Ported from first doc

Information is related to probability
Information is a measure of uncertainty (or "surprise")

Self-information: information of the event itself

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A)$$

if b = 2, unit of information is bits

Entropy: Average self information of data set
a data source generates output sequence from a set {A1, A2, …, AN}
P(Ai): Probability of Ai

$$H = \sum_i -P(A_i)\log_2 P(A_i)$$

The first-order entropy represents the minimal number of bits needed to losslessly represent one output of the source.

March 4th - Compression madness

Memoryless Source:

- an information source that is independently distributed.
- i.e., the value of the current symbol does not depend on the values of the previously appeared symbols.

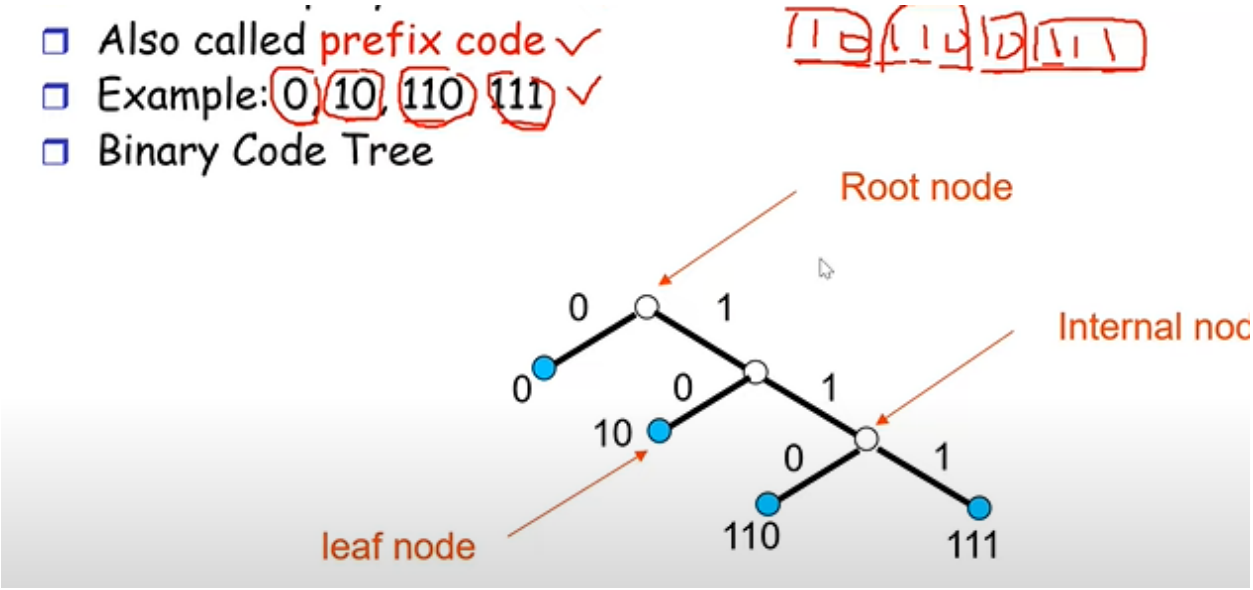
Instead of assuming a memoryless source, RunLength Coding (RLC) exploits memory present in the information source.
Rationale for RLC:

- If the information source has the property that symbols tend to form continuous groups, then such symbols and the length of the group can be coded.

Entropy is lower bound of lossless compression

Entropy coding

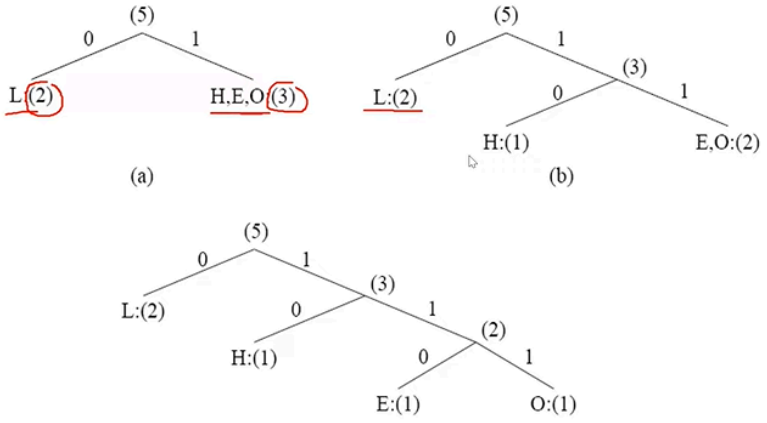
Prefix-free code. No codeword is prefix of another one. Can be uniquely decoded.



