

Coding Techniques

- Variable-sized codes
- Shannon-Fano coding
- Huffman coding
- Tunstall codes
- Arithmetic coding
- Coding increasing integer sequences

Variable-sized Codes (not fixed)

- can match code to model much more effectively
- encoder needs to transmit code to decoder
- model supplies probability for each character (or string)

Memory Codes

- entropy bounds the compression rate of memoryless codes
- it is possible to improve the compression rate with codes that use memory
- types of memory codes
 - **block coding**
group source elements, k at a time
 - **finite-memory codes**
use the previous k elements to determine which of k memoryless codes is used next
 - **finite-state codes**
combines block and finite-memory coding
use k -blocks and a set of encoder states

Prefix Sets

- for a set S of binary strings,
 - a *conservative* expansion $S \rightarrow S'$
replaces $w \in S$ with **both** $w0$ and $w1$
 - a *liberal* expansion $S \rightarrow S'$
replaces $w \in S$ with **either** $w0$ or $w1$
- a *prefix set* is a set of binary strings such that no element is a prefix of any other element
 - the *trivial* prefix set is $S_0 = \{\lambda\}$
 - all other prefix sets are termed *nontrivial*
- any nontrivial prefix set S can be obtained from the trivial prefix set S_0 by a sequence of expansions $S_0 \rightarrow \dots \rightarrow S$

Kraft vectors

- a **Kraft** vector (L_1, \dots, L_n) satisfies $\sum_{i=1}^n 2^{-L_i} \leq 1$
 - for prefix set $S = \{w_1, \dots, w_n\}$,
define $v(S) = (L_1, \dots, L_n)$, where $L_i = \text{length}(w_i)$
 - for any prefix set S , $v(S)$ is a Kraft vector
- Proof** by induction
- true for the trivial prefix set, S_0
 - if true for S then true for S' , where $S \rightarrow S'$
- a Kraft vector is **proper** if the sum = 1
improper if sum < 1

$v(S)$ is proper iff $S_0 \rightarrow \dots \rightarrow S$
uses only conservative expansions

Compact Codes

- a prefix code is **compact** if its Kraft vector is proper
- the **canonical** Kraft vector is in sorted order

Example: $\{0, 1000, 1001, 11\}$

has canonical Kraft vector $(1,2,4,4)$

but it is not a compact code

if the most frequent source el'ts are assigned to the shortest codewords,
then codes having the same canonical Kraft vector are equivalent
so, refer to *the* $(1,2,3,3)$ compact code (**CC**)

- only one CC of size 2: $v = (1,1)$
- only one CC of size 3: $(1,2,2)$
- two of size 4: $(1,2,3,3)$ & $(2,2,2,2)$
- 3 size-5, 5 size-6, 9 size-7, 16 size-8, ..., 565168 size-26
- asympt. $(0.141853) * 1.794147^k$ size- k

see: On-line Encyclopedia of Integer Sequences

<http://www.research.att.com/~njas/sequences>

Compact Codes

- can obtain a size- k CC by a
 - conservative expansion of a size- $(k - 1)$ CC
 - implement a conservative expansion by
 - replacing a length- a code with two codes of length $a + 1$

Example: $(0) \rightarrow (1, 1) \rightarrow (1, 2, 2) \rightarrow (1, 2, 3, 3) \rightarrow (2, 2, 2, 3, 3)$

- may be many expansion sequences between (0) and a particular size- k CC
 - \exists exactly one expansion sequence in which each expansion $S \rightarrow S'$ is either
 - an expansion of the longest codelength L , or
 - an expansion of the codelength $L - 1$ in S
- Example:** $(0) \rightarrow (1, 1) \rightarrow (1, 2, 2) \rightarrow (2, 2, 2, 2) \rightarrow (2, 2, 2, 3, 3)$

Kraft-MacMillan Inequality

- If an unambiguous variable-size code C over an alphabet of size q has n codewords of sizes L_i ,
then

$$K(C) = \sum_{i=1}^n q^{-L_i} \leq 1 \quad (**)$$

- If n positive integers $\{L_1, L_2, \dots\}$ satisfy (**),
then \exists a prefix-free code such that
the L_i are the sizes of its individual codes

- Proofs

- (1949) Kraft: prefix codes satisfy (**)
 - (1956) MacMillan: uniquely decodable codes
 - (1961) Karush: simplified proof
- Codeword set C is complete iff C is prefix-free and $K(C) = 1$

Kraft-MacMillan Inequality – Proof

A q -ary prefix code exists with wordlength $\{L_i\}$

iff $\sum_{i=1}^n q^{-L_i} \leq 1$ (**)

- can embed prefix code in a q -ary tree with leaves at depths $\{L_i\}$
- a leaf at depth L_i prunes q^{L-L_i} nodes from depth L
- the sum of the nodes pruned by all leaves cannot exceed the # of nodes at depth L
- therefore: uniquely decodable \Rightarrow (**)
- can choose nodes becoming leaves and show that sufficient nodes remain to prove the converse

Universal Compression

Theorem (*Impossibility of Universal Compression*)

For any given integer $n \geq 0$,
no program can compress without loss **all** strings of size $\geq n$ bits,

Proof

- Assume P can compress without loss all strings of size $\geq n$ bits.
- Using P , compress all the 2^n strings that have exactly n bits, so that each resulting compressed string has at most $n - 1$ bits.
- But, there are only $2^n - 1$ different strings having lengths in the range 1 to $n - 1$ and, therefore, at least two strings compress to the same string S .
- Decompressing S can recover at most one of those two strings, so P is not lossless.

Shannon's Noiseless-Coding Theorem

- For a random source emitting characters with probabilities $\{p_i\}$, the minimal expected wordlength, E , for an instantaneous code in an alphabet of $c \geq 2$ symbols satisfies

$$\frac{H}{\lg c} \leq E \leq \frac{H}{\lg c} + 1$$

- The lower bound is attained **iff** each $p_i = c^{-l_i}$, for integers l_i

Entropy Bound – Proof

For the binary case (which does generalize),

$$H = -\sum_i p_i \lg p_i \quad (1)$$

$$E = \sum_i p_i L_i \quad (2)$$

subject to Kraft,

$$\sum_i 2^{-L_i} \leq 1 \quad (3)$$

The curve $y = \ln x$ is strictly convex,

and lies strictly below $y = x - 1$, except at $x = 1$.

$$\ln x \leq x - 1 \quad (4)$$

Let $x = 2^{-L_i}/p_i$,

$$\ln(2^{-L_i}/p_i) \leq (2^{-L_i}/p_i) - 1 \quad (5)$$

Multiply by p_i and sum

$$\sum_i p_i \ln(2^{-L_i}/p_i) \leq \sum_i 2^{-L_i} - \sum_i p_i \leq 0 \quad (6)$$

Multiply to change to \lg , and expand $\lg(x/y)$

$$\sum_i p_i L_i \geq \sum_i -p_i \lg p_i \Rightarrow E \geq H \quad (7)$$

Entropy Bound – Proof

Finally, consider a **Shannon code** (discussed next) with

$$L'_i = \lceil -\lg p_i \rceil \quad (8)$$

This satisfies Kraft, because $\sum_i 2^{-\lg p_i} = \sum_i p_i \leq 1$

and we now use even larger lengths: $\lceil x \rceil \geq x$

Since we assumed that E was minimum, we get

$$E' \geq E \quad (9)$$

Then,

$$H \leq E \leq E' = \sum_i p_i \lceil -\lg p_i \rceil < \sum_i p_i (1 - \lg p_i) = H + 1$$

Shannon Coding

Shannon's theorem leads to a simple encoding method.

