
# CT2109

## OOP: Data Structures and Algorithms

**Dr. Frank Glavin**
Room 404, CS Building
Frank.Glavin@University*of*Galway.ie
School of Computer Science

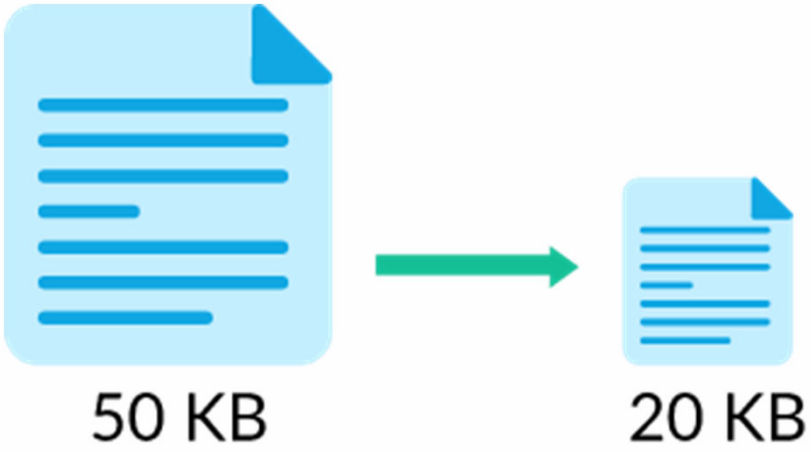
University *of* Galway.ie

# What You'll Achieve in This Topic

- Explain the concepts of lossy and lossless compression
- Describe an algorithm for run-length encoding
- Describe the Huffman encoding algorithm, including the algorithms and data structures used in it
- Demonstrate the application of run-length encoding and Huffman encoding to compressing data such as text

# Data Compression & Terminology

- Important part of data storage & transmission
- Bitstream:
  - Data is stored/transmitted in binary from
  - Stream of bits may be a file or a message
- Lossy compression:
  - Data size is reduced, but some information is lost
  - *Is this ever reasonable?*
  - *Example?*
- Lossless compression:
  - No data is lost
  - Compression is **reversible** to recover original bitstream
  - *Example?*



# Data Compression

Simple example:

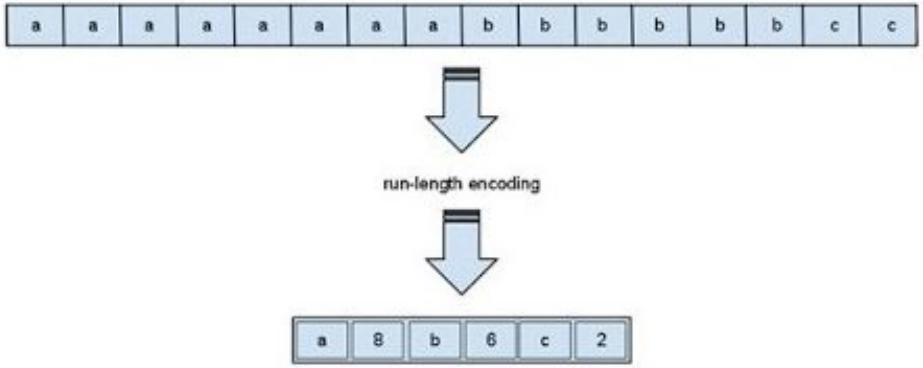
- "aaaaaabbbbbbfff" is a string
- "6a6b3f" is a simple compressed representation:

Notes:

- This is a simple form of *run-length encoding*
- Introduces new symbols to describe original sequence
- Original sequence had 15 chars, new one has 6 chars
- Can be reversed to recover original string

What about these sequences?

- "abbaafbafbbaafb"
- "aabbfaabbfaabbf"



# Can Every Bitstream Be Compressed?

Assuming **lossless** compression,
is it always possible to make a bitstream smaller?

What do you think?

# Can Every Bitstream Be Compressed?

Assuming lossless compression (i.e. reversible),
is it always possible to make a bitstream smaller?