Shannon-Fano coding

Sort symbols according to frequency count of occurrences.

Recursively divide symbols into two parts, each with approximately with same number of counts, until all parts contain only one symbol. Example: hello



This isn't unique, we can also have a coding tree that looks different, but still has single symbols on leaves.

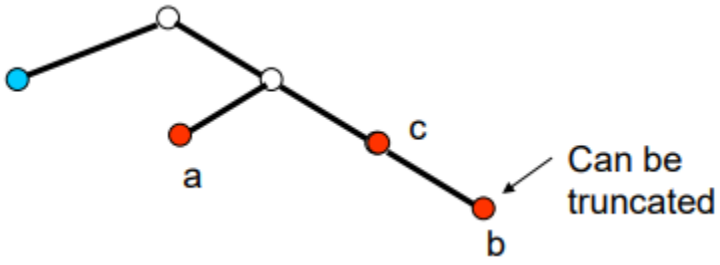
Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
TOTAL number of bits:				10

Huffman coding

Binary tree for prefix-free code. This will produce optimal prefix-free code.

Observations:

- Frequent symbols have short codes.
- In an optimum prefix-free code, the two codewords (symbol) that occur least frequently will have the same length.



Form subtree for c and b

Instead of Shannon, we start bottom up and form subtree for least frequent symbols.

Human Coding - a bottom-up approach

Initialization: Put all symbols on a list sorted according to their frequency counts.

• This might not be available! (like live streaming)

Repeat until the list has only one symbol left:

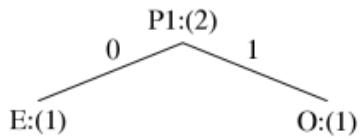
(1) From the list pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node.

(2) Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.

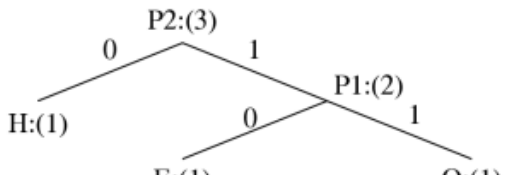
(3) Delete the children from the list.

Assign a codeword for each leaf based on the path from the root.

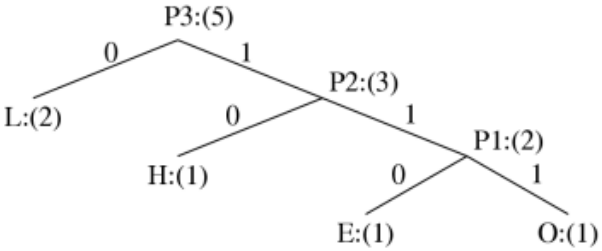
Hello example



(a)



(b)



Properties of Huffman coding

Unique Prefix Property:

- No Huffman code is a prefix of any other Huffman code - precludes any ambiguity in decoding.

Optimality:

- minimum redundancy code - proved optimal for a given data model (i.e., a given, accurate, probability distribution) under certain conditions.
- The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
- Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.

Average Huffman code length for an information source S is strictly less than entropy+ 1.

