CMPN206: Multimedia

Lecture 2: Entropy and Huffman Coding

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Spring 2014

Agenda

- Compression
- Self Information and Entropy
- Coding and Prefix Codes
- Huffman Coding
- Applications

Acknowledgments: Most slides are adapted from Richard Ladner and from Li and Drew.

Compression

The process of coding that will effectively reduce the total number of *bits* needed to represent certain information.

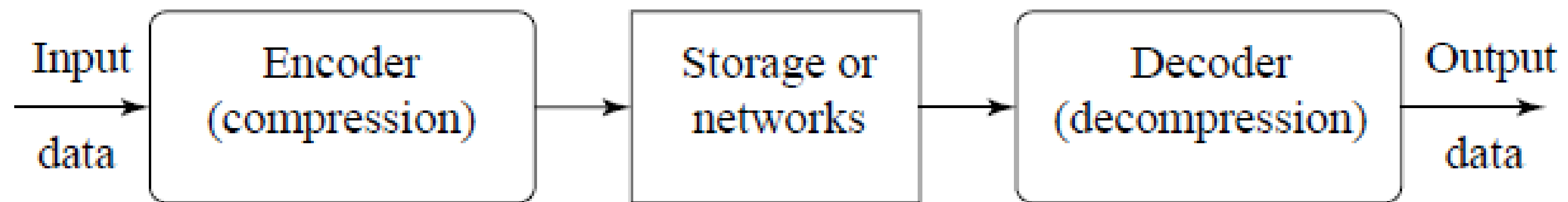


Fig. 7.1: A General Data Compression Scheme.

Lossless vs. Lossy

- Lossless: no information loss in the compression and decompression process

$$\hat{x} = x$$

- Lossy: some information is lost in the compression and decompression process

$$\hat{x} \neq x$$

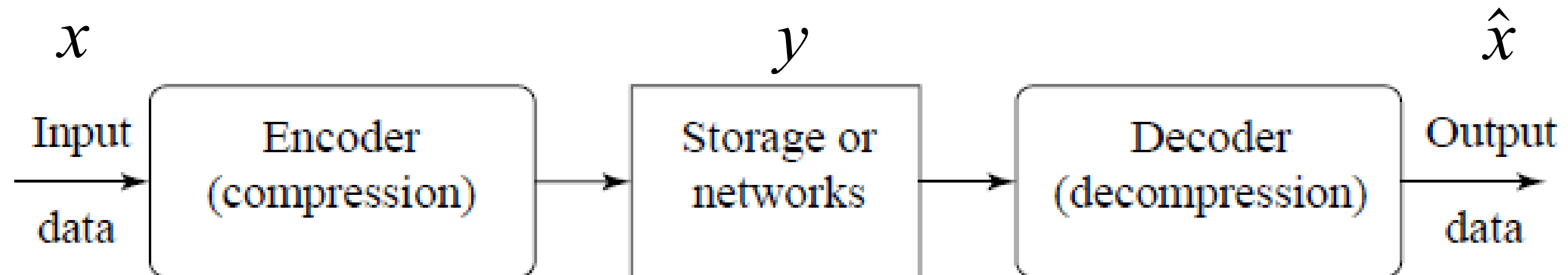


Fig. 7.1: A General Data Compression Scheme.

Compression Ratio

- Compression Ratio: $\frac{|x|}{|y|}$

  $|x|$ is the number of bits *before* compression
  $|y|$ is the number of bits *after* compression

- Will focus first on *lossless* compression.

Why Compression?

- Save storage: less bits to store
- Reduce transmission bandwidth and time
- Progressive transmission: some techniques send the most important information first giving a low resolution version before receiving the full quality version

Lossless Compression

- No data loss: important for
  - text compression
  - binary files
  - databases
- Statistical techniques
  - Huffman coding
  - Arithmetic coding
- Dictionary Based techniques
  - LZW, LZ77
- Available in many standards
  - gzip, bkzip, GIF, Lossless JPEG … etc

How is Compression Possible?

The keyword is: *Redundancy*

- There is more data than there is information
- Squeezing out the excess data results in compression
- Example:
  - The binary string 0000 0001 1111 0000 0000
  - How many bits does it take?
  - How can we compress that?
  - Using the idea of Run Length Coding, for example write how many consecutive zeros and ones we have: 7 5 8 which can be coded as: 111 101 1000
  - Compression ratio?

Digital Data

- Finite set of symbols $S = \{a_1, a_2, \dots, a_m\}$ called the alphabet where individual symbols are called letters
- Data represented as sequences (strings) of letters
- Example
  - $S = \{a, b, c, d\}$
  - sequence = *abbcaabcdddcab*
- Another Example
  - S = set of ASCII symbols
  - sequence = "Hello Information Theory!"

Binary

- $S = \{0, 1\}$
- Sequences are strings of 0's and 1's
- All digital information is represented using binary strings
- We can represent other alphabets using binary strings
- This is called *coding*

Coding

The assignment of *binary* sequences to letters of an alphabet

- The set of binary sequences is called a *code*
- The individual members are called *codewords*

- Example:
  - $S = \{a, b, c, d\}$ has a this fixed-length representation

symbol	a	b	c	d
binary	00	01	10	11

  - 2 bits per symbol
- How many bits per symbol are needed to represent an alphabet of n symbols in a fixed-length binary representation?

$$\lceil \log_2 n \rceil$$

Information Theory

- Developed in the 1940's and 1950's by Claude Shannon
- Attempts to explain the limits of communication, especially the ultimate channel capacity and the ultimate compression rate
- Useful in many other fields e.g. computer science
- Quantifies the notion of *information* (or *surprise*) in a mathematical way

Self Information

- Suppose we have a *source* that emits symbols from an alphabet $S = \{a_1, a_2, \dots, a_m\}$
- Let $P(a_i)$ = the probability of emitting symbol a_i
- The self information for symbol a_i is defined as

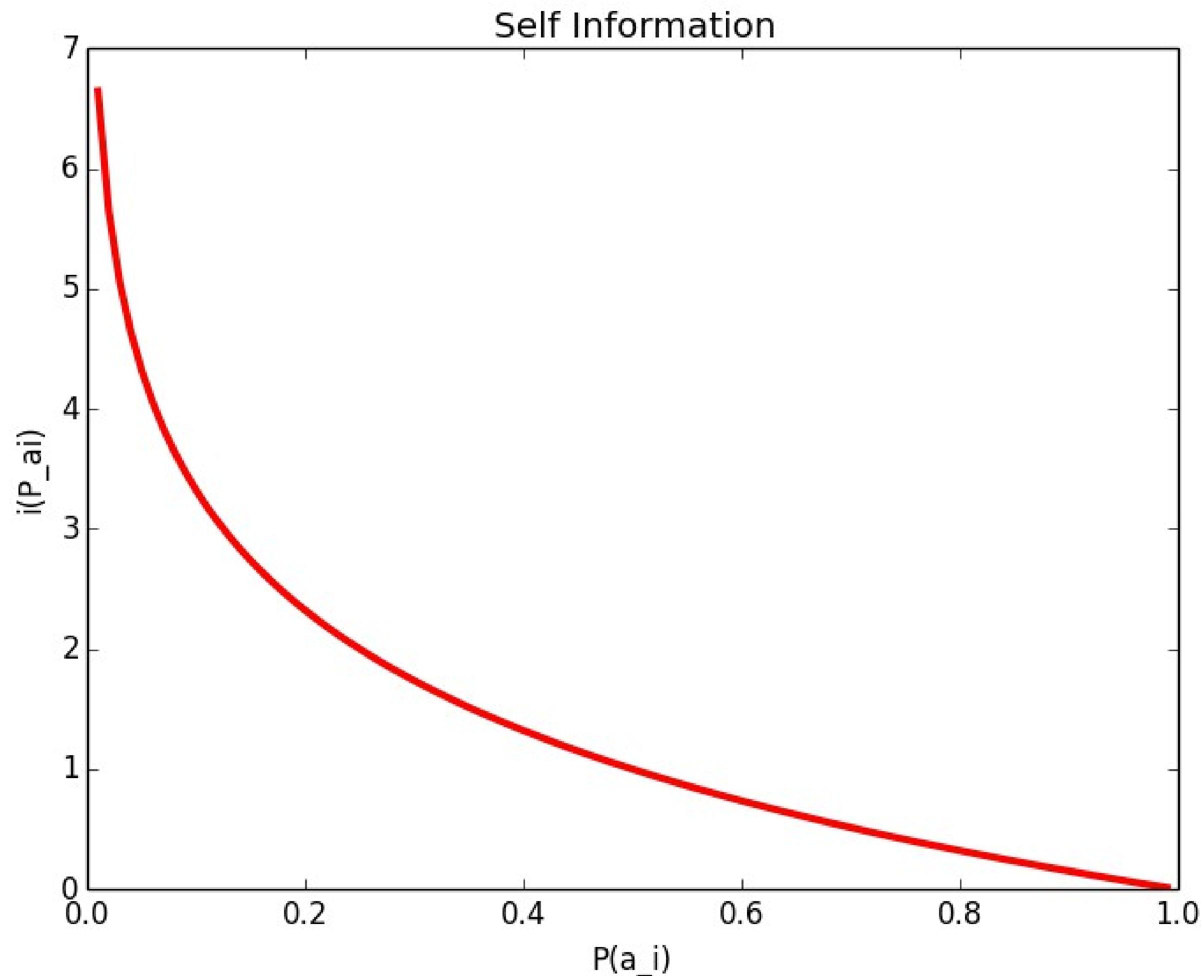
$$i(a_i) = \log_2 \frac{1}{P(a_i)} = -\log_2 P(a_i)$$