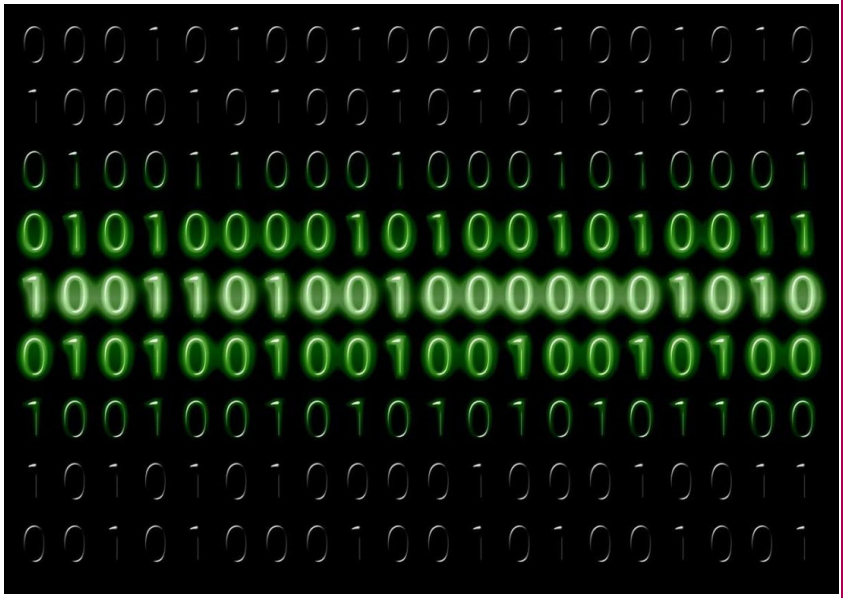
- No!

Proof by *contradiction*:

- Assume such an algorithm exists
- After applying algorithm, reapply it to resulting stream
- Continue until length is 1: impossible to reverse

Proof by *counting*:

- Assume bitstream of length N
- There are $2^N$ different such bitstreams
- There are only $2^N$-1 bitstreams of length shorter than N
- => will be at least one *collision*
- => Cannot be a reversible mapping between these



# Can Every Bitstream Be Compressed?

To be compressible, bitstream must have structure/ regularities that can be *summarized*

Amenable to compression:

- Natural text data
  - Contains frequent words that may appear often
  - Some letters appear with high frequency, but all letters have same length encoding
  - XML is routinely compressed
- Binary image data
  - Blocks of single colours:
  - long runs of 1, 0

# Can Every Bitstream Be Compressed?

- Not amenable to compression
  - Random data
  - Data that is already compressed
- Try this at home (optional!)
  - Write program to output a random binary stream, GZIP it
  - Does the file size reduce much or at all?
  - What about if you output a random **text** stream?
- Compressibility relates to **entropy**
  - Amount of disorder in a data stream
  - High entropy: low compressibility
- Note on GZIP
  - Combines **LZ77** (a dictionary encoder) & **Huffman** coding

Ollscoil na Gaillimhe
University of Galway

# Run-Length Encoding (1)

A simple compression method

Example bitstream:

000000000000111111110000000011111111111

12 **0**s + 8 **1**s + 9 **0**s + 11 **1**s

Method

Encode this as the numbers of alternating **0**s and **1**s

Always begin with the number of **0**s (which might be *none*)

Assume we use 4 bits to encode each number

Result:

1100|1000|1001|1011

12=1100, 8=1000, 9=1001, 11=1011

Compression Ratio = 16/40 = 40%

Ollscoil na Gaillimhe
University of Galway

School of Computer Science

# Run-Length Encoding (2)

## Algorithm issues

- How many bits should be used to store the counts?
- What do we do if a run is too long?
- What about runs that are shorter than the corresponding encoding?