Cons:

- Need a probability distribution
- Usually estimated from a training set, but the practical data could be quite different.
- Hard to adapt to changing statistics.
- Minimum codeword length is 1 bit

Example:
Source alphabet A = {a, b, c, d, e} Probability distribution: {0.2, 0.4, 0.2, 0.1, 0.1}
Code: {01, 1, 000, 0010, 0011}

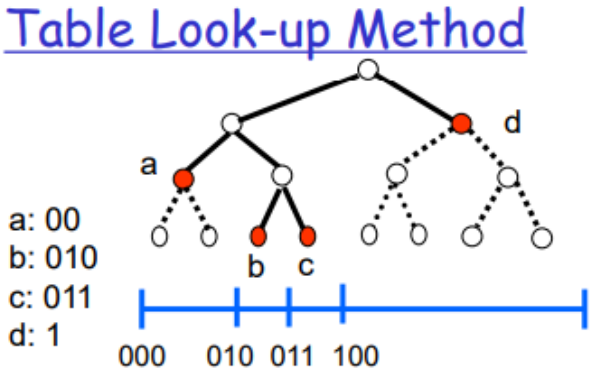
Entropy: $H(S) = -(0.2*\log_2(0.2)*2 + 0.4*\log_2(0.4)+0.1*\log_2(0.1)*2) = 2.122$ bits/symbol
Average Huffman codeword length: $L = 0.2*2+0.4*1+0.2*3+0.1*4+0.1*4 = 2.2$ bits / symbol
In general: $H(S) \leq L < H(S) + 1$

What about decoding?
Table Look-up Method N: # of codewords L: max codeword length

Expand to a full tree:

- Each Level-L node belongs to the subtree of a codeword.
- Equivalent to dividing the range [0, 2^L] into N intervals, each corresponding to one codeword.

bar[5]: {000, 010, 011, 100, 1000} Read L bits, and find which internal it belongs to. How to do it fast?




```
char HuffDec[8][2] = {
  {'a', 2},
  {'a', 2},
  {'b', 3},
  {'c', 3},
  {'d', 1},
  {'d', 1},
  {'d', 1},
  {'d', 1}
};
```

```
x = ReadBits(3);
k = 0;    //# of symbols decoded
While (not EOF) {
   symbol[k++] = HuffDec[x][0];
   length = HuffDec[x][1];
   x = x << length;
   newbits = ReadBits(length);
   x = x | newbits;
  x = x & 111B;
}
```

We can fix some of these problems with extended huffman code

Extended huffman code

Code multiple symbols jointly

- Composite symbol: (X1, X2, …, Xk). Alphabet increased exponentially: N^k Also called Joint Huffman code
- 2 symbol joint huffman code E.g., ABABBAAB vs AB AB BA AB.
  - This will change prob distribution from A: ½, B:½ to AB: ¾, BA: ¼.

Another example

$P(X_i = 0) = P(X_i = 1) = 1/2$

- Entropy $H(X_i) = 1$ bit / symbol

Joint probability: $P(X_{i-1}, X_i)$: P(0, 0) = 3/8, P(0, 1) = 1/8 P(1, 0) = 1/8, P(1, 1) = 3/8

Second order entropy: $H(X_{i-1},X_i)$ 1.8113bits / 2 symbols, or 0.9056 bits / symbol

Huffman code for X_i	0, 1
Average code length	1 bit / symbol
Huffman code for (X_{i-1}, X_i)	1, 00, 010, 011
Average code length	0.9375 bit /symbol

LZW Coding

Dictionary-based code: for each symbol we will have code to describe it.

Uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together

- e.g., words in English text.
- Encoder and decoder build up the same dictionary dynamically while receiving the data.
- Places longer and longer repeated entries into a dictionary, and then emits the code for an element, rather than the string itself, if the element has already been placed in the dictionary.

```
BEGIN
        s = next input character;
        while not EOF
        {
          c = next input character;

          if s + c exists in the dictionary
                s = s + c;
          else
            {
                output the code for s;
                add string s + c to the dictionary with a new code;
                s = c;
            }
        }
        output the code for s;
END
```

Example for input string ABABBABCABABBA

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	A	4	10	ABA
A	B			
AB	B			
ABB	A	6	11	ABBA
A	EOF	1		

