

## 15-853: Algorithms in the Real World

### Data Compression III

15-853

Page 1

## Compression Outline

**Introduction:** Lossy vs. Lossless, Benchmarks, ...

**Information Theory:** Entropy, etc.

**Probability Coding:** Huffman + Arithmetic Coding

**Applications of Probability Coding:** PPM + others

➔ **Lempel-Ziv Algorithms:**

- LZ77, gzip,

- LZ78, compress (Not covered in class)

**Other Lossless Algorithms:** Burrows-Wheeler

**Lossy algorithms for images:** JPEG, MPEG, ...

**Compressing graphs and meshes:** BBK

15-853

Page 2

## 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

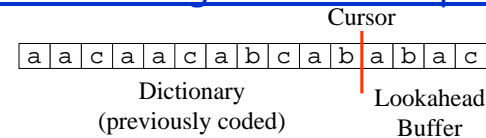
**Applications:** compress, GIF, CCITT (modems),  
ARC, PAK

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

15-853

Page 3

## LZ77: Sliding Window Lempel-Ziv



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

**Repeat:**

Output (**p**, **l**, **c**) where

**p** = position of the longest match that starts in  
the dictionary (relative to the cursor)

**l** = length of longest match

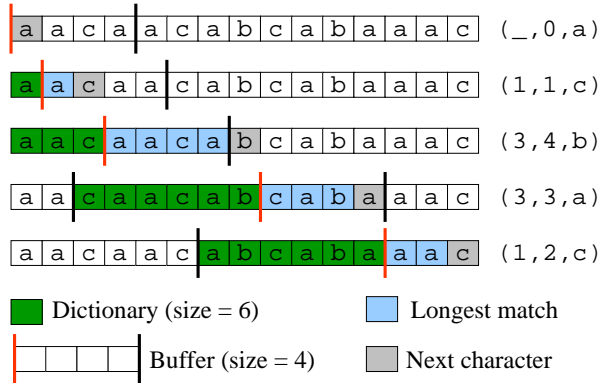
**c** = next char in buffer beyond longest match

Advance window by **l + 1**

15-853

Page 4

## LZ77: Example



## LZ77 Decoding

Decoder keeps same dictionary window as encoder. For each message it looks it up in the dictionary and inserts a copy at the end of the string

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

E.g. dict = abcd, codeword = (2, 9, e)

- Simply copy from left to right  

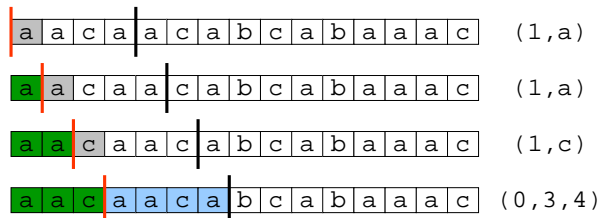
```
for (i = 0; i < length; i++)
    out[cursor+i] = out[cursor-offset+i]
```
- Out = abcdcdcdcdcdce

## LZ77 Optimizations used by gzip

LZSS: Output one of the following two formats

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

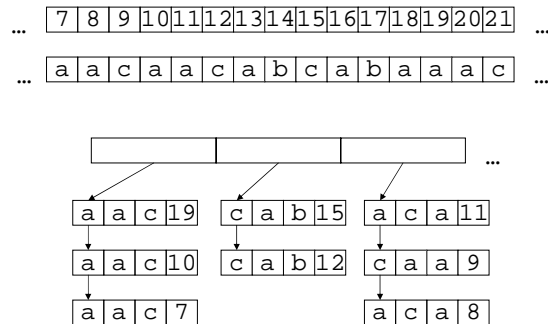
Uses the second format if length < 3.



## Optimizations used by gzip (cont.)

1. Huffman code the positions, lengths and chars
2. Non greedy: possibly use shorter match so that next match is better
3. Use a hash table to store the dictionary.
  - Hash keys are all strings of length 3 in the dictionary window.
  - Find the longest match within the correct hash bucket.
  - Puts a limit on the length of the search within a bucket.
  - Within each bucket store in order of position

## The Hash Table



15-853

Page 9

## Theory behind LZ77

Sliding Window LZ is Asymptotically Optimal  
[Wyner-Ziv,94]

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

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

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

Uses logarithmic code (e.g. gamma) for the position.  
Problem: "long enough" is really **really** long.

15-853

Page 10

## Comparison to Lempel-Ziv 78

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 (LZ78 is a trie)
- How it is extended (LZ78 only extends an existing entry by one character)
- How it is indexed (LZ78 indexes the nodes of the trie)
- How elements are removed

15-853

Page 11

## Lempel-Ziv Algorithms Summary

Adapts 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 performed poorly in terms of compression. More modern versions (e.g. gzip) do use probability coding as "second pass" and compress much better.

