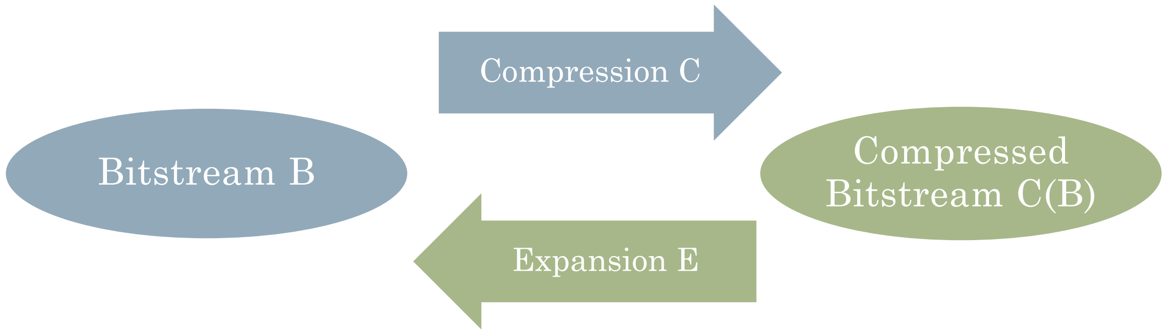
Compression

Kathrin Stark
Heriot-Watt University, 2024

Compression – Basic Model

Motivation:

- Reduce file space needed (e.g., video clips)
- Often built into disk controllers/Database Management Systems
- Reduce time to transfer information over network (e.g., streaming)



Lossless compression: $E(C(B)) = B$ Typically referred to as **correctness**

For **images and videos**, lossy methods are ok as long as only fine detail is lost.
(e.g., JPEG for images, MPEG for videos)

What You'll Learn

- Different **algorithms** for **lossless** compression: Run-length encodings, Huffman code, and LZW
- How the **compression ratio** allows us to measure the efficiency of compression
- Why no algorithm exists that can compress all bitstrings

Compression Ratio

Definition

- **Compression Ratio** $= \frac{\text{uncompressed size}}{\text{compressed size}}$
  - Attention when using different data types!
  - For example:
    - Compressing 8 byte into 3 int yields a compression rate of $\frac{8}{3 * 4} = \frac{2}{3}$
- Typically depends on how much structure exists in the input file (technically: how much **entropy** ("information") exists)
  - Easiest form: Repetition
  - But also: Known structure, i.e. generated by a specific program

Run-Length Encoding

- **Idea**: Look for **runs** of repeated characters and replace them with count + relevant character:

  aaaaaaabbbbbbbbcccc => 7a8b4c

  What's the compression ratio?

- **Binary files**: Don't need to specify the character; assume files always start with a zero.

  111111110000111111 => 0846

  What's the compression ratio?

- **Usage:** Popular way to encode **bitmaps**, a component of image compression (in particular, black and white)

- **Properties**:
  - Easy to encode
  - Not always efficient

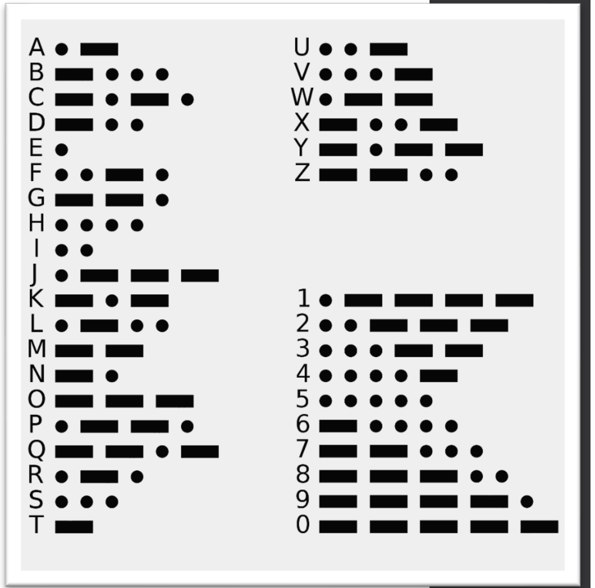
```
0 1 1 1 1 1 1 0
0 0 0 0 0 0 1 0
0 0 0 0 0 1 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0
0 0 1 0 0 0 0 0
```