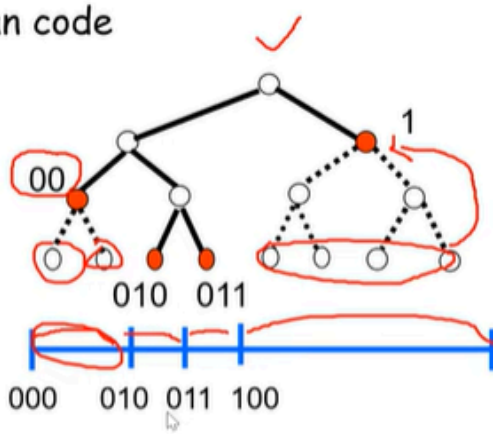
Output codes: 1 2 4 5 2 3 4 6 1.
Instead of sending 14 characters, only 9 codes need to be sent (compression ratio = 14/9 = 1.56).
By column: Assign A, B, C
And remap

Arithmetic coding

Can resolve many limitations of huffman coding such as needing a prob distribution and minimum codeword length being 1 bit.

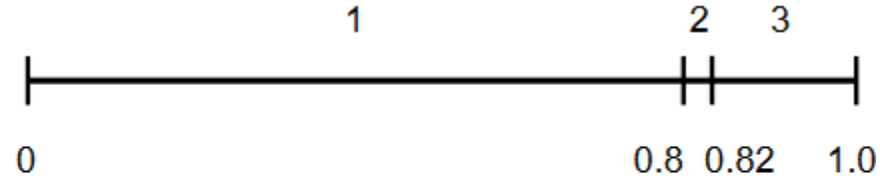
Arithmetic coding divides interval recursively.

- Recall table look-up decoding of Huffman code
  - N: alphabet size
  - L: Max codeword length
  - Divide [0, 2^L] into N intervals
  - One interval for one symbol
  - Interval size is roughly proportional to symbol prob.



- Arithmetic coding applies this idea recursively
  - Normalizes the range [0, 2^L] to [0, 1].
  - Map a sequence to a unique tag in [0, 1).

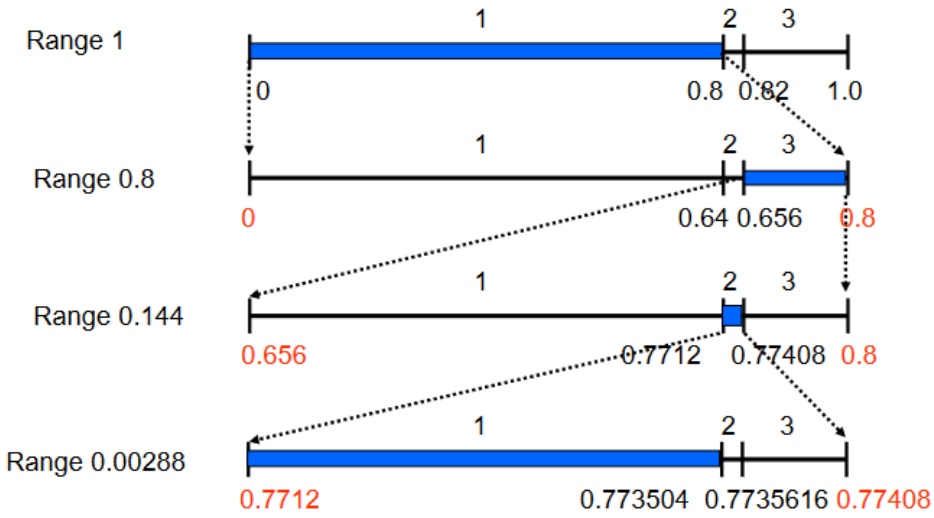
Example) disjoint and complete partition of the range [0,1)
[0,0.8), [0.8,0.82), [0.82,1)
Each interval corresponds to one symbol
Interval size is proportional to symbol probability.