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

15-853

Page 12

## Compression Outline

**Introduction:** Lossy vs. Lossless, Benchmarks, ...

**Information Theory:** Entropy, etc.

**Probability Coding:** Huffman + Arithmetic Coding

**Applications of Probability Coding:** PPM + others

**Lempel-Ziv Algorithms:** LZ77, gzip, compress, ...

➔ **Other Lossless Algorithms:**

- Burrows-Wheeler
- ACB

**Lossy algorithms for images:** JPEG, MPEG, ...

**Compressing graphs and meshes:** BBK

15-853

Page 13

## Burrows -Wheeler

Currently near best "balanced" algorithm for text  
Breaks file into fixed-size blocks and encodes each block separately.

**For each block:**

- Sort each character by its full context.  
This is called the **block sorting transform**.
- Use **move-to-front transform** to encode the sorted characters.

The ingenious observation is that the decoder only needs the sorted characters and a pointer to the first character of the original sequence.

15-853

Page 14

## Burrows Wheeler: Example

Let's encode:  $d_1e_2c_3o_4d_5e_6$

We've numbered the characters to distinguish them.

Context "wraps" around. Last char is most significant.

<u>Context</u>	<u>Char</u>		<u>Context</u>	<u>Output</u>
ecode <sub>6</sub>	d <sub>1</sub>	<b>Sort Context</b> ➔	dedec <sub>3</sub>	o <sub>4</sub>
coded <sub>1</sub>	e <sub>2</sub>		coded <sub>1</sub>	e <sub>2</sub>
odede <sub>2</sub>	c <sub>3</sub>		decod <sub>5</sub>	e <sub>6</sub>
dedec <sub>3</sub>	o <sub>4</sub>		odede <sub>2</sub>	c <sub>3</sub>
edeco <sub>4</sub>	d <sub>5</sub>		ecode <sub>6</sub>	d <sub>1</sub> ←
decod <sub>5</sub>	e <sub>6</sub>		edeco <sub>4</sub>	d <sub>5</sub>

15-853

Page 15

## Burrows-Wheeler (Continued)

**Theorem:** After sorting, equal valued characters appear in the same order in the output as in the most significant position of the context.

**Proof sketch:** Since the chars have equal value in the most-significant-position of the context, they will be ordered by the rest of the context, *i.e.* the previous chars. This is also the order of the output since it is sorted by the previous characters.

<u>Context</u>	<u>Output</u>
c <sub>3</sub>	o <sub>4</sub>
d <sub>1</sub>	e <sub>2</sub>
d <sub>5</sub>	e <sub>6</sub>
e <sub>2</sub>	c <sub>3</sub>
e <sub>6</sub>	d <sub>1</sub>
o <sub>4</sub>	d <sub>5</sub>

15-853

Page 16

## Burrows-Wheeler: Decoding

Consider dropping all but the last character of the context.

	<u>Context</u>	<u>Output</u>
	a c	
- What follows the underlined <u>a</u> ?	a <u>b</u>	
- What follows the underlined <u>b</u> ?	a b	
	b a	
- What is the whole string?	b <u>a</u> ←	
<b>Answer:</b> b, a, abacab	c a	

15-853

Page 17

## Burrows-Wheeler: Decoding

What about now?

**Answer:** cabbaa

Can also use the "rank".  
The "rank" is the position of a character if it were sorted using a stable sort.

	<u>Context</u>	<u>Output</u>	<u>Rank</u>
	a c	←	6
	a a		1
	a b		4
	b b		5
	b a		2
	c a		3

15-853

Page 18

## Burrows-Wheeler Decode

```

Function BW_Decode(In, Start, n)
  S = MoveToFrontDecode(In, n)
  R = Rank(S)
  j = Start
  for i=1 to n do
    Out[i] = S[j]
    j = R[j]
  
```

Rank gives position of each char in sorted order.

15-853

Page 19

## Decode Example

<u>S</u>	<u>Rank(S)</u>	<u>Out</u>
o <sub>4</sub>	6	e <sub>6</sub> d <sub>1</sub> ←
e <sub>2</sub>	4	d <sub>1</sub> e <sub>2</sub>
e <sub>6</sub>	5	e <sub>2</sub> c <sub>3</sub>
c <sub>3</sub>	1	c <sub>3</sub> o <sub>4</sub>
d <sub>1</sub>	2	o <sub>4</sub> d <sub>5</sub>
d <sub>5</sub>	3	d <sub>5</sub> e <sub>6</sub>

15-853

Page 20

## Overview of Text Compression

**PPM** and **Burrows-Wheeler** both encode a single character based on the immediately preceding context.

**LZ77** and **LZ78** encode multiple characters based on matches found in a block of preceding text

Can you **mix these ideas**, *i.e.*, code multiple characters based on immediately preceding context?

- **BZ** does this, but they don't give details on how it works - current best compressor
- **ACB** also does this - close to best

15-853

Page 21

## ACB (Associate Coder of Buyanovsky)

Keep dictionary sorted by context (the last character is the most significant)

- Find longest match for context
- Find longest match for contents
- Code
  - Distance between matches in the sorted order
  - Length of contents match

Has aspects of Burrows-Wheeler, and LZ77

<u>Context</u>	<u>Contents</u>
	decode
	dec ode
	d ecode
	decod e
	de code
	deco de

15-853

Page 22