## Standard choices

- Use 8 bits (runs are between 0 and 255 in length)
- If a run is longer than 255, insert a run of length 0
- (300 **1**s is encoded as 255 **1**s + 0 **0**s + 45 **1**s
- Encode short runs, even if this lengthens the output

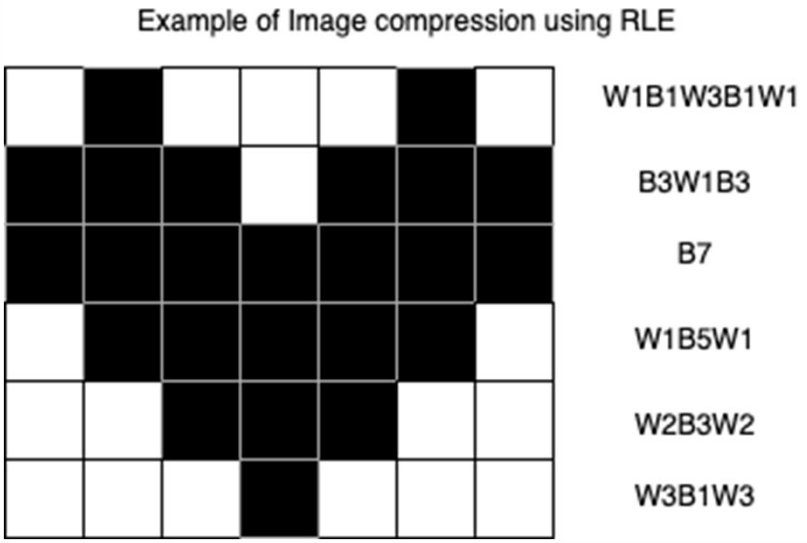
# Run-Length Encoding (3)

Popular for bitmaps

- If a bitmap's resolution is doubled:
  - bitstream increases x4
  - RLE compressed version typically increases x2

Works in a single pass:

- No need to look ahead when compressing or decompressing



# Run-Length Encoding: Limitations

If bitstream has large numbers of short runs
- RLE encoded version can be **longer** than the original!
- Worst case: 10101010
- *How would this be encoded?*

Natural text contains few repeated letters
- Traditional binary representation is 7-bit or 8-bit ASCII
- This tends to contain short runs also:
- 8-bit run encoding is definitely too long;
- 4-bit run encoding does not do much for it either

# ASCII Binary Representation (Excerpt)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 48 | 0 | 00110000 | 67 | C | 01000011 | 86 | V | 01010110 | 105 | i | 01101001 |
| 49 | 1 | 00110001 | 68 | D | 01000100 | 87 | W | 01010111 | 106 | j | 01101010 |
| 50 | 2 | 00110010 | 69 | E | 01000101 | 88 | X | 01011000 | 107 | k | 01101011 |
| 51 | 3 | 00110011 | 70 | F | 01000110 | 89 | Y | 01011001 | 108 | l | 01101100 |
| 52 | 4 | 00110100 | 71 | G | 01000111 | 90 | Z | 01011010 | 109 | m | 01101101 |
| 53 | 5 | 00110101 | 72 | H | 01001000 | 91 | [ | 01011011 | 110 | n | 01101110 |
| 54 | 6 | 00110110 | 73 | I | 01001001 | 92 | \ | 01011100 | 111 | o | 01101111 |
| 55 | 7 | 00110111 | 74 | J | 01001010 | 93 | ] | 01011101 | 112 | p | 01110000 |
| 56 | 8 | 00111000 | 75 | K | 01001011 | 94 | ^ | 01011110 | 113 | q | 01110001 |
| 57 | 9 | 00111001 | 76 | L | 01001100 | 95 | _ | 01011111 | 114 | r | 01110010 |
| 58 | : | 00111010 | 77 | M | 01001101 | 96 | ` | 01100000 | 115 | s | 01110011 |
| 59 | ; | 00111011 | 78 | N | 01001110 | 97 | a | 01100001 | 116 | t | 01110100 |
| 60 | < | 00111100 | 79 | O | 01001111 | 98 | b | 01100010 | 117 | u | 01110101 |
| 61 | = | 00111101 | 80 | P | 01010000 | 99 | c | 01100011 | 118 | v | 01110110 |
| 62 | > | 00111110 | 81 | Q | 01010001 | 100 | d | 01100100 | 119 | w | 01110111 |
| 63 | ? | 00111111 | 82 | R | 01010010 | 101 | e | 01100101 | 120 | x | 01111000 |
| 64 | @ | 01000000 | 83 | S | 01010011 | 102 | f | 01100110 | 121 | y | 01111001 |
| 65 | A | 01000001 | 84 | T | 01010100 | 103 | g | 01100111 | 122 | z | 01111010 |
| 66 | B | 01000010 | 85 | U | 01010101 | 104 | h | 01101000 | 123 | { | 01111011 |