Bitmap for 7

Variable-Length Encoding

- **Idea**: Some characters are more common than other – use special codes for them.
  - *Example*: Morse Code.
- **Code** = mapping of each character of an alphabet to a binary code word.
- **Variable length encoding =** words can be of variable length
  - … do we need a delimiter between variable-length encodings?
    - For example, in Morse code: Without delimiters ● ● could denote both *ee* or *i*.
  - **Prefix code** = binary code such that no code word is the prefix of another code word.

    Question: What does 01101110 decode with the right table? Why is it unique?
- **Encoding tree** = Represents a prefix code.
  - Usually presented by **tries** (see next slide).



Morse Code

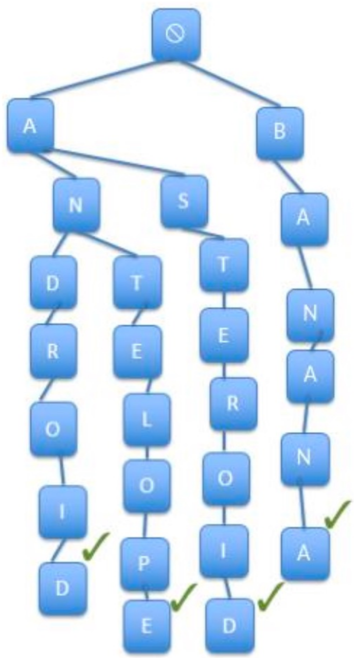
Character	Code
a	0
b	110
c	100
d	101
r	111

Example Prefix Code

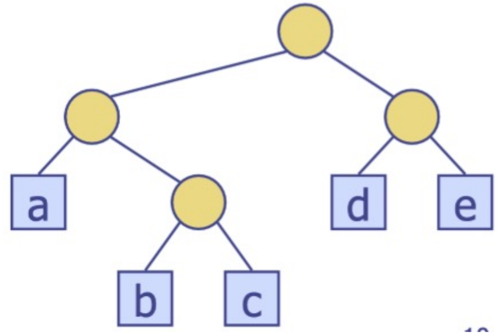
Reminder: Tries

Tries

- A form of n-ary tree (pronounced *try*)
- Efficient way to store a dictionary
- Each level stores a character position
- *Nth* level stores the *nth* character of word
- A word is valid if
  1. Each character in word appears at correct level of tree
  2. Node containing final character is either:
     - a leaf
     - Marked as valid word
  3. That node is **marked** as a **valid word**
- Lookup O(1) worst case
  – where N is number of words



Source: F28SG



00	010	011	10	11
a	b	c	d	e

Idea: Symbols are stored in leaves.
Encoding = path to leaf.

Operation	**Average**	**Worst case**
Search	O(n)	O(n)
Insert	O(n)	O(n)
Delete	O(n)	O(n)

Variable Length Encoding
Pseudocode

Compress
Input: A code in form of a trie
Algorithm:
For all symbols s:
1. Start at leaf of trie corresponding to symbol s.
2. Follow the path up to the root.
3. Print bits in reverse order.

Enhance
Input: A code in form of a trie
Algorithm:
For all bit sequences:
1. Start at the root of the tree.
2. Take the right branch if the bit is 0; left branch if 1.
3. When at a leaf node, print symbol and return to root.

Given specific data, what should be the encoding we use?
Here: How can we find a code that gives the shortest encoding given information on how frequently different characters occur.

