Outline

PA1 Timing Analysis Report Due Now

Last time:

- Trie
- RLE and Huffman encoding

Today:

- Dictionary-based compression
- LZ77 and LZSS
- LZ78 and LZW
Dictionary-based Compression

Huffman encoding represents frequently seen symbols with small codes.

Often sequences of symbols are frequently seen.

Example: in English text, “the”, “and”, “ion”, “ing”, “of the” (and “like”, “so”, “right”) are seen more frequently than other sequences.

Idea: assign codes to entire sequences.

Advantages over Huffman:

- better compression
- universal code: encode a piece of text “on-the-fly,” in one-pass, “streaming” mode
- no need to store code table (dictionary) in compressed file

Sugih Jamin (jamin@eecs.umich.edu)
Dictionary-based Taxonomy

All of dictionary-based methods can be divided into two:

- if a sequence has been previously encountered in the input data, output a pointer to the earlier occurrence; the dictionary is implicitly represented by previously processed and output data, example: LZ77

```
Input Stream: [sequence]
Output Stream: [sequence]
```

- enter *phrases* into the dictionary; when the same phrase is encountered, output its index into the dictionary, example: LZW

```
Input Stream: [sequence]
Output Stream: [sequence]
Dictionary: [dictionary entries]
```
Lempel-Ziv 77 Compression

LZ77:

- find duplicated strings in input data
- replace latter occurrence of a string by a pointer to the previous occurrence

Pointer is encoded as \((P, L)\), where

- \(P\) is the distance between the two occurrences of the string
- \(L\) is the length of the match

**Example:** Blah blah blah blah blah blah!  
**Output:** Blah b(5,18)!
LZ77

Algorithm:

- uses two variables: prefix, and lookahead buffer of same fixed-size
- prefix initially empty, buffer contains input block
- while (buffer not empty) {
  find longest match in prefix+buffer of buffer
  if (match) {
    output (P, L)
    move prefix and buffer forward L chars
  } else {
    output (0, 0)
  }
} else {
  output the char at head of buffer
  move prefix and buffer forward 1 char
}
LZ77 Example

String: aaabbcab

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>–</td>
<td>aaabbcab</td>
<td>–</td>
<td>(0,0)a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>a</td>
<td>aaabbcab</td>
<td>aa</td>
<td>(1,2)b</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>aaab</td>
<td>bcab</td>
<td>b</td>
<td>(1,1)c</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>aaabbc</td>
<td>ab</td>
<td>ab</td>
<td>(4,2)\0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>aaabbcab</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Output: 00a12b11c42 (56 bits)

(P, L) is represented with 6 bits, so each (P, L) character is 14 bits
LZSS

LZ-Storer-Szymanski observations:

- null pointer redundant
- actually, if $L < 2$, use of pointer redundant
- the mismatch after a pointer could be a match

LZSS algorithm:

while (buffer not empty) {
    find longest match in prefix+buffer of buffer
    if (match and $L \geq \text{MINLENGTH}(2))$
    {
        output ($P, L$
        move prefix and buffer forward $L$ chars
    }
    else {
        output the char at head of buffer
        move prefix and buffer forward 1 char
    }
}
### LZSS Mode Example

**String:** `aaabbcab`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>–</td>
<td><code>aaabbcab</code></td>
<td>–</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td><code>a</code></td>
<td><code>aaabbcab</code></td>
<td>aa</td>
<td><code>(1,2)</code></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td><code>aaa</code></td>
<td><code>bbcab</code></td>
<td>–</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td><code>aaab</code></td>
<td><code>bcbab</code></td>
<td>–</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td><code>aaabb</code></td>
<td><code>cab</code></td>
<td>–</td>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td><code>aaabbc</code></td>
<td><code>ab</code></td>
<td>ab</td>
<td><code>(4,2)</code></td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td><code>aaabbcab</code></td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

**Output:** `a12bbc42` (54 bits)

Assume each char is represented as 9 bits:
1 bit to differentiate between char and pointer

Sugih Jamin (jamin@eecs.umich.edu)
LZSS Example

String: Blah_blah_blah_blah_blah_blah!

