Lecture notes on Data Compression

Arithmetic Coding

Contents

- Huffman coding revisited
- History of arithmetic coding
- Ideal arithmetic coding
- Properties of arithmetic coding

Huffman Coding Revisited -- How to Create Huffman Code

- Construct a Binary Tree of Sets of Source Symbols.
  - Sort the set of symbols with non-decreasing probabilities.
  - Form a set including two symbols of smallest probabilities.
  - Replace these by a single set containing both the symbols whose probability is the sum of the two component sets.
- Repeat the above steps until the set contains all the symbols.
  - Construct a binary tree whose nodes represent the sets. The leaf nodes representing the source symbols.
  - Traverse each path of the tree from root to a symbol, assigning a code 0 to a left branch and 1 to a right branch. The sequence of 0's and 1's thus generated is the code for the symbol.

Properties of Huffman Coding

- Huffman codes are minimum redundancy codes for a given probability distribution of the message.
- Huffman coding guarantees a coding rate l_H within one bit of the entropy H.
  - Average code length l_H of the Huffman coder on the source S is bounded by $H(S) <= l_H <= H(S) + 1$
- Studies showed that a tighter bound on the Huffman coding exists.
  - Average code length $l_H < H(S) + p_{max} + 0.086$, where p_{max} is the probability of the most frequently occurring symbol.
  - So, if the p_{max} is quite big (in case that the alphabet is small and the probability of occurrence of the different symbols is skewed), Huffman coding will be quite inefficient.

Properties of Huffman Coding (continued)

- Huffman code does not achieve 'minimum redundancy' because it does not allow fractional bits.
- Huffman needs at least one bit per symbol.
  - For example, given alphabet containing two symbols with probability: $p_1 = 0.99, p_2 = 0.01$
  - The optimal length for the first symbol is: $-\log(0.99) = 0.0145$
  - The Huffman coding, however, will assign 1 bit to this symbol.
- If the alphabet is large and probabilities are not skewed, Huffman rate is pretty close to entropy.

Properties of Huffman Coding (continued)

- If we block m symbols together, the average code length l_H of the Huffman coder on the source S is bounded by

$$H(S) <= l_H <= H(S) + 1/m$$

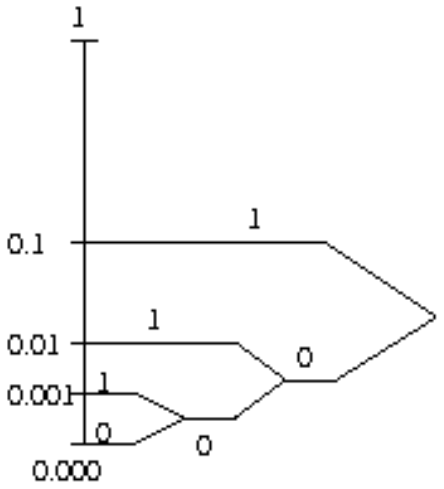
- However, the problem here is that we need a big codebook. If the size of the original alphabet is K, then the size of the new code book is K^m.

- Thus, Huffman's performance becomes better at the expense of exponential codebook size.

Another View of Huffman Coding

- Huffman code re-interpreted here by mapping the symbols to subintervals of [0,1) at the base value of the subintervals.
- The code words, if regarded as **binary fractions**, are pointers to the particular *interval* in the binary code.
- An extension to this idea is to encode the symbol sequence as a subinterval leads to arithmetic coding.

symbol	probability	code	binary fraction
W	0.5	1	0.1
X	0.25	01	0.01
Y	0.125	001	0.001
Z	0.125	000	0.000



Arithmetic Coding

- The idea is to code string as a binary fraction pointing to the *subinterval* for a particular symbol sequence.
- Arithmetic coding is especially suitable for small alphabet (binary sources) with highly skewed probabilities.
- Arithmetic coding is very popular in the image and video compression applications.

