Outline

- Definition of **Entropy**
- Three Entropy coding techniques:
  - Huffman coding
  - Arithmetic coding
  - Lempel-Ziv coding

Entropy Coding

(taken from the Technion)

Entropy

Entropy of a set of elements $e_1,\ldots,e_n$ with probabilities $p_1, \ldots p_n$ is:

$$H\left(p_1 \ldots p_n\right) \equiv -\sum_{\forall i} p_i \log_2 p_i$$

Entropy (in our context) - smallest number of bits needed, **on the average**, to represent a symbol (the average on all the symbols code lengths).

Note: $\log_2 p_i$ is the **uncertainty** in symbol e_i (or the “surprise” when we see this symbol). Entropy – average “surprise”

Assumption: there are no dependencies between the symbols’ appearances

Definitions

Alphabet: A finite set containing at least one element: **A** = {a, b, c, d, e}

Symbol: An element in the alphabet: $s \in$ **A**

A string over the alphabet: A sequence of symbols, each of which is an element of that alphabet: ccdabdcaad…

Codeword: A sequence of bits representing a coded symbol or string: 110101001101010100…

p_i: The occurrence probability of symbol s_i in the input string. $\sum_{\forall i \in A} p_i = 1$

L_i: The length of the codeword of symbol s_i in bits.

Entropy example

Entropy calculation for a two symbol alphabet.

Example 1:	A	p_A=0.5
	B	p_B=0.5

$$\mathrm{H}(\mathrm{A},\mathrm{B}) = -\mathrm{p_A}\log_2 \mathrm{p_A} - \mathrm{p_B}\log_2 \mathrm{p_B} = \\ = -0.5\log_2 0.5 - 0.5\log_2 0.5 = 1$$

It requires one bit per symbol on the average to represent the data.

Example 2:	A	p_A=0.8
	B	p_B=0.2

$$H(A,B) = -p_A\log_2 p_A - p_B\log_2 p_B = \\ = -0.8\log_2 0.8 - 0.2\log_2 0.2 \cong 0.7219$$

It requires less than one bit per symbol on the average to represent the data.

How can we code this ?

Entropy examples

- Entropy of $e_1,\ldots e_n$ is maximized when

  $p_1=p_2=\ldots=p_n=1/n \rightarrow \mathrm{H}(e_1,\ldots,e_n)=\log_2 n$

  - No symbol is “better” than the other or contains more information
  - 2^k symbols may be represented by k bits

- Entropy of $p_1,\ldots p_n$ is minimized when

  $p_1=1,\ p_2=\ldots=p_n=0 \rightarrow \mathrm{H}(e_1,\ldots,e_n)=0$

Entropy coding

- Entropy is a ***lower bound*** on the average number of bits needed to represent the symbols (the data compression limit).
- Entropy coding methods:
  - Aspire to achieve the entropy for a given alphabet, BPS→Entropy
  - A code achieving the entropy limit is optimal

BPS : bits per symbol

$$\mathrm{BPS} = \frac{|\text{encoded message}|}{|\text{original message}|}$$

Code types

- **<u>Fixed-length codes</u>** - all codewords have the same length (number of bits)

  A – 000, B – 001, C – 010, D – 011, E – 100, F – 101

- **<u>Variable-length codes</u>** - may give different lengths to codewords

  A – 0, B – 00, C – 110, D – 111, E – 1000, F - 1011

Huffman coding

- Each symbol is assigned a variable-length code, depending on its frequency. The higher its frequency, the shorter the codeword
- Number of bits for each codeword is an integral number
- A <u>prefix code</u>
- A <u>variable-length code</u>
- Huffman code is the **<u>optimal</u>** prefix and variable-length code, **<u>given</u>** the symbols' probabilities of occurrence
- Codewords are generated by building a Huffman Tree

Code types (cont.)

- **<u>Prefix code</u>** - No codeword is a prefix of any other codeword.

  A = 0;   B = 10;   C = 110; D = 111

- **<u>Uniquely decodable code</u>** - Has only one possible source string producing it.
  - Unambigously decoded
  - Examples:
    - Prefix code - the end of a codeword is immediately recognized without ambiguity: 010011001110 → 0 | 10 | 0 | 110 | 0 | 111 | 0
    - Fixed-length code

Huffman encoding

Use the codewords from the previous slide to encode the string "BCAE":

String:	B	C	A	E
Encoded:	10	00	01	111

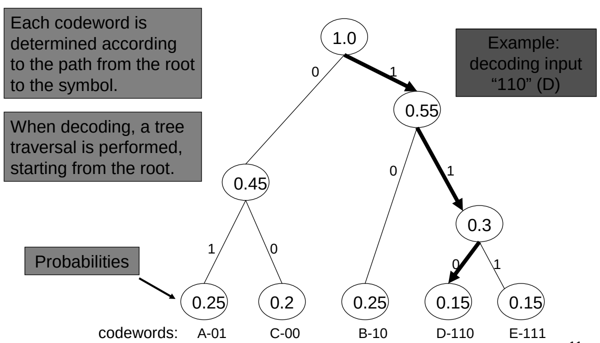
Number of bits used: 9

The BPS is (9 bits/4 symbols) = **2.25**

Entropy: - 0.25log0.25 - 0.25log0.25 - 0.2log0.2 - 0.15log0.15 - 0.15log0.15 = **2.2854**

BPS is lower than the entropy. **WHY ?**

Huffman tree example



Huffman encoding

- Build a table of per-symbol encodings (generated by the Huffman tree).
  - Globally known to both encoder and decoder
  - Sent by encoder, read by decoder
- Encode one symbol after the other, using the encoding table.
- Encode the pseudo-eof symbol.