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>Blah_bla</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>2</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>3</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>4</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>5</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>6</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>(5, 1)</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>19</td>
<td>24</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>(5, 18)</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
<tr>
<td>21</td>
<td>26</td>
<td>Blah_bla</td>
<td>Blah_bla</td>
<td>!</td>
</tr>
</tbody>
</table>

Output: Blah b518! (requires 5 bits for L)
LZ77/SS Wrap Up

Implementation issue: how big should the prefix+buffer window be? Implications?

LZSS is what people usually mean when they say LZ77

gzip: LZ77/SS output is further compressed using Huffman

Also used in PKZip, ARJ, LHarc, Zoo, etc.
Dictionary-based Taxonomy

All of dictionary-based methods can be divided into two:

- if a sequence has been previously encountered in the input data, output a pointer to the earlier occurrence; the dictionary is implicitly represented by previously processed and output data, example: LZ77

  ![Diagram of LZ77](image)

- enter *phrases* into the dictionary; when the same phrase is encountered, output its index into the dictionary, example: LZW

  ![Diagram of LZW](image)
LZ78

Idea: enter sequences of symbols into a dictionary; when a sequence is seen again, output its index in the dictionary

Algorithm:
1. define a phrase recursively as a previously seen phrase (prefix) plus an additional character (such that the concatenation has not been seen before)
2. enter each new phrase into a dictionary maintained as a trie
3. reset prefix after entering a new phrase
4. nodes on the trie are labeled in order of addition (phrase index), root has index 0
5. trie ordered by index
6. when a phrase is repeated, output the phrase’s index instead of the phrase, otherwise output 0
7. output the last (mismatching) character from Step 1

Sugih Jamin (jamin@eecs.umich.edu)
LZ78 Encoding Example

String: how now brown cow in town.

Encoding: 0h0o0w0_0n2w4b0r6n4c6_0i5_0t9.

Time complexity ($n$ string length, $|\Sigma|$ alphabet size): $O(n|\Sigma|)$

Sugih Jamin (jamin@eecs.umich.edu)
LZ78 Decoding

String: how now brown cow in town.
Encoding: 0h0o0w0_0n2w4b0r6n4c6_0i5_0t9.

How to decode the encoded string in $O(n)$, $n$ the size of the encoded string?
LZW Compression

LZ78 encoding is $O(n|\Sigma|)$, want an $O(n)$ algorithm, $n$: input size

Welch’s adaptation of LZ78:

- encoding and decoding use an array (dictionary)
  (similar to LZ78 decoding, each trie node encoded as an array index)
- all 8-bit chars assumed already in array, at indices 0-255
- each new phrase is assigned the next index
- phrase matching doesn’t start at root node, instead use last character
  in previous phrase as start of new phrase
- finding a phrase in the dictionary is like finding a path in the LZ78 trie,
  reading in each new char corresponds to going down one level into
  the trie, hence no code output until mismatch
- upon mismatch, output the index of the matched prefix, and enter the
  new phrase into the dictionary
- use hashing to match phrases in $O(1)$ time $\Rightarrow$ total time $O(n)$
LZW Compression Example

String: /WED/WE/WEE/WEB

LZW compression algorithm:

phrase = read nextchar
while input not empty {
    c = read nextchar
    if phrase+c is in dictionary {
        phrase += c
    } else {
        output code for phrase
        add phrase+c to dictionary
        phrase = c
    }
}
output code for phrase