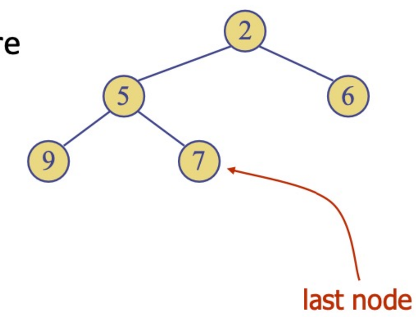
Reminder: Priority Queue

Priority Queue ADT

- Each entry has a key and value
  - **Key**: could be price, miles, age, …
  - **Value**: The thing that we store in the queue
- Operations
  - **size()** - how many entries
  - **isEmpty()** - is it empty or not?
  - **insert(key,value)** - Insert a key/value pair
  - **removeMin()** - Removes and returns minimum element
  - **min()** - Returns the minimum element (but keeps it)

Min Heaps

- A **min heap** is a binary tree that satisfies the following properties:
  - **Each node** except root has a value that is greater than (or equal to) its parent
  - Each level is filled up before moving to next
  - From left to right
- The last node is the rightmost node at the last level



Source: F28SG

Min Heaps are an implementation of the priority queue ADT.

Huffman's Algorithm
Computing the Encoding

Goal: Given a string X, yield an optimal variable-length encoding.

Algorithm HuffmanEncoding(X)
Input: string X of size n
Output: Optimal encoding trie for X

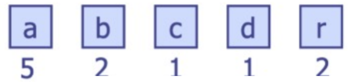
1. Compute the frequency f(c) of all distinct characters.
2. For each character c, put the character c as the one-node trie + its frequency f(c) into a priority queue Q.
3. Repeat until there is only one element in the priority queue Q:
   i. Get the two minimum elements (f1, T1) and (f2, T2).
   ii. Compute the trie combining T1 and T2, put it in Q with key f1 + f2.

Input = abracadabra

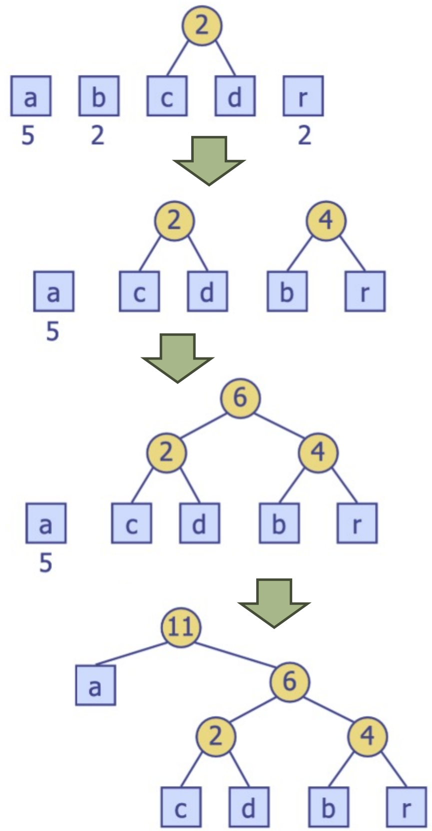
Step 1:

a	**b**	**c**	**d**	**r**
5	2	1	1	2

Step 2:



Step 3:



Huffman's Algorithm
Pseudocode + Runtime Analysis

Operation	**Average**	**Worst case**
insert	O(1)	O(log n)
minKey	O(1)	O(1)
removeMin	O(log n)	O(log n)

Reminder: Runtimes min-heap

```
Algorithm HuffmanEncoding(X)
  Input string X of size n
  Output optimal encoding trie for X
  C ← distinctCharacters(X)
  computeFrequencies(C, X)
  Q ← new empty heap
  for all c ∈ C
    T ← new single-node tree storing c
    Q.insert(getFrequency(c), T)
  while Q.size() > 1
    f1 ← Q.minKey()
    T1 ← Q.removeMin()
    f2 ← Q.minKey()
    T2 ← Q.removeMin()
    T ← join(T1, T2)
    Q.insert(f1 + f2, T)
  return Q.removeMin()
```

1. Compute the frequency f(c) of all distinct characters.

Worts-case runtime: O(n)