- Map to real line range [0, 1)
- Order does not matter
  - Decoder need to use the same order

Encoding

- Input sequence: "1321"



Encoder Pseudo Code

- Keep track of LOW, HIGH, RANGE
  - Any two are sufficient, e.g., LOW and RANGE.

```
low=0.0, high=1.0;
while (not EOF) {
    n = ReadSymbol();
    RANGE = HIGH - LOW;
    HIGH = LOW + RANGE * CDF(n);
    LOW = LOW + RANGE * CDF(n-1);
}
output LOW;
```

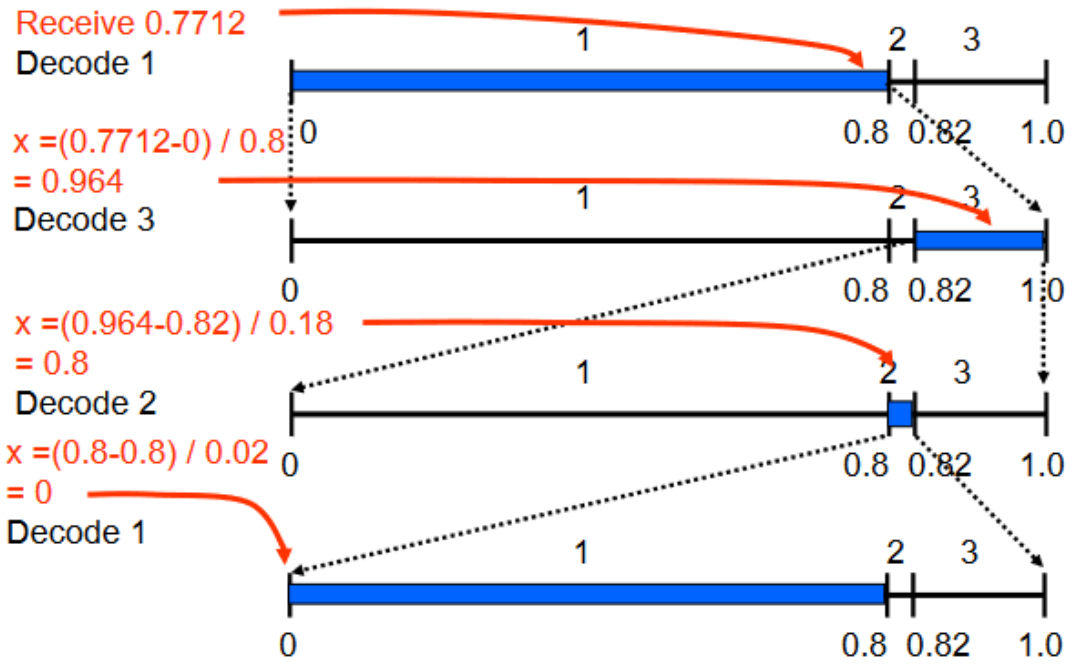
Symbol	Prob.
1	0.8
2	0.02
3	0.18

Input	HIGH	LOW	RANGE
Initial	1.0	0.0	1.0
1	0.0+1.0*0.8=0.8	0.0+1.0*0 = 0.0	0.8
3	0.0 + 0.8*1=0.8	0.0 + 0.8*0.82=0.656	0.144
2	0.656+0.144*0.82=0.77408	0.656+0.144*0.8=0.7712	0.00288
1	0.7712+0.00288*0.8=0.773504	0.7712+0.00288*0=0.7712	0.002304

Simplified Decoding

- Normalize RANGE to [0, 1) each time
- No need to recalculate the thresholds.

$$x \leftarrow \frac{x - low}{range}$$




```
Low = 0; high = 1;
x = Encoded_number
While (x ≠ low) {
    n = DecodeOneSymbol(x);
    output symbol n;
    x = (x - CDF(n-1)) / (CDF(n) - CDF(n-1));
};
```

There are still problems here, we need high precision and no output is generated until the entire sequence is encoded

- Key Observation:
  - As the RANGE reduces, many MSB's of LOW and HIGH become identical:
    - Example: Binary form of 0.7712 and 0.773504:
      0.1100010.., 0.1100011..,
  - We can output identical MSB's and re-scale the rest:
    - ➔ Incremental encoding
  - This also allows us to achieve infinite precision with finite-precision integers.

Comparison with Huffman

Input Symbol 1 does not cause any output
Input Symbol 3 generates 1 bit
Input Symbol 2 generates 5 bits
Symbols with larger probabilities generates less number of bits.
Sometimes no bit is generated at all
→ Advantage over Huffman coding
Large probabilities are desired in arithmetic coding
Can use context-adaptive method to create larger probability
and to improve compression ratio.