Given the $\{p_i\}$,

- $l_i \leftarrow \lceil -\log_c p_i \rceil$
- $f_i \leftarrow$ the c -ary expansion, to l_i places, of $\sum_{j=1}^{i-1} p_j$

Example:

| i | p_i | $-\lg p_i$ | l_i | f_i |
|-----|-------|------------|-------|-------|
| 1 | 0.4 | 1.32 | 2 | 00 |
| 2 | 0.2 | 2.32 | 3 | 011 |
| 3 | 0.2 | 2.32 | 3 | 100 |
| 4 | 0.1 | 3.32 | 4 | 1100 |
| 5 | 0.1 | 3.32 | 4 | 1110 |

- $f_2 = p_1 = 0.4$
- .4 expanded in binary to 3 places is .011
- $f_3 = p_1 + p_2 = 0.6$
- .6 expanded in binary to 3 places is .100
- $H = 2.12$ and $E = 2.8$
- wasteful code: could drop last digit for symbols 2,3,4,5

Shannon Coding

Example:

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| | <i>e</i> | <i>t</i> | <i>a</i> | <i>n</i> | <i>o</i> | <i>i</i> | <i>l</i> |
| <i>p</i> | .25 | .20 | .15 | .12 | .10 | .10 | .08 |
| $-\lg p$ | 2 | 2.3 | 2.7 | 3.1 | 3.3 | 3.3 | 3.6 |
| <i>l</i> | 2 | 3 | 3 | 4 | 4 | 4 | 4 |

- $H = 2.698$ and $E = 3.15 \Rightarrow$ redundancy = 0.452

If block k source letters and use Shannon on the result stream, the average code length per source letter

$$= \frac{1}{k} \sum_{\sigma} \Pr(\sigma) \text{len}(\text{codeword}(\sigma))$$

$$\leq \frac{1}{k} \sum_{\sigma} \Pr(\sigma) (-\log \Pr(\sigma) + 1)$$

$$= H_k + \frac{1}{k}, \text{ which approaches } H \text{ as } k \text{ increases}$$

Shannon Coding – wrong code

What is expected code length E if the code is designed for wrong pdf?

Ex: Using estimate $q(x)$ of unknown pdf $p(x)$.

Theorem Using code lengths $l(x) = \lceil \log \frac{1}{q(x)} \rceil$, when pdf is $p(x)$, results in expected length E , where

$$H(p) + D(p||q) \leq E < H(p) + D(p||q) + 1$$

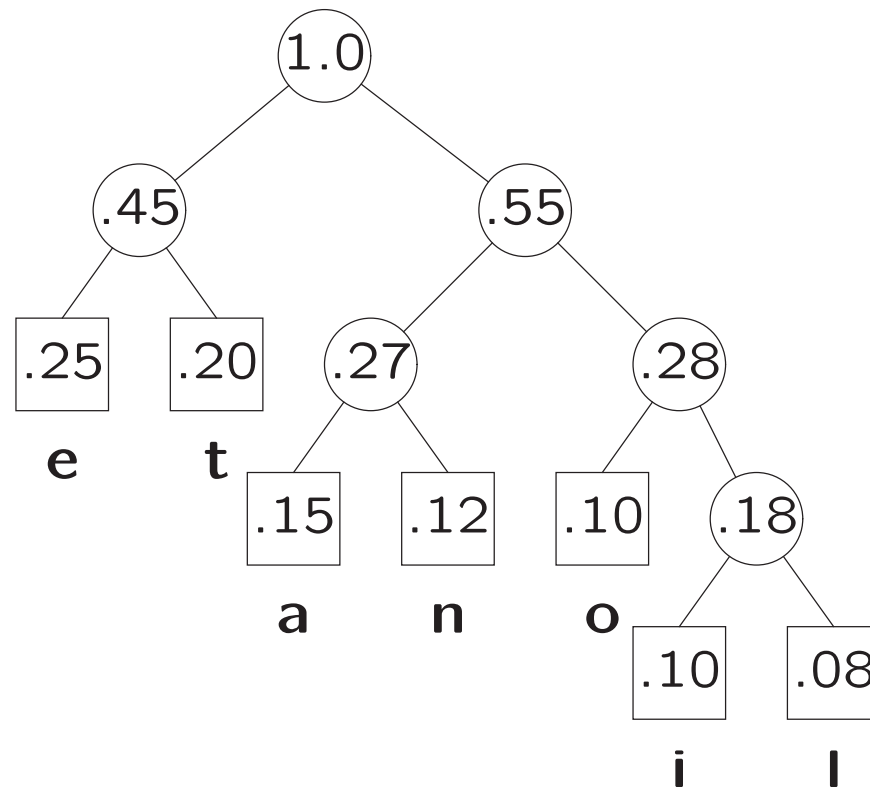
Proof [upper bound]

$$\begin{aligned} E &= \sum_x p(x) \lceil \log \frac{1}{q(x)} \rceil \\ &< \sum_x p(x) (\log \frac{1}{q(x)} + 1) = \sum_x p(x) \log \left(\frac{p(x)}{q(x)} \frac{1}{p(x)} \right) + 1 \\ &= \sum_x p(x) \log \frac{p(x)}{q(x)} + \sum_x p(x) \log \frac{1}{p(x)} + 1 \\ &= D(p||q) + H(p) + 1 \end{aligned}$$

Shannon-Fano Coding

- list elements in order of decreasing prob
- divide list into 2 parts (P_1, P_2) of \approx equal prob
- el'ts in P_1 start with 0-bit, in P_2 with 1-bit
- continue recursively with lists P_1 and P_2

Example: e t a n o i l
.25 .20 .15 .12 .10 .10 .08



- $H = 2.698$, avg code length = 2.73, redundancy = 0.032

Static Huffman Coding

- given freq distrib F and code $C = (L_1, \dots, L_n)$
encoding length = $F \cdot C = \sum f_i L_i$
 - optimal code if minimum encoding length
 - an optimal code may not be unique

Example: frequency distribution = (4,2,2,1,1)

size-5 compact code $C_1 = (1,2,3,4,4)$

size-5 compact code $C_2 = (1,3,3,3,3)$

size-5 compact code $C_3 = (2,2,2,3,3)$

each yields dot product = 22 bits

- in general, optimal code might be unique but annoying to search,
for example, all 1 billion size-39 compact codes
- Huffman code finds an optimal compact code quickly