# 8-Bit ASCII Encoding

Consider the text “Mississippi”

8-bit ASCII:

01001101011010010111001101110011011010010111
00110111001101101001011100000111000001101001

Every character encoded with 8 bits

88 bits

RLE will not compress it well

How can we do better?

Do we need all 8 bits?

Yes, in general case if we use all ASCII characters

Do all characters need a full 8 bits to encode them?

$2^8$-1 possibilities if we used all seqs of length 1-7

| | |
|---|---|
| M | 01001101 |
| i | 01101001 |
| s | 01110011 |
| s | 01110011 |
| i | 01101001 |
| s | 01110011 |
| s | 01110011 |
| i | 01101001 |
| p | 01110000 |
| p | 01110000 |
| i | 01101001 |

# Huffman Encoding (1)

Main idea: variable-length codes

- Use short-length codes for more frequently occurring characters

These should be prefix-free

- If we used a simple encoding such as A=0, B=1,
  C=10, …, would not know if 10 encoded "BA" or "C"
- We want codes to be uniquely decodable without needing any delimiters or prefixes
  (Fixed-length codes like 8-bit ASCII are also prefix-free)

# Huffman Encoding (2)

To come up with a prefix-free variable encoding,
we construct a Huffman Tree:

- This is a binary tree that will give us the final encoding
- See next slides

Using the tree, read off variable-length codes for each character

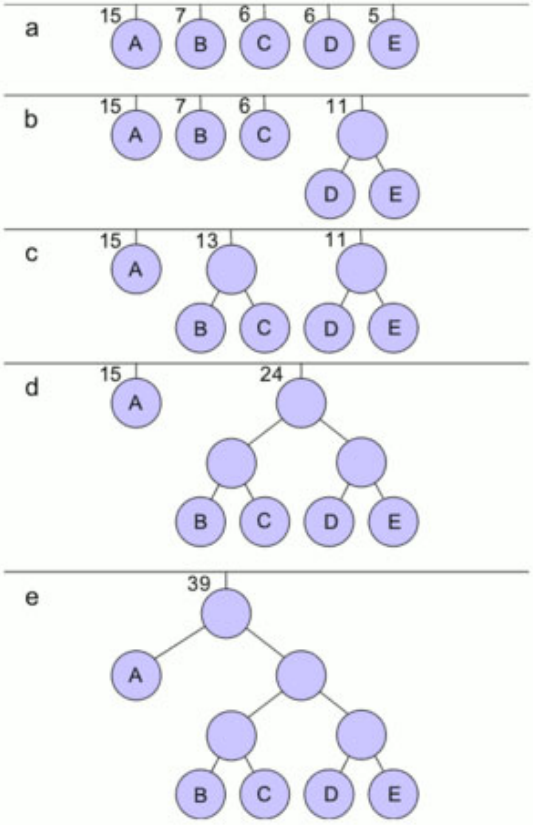
Encode this message using the tree

Note: we need to store the encoding as well as the message being encoded

- Reduces compression ratio a lot for short messages
- Can use an agreed encoding, e.g. for English text
- With an agreed encoding, computation time of constructing the tree is also avoided

