-Ziv [LZ77]

- Gzip is a fast implementation

Dictionary Lempel-Ziv [LZ78]

- Not as compact compression, but faster

Sliding Window Lempel-Ziv [LZ77]

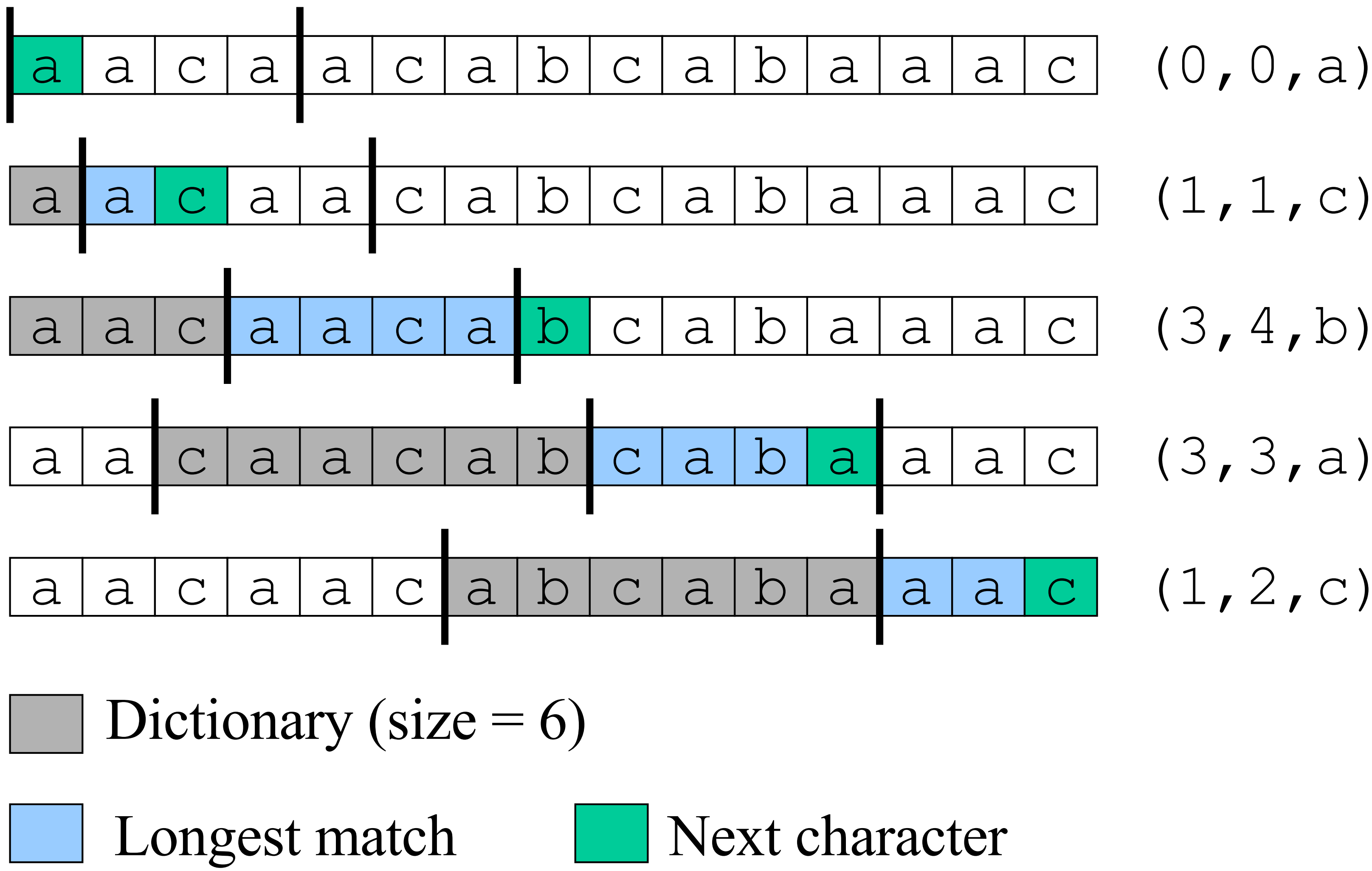


Dictionary and buffer "windows" are fixed length and slide with the cursor

On each step:

- Output (P,L,C) to encode the next substring compressed.
  P = relative position (going backwards) 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

Sliding Window Lempel-Ziv [LZ77] Example



Sliding Window Lempel-Ziv [LZ77] Decoding

Decoder keeps same dictionary window as encoder.

- For each message it looks it up in the dictionary and inserts a copy

What if L > P? (only part of the message is in the dictionary.)