Static Huffman Coding

- **Huffman is bottom-up** (Shannon-Fano is top-down)

initialize list with ordered set of probabilities

while $len(\text{list}) > 1$ **do**

merge 2 smallest values (i, j) into one value (x)

represent x by creating a parent node having children i and j

insert x in proper place

- **avoid insertion search**: use 2 lists (leaf+internal)

while lists contain more than one value **do**

merge 2 smallest values (i, j) into one value (x)

represent x by creating a parent node having children i and j

place x at end of 'internal' list

Static Huffman Coding

- **weighted codewords**

Huffman's algorithm to minimize $\sum p_i l_i$ works even when $\sum p_i \neq 1$, as long as all $p_i \geq 0$

- Huffman can be used on frequencies, not just probabilities

- **redundancy** = avg code length - entropy

- Shannon redundancy < 1
- Shannon-Fano redundancy < 2

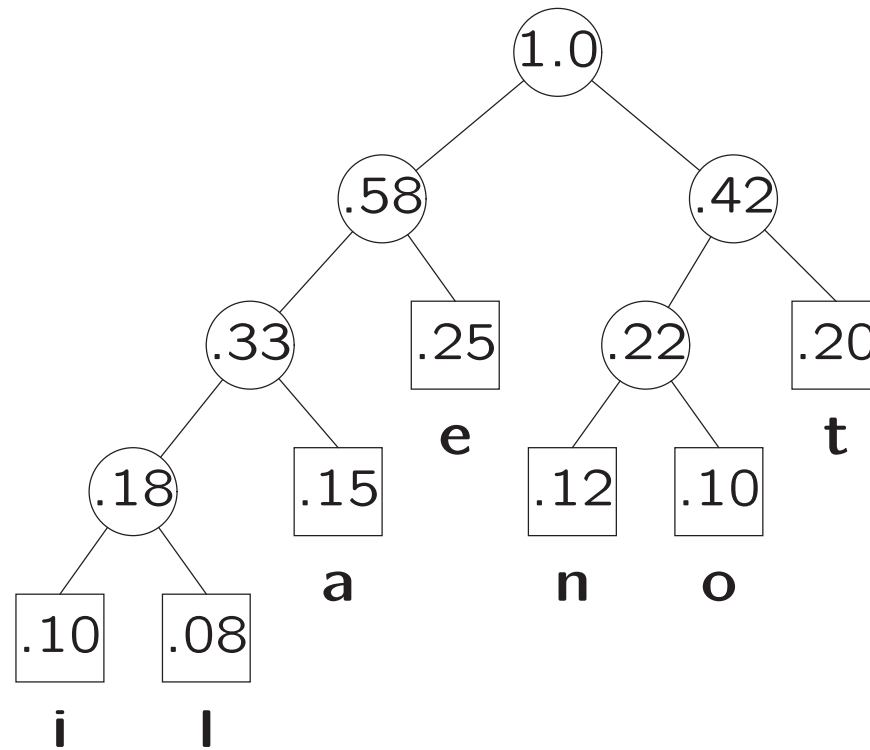
- Huffman redundancy $\leq \begin{cases} p + 0.086 = \lg \frac{2 \lg e}{e} & \text{if } p \geq 0.5 \\ p & \text{if } p < 0.5 \end{cases}$

where p = prob of most likely symbol

- for English text, $\text{Pr}(\text{space}) = 0.18$

Static Huffman Coding

Example: e t a n o i l
.25 .20 .15 .12 .10 .10 .08



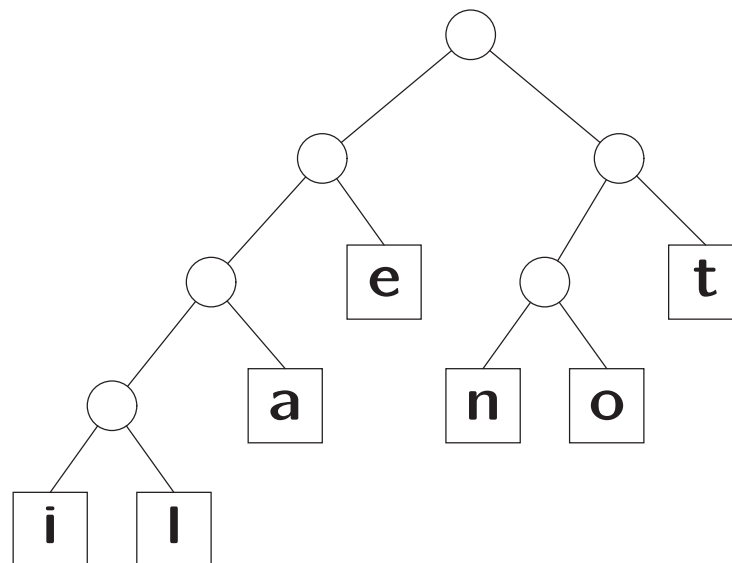
Static Huffman Coding

Decoding:

- encoder transmits the code in some form
- decoder iterates determining next char by
 - start at top (root) of tree
 - branch L/R (for 0- or 1-bit) until a Leaf
 - output character associated with Leaf

Example:

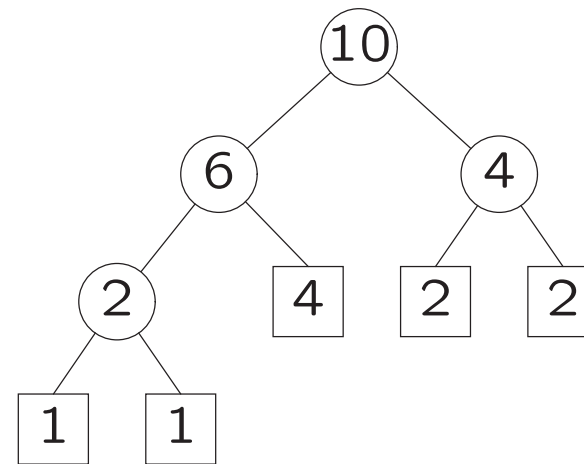
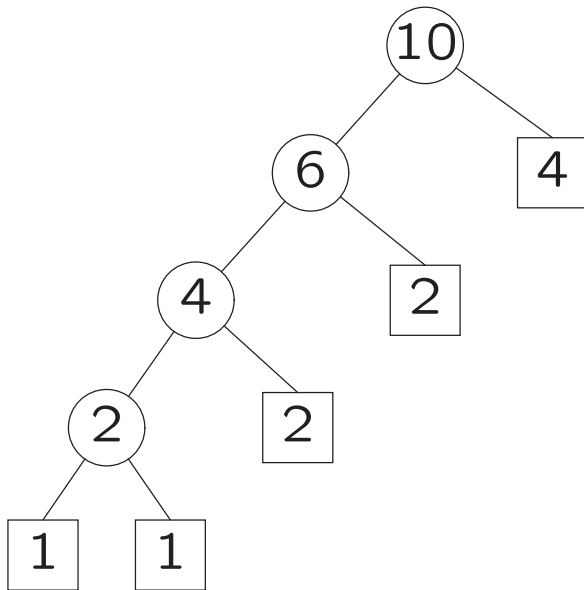
1 1 1 0 1 0 0 0 0 0 0 0 1
t o i l