# Huffman Encoding – Example (1)

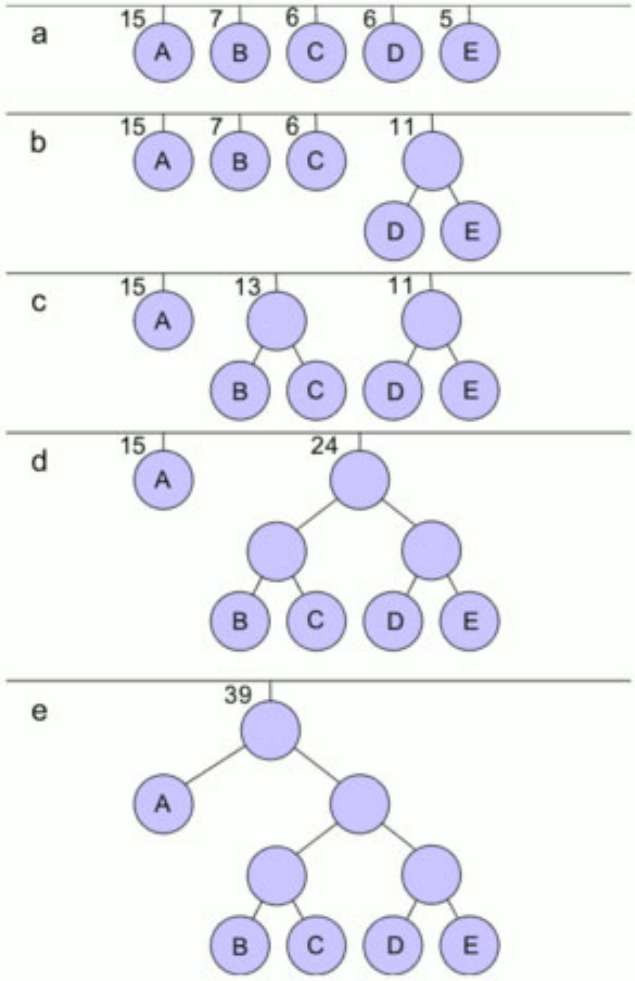
1. Create a set of trees, each consisting of one leaf; each leaf represents a symbol
2. Remove the two trees with the lowest probability, merge them into a single tree, sum up their probabilities and return the new tree into the pool
3. Repeat step 2 until a single tree is left over
4. Generate code words as seen before



# Huffman Encoding – Example (2)

In the example on the right the following code is generated:

| Symbol | Code |
|---|---|
| A | 0 |
| B | 100 |
| C | 101 |
| D | 110 |
| E | 111 |



# Algorithm to Construct Huffman Tree Using a Priority Queue

1. Count frequencies of all letters
2. Put them as nodes in a Priority Queue, with *lowest* count having *highest* priority

   A queue where items are inserted according to priority, not at the end; dequeued as normal from front

   In case of a tie, put more recently enqueued item after older items
3. While there are at least 2 nodes on the queue:

   Dequeue the 2 nodes at the front

   Make a new node with them as its 2 children

   Value of new node = sum of children's counts

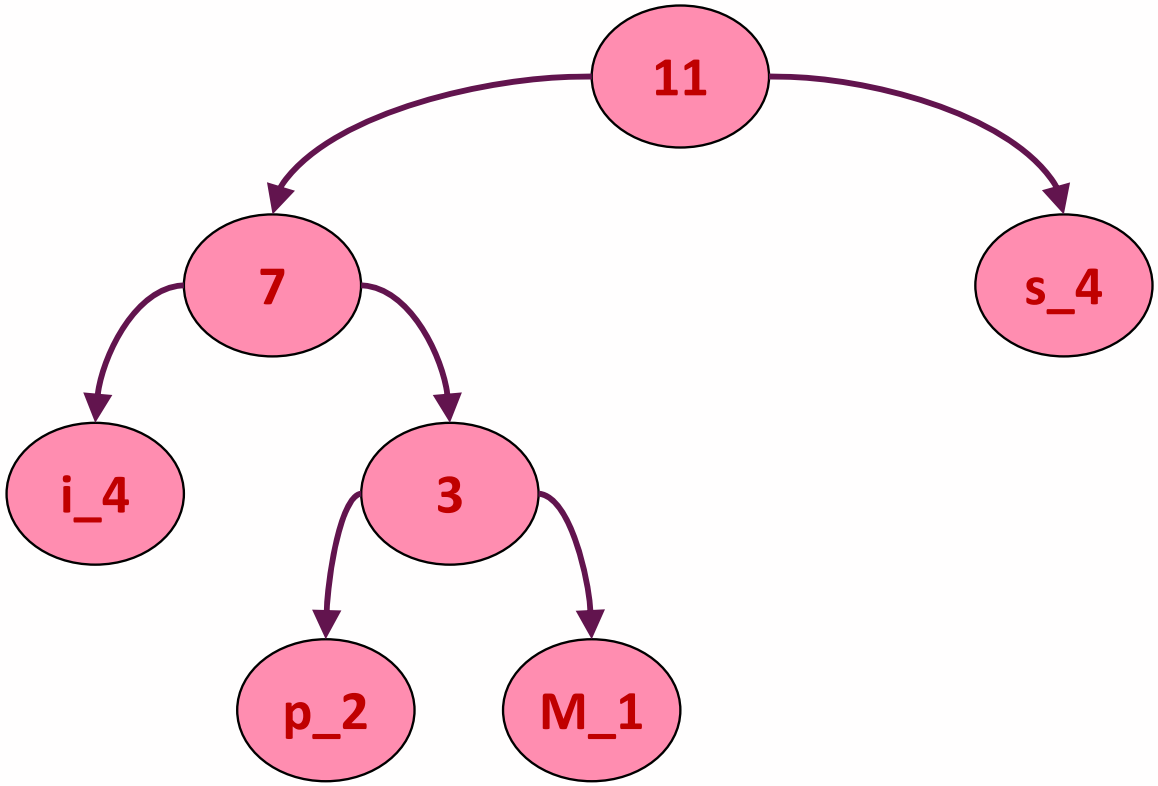
   Enqueue this new node

# Let's Do It …

M:1 p:2 i:4 s:4

Each node, apart from the root, represents a bit in the Huffman code: each left child is a 0 and each right child is a 1

# One Possible Result




Ollscoil na Gaillimhe
University of Galway

School of Computer Science

# Resultant Encoding

With this, "Mississippi"
encodes to:

10011001100111011011 1

Just 21 bits

Excluding the code,

Compression Ratio = 23.9%

Space Savings = 1 – CR = 76.1%

Can you decode this unambiguously?

For decoding, can use the code directly, or use the tree:

Start at root

0:left, 1:right, until a leaf is reached

| | |
|---|---|
| s | 0 |
| i | 11 |
| p | 101 |
| M | 100 |

| | |
|---|---|
| M | 100 |
| i | 11 |
| s | 0 |
| s | 0 |
| i | 11 |
| s | 0 |
| s | 0 |
| i | 11 |
| p | 101 |
| p | 101 |
| i | 11 |

Ollscoil na Gaillimhe
University of Galway

School of Computer Science

# Aside: Relationship to Machine Learning

Compression relates to Machine Learning:

- Goal is to find descriptions (hypotheses) that partially/fully describe a larger set of data

Usually the hypotheses are expressed differently than in zip algorithms

- E.g. rules; equations; graphs
- Almost always **lossy**: focus on main features of data, not every possible detail
- Find hypotheses to fit data through **heuristic search**

Standard ZIP algorithms have been used as similarity metrics for large documents

# What You Achieved in This Topic

- Explain the concepts of lossy and lossless compression
- Describe an algorithm for run-length encoding
- Describe the Huffman encoding algorithm, including the algorithms and data structures used in it
- Demonstrate the application of run-length encoding and Huffman encoding to compressing data such as text