and is measured in *bits*.

Self Information

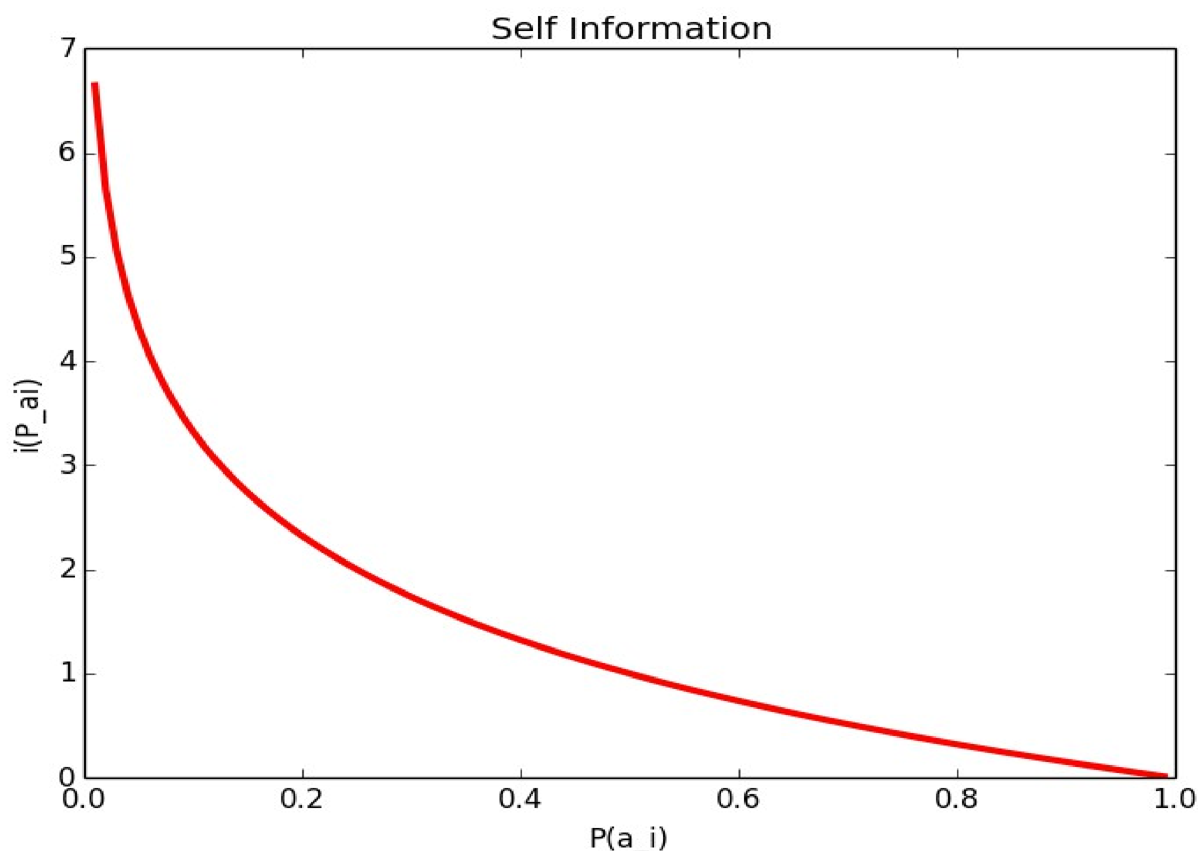
- The self information for symbol a_i is defined as

$$i(a_i) = \log_2 \frac{1}{P(a_i)} = -\log_2 P(a_i)$$



Properties of Self Information

- The information is *low* when $P(a_i)$ is high i.e. frequent symbols have *less* information (less surprise)
- The information is *high* when $P(a_i)$ is low i.e. rare symbols have *more* information (more surprise)



Properties of Self Information

- The self information contained in two independent symbols is the *sum* of their self information

$$
\begin{aligned}
i(a_i a_j) &= -\log_2 P(a_i a_j) \\
&= -\log_2 P(a_i) P(a_j) \\
&= -\log_2 P(a_i) - \log_2 P(a_j) \\
&= i(a_i) + i(a_j)
\end{aligned}
$$

Example

- $S = \{a, b, c\}$ with $P(a) = 1/8$, $P(b) = 1/4$, $P(c) = 5/8$
  - $i(a) = \log_2(8) = 3$
  - $i(b) = \log_2(4) = 2$
  - $i(c) = \log_2(8/5) = .678$
- Receiving a more information than b or c because it's the rarest

$$i(a_i) = \log_2 \frac{1}{P(a_i)} = -\log_2 P(a_i)$$

Entropy

- The entropy of an alphabet $S = \{a_1, a_2, \ldots, a_m\}$ with probabilities $P(a_i)$ is

$$H = \sum_{i=1}^{m} P(a_i) \log_2 \frac{1}{P(a_i)} = -\sum_{i=1}^{m} P(a_i) \log_2 P(a_i)$$

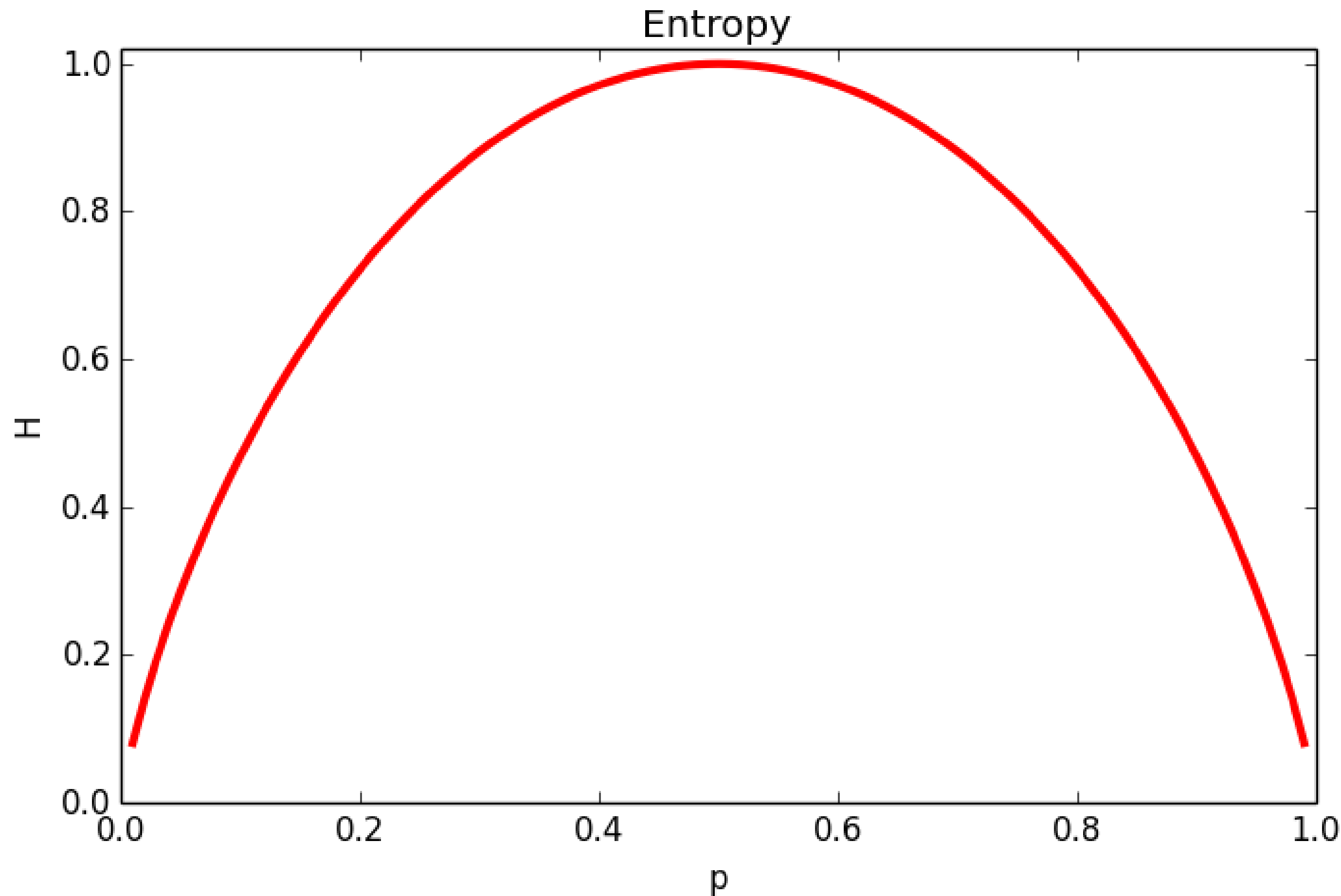
- H represents the average number of bits required to code up the symbols in the alphabet if we know their probabilities
- It represents the lower bound on the average number of bits required to code a symbol in this *source model* i.e. the best a lossless compression algorithm can do!