2. For each character c, put the character c as the one-node trie + its frequency f(c) into a priority queue Q.

Worst-case runtime: O(d log d)

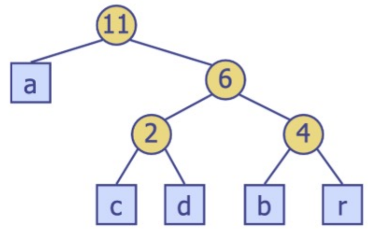
3. Repeat until there is only one element in the priority queue Q:
   i. Get the two minimum elements (f1, T1) and (f2, T2).
   ii. Compute the trie combining T1 and T2, put it in Q with key f1 + f2.

Worst-case runtime: O(d log d)

Worst-case runtime: O(n + d log d)

Huffman's Algorithm
Efficiency (I)

a	b	c	d	r
5	2	1	1	2



Goal: Prove that Huffman's encoding yields the optimal prefix code.

Let's start with some definitions!

Let H be the Huffman encoding for a text t with alphabet r.
Then, the **total encoding length** of the compressed text is:

$$\mathrm{L(H)} = \sum_{s\in r} f(s) * \underbrace{|H(s)|}_{\text{depth in the trie/number of bits}}$$

A prefix-free code c is **optimal** if for all other prefix-free codes c':

$$\sum_{s\in r} f(s) * |c(s)| \leq \sum_{s\in r} f(s) * |c'(s)|$$

Observation 1: Whenever we have an optimal encoding, the code with the lowest frequency has to have the largest depth.

Observation 2: The total encoding length corresponds to the **weighted external path length**, i.e. above:

$$L(H) = 11 + 6 + 2 + 4$$

Huffman's Algorithm
Efficiency (II)

Theorem: Given a set *r* of symbols and frequencies *f*, the Huffman algorithm builds an optimal prefix-free code.

Proof. By induction on the number of symbols.
If $r \leq 2$, the code is optimal, as we can't encode any symbols with less than a bit.

If $r > 2$, assume that s_i and s_j are the first two symbols chosen in the Huffman encoding H.
If we replace s_i and s_j by a symbol s_{ij} with $f_{ij} = f_i + f_j$,this results in the Huffman tree H_{ij} where the node with leaves for s_i and s_j is replaced by a single leaf s_{ij} and hence:

$$L(H) = L(H_{ij}) + f_i + f_j$$

By the induction hypothesis, the Huffman code is optimal for this new set of symbols.

Assume that *H'* is an optimal prefix tree for *r*.
By observation 1, we know that f_i and f_j are at the lowest level of the tree, and we can assume that f_i and f_j are siblings (otherwise: swap f_j's sibling with f_i). Let H'_{ij} be the prefix tree where we replace the node for s_i and s_j with (s_{ij}, $f_i + f_j$). H'_{ij} Is now a (not necessarily optimal) prefix tree with $L(H') = L\left(H'_{ij}\right) + f_i + f_j$. Hence:

Then $L(H) = L(H_{ij}) + f_i + f_j <= L(H'_{ij}) + f_i + f_j = L(H')$, i.e. H is optimal.

LZW Algorithm

- **Substitutional compression:**
  Find repeated sequences, not just runs (i.e., *the* cat and *the* dog)
- LZW = Lempel-Ziv-Welch (1984)
- **Compression**:
  1. Start with an initial dictionary (e.g., ASCII/UNICODE).
     Let n be the length/highest value of the dictionary.
  2. *Repeat* until there are no more input characters:
     a) Find the longest string s in the symbol table a prefix of the unscanned input.
     b) Write the value associated with s.
     c) Scan one character c past s in the input and put c ++ s in the dictionary with the value $n + 1$.