<table>
<thead>
<tr>
<th>phrase</th>
<th>c</th>
<th>output</th>
<th>add to dict</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>W</td>
<td>/</td>
<td>256 = /W</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>W</td>
<td>257 = WE</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>E</td>
<td>258 = ED</td>
</tr>
<tr>
<td>D</td>
<td>/</td>
<td>D</td>
<td>259 = D/</td>
</tr>
<tr>
<td>/</td>
<td>W</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>/W</td>
<td>E</td>
<td>256</td>
<td>260 = /WE</td>
</tr>
<tr>
<td>E</td>
<td>/</td>
<td>E</td>
<td>261 = E/</td>
</tr>
<tr>
<td>/</td>
<td>W</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>/W</td>
<td>E</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>/WE</td>
<td>E</td>
<td>260</td>
<td>262 = /WEE</td>
</tr>
<tr>
<td>E</td>
<td>/</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>E/</td>
<td>W</td>
<td>261</td>
<td>263 = E/W</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>WE</td>
<td>B</td>
<td>257</td>
<td>264 = WEB</td>
</tr>
<tr>
<td>B</td>
<td>–</td>
<td>B</td>
<td>–</td>
</tr>
</tbody>
</table>

Output: / W E D 256 E 260 261 257 B
LZW Decompression

Similar idea to encoding: build the dictionary while decoding

Input string: / W E D 256 E 260 261 257 B

c = phrase = read nextcode
output lookup(phrase)
while (read nextcode != eof) {
  if (nextcode in dict) {
    string = lookup(nextcode)
  } else {
    string = lookup(phrase)
    string += c
  }
output string
c = first char in string
add lookup(phrase)+c to dict
phrase = nextcode
}

<table>
<thead>
<tr>
<th>phrase</th>
<th>nextcode</th>
<th>string/output</th>
<th>c</th>
<th>add to dict</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>/</td>
<td>W</td>
<td>W E</td>
<td>W</td>
<td>256 = /W</td>
</tr>
<tr>
<td>W</td>
<td>E</td>
<td>E D</td>
<td>E</td>
<td>257 = WE</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>D /W</td>
<td>D</td>
<td>258 = ED</td>
</tr>
<tr>
<td>D</td>
<td>256</td>
<td>/W E</td>
<td>/</td>
<td>259 = D/</td>
</tr>
<tr>
<td>256</td>
<td>E</td>
<td>E /WE</td>
<td>E</td>
<td>260 = /WE</td>
</tr>
<tr>
<td>E</td>
<td>260</td>
<td>/WE E</td>
<td>/</td>
<td>261 = E/</td>
</tr>
<tr>
<td>260</td>
<td>261</td>
<td>E/ E W</td>
<td>E</td>
<td>262 = /WEE</td>
</tr>
<tr>
<td>261</td>
<td>257</td>
<td>WE W</td>
<td>W</td>
<td>263 = E/W</td>
</tr>
<tr>
<td>257</td>
<td>B</td>
<td>B B</td>
<td>B</td>
<td>264 = WEB</td>
</tr>
</tbody>
</table>

Output: /WED/WE/WEE/WEB
LZW Implementation Issues

During encoding, string not stored in index order, e.g., hash string to dictionary, store index value with string

Search for longest match!
String $abccab$ has prefixes that match all of $ab$, $abc$, $abcc$, pick the longest one

For string $abccab$ and dictionary entries $abc$ and $bccab$, which entry to match?

Dictionary size limited to codeword/index size;
for 12-bit code word, dictionary can hold 4096 entries, including the 256 8-bit chars

What to do when dictionary is full?

Sugih Jamin (jamin@eecs.umich.edu)
Dealing with Full Dictionary

Options:

- stop entering new phrases, use existing ones only (but watch compression rate and switch to a different method if compression rate drops significantly)
- flush the dictionary (works well if different parts of input have different patterns)
- remove least frequently used old strings from dictionary (must keep access statistics, and make sure you don’t leave dangling prefix pointers!)
- switch to a larger codeword (must mark transition in output file)
LZ78/W Wrap Up

Faster than LZ77/SS due to less string comparisons

LZW is what people usually mean when they say LZ

Used in `compress` and also in the `gif` file format

Patented by Unisys Corp., but allowed for free non-commercial use