A Bit of History

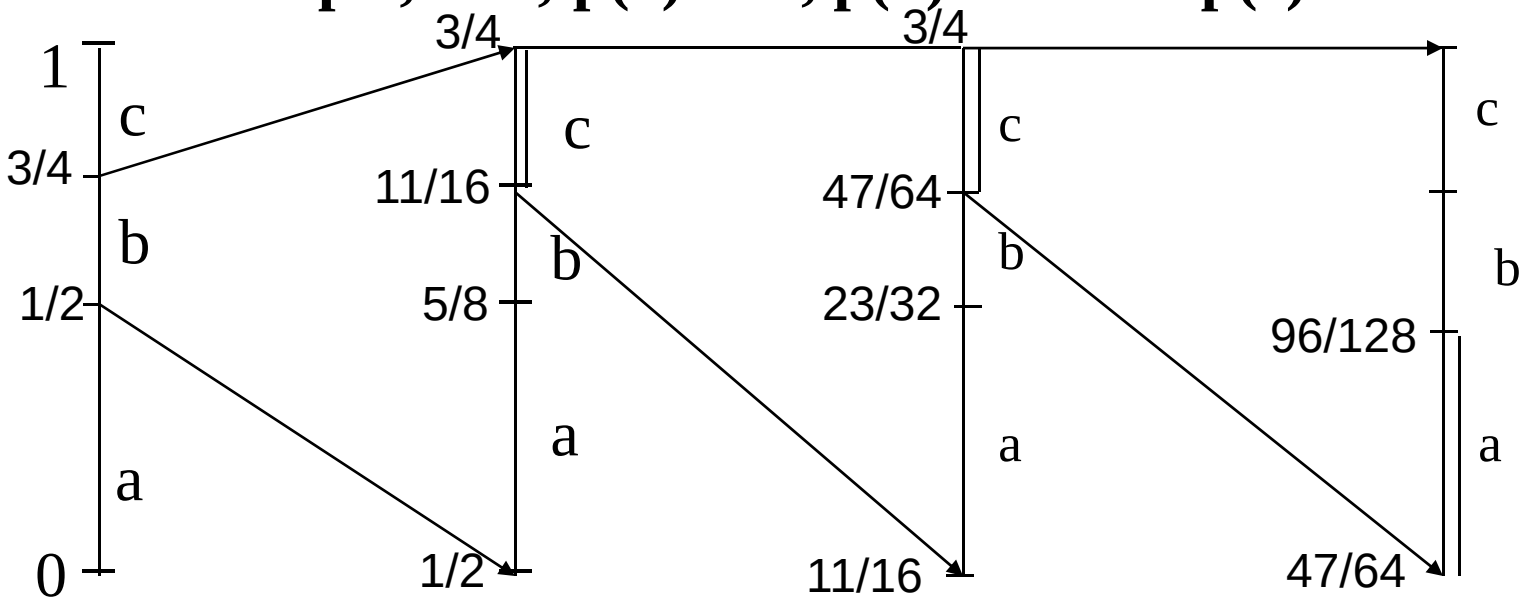
- The idea that code string can be a binary fraction pointing to the subinterval for a particular symbol sequence is due to Shannon [1948]; and was used by Elias [1963] to successive subdivision of the intervals.
- Shannon observed that if the probabilities were treated as high precision binary numbers, then it may be possible to decode messages unambiguously.
- David Huffman invented his code around the same time and the observation was left unexplored until it re-surfaced in 1975.

A Bit of History (continued)

- The idea of arithmetic coding was suggested by Rissanen [1975] from the theory of enumerative coding by Pasco [1976].
- The material of this notes is based on the most popular implementation of arithmetic coding by Witten, etc., published in Communications of the Association for Computing Machinery (1987).
- Moffat, etc (1998) also proposed some improvements upon the 1987 paper; however, the basic idea remains same.

Static Arithmetic Coding

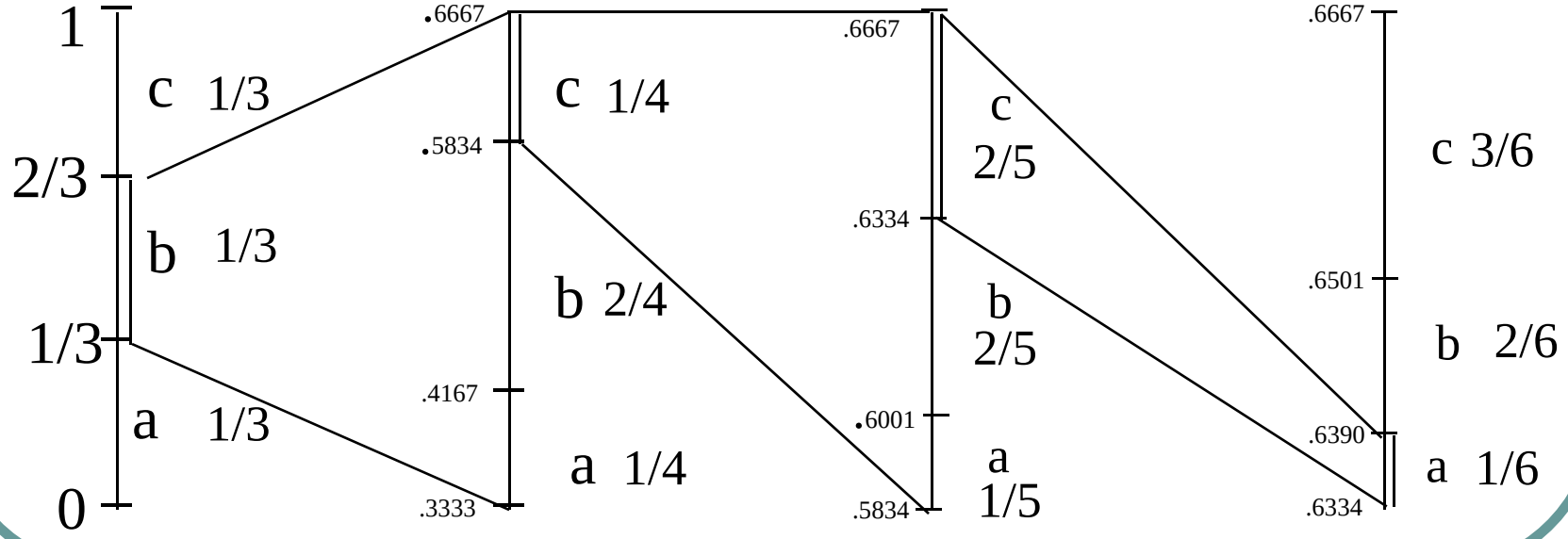
Consider an half open interval [*low*,*high*). Initially, interval is set as [0,1) and *range= high -low* = 1-0 = 1.
Interval is divided into cumulative probabilities of n symbols.
For this example, n=3; p(a)=1/2, p(b)=1/4 and p(c)=1/4.