Example

- $S = \{a, b\}$ with $P(a) = p$, $P(b) = 1 - p$

$$H = -p\log_2(p) - (1-p)\log_2(1-p)$$

- Maximum $H = 1$ at $p = 0.5$ (equally probable) and *zero* at $p = 0$ or 1 (no uncertainty)



More Examples

- $S = \{a, b, c\}$ with $P(a) = 1/8$, $P(b) = 1/4$, $P(c) = 5/8$

$$H = \frac{1}{8} \times 3 + \frac{1}{4} \times 2 + \frac{5}{8} \times 0.678 = 1.3 \text{ bits/symbol}$$

- $S = \{a, b, c\}$ with $P(a) = 1/3$, $P(b) = 1/3$, $P(c) = 1/3$

$$H = 3 \times \frac{1}{3} \times \log_2 3 = 1.6 \text{ bits/symbol}$$

  this is the worst case. Why?

- How many bits per symbol are required in a fixed-length coding?
- 2 bits/symbol

symbol	a	b	c
binary code	00	01	10

Goals of Coding

Represent any alphabet in binary codewords while:

- Minimizing the average number of bits per symbol, called the *rate* of the code
- Allowing the receiver to be able to correctly *decode* the codewords and retrieve the original symbols sent. These codes are called *uniquely decodable codes*

Example

$S = \{a_1, a_2, a_3, a_4\}$ with $P(a_1) = ½$, $P(a_2) = ¼$, $P(a_3) = P(a_4) = 1/8$

- How much is the entropy H?

$$H = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{2}{8}\log_2 8 = 1.75 \text{ bits/symbol}$$

- Consider the codes below:

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length					

- Average Length for a code = $\sum_i P(a_i) l_i$

where l_i is the length of the codeword for letter a_i

Example

$$H = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{2}{8}\log_2 8 = 1.75 \text{ bits/symbol}$$

- Code 1
  - Is it uniquely decodable?
    - No. Why?
    - Two symbols have the same codeword
  - What's the average length? $= 0.5 + 0.25 + 0.125 + 0.125 \times 2 = 1.125$

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length					

Example

$$H = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{2}{8}\log_2 8 = 1.75 \text{ bits/symbol}$$

- Code 2
  - Is it uniquely decodable?
    - No. Why?
    - Though no two codewords are the same, but consider the sequence 100, it can be decoded as either $a_2 a_3$ or $a_2 a_1 a_1$

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length		1.125	1.25		

Example

$$H = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{2}{8}\log_2 8 = 1.75 \text{ bits/symbol}$$

- Code 3
  - Is it uniquely decodable?
    - Yes!
  - Each codeword ends with a 0 except the last one.
  - Decoding: accumulate bits until you see a 0 or you see three 1's

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length		1.125	1.25	1.75	

Example

$$H = \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{2}{8}\log_2 8 = 1.75 \text{ bits/symbol}$$

- Code 4
  - Is it uniquely decodable?
    - Yes!
  - Each codeword starts with a 0 except the last one.
  - Decoding: accumulate bits until you see a 0. The 0 starts the new codeword.

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length		1.125	1.25	1.75	1.875

Instantaneous Codes

A code where the decoder knows the moment a codeword is complete without having to examine bits from the *next* codeword.

- Which of Codes 3 and 4 is *instantaneous*?
- Code 3, because in Code 4 the decoder has to see the 0 of the following codeword to know it finished the current codeword. Code 4 is a *near*-instantaneous code.

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length		1.125	1.25	1.75	1.875

Prefix Codes

A code where no codeword is a *prefix* of another.

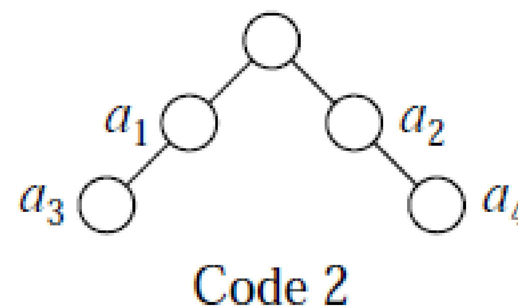
- Which of Codes 3 and 4 is a *prefix code*?
- Code 3, because in Code 4 we have codeword 0 a prefix of all the other codewords …
- Prefix Codes are guaranteed to be *uniquely decodable*

Letters	Prob.	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average Length		1.125	1.25	1.75	1.875

Prefix Codes

- How to check if a given code is a *prefix code*?
- Draw the rooted binary tree corresponding to the code.
- From each node draw a *left* branch if the bit is a 0 and a *right* branch if a bit is a 1.
- Mark any node with the codeword it represents
- If all the codewords are at the *leaves*, then it's a *prefix code*

Letters	Code 2	Code 3	Code 4
a_1	0	0	0
a_2	1	10	01
a_3	00	110	011
a_4	11	111	0111

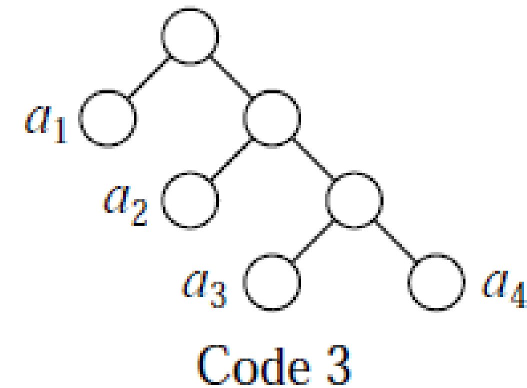


not a prefix code

Prefix Codes

- How to check if a given code is a *prefix code*?
- Draw the rooted binary tree corresponding to the code.
- From each node draw a *left* branch if the bit is a 0 and a *right* branch if a bit is a 1.
- Mark any node with the codeword it represents
- If all the codewords are at the *leaves*, then it's a *prefix code*

Letters	Code 2	Code 3	Code 4
a_1	0	0	0
a_2	1	10	01
a_3	00	110	011
a_4	11	111	0111

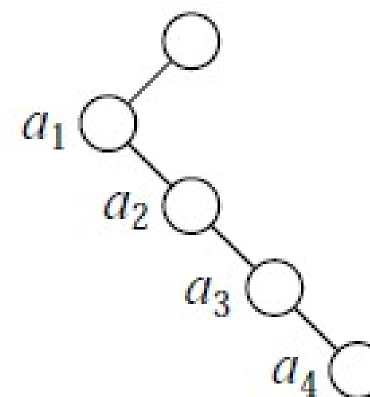


is a prefix code

Prefix Codes

- How to check if a given code is a *prefix code*?
- Draw the rooted binary tree corresponding to the code.
- From each node draw a *left* branch if the bit is a 0 and a *right* branch if a bit is a 1.
- Mark any node with the codeword it represents
- If all the codewords are at the *leaves*, then it's a *prefix code*