Static Huffman Coding

- prefer merging leaf will minimize variance

Example: 1, 1, 2, 2, 4



Static Huffman Coding

- **canonical** Huffman tree (leaf-heights ordered)
can be represented in an especially space efficient way
- **M -ary** Huffman trees
 - express code in base M
 - $M = 16 \Rightarrow$ manipulate nybbles instead of bits

Static Huffman Coding – Block Coding

- k -block Huffman coding
 - block input into groups of k characters each and treat these blocks as atomic characters
 - assumes independence of probabilities
 - tree size increases exponentially with k
 - redundancy (per char) $\leq 1/k$

Example: input = a,b,b,a,b,a,a,b,b,b,a,a,b,a,b
2-blocked = ab,ba,ba,ab,bb,ba,ab,ab
new alphabet = {ab,ba,bb} with frequencies {4,3,1}
 encoded as {**0,10,11**}
output = **010100111000**

Redundancy of Huffman BlockCoding $\leq 1/k$

Definitions

X = input (of len n), X^k = k -blocked input (has len n/k)

$B(X^k)$ = output using Huffman, has len L^k

$K(X)$ = output using k -block Huffman, has len L

$R(X^k)$ = compression rate of $B(X^k)$ in bits/symbol

$R_k(X)$ = compression rate of $K(X)$

entropy = (bits/symbol)*(# of symbols) = $H_k(X)*n = H(X^k)*n/k$

- $B(X^k) = K(X)$, and so $L^k = L$
 $R(X^k) = L^k/(n/k)$
 $R_k(X) = L/n$
and so $R_k(X) = R(X^k)/k$
- from Shannon's Theorem, $H(X^k) \leq R(X^k) \leq H(X^k) + 1$
divide through by $k \Rightarrow H_k(X) \leq R_k(X) \leq H_k(X) + 1/k$
note $H_k(X) \leq H(X)$,
so $R_k(X) \leq H(X) + 1/k$

Adaptive Codes

- code changes over time

Example:

initially

A 00

B 01

C 1

after **A** is found to occur very frequently
and **C** rarely,

A 0

B 10

C 11

- encoder and decoder must execute
same algorithm so as to maintain synchrony

Adaptive Huffman Coding

- symbol freqs reflect *only* what was seen
initial default tree (empty) avoids pre-scan
- Sibling property
nodes can be listed in frequency order
 - to make sibling nodes adjacent
 - nodes with same frequency grouped into a **block**
- the code tree changes as symbols are read
expensive to rebuild tree after each char
updating tree after each char is reasonable
- encode symbol and *then* update tree
decoder mirrors encoder: decode & update

Adaptive Huffman Coding

- Encoding
 - if symbol previously unseen
 - encode `escape` and new symbol
 - insert new symbol in codetree
 - else (symbol previously seen)
 - encode codeword of prev seen sym
 - swap node with high node in block
 - increment frequency of symbol
 - update tree

Adaptive Huffman Coding

Symbol previously UNSEEN

- encode escape code
node NYT (not yet transmitted) for *escape*
same as encoding any other node (see below)
- encode new ascii symbol
use a phased-in binary code
- insert new symbol in codetree
node NYT (*escape*) becomes
interior node (having *freq* 1)
with children:
 - NYT (*freq* 0)
 - new symbol (*freq* 1)

Adaptive Huffman Coding

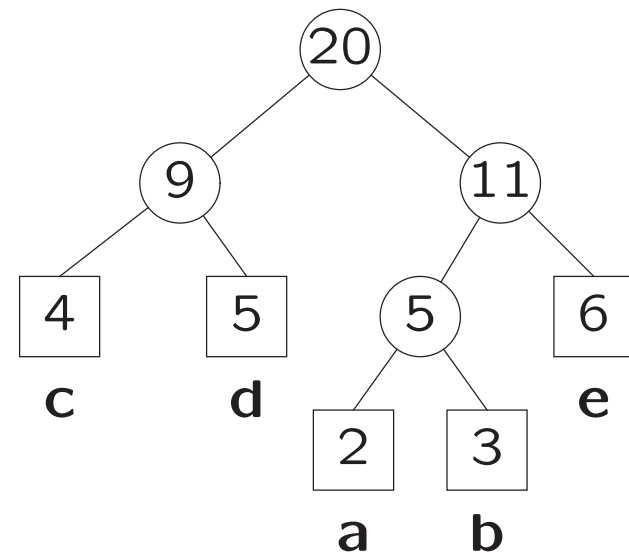
Symbol previously SEEN

- encode codeword of symbol S
 - determine node X associated with S (array look-up or linear search)
 - move up tree from X to *root*
 - going from L(R) child to parent \rightarrow append 1 (0) bit to code at end, emit code bits in *reverse order*
- update tree after read symbol of node X
until $X = \text{root}$ do
 - swap X with the first node of block \neq parent(X)
 - increment $\text{freq}(X)$, $X \leftarrow \text{parent}(X)$
- frequency counter overflow: re-scale
 - halve leaf frequencies, recalculate others
 - requires rebuilding tree
 - induces exponential freq decay \Rightarrow locality of ref

Adaptive Huffman Coding

Example 1: aabbbccccdddddeeeee 2: eeeeeeddddccccbbbaa

- static Huffman uses 93 bits

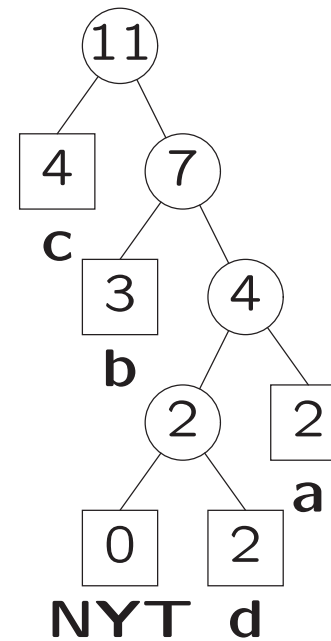


tree needs $5 \cdot 8$ bits (chars) + 8 bits (struct) = 48
string needs $3(2+3) + 2(4+5+6) = 45$ bits

- adaptive Huffman uses 89 bits (Ex 2: 83 bits)
transmit 5 chars intermingled = 40 bits
string (including escape codes) = 49 bits (Ex 2: 43 bits)