Huffman tree construction

- Initialization:
  - Leaf for each symbol x of alphabet **A** with *weight*=p_x.
  - Note: One can work with integer weights in the leafs (for example, number of symbol occurrences) instead of probabilities.
- **while** (tree not fully connected) **do begin**
  - *Y, Z* $\leftarrow$ lowest_root_weights_tree()
  - r $\leftarrow$ new_root
  - r->attachSons(Y, Z) // attach one via a 0, the other via a 1 (order not significant)
  - weight(r) = *weight(Y)+weight(Z)*

Symbol probabilities

- How are the probabilities known?
  - Counting symbols in input string
    - Data must be given in advance
    - Requires an extra pass on the input string
  - Data source’s distribution is known
    - Data not necessarily known in advance, but we know its distribution

Huffman decoding

- Construct decoding tree based on encoding table
- Read coded message bit-by-bit:
  - Travers the tree top to bottom accordingly.
  - When a leaf is reached, a codeword was found $\rightarrow$ corresponding symbol is decoded
- Repeat until the pseudo-eof symbol is reached.

No ambiguities when decoding codewords (prefix code)

Huffman Entropy analysis

Best results (entropy wise) - **only** when symbols have occurrence probabilities which are negative powers of 2 (i.e. ½, ¼, …). Otherwise, BPS > entropy bound.

Example:

Symbol	Probability	Codeword
A	0.5	1
B	0.25	01
C	0.125	001
D	0.125	000

Entropy = **1.75**
A representing probabilities input stream : **AAAABBCD**
Code: **11110101001000**
BPS = (14 bits/8 symbols) = **1.75**

Example

"Global" English frequencies table:

Letter	Prob.	Letter	Prob.
A	0.0721	N	0.0638
B	0.0240	O	0.0681
C	0.0390	P	0.0290
D	0.0372	Q	0.0023
E	0.1224	R	0.0638
F	0.0272	S	0.0728
G	0.0178	T	0.0908
H	0.0449	U	0.0235
I	0.0779	V	0.0094
J	0.0013	W	0.0130
K	0.0054	X	0.0077
L	0.0426	Y	0.0126
M	0.0282	Z	0.0026
Total: 1.0000			

Huffman summary

- Achieves entropy when occurrence probabilities are negative powers of 2
- Alphabet and its distribution must be known in advance
- Given the Huffman tree, very easy (and fast) to encode and decode
- Huffman code is not unique (because of some arbitrary decisions in the tree construction)

Huffman tree construction complexity

- Simple implementation - $o(n^2)$.
- Using a Priority Queue - $o(n \cdot \log(n))$:
  - Inserting a new node – $o(\log(n))$
  - n nodes insertions - $o(n \cdot \log(n))$
  - Retrieving 2 smallest node weights – $o(\log(n))$

Arithmetic coding

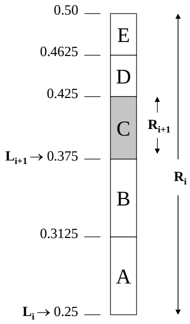
- Assigns one (normally long) codeword to entire input stream
- Reads the input stream symbol by symbol, appending more bits to the codeword each time
- Codeword is a **number**, representing a segmental sub-section based on the symbols' probabilities
- Encodes symbols using a non-integer number of bits → very good results (entropy wise)

Arithmetic coding

Mathematical definitions

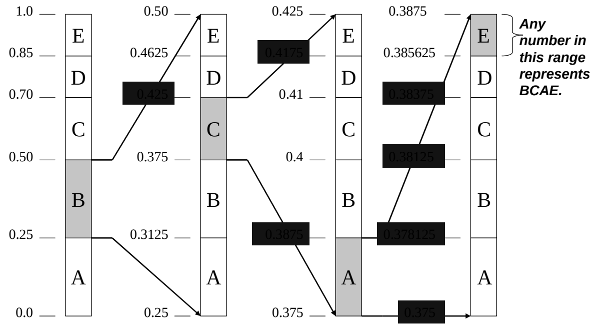
L – The smallest binary value consistent with a code representing the symbols processed so far.

R – The product of the probabilities of those symbols.



Example

Coding of **BCAE** $p_A = p_B = 0.25$, $p_C = 0.2$, $p_D = p_E = 0.15$



Arithmetic encoding (cont.)

Two possibilities for the encoder to signal to the decoder end of the transmission:

1. Send initially the number of symbols encoded.
2. Assign a new EOF symbol in the alphabet, with a very small probability, and encode it at the end of the message.