Letters	Code 2	Code 3	Code 4
a_1	0	0	0
a_2	1	10	01
a_3	00	110	011
a_4	11	111	0111



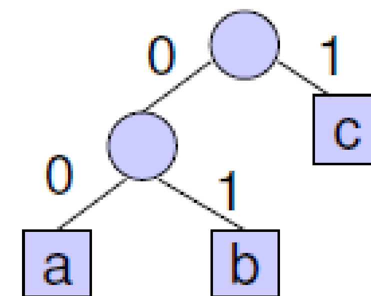
not a prefix code

Encoding Prefix Codes

```
repeat:
   if there is input
   report codeword for input symbol
until end of code
```

Input: ccab

Output:

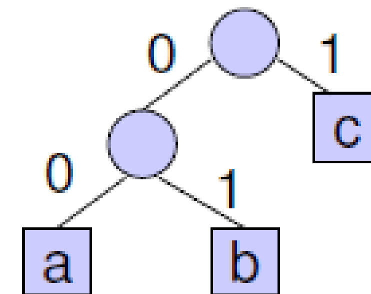


Encoding Prefix Codes

```
repeat:
   if there is input
   report codeword for input symbol
until end of the code
```

Input: ccab

Output: 1

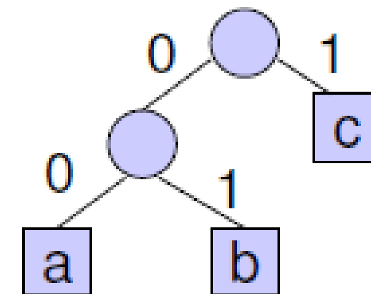


Encoding Prefix Codes

```
repeat:
   if there is input
   report codeword for input symbol
until end of code
```

Input: ccab

Output: 11

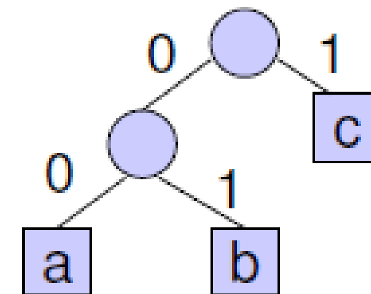


Encoding Prefix Codes

```
repeat:
   if there is input
   report codeword for input symbol
until end of the code
```

Input: ccab

Output: 1100

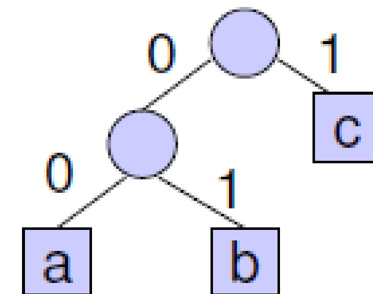


Encoding Prefix Codes

```
repeat:
   if there is input
   report codeword for input symbol
until end of the code
```

Input: ccab

Output: 110001

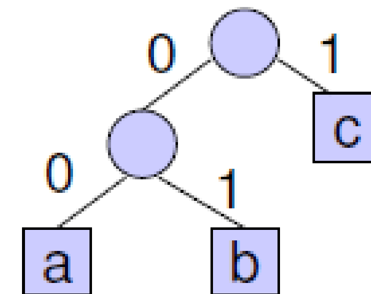


Encoding Prefix Codes

```
repeat:
   if there is input
   report codeword for input symbol
until end of the code
```

Input: ccabccca

Output: 110001111100

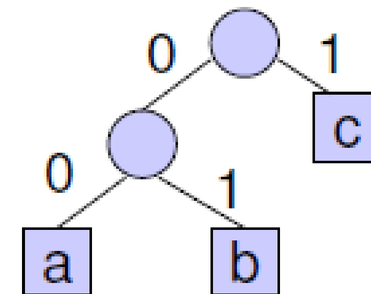


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output:

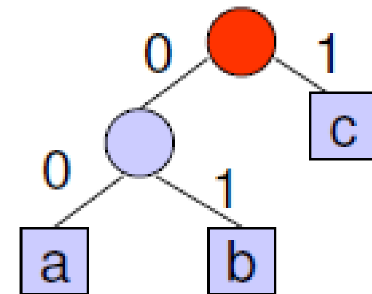


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output:

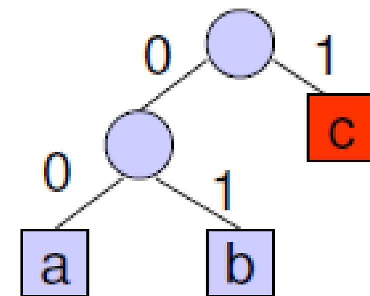


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: c

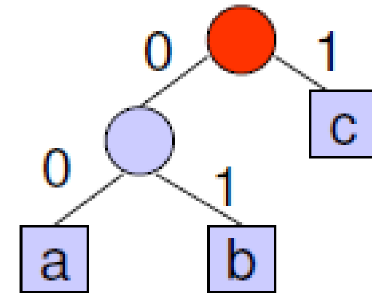


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

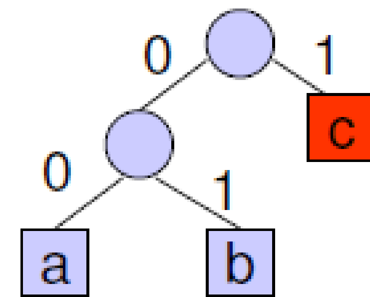
Input: 11000111100

Output: c



Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

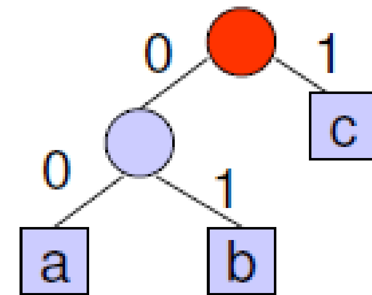


Input: 1100011100

Output: cc

Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```



Input: 11000111100

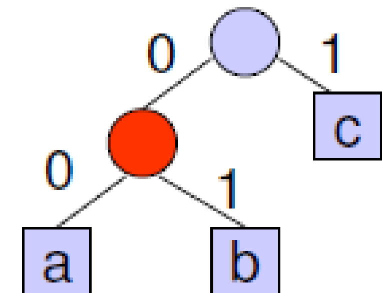
Output: cc

Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: cc

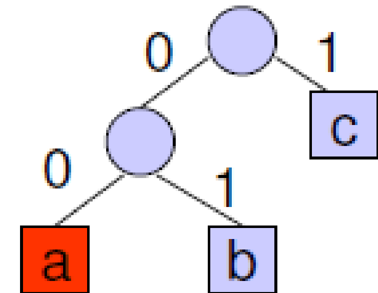


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: cca

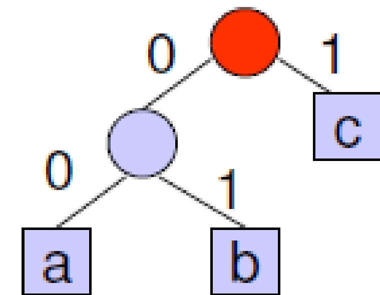


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: cca

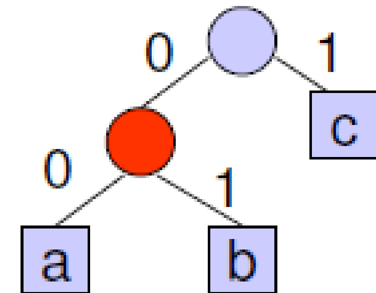


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: cca

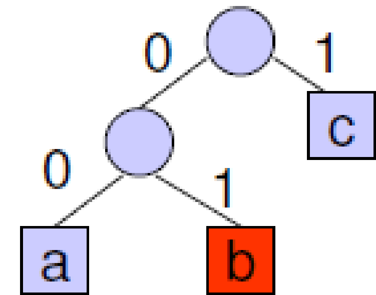


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: ccab

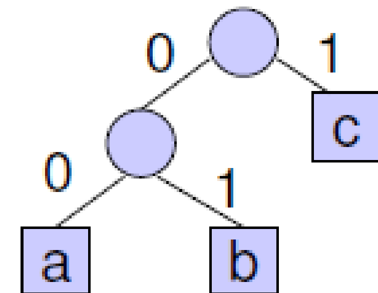


Decoding Prefix Codes

```
repeat:
   start at root of tree
   repeat:
      if read bit = 1 then go right
      else go left
   until node is a leaf
   report leaf
until end of the code
```