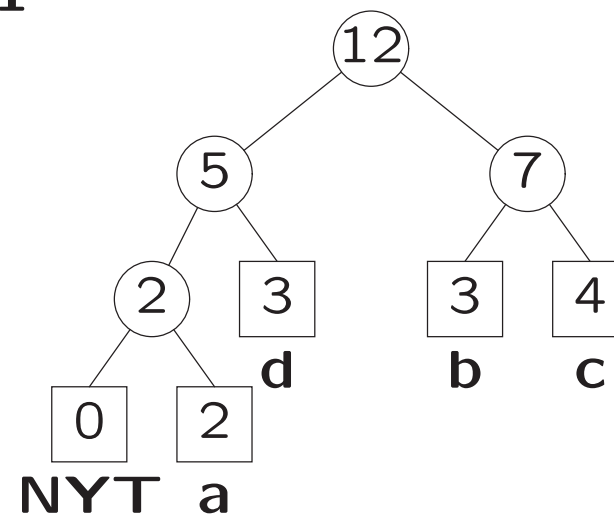
Adaptive Huffman Coding

after: aabbcccccdd



symbol **d** is read and encoded by: **1101**

tree is updated:



Length-Limited Huffman Coding

- motivation

size of high-speed register or buffer

- problem specification

- given: $p = p_1 < \dots < p_n, L$

- find: $l = l_i, \text{ each } \leq L$

so as to

satisfy Kraft: $K = \sum 2^{-l_i} \leq 1$

minimize cost $C = \sum p_i l_i$

- easy to assign codewords after finding codelengths

Length-Limited Huffman – Approach

- initially: all $l_i = 0$
 \Rightarrow cost $C=0$, len limit ok, but $K = n > 1$
- can increment $l_1 \leftarrow 1 \Rightarrow$ reduces K by 0.5
- can increment $l_2 \leftarrow 1 \Rightarrow$ reduces K by 0.5
- **choice:** increment $l_3 \leftarrow 1$ or increment l_1 and $l_2 \leftarrow 2$
- iterate $2n - 2$ times reducing K by 0.5
 each time increment lens of cheapest item set
 whose sum of exponentiated neg lens = 1
- $q[1]$ is a list, ordered by impact on C , of packages of items
 that, when incremented by 1, would reduce K by 2^{-L}
- $P \in q[k + 1]$ reduces K by twice as much as $P' \in q[k]$

Length-Limited Huffman Coding

- definitions

item = leaf at a certain depth

package = an item or a collection of items

the sum of whose widths is a constant

merge() pairs objects (items or packages)

to produce ordered list of packages having twice the width

- algorithm

if $L < \lceil \lg n \rceil$ **then** impossible

$q[1] \leftarrow p$

for $i \leftarrow 1$ **to** $L - 1$

$q[i + 1] \leftarrow \text{merge}(p, \text{package}(q[i]))$

$l \leftarrow \{0, 0, \dots, 0\}$

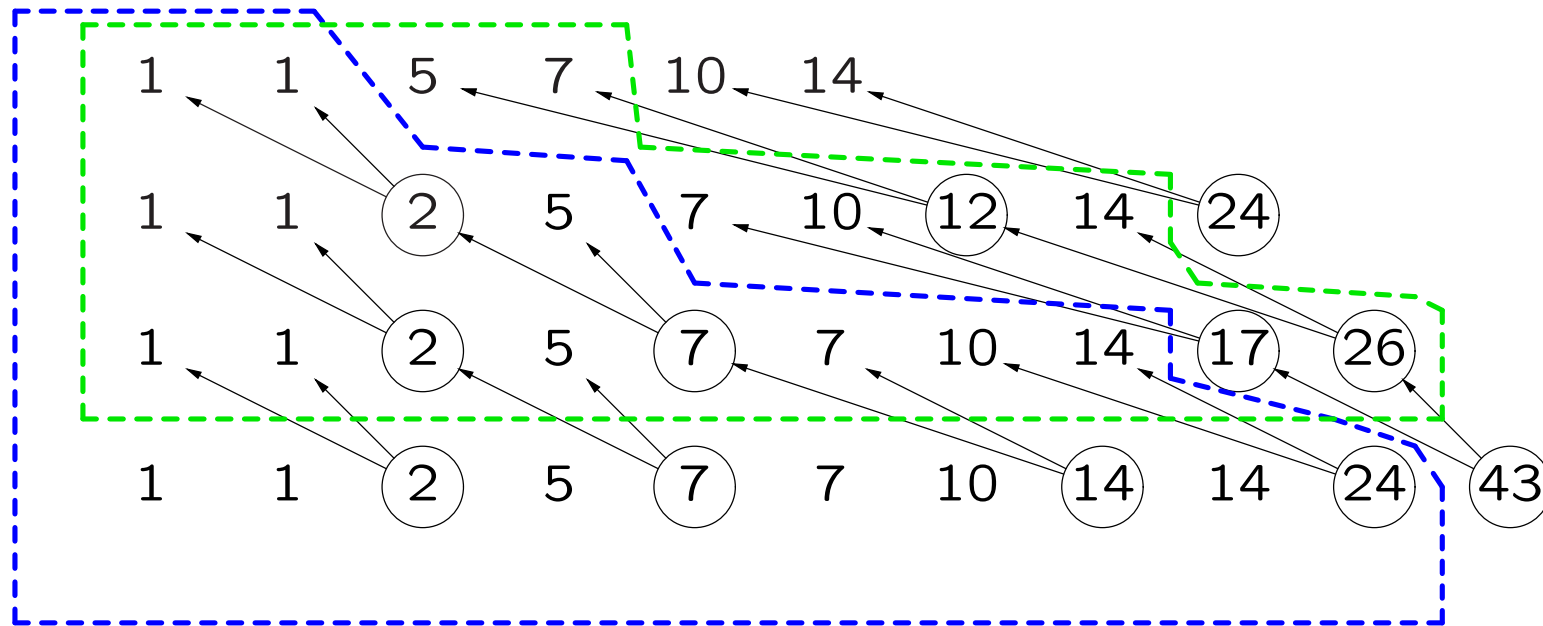
for $j \leftarrow 1$ **to** $2n - 2$

for each leaf p_k in set of items rooted at $q[L, j]$

$l_k \leftarrow l_k + 1$

Length-Limited Huffman Coding

Example: for $p = \{1,1,5,7,10,14\}$ with $L = 4$



- $L = 4 \rightarrow$ lengths = $\{4,4,3,2,2,2\}$
- $L = 3 \rightarrow$ lengths = $\{3,3,3,3,2,2\}$

Length-Limited Huffman Coding

complexity

- $O(nL)$ time and $O(n)$ space
- can reduce space to $O(L^2)$ [Katajainen et al. 1995]
- faster algorithms
 - $O(n(L - \lg n + 1))$ time & space [Turpin 1998]
 - $O(n(H - L + 1))$ time & $O(n)$ space
where $H =$ longest codeword length from Huffman alg
[Liddell, Moffat 2003]

Information Theory – Huffman Code

- Huffman code is the **most efficient prefix code** but is not necessarily optimal
- length of code for symbol $i \approx \lg(1/p_i)$
not necessarily bounded by $\lceil \lg(1/p_i) \rceil$
- Huffman coding is **optimal** (achieves entropy)
only if all probs are integer powers of $1/2$
- (exaggerated) example of Huffman code inefficiency:
 - 2 chars with probabilities 0.01 and 0.99
 - file of 100 chars
 - Huffman algorithm gives each char a 1-bit code
result: compressed file size = 100 bits
 - entropy predicts file size to be 8 bits