Value = ASCII in hex

Input: ABRACADABRABRABRA
Output:

A	41
…	
EOF	80

Input: ABRACADABRABRABRA
Output: 41

A	41
…	
EOF	80
AB	81

Input: ABRACADABRABRABRA
Output: 4142

A	41
…	
EOF	80
AB	81
BR	82

…

LZW Algorithm
Full Example Compression

Input: ABRACADABRABRABRA

Output:

41	**42**	**52**	**41**	**43**	**41**	**44**	**81**	**83**	**82**	**88**	**41**	**80**
A	B	R	A	C	A	D	AB	RA	BR	ABR	A	EOF

Q: What is the compression rate?

Key

Value

A	41
…	
EOF	80
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

LZW Algorithm

Expansion:

1. Start with the **inversed** initial dictionary (e.g., ASCII/UNICODE).
   Let n be the length/highest key of the dictionary.
2. Repeat until there are no more code words s:
   a) Write the value val associated with s.
   b) If possible, scan one code word s' past s in the input.
   c) Decode s' to value val' with first character c.
   d) Put the value val ++ c at key n + 1.

Note: No full dictionary has to be given for the full encoding!

Input: 41425241434144818382884180
Output:

41	A
…	…
80	EOF

Input: 41425241434144818382884180
Output: A

41	A
…	…
80	EOF
81	AB

Input: 41425241434144818382884180
Output: AB

41	A
…	…
80	EOF
81	AB
82	BR

…

LZW Algorithm
Full Example Expansion

Input: 4142524143414481838288 4180

Output:

A	B	R	A	C	A	D	AB	RA	BR	ABR	A	EOF

A	41
…	
EOF	80
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

LZW Algorithm
Implementation Details

- **Usage**: Used in Unix' compress (e.g., part of tar), GIF image format
- **Compression ratio:** A large English text file can typically be compressed via LZW to about half of its original size
- Some annoyance: How do we know how long a code word is?
  - Standard: The compression algorithm increases the width when a code c is added to the table at position 2^n . From the encoding of the next input character, increase the width to 1 + n.
  - Alternative (”early change”): Encode already c with width 1 + n.
    This used to be standard in earlier implementations. (I.e., Adobe allows both variants)

Does an Universal Encoding Exist?

Lemma. No algorithm can compress every bitstring b.

Proof. Assume the bitstrings b = 0 and b = 1. Both would have to be encoded by the empty bitstring, allowing no unique expansion.

Lemma. No algorithm can compress every bitstring b of length l > n.

Proof. Proof by contradiction. Assume an algorithm $f = (c, e)$ that compresses every bitstring b of length > n, i.e. let n be the number of possible bitstrings of length l and let m be the number of possible compressed bitstrings of length less than l.
By the **Pigeonhole Principle**, at least one compressed bitstring must correspond to more than one bitstring.

But then the encoding cannot be lossless, as for at least one of the original bitstrings *b:*

$e(c(b)) \neq b$

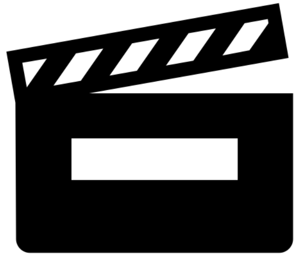
Lossy Compression

Audio Compression

- Lossless compression: Find patterns/redundancy in audio data (i.e., FLAC/ALAC)
- Lossy compression: Exploit that some parts of the audio signal are less perceptible to the human ear and can be discarded/approximated (e.g., MP3)

Picture Compression

- Lossless compression: Find patterns/redundancy in image data (i.e., PNG, lossless PEG)
- Lossy compression: Discard data that is less perceptible to the human eye (i.e., it's ok if fine detail is lost).
- Can reduce by a factor of 5 without a perceptible loss in quality.



Video Compression

- Roughly: Uses JPEG compression for frame, looks at differences between frames rather than recoding everyone (i.e., MPEG)

What You Learned

- Different **algorithms** for **lossless** compression: Run-length encodings, Huffman code, and LZW
- How the **compression ratio** allows us to measure the efficiency of compression
- Why no algorithm exists that can compress all bitstrings

Further Reading

- Chapter 5.0, chapter 5.5 of the book.
- Chapter 5.2 if you need a reminder of tries.