Input: 11000111100

Output: ccabccca



Kraft McMillan Inequality

- Prefix codes are guaranteed to be uniquely decodable
- But, do we lose performance (i.e. having the shortest average codeword) by limiting ourselves to prefix codes?
- The answer is: No!
- This is proved using the Kraft-McMillan Inequality:

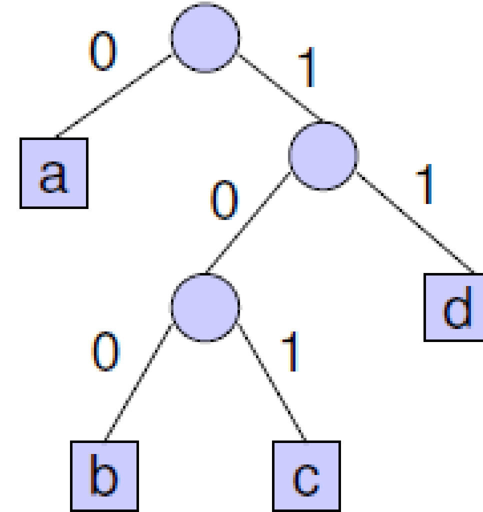
$$\sum_{i=1}^{N} 2^{-l_i} \leqslant 1$$

where l_i is the length of codeword i from N codewords.

- Proof: Section 2.4.3 [**IDC**].

Exercise

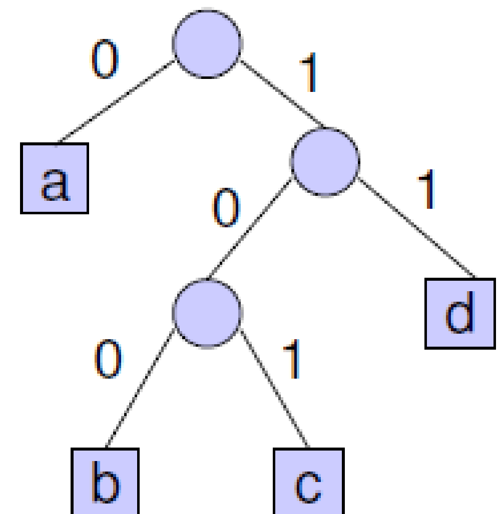
- Encode a symbol string
- Decode the symbol string
- Make sure they match!



Huffman Coding

- Huffman in 1951
- Uses *probabilities* of symbols (or their *frequencies*) to build a *variable length prefix code*
  - Codewords have different lengths
  - Each symbol maps to a unique codeword
  - More frequent symbols have shorter codewords
  - No codeword is a prefix of another

a	0
b	100
c	101
d	11



Cost of a Huffman Tree

- Let $p_1, p_1, \ldots p_m$ be the probabilities for symbols $a_1, a_1, \ldots a_m$
- The cost of a Huffman Tree T is defined as:

$$C(T) = \sum_{i=1}^{m} p_i l_i$$

  where l_i is the length of the path from the root to a_i i.e. the length of the codeword for symbol a_i

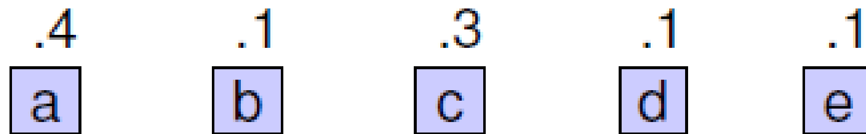
- $C(T)$ represents the *expected* (average) length of a codeword, and is the *bit rate* of the code
- Huffman Trees are *optimal* in the sense that they have the *minimum* cost among all trees

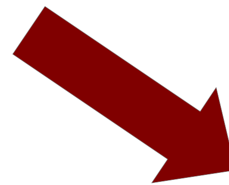
Huffman Coding

```
Form a node for each symbol a_i with probability p_i
Insert the nodes into a priority queue Q
while there nodes in Q
    min1 = extract_min(Q)
    min2 = extract_min(Q)
    create a new node n

    // make the new node their parent
    n.left = min1
    n.right = min2

    // set its probability to the sum of their prob.
    n.p = min1.p + min2.p
    insert(Q, n)
return extract_min(Q)
```