Alternating Runlength Huffman (ARH)

- P = prob of most probable symbol (MPS)
- if source is IID and $P > \frac{\sqrt{5}-1}{2} \approx 0.618$
more efficient to encode as an alternating sequence

(1) runs of MPS – each RLE via Huffman

$$\Pr[\text{runlength}=i] = (1 - P)P^i$$

use Golomb code with parameter $m = \lceil \frac{-\log(1+P)}{\log P} \rceil$

| m | max P | m | max P | m | max P |
|-----|---------|-----|---------|-----|---------|
| 2 | .7548 | 5 | .8812 | 8 | .9215 |
| 3 | .8191 | 6 | .8986 | 9 | .9295 |
| 4 | .8566 | 7 | .9115 | 10 | .9360 |

(2) other symbols – each coded by Huffman

Example:

input 0, 1, 0, 0, 0, 2,-1

viewed as $0^1, 1, 0^3, 2, 0^0, -1$

expressed as 1, 1, 3, 2, 0,-1

interleaved seqs coded via separate Huffman codes

Alphabet Partitioning – Modified Huffman

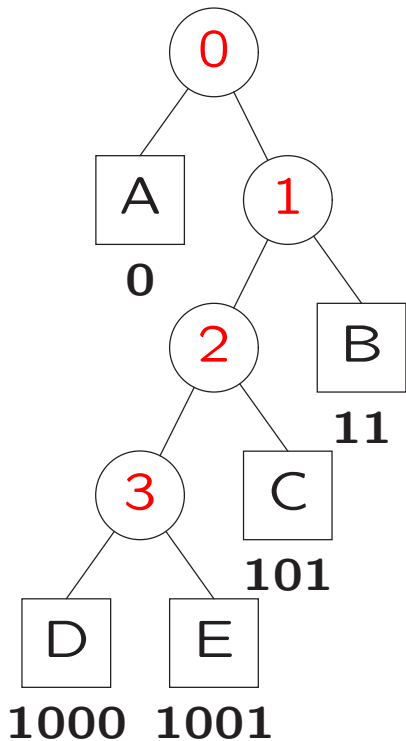
- partition a large alphabet into smaller sets
identify symbol by set number, then by element within set
- **Modified Huffman Coding:** each set's size is a power of 2
- most applications use predefined partitions
based on assumptions of pdf
- when pdf is known, can determine an optimal partitioning
(not bound by the power-of-2 constraint)
if N input values, and use M partitions,
dynamic pgm'ng finds opt in time $O(MN^2)$
- a heuristic can determine good partition in time $O(N)$

Ref: Chen, Chiang, etal., DCC 2003, pp.372-381

Fast Decoding of Huffman Encodings

Partial Decoding Table

replace tree traversal with table look-ups



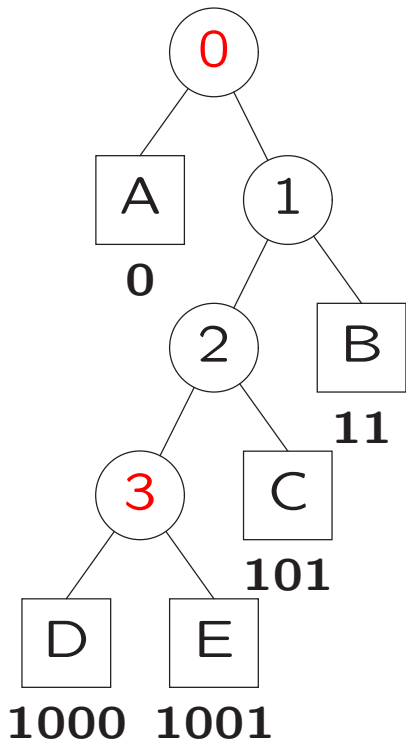
| code | table 0 | | table 1 | | table 2 | | table 3 | |
|------|---------|-----|---------|-----|---------|-----|---------|-----|
| | out | tbl | out | tbl | out | tbl | out | tbl |
| 000 | AAA | 0 | D | 0 | DA | 0 | DAA | 0 |
| 001 | AA | 1 | E | 0 | D | 1 | DA | 1 |
| 010 | A | 2 | CA | 0 | EA | 0 | D | 2 |
| 011 | AB | 0 | C | 1 | E | 1 | DB | 0 |
| 100 | - | 3 | BAA | 0 | CAA | 0 | EAA | 0 |
| 101 | C | 0 | BA | 1 | CA | 1 | EA | 1 |
| 110 | BA | 0 | B | 2 | C | 2 | E | 2 |
| 111 | B | 1 | BB | 0 | CB | 0 | EB | 0 |

- Example:** 100 101 110 000 101
 100 \Rightarrow out: λ , new tbl 3 101 \Rightarrow out: EA, new tbl 1
 110 \Rightarrow out: B, new tbl 2 000 \Rightarrow out: DA, new tbl 0
 101 \Rightarrow out: C, new tbl 0

Fast Decoding of Huffman Encodings

Reduced Partial Decoding Table

delete some tables, replace references by backing-up count



| code | table 0 | | | table 3 | | |
|------|---------|-----|---|---------|-----|---|
| | out | tbl | b | out | tbl | b |
| 000 | AAA | 0 | 0 | DAA | 0 | 0 |
| 001 | AA | 0 | 1 | DA | 0 | 1 |
| 010 | A | 0 | 2 | D | 0 | 2 |
| 011 | AB | 0 | 0 | DB | 0 | 0 |
| 100 | - | 3 | 0 | EAA | 0 | 0 |
| 101 | C | 0 | 0 | EA | 0 | 1 |
| 110 | BA | 0 | 0 | E | 0 | 2 |
| 111 | B | 0 | 1 | EB | 0 | 0 |

- Example:** 100 101 110 000 101
 100 \Rightarrow out: λ , new tbl 3 101 \Rightarrow out: EA, new tbl 0, backup 1
 111 \Rightarrow out: B, new tbl 0, backup 1 100 \Rightarrow out: λ , new tbl 3
 001 \Rightarrow out: DA, new tbl 0, backup 1 101 \Rightarrow out: C, new tbl 0

Tunstall Codes

- variable-to-fixed rate code
- $s =$ size of source alphabet $S = \{x_1, \dots, x_s\}$
 $c =$ size of code alphabet, typically 2^m
- initialize *root* to have weight 1
while there are $\leq c - s + 1$ leaves **do**
 let L be leaf with largest weight
 create s children of L , one for each x_i
 label edge ($L \rightarrow$ its i -th child) with x_i
 weight(child i of L) = $\text{Pr}(x_i) \cdot \text{wt}(L)$
- associate a unique code y_j to each leaf L_j
the string of labels on path $\text{root} \rightarrow L_j$
is the source string that maps to y_j

Tunstall Codes

- optimal uniquely parsable code of given size
plurally parsable codes can do better

