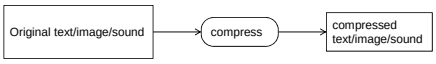
Compression and Huffman Coding

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Compression

- Reducing the memory required to store some information.



- Lossless compression vs lossy compression
- Lossless compression only possible if there is redundancy in the original
  - eg, repeated patterns
  - compression identifies and removes some of the redundant elements.

Lempel-Ziv

- Lossless compression.
- LZ77 = simple compression, using repeated patterns
  - basis for many later, more sophisticated compression schemes.
- Key idea:
  - If you find a repeated pattern, replace the later occurrences by a link to the first:

```
a contrived text containing riveting contrasting

a contrived text
```

Lempel-Ziv

```
a contrived text containing riveting contrasting t
```
➔
[0,0,a][0,0,][0,0,c][0,0,o][0,0,n][0,0,t][0,0,r][0,0,i][0,0,v][0,0,e][0,0,d][10,1,t]
[4,1,x][3,1,][15,4,a][15,1,n][2,2,g][11,1,r][22,3,t][9,4,c][35,4,a][0,0,s][12,5,t]

```
a contrived text containing riveting contrasting
```

```
a contrived text
```

Lempel-Ziv 77

- Outputs a string of tuples
  - tuple = [offset, length, nextCharacter] or [0,0,character]
- Moves a cursor through the text one character at a time
  - cursor points at the next character to be encoded.
- Drags a "sliding window" behind the cursor.
  - searches for matches only in this sliding window
- Expands a lookahead buffer from the cursor
  - this is the string it wants to match in the sliding window.
- Searches for a match for the longest possible lookahead
  - stops expanding when there isn't a match
- Insert tuple of match point, length, and next character

Lempel-Ziv 77

Algorithm

```
cursor ← 0
windowSize ← 100 // some suitable size
while cursor < text.size
    lookahead ← 0
    prevMatch ← 0
    loop
        match ← stringMatch(  text[cursor.. cursor+lookahead],
                text[(cursor<windowSize)?0:cursor-windowSize .. cursor-1] )
        if match succeeded then
            prevMatch = match
            lookahead++
        else
            output( [suitable value for prevMatch, lookahead - 1, text[cursor+lookahead - 1])
            cursor ← cursor + lookahead
            break
```

Cursor – WindowSize should never point before 0,

cursor+lookahead mustn't go past end of text

Decompression

```
a contrived text containing riveting contrasting t
```

➔

[0,0,a][0,0,][0,0,c][0,0,o][0,0,n][0,0,t][0,0,r][0,0,i][0,0,v][0,0,e][0,0,d][10,1,t]
[4,1,x][3,1,][15,4,a][15,1,n][2,2,g][11,1,r][22,3,t][9,4,c][35,4,a][0,0,s][12,5,t]

- Decompression: Decode each tuple in turn:

  cursor ← 0

  for each tuple

    [0, 0, *ch*] → output[cursor++] ← *ch*

    [*offset*, *length*,*ch*] →

      for j = 0 to length-1

        output [cursor++] ← output[cursor-*offset*]

      output[cursor++] ← *ch*

Huffman Encoding

Problem:

- Given a set of symbols/messages
- encode them as bit strings
- minimising the total number of bits.

- Messages:
  - characters
  - numbers.

Equal Length Codes

Equal length codes:

msg:	0	1	2	3	4	5	6	7	8	9	10
code:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010

msg:	a	b	c	d	e	f	g	…	z	_
	00001	00010	00011	00100	00101	00110	00111	…	11010	11011

N different messages, $\log_2 N$ bits per message

10 numbers, message length = 4

26 letters, message length = 5

If there are many repeated messages, can we do better?

Frequency based encoding

msg:	0	1	2	3	4	5	6	7	8	9	10
code:	1010	0	1	10	11	100	101	110	111	1000	1001

Suppose

0 occurs 50% of the time,
1 occurs 20% of time,
2-5 5% each,
6-10 2%

encode with variable length code:

0	by	'0'	most common
1	by	'1'	msgs have
2	by	'10'	shorter codes
3	by	'11'	
4	by	'100'	
5	by	'101'	
10	by	'1010'	

Variable length encoding

- More efficient to have variable length codes
- Problem: where are the boundaries?

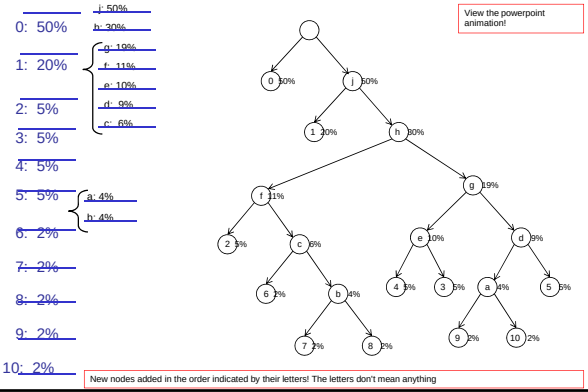
00001001010100110100110000111000111100001

- Need codes that tell you where the end is:
- msg: 0 1 2 3 4 5 6 7 8 9
- code: 0 10 1100 11101 11100 11111 11010 110110 ....

"Prefix" coding scheme
no code is the prefix of another code.

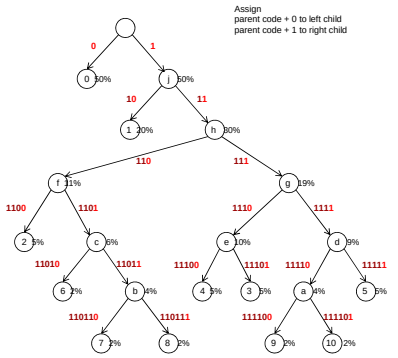
Example: Building the tree

0: 50%
1: 20%
2: 5%
3: 5%
4: 5%
5: 5%
6: 2%
7: 2%
8: 2%
9: 2%
10: 2%



Example: Assigning the codes

	Code
0: 50%	0
1: 20%	10
2: 5%	1100
3: 5%	11101
4: 5%	11100
5: 5%	11111
6: 2%	11010
7: 2%	110110
8: 2%	110111
9: 2%	111100
10: 2%	111101



average msg length = (1*.5)+(2*.2)+(4*.05)+(5*.17)+(6*.08) = 2.43 bits

Huffman Coding

- Generates the best set of codes, given frequencies/probabilities on all the messages.
- Creates a binary tree to construct the codes.

Construct a leaf node for each message with the given probability

Create a priority queue of messages,
(lowest probability = highest priority)

while there is more than one node in the queue:
remove the top two nodes
create a new tree node with these two nodes as children.
node probability = sum of two nodes
add new node to the queue

final node is root of tree.

Traverse tree to assign codes:
if node has code c, assign c0 to left child, c1 to right child

Video on YouTube: Text compression with Huffman coding