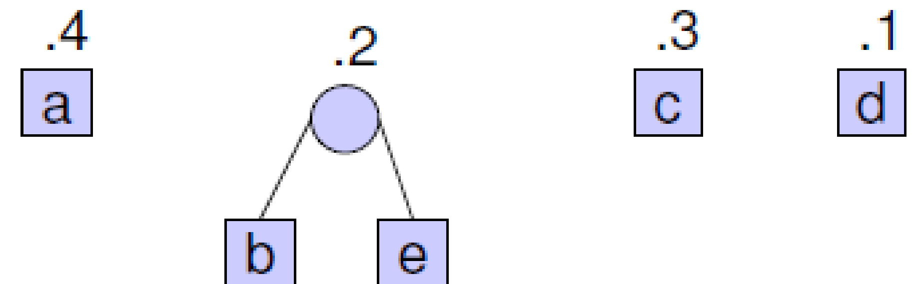
Example

Symbol	*a*	*b*	*c*	*d*	*e*
Prob.	0.4	0.1	0.3	0.1	0.1



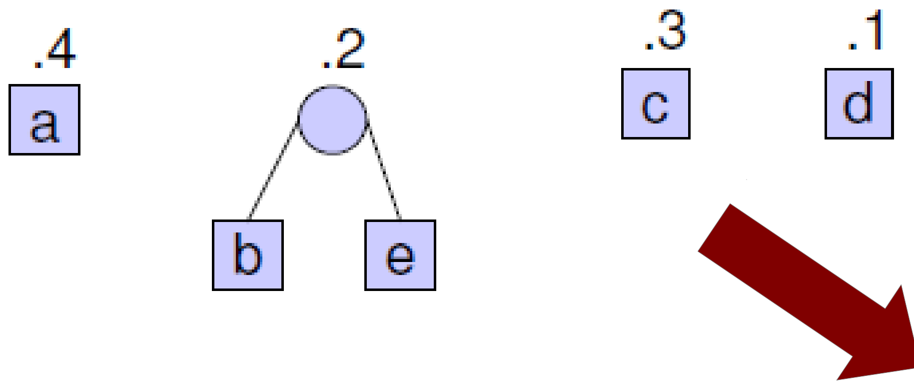


Combine *b* and *e*
into a new symbol
with prob. = 0.2

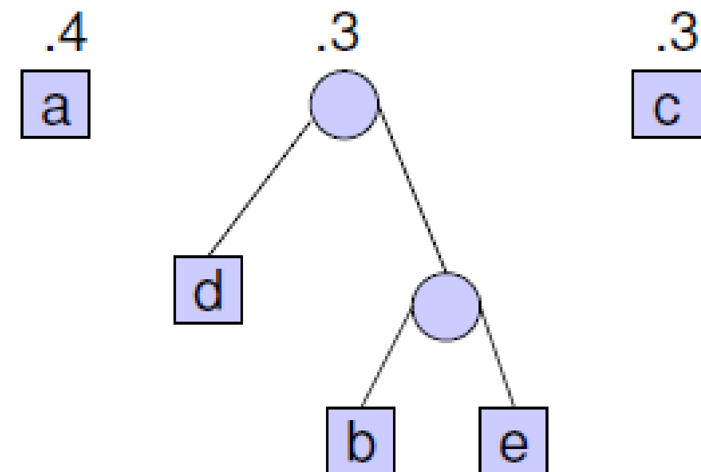


Example

Symbol	*a*	*b*	*c*	*d*	*e*
Prob.	0.4	0.1	0.3	0.1	0.1

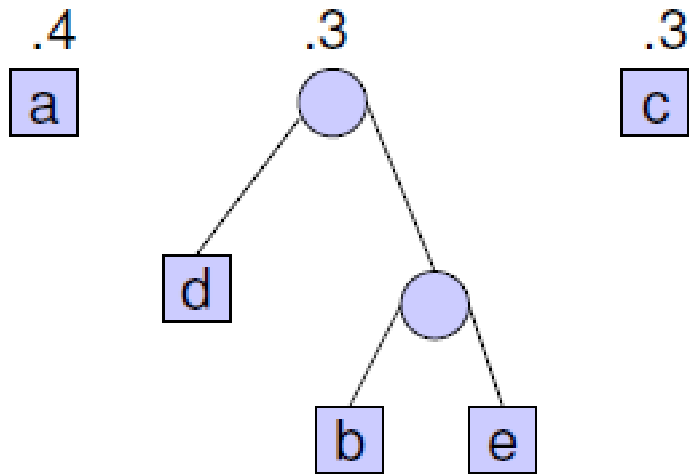


Combine (*b, e)* and *d* into a new symbol with prob. = 0.3

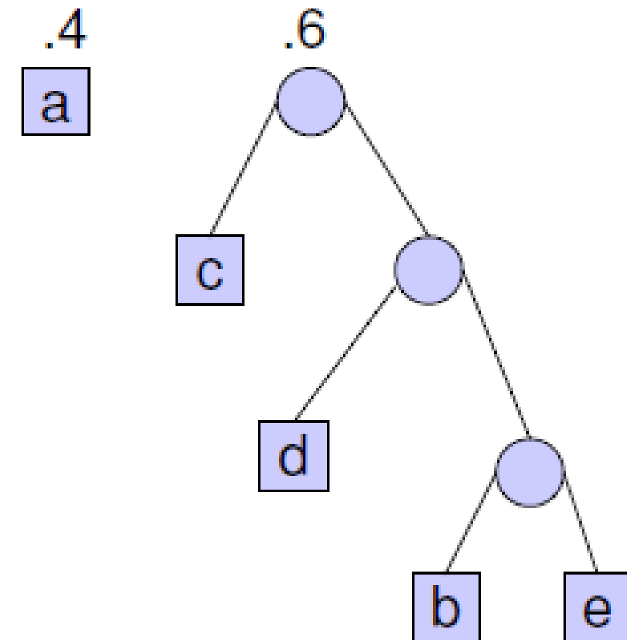


Example

Symbol	*a*	*b*	*c*	*d*	*e*
Prob.	0.4	0.1	0.3	0.1	0.1

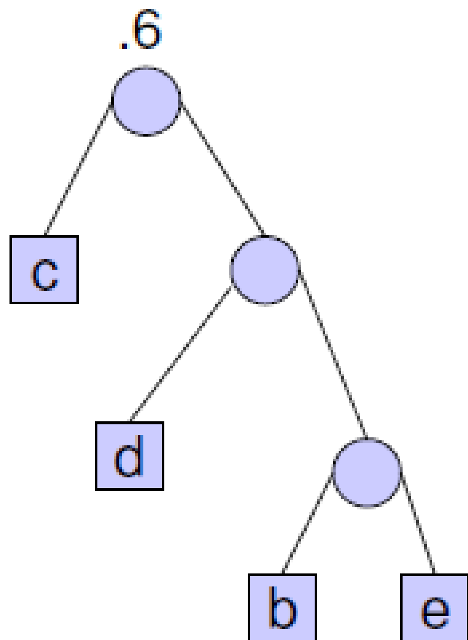


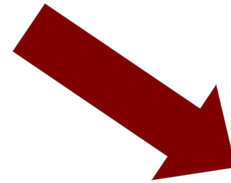
Combine (*d*, *b, e)* and *e*
into a new symbol
with prob. = 0.6

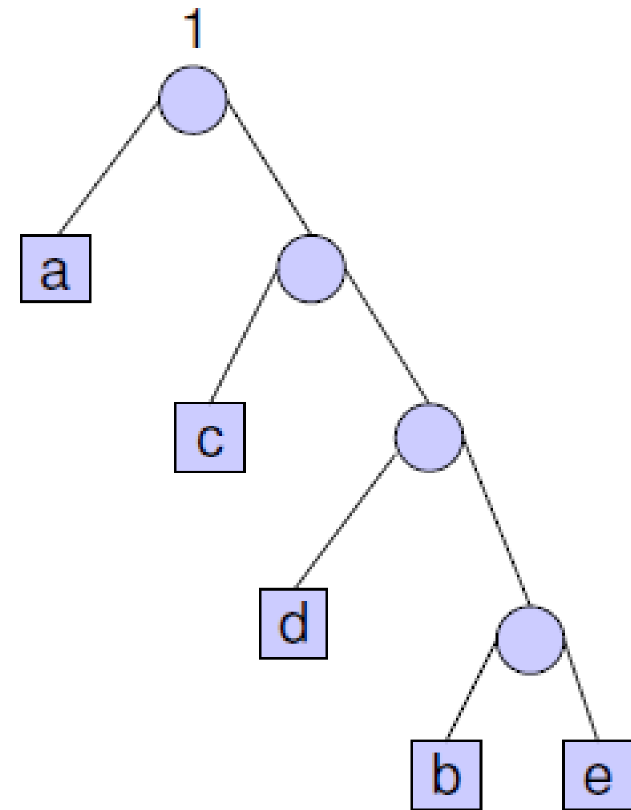


Example

Symbol	*a*	*b*	*c*	*d*	*e*
Prob.	0.4	0.1	0.3	0.1	0.1







Combine (*c, d*, *b, e)* and *a*
into a new symbol
with prob. = 1 and return the
root of the tree

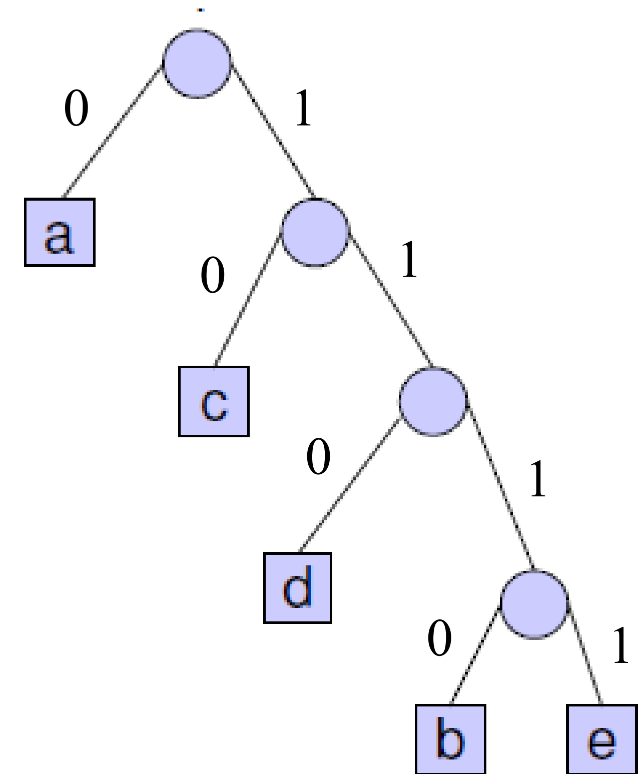
Example

Symbol	a	b	c	d	e
Prob.	0.4	0.1	0.3	0.1	0.1
Code	0	1110	10	110	1111

Cost of the tree:

$$C(T) = \sum_i P(a_i) l_i$$

$$= 0.4 \times 1 + 0.1 \times 4 + 0.3 \times 2 + 0.1 \times 3 + 0.1 \times 4$$
$$= 2.1 \text{ bits/symbol}$$