Example

$\{0, 000, 1\}$ when $p_0 = .83$, $p_1 = .17$

- can generalize to k -block inputs (as in Huffman)
- adaptive Tunstall (analogous to adaptive Huffman)

Arithmetic Coding

- does not use paradigm of codeword set
represent string by number in subinterval of $[0., 1.)$
- approaches theoretical best compression arbitrarily closely
 - near-optimal for any distribution
 - uses a total of ≤ 2 bits more than required by entropy
 - avg codelength $\leq H + 2/m$, where $m =$ string length
- more efficient adaptive implementation
- uniquely decodable,
but can have unbounded coding delay

Arithmetic Coding

- input symbols with known/assumed probabilities
 - $p_i = \text{Pr}(\text{symbol } i)$
 - cumulative prob $P_i = \sum_1^i p_k$
- **maintain interval** $I = [L, H)$, initially: $I = [0, 1)$
iterate
 - read next symbol s_i
 - update interval:
$$L' \leftarrow L + P_{i-1} \cdot (H - L)$$
$$H' \leftarrow L + P_i \cdot (H - L)$$
- output sufficient bits to distinguish I from other intervals
- the representation of any input sequence
is a real number between 0 and 1
decoder can reconstruct the sequence from the real number
- high-probability char can contribute < 1 bit to output

Arithmetic Coding – Example

| | X | | Y | | # |
|---|----------|--|----------|----|----------|
| 0 | .2 | | | .9 | 1 |

| | X | | Y | | # |
|---|----------|-----|----------|-----|----------|
| 0 | .2 | .34 | | .83 | .9 |
| 1 | | | | .9 | 1 |

| | X | Y | # |
|---|----------|----------|----------|
| 0 | .2 | .34 | |
| 1 | | | 1 |

- transmit any real in the range $[\text{.326}, \text{.34})$
- # of bits required = $-\lg(\text{interval size})$
- a hi-prob symbol narrows interval less than does a lo-prob symbol, and so contributes fewer bits

Arithmetic Coding

Problems:

1. no output until end
2. high precision arithmetic

Solutions:

1. output each bit as it becomes known
2. double length of interval
reflect only the unknown part of the output

Problem:

3. if interval shrinks while straddling $\frac{1}{2}$
we won't know next bit b_j

Solution:

3. but we will know that the following bit $b_{j+1} = \bar{b}_j$
so note that & expand

Arithmetic Coding

After selecting subinterval, modify I and output bits:

until $I \notin [0, \frac{1}{2})$ & $I \notin [\frac{1}{4}, \frac{3}{4})$ & $I \notin [\frac{1}{2}, 1)$

do

if $I \in [0, \frac{1}{2})$ **then**

output 0 & leftover 1's

$[L', H'] \leftarrow [2L, 2H]$

else if $I \in [\frac{1}{2}, 1)$ **then**

output 1 and leftover 0's

$[L', H'] \leftarrow [2L - 1, 2H - 1]$

else $\{I \in [\frac{1}{4}, \frac{3}{4})\}$

increment leftover

$[L', H'] \leftarrow [2L - \frac{1}{2}, 2H - \frac{1}{2}]$

Arithmetic Coding

- decoder reconstructs using reverse reasoning
- need EOF char (**#**) to know when to stop
- decoding can also be done incrementally
 - encoder transmits bits as they become known
 - decoder recovers chars as subinterval becomes known
- no need for infinite precision, use integer arithmetic
 - instead of $[0.0, 1.0)$ use $[0, 65536)$
 - re-scale when necessary
 - keep cumulative counts C_i , let $T = C_n$
$$L' \leftarrow L + \lfloor \frac{C_{i-1} \cdot (H-L)}{T} \rfloor$$
$$H' \leftarrow L + \lfloor \frac{C_i \cdot (H-L)}{T} \rfloor$$
 - instead of assigning fixed % for EOF
 - assign minimal-size block of slots

Arithmetic Coding

Example of integer arithmetic

- **current state**
 - symbol counts: $ct(\mathbf{a})=4$, $ct(\mathbf{b})=5$, $ct(\mathbf{EOF})=1$
 - current interval: $[25, 89)$
 - full interval: $[0, 128)$

- **input: b**
 - $L' \leftarrow 25 + \lfloor \frac{4(89-25)}{10} \rfloor = 50$
 - $H' \leftarrow 25 + \lfloor \frac{9(89-25)}{10} \rfloor = 82$
 - $|I'| \leq 32$, so expand
 - I' straddles 64, so leftover++
 - new interval = $[2 \cdot 50 - 64, 2 \cdot 82 - 64)$
= $[36, 100)$

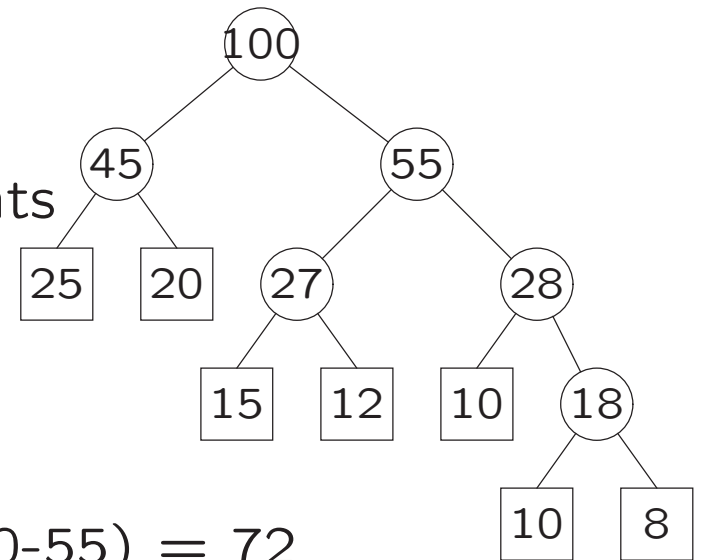
Adaptive Arithmetic Coding

- requires only freqs to narrow the interval
okay for freqs to change between iterations

- adaptive Huffman is more cumbersome
but AC does not incur increased overhead

- can use trees to maintain cumulative counts
(much faster, minimizes update time)

- all counts initialized at 1



Example $\text{cdf}[12] = 12 + (27-12) + (100-55) = 72$

- need sufficient *precision* to distinguish low-prob symbols
- freq ct overflows, so use *scaling*

Scaling

- useful to reduce values periodically
 - limit on precision (bit overflow)
 - give more weight to recent events (locality of reference)
- “lumpy” implementation
when cum values reach cut-off: halve frequency counts, *round up*
 - practical version of exponential aging
- continuous decay, half-life of d
 - at time t multiply wt by $(1 + x)^t$, with $x = \frac{\ln 2}{d}$
 - scale to retain fractional values

Example: half-life $d = 1000$ symbols ($\rightarrow x = 1.00069$)
scale by factor of 10,000
24-bit cells \rightarrow hold value ≈ 16 million (=1600 scaled)