- E.g. dict = `abcd`, codeword = `(2,9,e)`
- Simply copy starting at the cursor

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

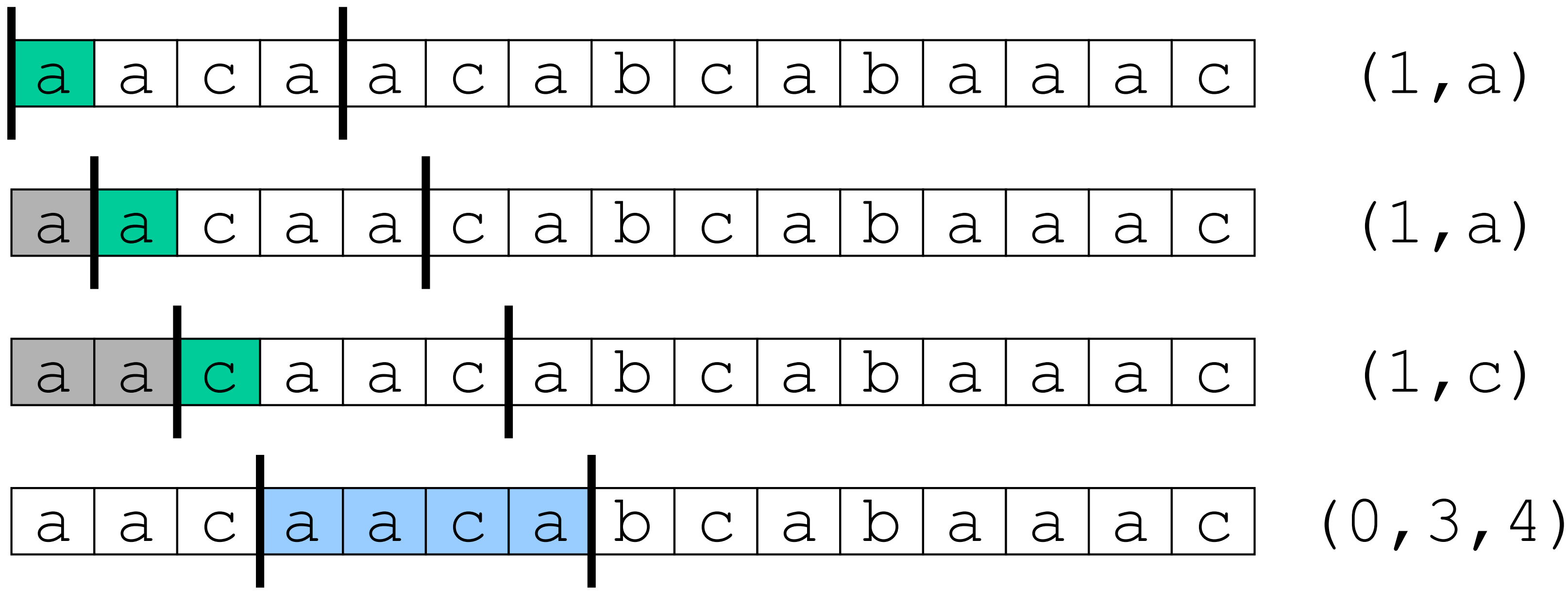
- Out = `abcdcdcdcdcdc`

Optimizations used by **`gzip`**

Output one of the following formats

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

Typically use the second format if length < 3.



Optimizations used by `gzip` (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

Sliding Window Lempel-Ziv [LZ77] In Theory is Asymptoticly Optimal

Will compress long enough strings to the source entropy as the window size goes to infinity.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Uses logarithmic code for position.

Problem: "long enough" is really **really** long.

Dictionary Lempel-Ziv [LZ78]

Basic algorithm:

- Keep dictionary of words with integer *id* for each entry (e.g. keep it as a trie).
- Coding loop
  - find the longest match *S* in the dictionary
  - Output the entry *id* of the match and the next character C past the match from the input *(id,C)*
  - Add the string *Sc* to the dictionary
- Decoding keeps same dictionary and looks up *id*s

Dictionary Lempel-Ziv [LZ78] Coding Example

													Output	Dict.
a	a	b	a	a	c	a	b	c	a	b	c	b	(0,a)	1 = a
a	a	b	a	a	c	a	b	c	a	b	c	b	(1,b)	2 = ab
a	a	b	a	a	c	a	b	c	a	b	c	b	(1,a)	3 = aa
a	a	b	a	a	c	a	b	c	a	b	c	b	(0,c)	4 = c
a	a	b	a	a	c	a	b	c	a	b	c	b	(2,c)	5 = abc
a	a	b	a	a	c	a	b	c	a	b	c	b	(5,b)	6 = abcb

Dictionary Lempel-Ziv [LZ78] Decoding Example

Input														Dict.
(0,a)	a													1 = a
(1,b)	a	a	b											2 = ab
(1,a)	a	a	b	a	a									3 = aa
(0,c)	a	a	b	a	a	c								4 = c
(2,c)	a	a	b	a	a	c	a	b	c					5 = abc
(5,b)	a	a	b	a	a	c	a	b	c	a	b	c	b	6 = abcb

Dictionary Lempel-Ziv [LZ78]
Practical Issues

What happens when the dictionary gets 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 Sliding-Window and Dictionary Lempel-Ziv and their variants:

- Keep a "dictionary" of recent strings that have been seen.
- Give much better compression than Huffman Coding
- Adapt well to changes in the file.

The differences between Sliding-Window and Dictionary Lempel-Ziv are:

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

Lempel-Ziv Algorithms Summary

The original published Lempel-Ziv 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.