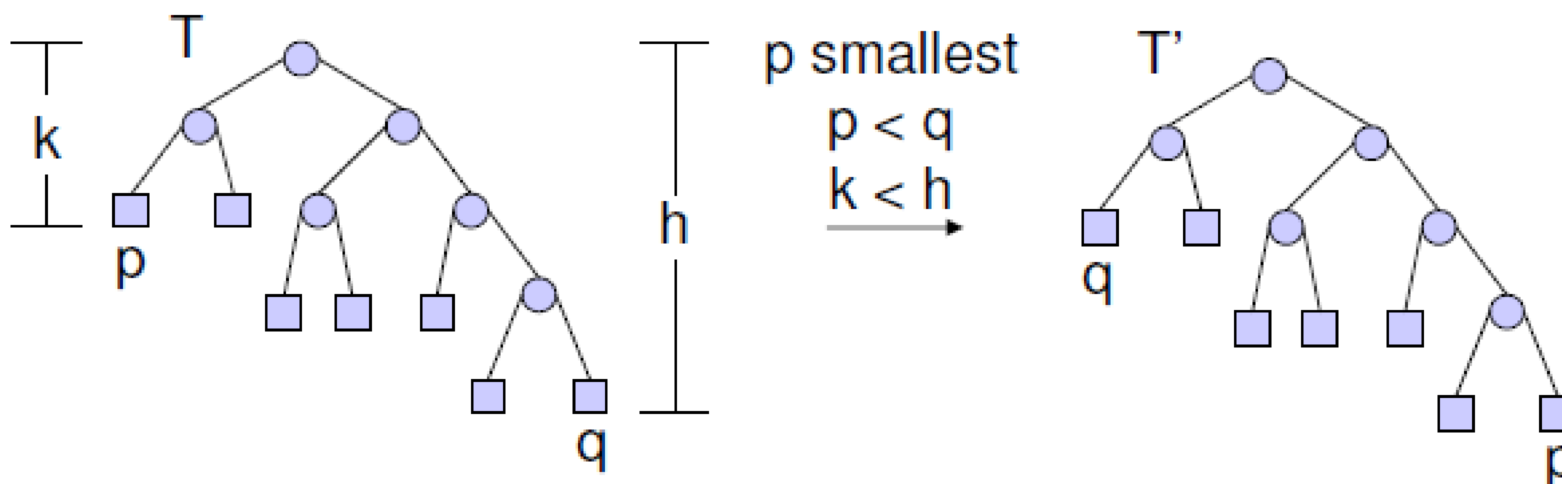
Entropy:

$$H = \sum_i P(a_i) \log_2 \frac{a}{P(a_i)}$$
$$= 2.05 \text{ bits/symbol}$$

Proof of Optimality I

In an optimal tree, the lowest probability symbol has the longest codeword i.e. the maximum distance from the root.

Proof:
If that's not the case, exchanging the lowest probability symbol with the one at maximum distance from the root will lower the cost of the tree. Contradiction.

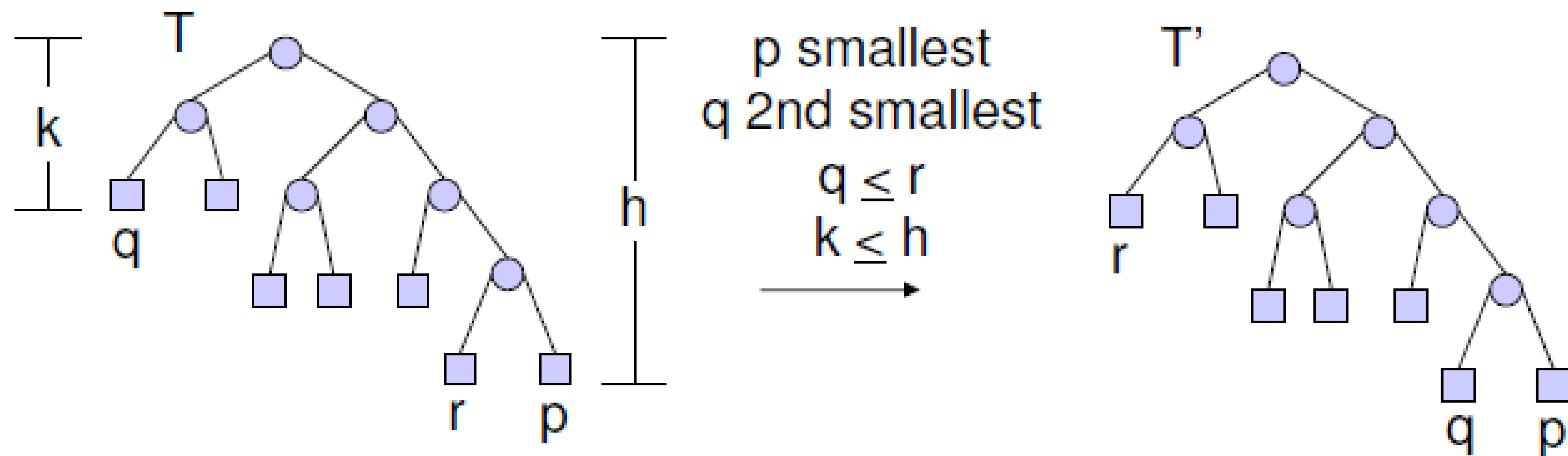


$$
\begin{aligned}
C(T') &= C(T) - (pk + qh) + (qk + ph) \\
&= C(T) - (q-p)(h-k) \\
&< C(T)
\end{aligned}
$$

Proof of Optimality II

In some optimal tree, the second lowest probability symbol is a sibling of the lowest probability symbol.

Proof:
If that's not the case, exchanging the second lowest probability symbol with the sibling of the lowest probability one will not raise the cost.

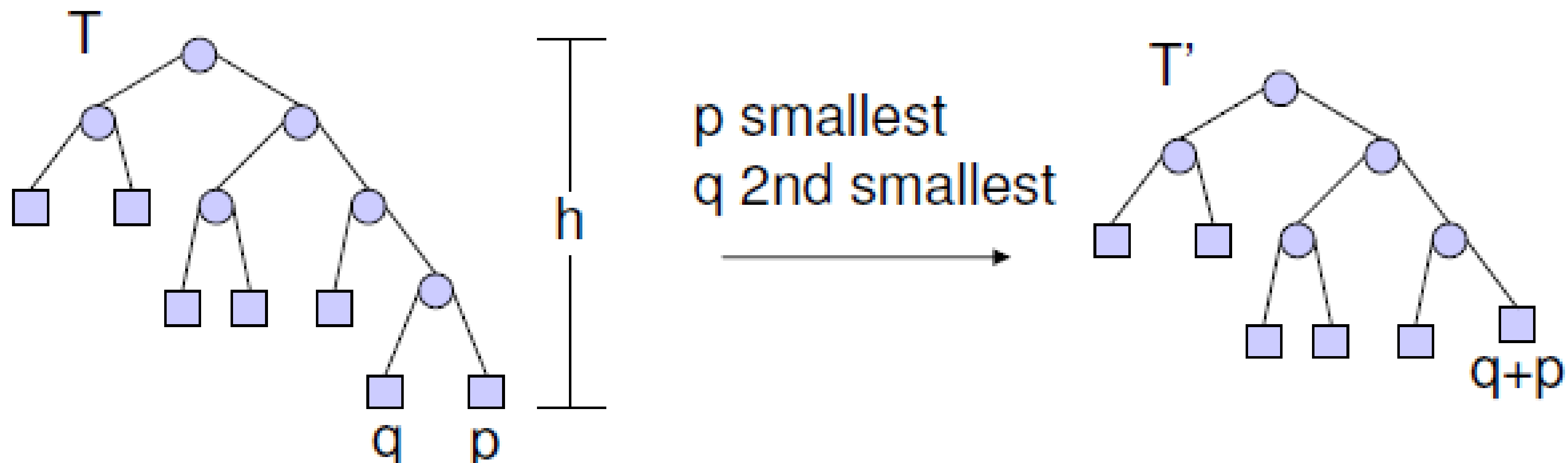


$$C(T') = C(T) - (qk + rh) + (qh + rk)$$
$$= C(T) - (r - q)(h - k)$$
$$\leqslant C(T)$$

Proof of Optimality III

In an optimal tree, if we replace the two lowest probability symbols by a node whose probability is the sum of their probabilities, the new tree T' is optimal for the reduced alphabet.

Proof:
If that's not the case, we can find another lower cost tree T'' which can lead to a tree T''' with lower cost than T.
Contradiction.