 - initial value = 10,000 (corresponds to 1)
 - next increment = 10,007, then 10,014, etc.
 - after every 1000, halve values & increments

Structured Arithmetic Coding

- correlation may not capture symbol relations

Example

now 10 symbols active (MTF output mostly 1-10)

later 20 symbols active (output also many 11-20)

occurrence of symbol 13

could suggest increased frequencies of symbols 12 and 14

- group (bucket) symbols (say, by powers of 2)
symbol idx = bucket selector + bucket idx

Example: $B_1 = \{1\}$, $B_2 = \{2,3\}$,

$B_3 = \{4-7\}$, $B_4 = \{8-15\}$, ...

symbol 10 = bucket 4 + index 2

- maintain pdf of selectors and within each bucket
selector has small alphabet, small half-life
bucket pdf has larger half-life, slower adaptation
- gives smaller boost to freqs of all symbols in bucket

Arithmetic Coding – Variations

- quasi-arithmetic coding
 - hybrid between Huffman and AC
 - intermediate intervals maintained as states
 - replaces many computations by table look-ups
 - uniquely decodable, bounded coding delay
 - much faster than AC
 - much more efficient than Huffman
 - virtually same efficiency as AC
- Q-coder
 - adaptive binary AC
 - multiplication-free
 - patented by IBM
- Z-coder
 - also adaptive and multiplication-free AC
 - faster and not patented

Variable-cost code alphabet

- code symbol transmission costs may vary

- Morse: dash = 3, dot = 1 time unit
- bus communications: need channel control + 256 normal alphabet
most syms in 1 byte, some need 2 bytes
- energy costs: change bit > same bit

Example: cost(dash) = 3, cost(dot) = 1

| p_i | code 1 | code 2 | code 3 | code 4 | code 5 |
|---------|--------|--------|--------|--------|--------|
| 0.45 | . | . | .. | .. | |
| 0.35 | .. | ... | - | .- | - |
| 0.15 | ... | -- | ... | .. | ..- |
| 0.05 | --- | -.- | .- | -- | ...- |
| E(cost) | 3.35 | 3.45 | 3.05 | 3.20 | 3.25 |

- optimal code might not be complete

code alphabet = {**a, b, c, d**}
with costs = {1, 1, 5, 9}

| source | A | B | C | D | E(cost) |
|--------|----------|-----------|-----------|----------|---------|
| pdf | 0.45 | 0.35 | 0.15 | 0.05 | |
| code 1 | a | b | c | d | 2.0 |
| code 2 | a | ba | bb | c | 1.7 |

Perfect variable-cost code

- **Channel symbol σ** : occurs with prob q_i and has cost c_i
 - has information content = $I(q_i) = -\lg q_i$
 - has information rate = $I(q_i)/c_i$
- **Perfect code**: symbols carry info at equal rates
 - [Shannon and Weaver 1949]
 - can occur only if $q_i = t^{c_i}$, for some t
 - t determined by $\sum q_i = \sum t^{c_i} = 1$
- **Expected transmission cost** per bit of info
$$T(C) = \sum q_i \frac{c_i}{I(q_i)} = \frac{c}{I(t^c)} = \frac{1}{I(t)}$$
 - $C = [1,1]$ (normal binary) $\rightarrow t = 0.5, T(C) = 1$
 - $C = [2,2]$ (expensive bits) $\rightarrow t \approx 0.71, T(C) = 2$
 - $C = [1,3]$ (Morse-like) $\rightarrow t \approx 0.68, T(C) \approx 1.81$

Perfect variable-cost code

Earlier example: $\text{cost}(\text{dash}) = 3, \text{cost}(\text{dot}) = 1$

| p_i | code 1 | code 2 | code 3 | code 4 | code 5 |
|---------|--------|--------|--------|--------|--------|
| 0.45 | . | . | .. | .. | ... |
| 0.35 | -. | ... | - | .- | - |
| 0.15 | ---. | -- | ...- | -. | .- |
| 0.05 | ---- | -.- | ...- | -- | ...- |
| E(cost) | 3.35 | 3.45 | 3.05 | 3.20 | 3.25 |

entropy $H(\text{pdf}) \approx 1.68$

\Rightarrow best Morse might do $= 1.68 \cdot 1.81 \approx 3.04/\text{symbol}$

so "code 3" was very good

Optimal Variable-Cost Codes

General unequal-unequal problem

- Prefix code

Bradford et al., *Proc 6th Ann. Euro Symp on Alg* (1998), LNCS 1461, pp.43-54.

- Arithmetic code

consider an AC decoder that is given a stream of random bits
and generates a symbol stream with the correct symbol pdf
coded symbol costs $C \Rightarrow$ get output pdf Q

- to encode message stream:

use AC encoder with pdf P of message

\Rightarrow stream with equiprobable bits

use AC decoder with pdf Q of channel syms

\Rightarrow stream with correct symbol pdf

- to decode

use AC encoder with pdf Q

use AC decoder with pdf P

Coding Increasing Integer Sequences

- **Binary Interpolative Coding** – two-pass
 - assumes knowing sequence length and upper bound on values
 - encode median value using centered minimal binary encoding
 - recursively encode beginning and end subsequence halves
 - most effective when sequence has clusters

Example: 1 2 8 13 25 28 35, known length 7, upper bound 50

encode median $V_4 = 13$ in $[4,47]$ since $V_3 \geq 3$ and $V_5 \leq 48$,
use $\lceil \lg 44 \rceil = 6$ bits

encode $V_2 = 2$ in $[2,11]$, use 4 bits

encode $V_1 = 1$ in $[1,1]$, use 0 bits

encode $V_3 = 8$ in $[3,12]$, use 3 bits with phase-in binary code

3:0000, 4:0001, 5:010, 6:011, 7:100, 8:101, 9:110, 10:111, 11:0010, 12:0011

encode $V_6 = 28$ in $[15,49]$, use 5 bits with phase-in binary code

35 slots, 6 use 6 bits (15–17, 47–49) & 29 use 5 bits (18–46)

encode $V_5 = 25$ in $[14,27]$, use 4 bits

encode $V_7 = 35$ in $[29,50]$, use 4 bits with phase-in binary code

22 slots, 10 use 5 bits (29–33, 46–50) & 12 use 4 bits (34–45)

Coding Increasing Integer Sequences

- **Adaptive Sequential Coding** – one-pass
 - **if** needed length $>$ prev used by n
then prepend n 0-bits
otherwise prepend one 1-bit
 - use $\max\{\text{needed length, previous needed length}\}$ bitsmost effective when sequence differences are homogeneous

Example

| | | | | | | | |
|-----------------|---|---|--------|------|-------|-------|------|
| integer values | 1 | 2 | 8 | 13 | 25 | 28 | 35 |
| binary diff - 1 | 0 | 0 | 101 | 100 | 1011 | 10 | 110 |
| adaptive | 1 | 1 | 000101 | 1100 | 01011 | 10010 | 0110 |

Variations:

replace **needed** with **used**, and/or limit decrease to 1