Note: The order of the symbols in the alphabet must remain consistent throughout the algorithm.

Arithmetic encoding

Initially L = 0, R = 1.

When encoding next symbol, L and R are refined.

$$L \leftarrow L + R\sum_{i=1}^{j-1} p_i \qquad R \leftarrow R \cdot p_j$$

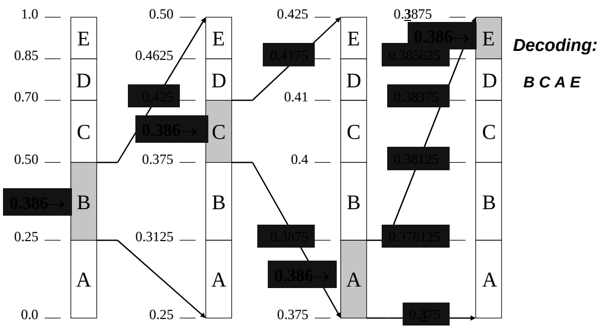
At the end of the message, a binary value between L and L+R will unambiguously specify the input message. The **shortest** such binary string is transmitted.

In the previous example:

- Any number between 385625 and 3875 (discard the '0.').
- Shortest number - 386, in binary: 110000010
- BPS = (9 bits/4 symbols) = **2.25**

Arithmetic decoding example

Decoding of **0.386**



Arithmetic decoding

In order to decode the message, the symbols order and probabilities must be passed to the decoder.

The decoding process is identical to the encoding. Given the codeword (the final number), at each iteration the corresponding sub-range is entered, decoding the symbol representing the specific range.

Distributions issues

- Until now, symbol distributions were known in advance
- What happens if they are not known?
  - Input string not known
  - Huffman and Arithmetic Codings have an adaptive version
    - Distributions are updated as the input string is read
    - Can work online

Arithmetic entropy analysis

- Arithmetic coding manages to encode symbols using non integer number of bits !
- One codeword is assigned to the entire input stream, instead of a codeword to each individual symbol
- This allows Arithmetic Coding to achieve the Entropy lower bound

Lempel-Ziv concepts

- What if the alphabet is **unknown** ? Lempel-Ziv coding solves this general case, where only a stream of bits is given.
- LZ creates its own dictionary (strings of bits), and replaces future occurrences of these strings by a shorter position string:
- In simple Huffman/Arithmetic coding, the dependency between the symbols is ignored, while in the LZ, these dependencies are identified and are exploited to perform better encoding.
- When all the data is known (alphabet, probabilities, no dependencies), it's best to use **Huffman** (LZ will try to find dependencies which are not there…)

Lempel-Ziv concepts

Lempel-Ziv algorithm

1. Initialize the dictionary to contain an empty string (D={ø}).
2. W ← longest block in input string which appears in D.
3. B ← first symbol in input string after W
4. Encode W by its index in the dictionary, followed by B
5. Add W+B to the dictionary.
6. Go to Step 2.

Lempel-Ziv compression

- Parses source input (in binary) into the shortest distinct strings:
  1011010100010 → 1, 0, 11, 01, 010, 00, 10
- Each string includes a prefix and an extra bit (010 = 01 + 0), therefore encoded as: (prefix string place, extra bit)
- Requires 2 passes over the input (one to parse input, second to encode). Can be modified to one pass.
- Compression: (n – number of distinct strings)
  - log(n) bits for the prefix place + 1 bit for the added bit
  - Overall – n·log(n) bits compressed

Compression comparison

Compressed to (percentage):	Lempel-Ziv (unix gzip)	Huffman (unix pack)
html (25k) *Token based ascii file*	**20%**	65%
pdf (690k) *Binary file*	**75%**	95%
ABCD (1.5k) *Random ascii file*	33%	**28.2%**
ABCD(500k) *Random ascii file*	29%	**28.1%**

ABCD – $\{p_A = 0.5, p_B = 0.25, p_C = 0.125, p_D = 0.125\}$

Lempel-Ziv is asymptotically optimal

Example

Input string: 1 0 1 1 0 1 0 1 0 0 0 1 0

Dictionary D	
Index	Entry
0	ø
1	1
2	0
3	11
4	01
5	010
6	00
7	10

W	ø	ø	1	0	01	0	1
B	1	0	1	1	0	0	0

Encoded string:

Pairs: (0,1) (0,0) (1,1) (2,1) (4,0) (2,0) (1,0)

Encoding: 0001 0000 0011 0101 1000 0100 0010

0001000000110101100001000010

Comparison

	Huffman	Arithmetic	Lempel-Ziv
Probabilities	Known in advance	Known in advance	Not known in advance
Alphabet	Known in advance	Known in advance	Not known in advance
Data loss	None	None	None
Symbols dependency	Not used	Not used	Used – better compression
Preprocessing	Tree building – O(n log n)	None	First pass on data (can be eliminated)
Entropy	If probabilities are negative powers of 2	Very close	Best results when alphabet not known
Codewords	One codeword for each symbol	One codeword for all data	Codewords for set of alphabet
Intuition	Intuitive	Not intuitive	Not intuitive