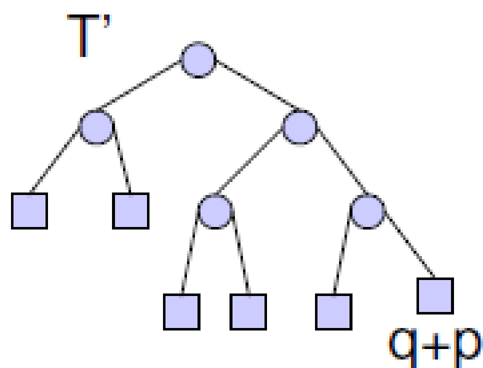
$$C(T') = C(T) - (qh + ph) + (q+p)(h-1) = C(T) - (p+q)$$

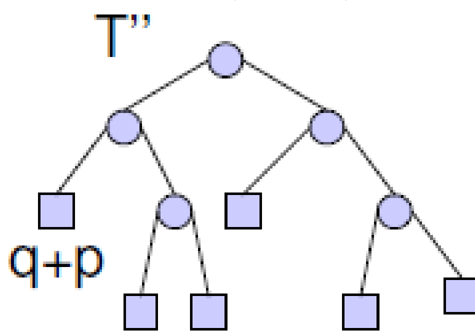
Proof of Optimality III

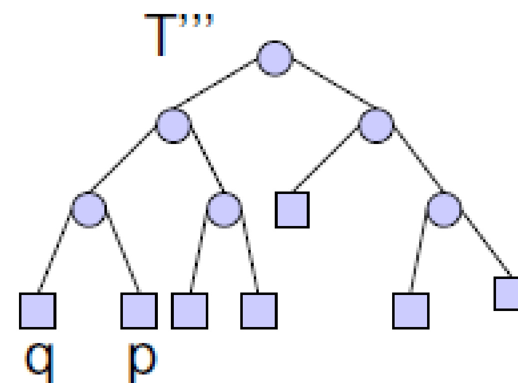
In an optimal tree, if we replace the two lowest probability symbols by a node whose probability is the sum of their probabilities, the new tree T' is optimal for the reduced alphabet.

Proof:
If that's not the case, we can find another lower cost tree T'' which can lead to a tree T''' with lower cost than T.
Contradiction.

$$C(T'') < C(T')$$







$$C(T''') = C(T'') + (p+q) < C(T') + (p+q) = C(T)$$

$$C(T') = C(T) - (qh + ph) + (q+p)(h-1) = C(T) - (p+q)$$

Quality of Huffman Coding

The *rate* of the Huffman Coding is within 1 bit of the entropy for the source alphabet i.e.

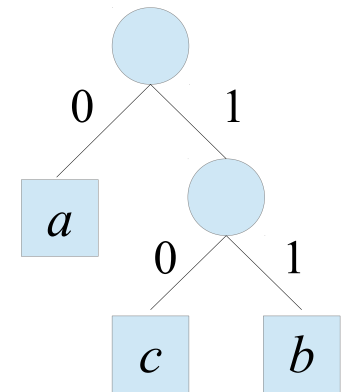
$$H \leqslant C(T) \leqslant H + 1$$

where H is the entropy and $C(T)$ is the cost of the Huffman Tree (or the expected length of the codeword)

Limitations

- Very efficient encoding/decoding
- Optimal codes if the probabilities are known
- Does not work well with smaller alphabets or when the probability distribution is *skewed*
- Example: Consider the following alphabet

Symbol	a	b	c
Probability	0.8	0.02	0.18
Huffman Code	0	11	10



  - $H = 0.816$ bits/symbol
  - Huffman's rate = 1.2 bits/symbol
  - An increase of 47% of the entropy!

Extended Huffman Coding

- Instead of coding the alphabet symbols individually, create a new *extended* alphabet by grouping together symbols from the original alphabet, and run Huffman coding on that.
- If we group together n symbols, the bounds for the Huffman Coding on the new alphabet become

$$H \leqslant C(T) \leqslant H + \frac{1}{n}$$

  i.e. by making the blocks bigger we can decrease the rate more
- Drawback: grouping n symbols increases the size of the alphabet *exponentially*

Example

- Extend this alphabet by grouping *two* symbols together, and assume they are independent when computing their probability

Symbol	Prob.
a	0.8
b	0.02
c	0.18

$$P(ab)=P(a)P(b)$$

Symbol	Prob.	Code
aa	0.64	0
ab	0.016	10101
ac	0.144	11
ba	0.016	101000
bb	0.0004	10100101
bc	0.0036	1010011
ca	0.1440	100
cb	0.0036	10100100
cc	0.0324	1011

- $H = 0.816$ bits/symbol
- Rate = 1.7228 bits/symbol in the extended alphabet
- Each such symbol contains two original symbols
- Actual rate = 0.8614 bits/symbol < 1.2 obtained previously!

Alphabet Probabilities

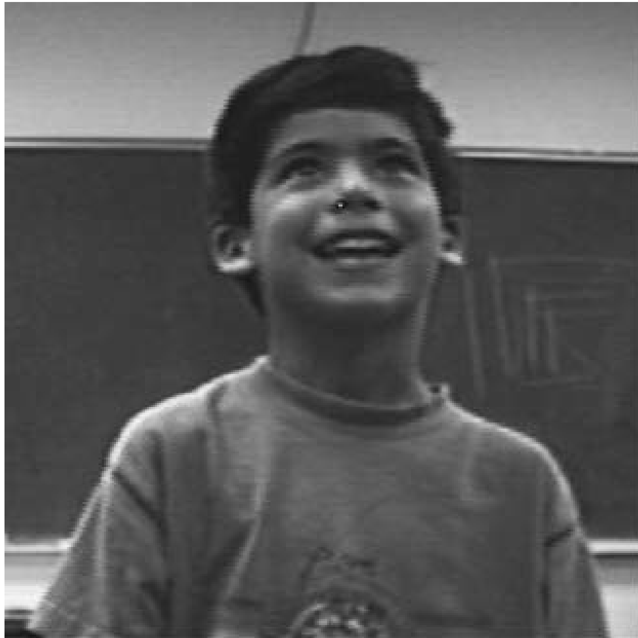
- If we know the alphabet probabilities beforehand, we can compute the Huffman Tree *once* and use it at the transmitter and receiver
- If we don't know the probabilities:
  - We can *estimate* them from the *frequency* of occurrence of each symbol. In that case, we need to:
    - transmit the Huffman Tree computed at the transmitter, or
    - perform two passes at the receiver, first to compute the frequencies then compute the Huffman Tree
  - Use Adaptive Huffman Coding, which performs one pass over the data and updates the tree as it scans the input

Applications

- Huffman Coding is used in many applications:
  - Image compression
  - Text compression
  - Audio compression
- Usually used as a building block in compression standards e.g. lossless JPEG

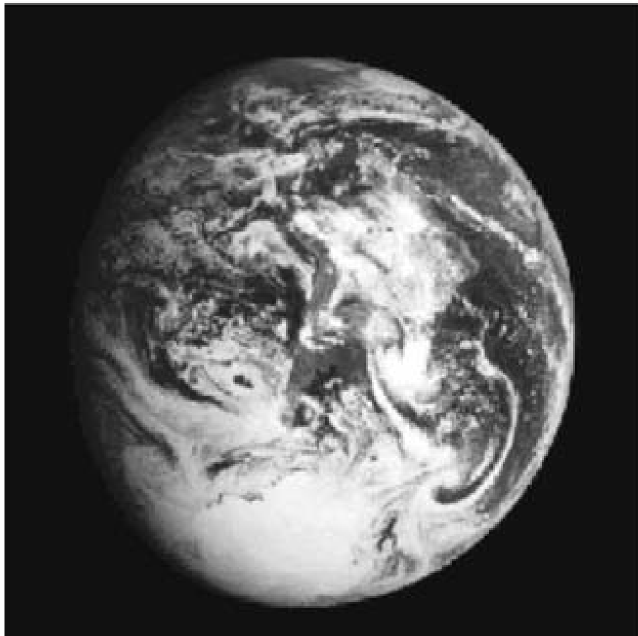
Example

Sena



Sensin

Earth



Omaha

Example

- Huffman Coding of the pixel values (statistics computed from the image)

TABLE 3.23 Compression using Huffman codes on pixel values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

- We can use the *correlation* between neighboring pixels by encoding some *difference* image e.g. encode the difference between a pixel and its left neighbor:

$$\hat{I}(i,j)=I(i,j-1)$$

- The *receiver* can then reconstruct the image by adding the decoded value to its left neighbor

Example

- Huffman Coding of the pixel values:

TABLE 3.23 **Compression using Huffman codes on pixel values.**

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

- Huffman Coding of the *residual* values:

TABLE 3.24 **Compression using Huffman codes on pixel difference values.**

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

Recap

- Compression
- Self Information and Entropy
- Coding and Prefix Codes
- Huffman Coding
- Applications
- Next:
  - Run Length Coding
  - Arithmetic Coding
  - Dictionary Techniques
- More information: Chapters 2 and 3 [**IDC**]