Any value in the range [47/64,96/128)) encodes 'bcca'

Adaptive Arithmetic Coding

Consider an half open interval [*low*,*high*). Initially, interval is set as [0,1) and *range= high -low* = 1-0 = 1.
Interval is divided into cumulative probabilities of n symbols, each having the same probability 1/n at the beginning.



Any value in the range [.6334,.6390) encodes ‘bcca’

Update Frequency in Arithmetic Encoding

- A *static zero-order model* is used in the first example.
- Dynamic (second example) update is more accurate.
  - Initially we have a frequency distribution.
  - Every time we process a new symbol, update the frequency distribution.

Properties of Arithmetic Coding

- **The dynamic version is not more complex than the static version.**

- **The algorithm allocates $-\log p_i$ number of bits to a symbol of probability p_i whether or not this value is low or high. Unlike Huffman codes which is a fixed-to-variable coding scheme, arithmetic coding is variable -to-fixed coding scheme, and is capable of allocating non-integral number of bits to symbols, producing a near-optimal coding. It is not absolutely optimal due to limited precision of arithmetic operations.**

- **Incremental transmission of bits are possible, avoiding working with higher and higher precision numbers.**

Update Interval in Arithmetic Encoding

- Two main parts in the arithmetic coding
  - Update frequency distribution
  - Update subinterval
- Initially we have the interval [L=Low, L+R=Range) as [0, 1)
- Symbols of the alphabet are mapped to the integers 1,2,…s, ,n. For each incoming symbol s, the interval is updated as
  - Low: $\mathrm{L} = \mathrm{L} + \mathrm{R} \times \sum_{j=1}^{s-1} P[j]$
  - Range: $\mathrm{R} = \mathrm{R} \times \mathrm{P}[\mathrm{s}]$

This summation can be pre-calculated. When s=1, the second term is set to 0.

Ideal Static Arithmetic Encoder

1. **ALGORITHM** ARITHMETIC_IDEAL_ENCODE(M)
   /* M is the message to be encoded */
1. set L= 0 and R = 1
2. FOR i = 1 to |M| DO
   - set s = M[i]
   - set $\mathrm{L} = \mathrm{L} + \mathrm{R} \times \sum_{j=1}^{s-1} P[j]$
   - set $\mathrm{R} = \mathrm{R} \times \mathrm{P}[\mathrm{s}]$
3. END FOR
4. /*If the algorithm has to be adaptive, code has to be inserted before the above 'end' statement to re-compute probabilities*/.
5. transmit the shortest (fewest bits) binary fractional number that lies in the interval [L, L+R)
6. **END ALGORITHM**

Ideal Static Arithmetic Decoder

1. **ALGORITHM** ARITHMETIC_IDEAL_DECODE(m)
2. /* Let *V* be the fractional value transmitted by the encoder. Let the message length *m* be also be transmitted . The probabilities of the symbols are same as that of the decoder. For adaptive algorithm, the probabilities are updated following the same algorithm as used by the encoder*/
3. FOR *i* = 1 to *m* DO

   Determine *s* such that $\sum_{j=1}^{s-1} P[j] \le V < \sum_{j=1}^{s} P[j]$

   Recover *L* and *R* from s $L = \sum_{j=1}^{s-1} P[j] \qquad R = P[s]$

   Update $V = (V-L)/R$

   Output *M[i]=s*
4. END FOR
5. RETRUN message *M*
6. **END ALGORITHM**

Ideal Static Arithmetic Coding

- The basic idea of the arithmetic coding is to use a high-precision fractional number to encode the probability of the message.
  - Message M = [abaaaeaaba], alphabet $\sum = \{a, b, c, d, e, f\}$
  - Probability distribution P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
  - Probability of the message is
    P[a] * P[b] * P[a] * P[a] * P[a] * P[e] * P[a] * P[a] * P[b] * P[a]
    = 0.67 * 0.11 * 0.67* 0.67* 0.67 * 0.05 * 0.67* 0.67 * 0.11 * 0.67
    = 0.00036666730521220415.
  - However, we can not simply use this probability to encode the message, because we know there exist many messages which have exactly the same probability, such as M1 = [b, a, a, a, a, e, a, a, b, a], or M2 = [a, a, b, a, a, e, a, a, b, a], etc.
  - In fact, all permutations of the symbols in the message M have the same probability as the message M. So, to encode this message, we need to enforce some order (=<) of the letters in M.
  - This ordering will allow the computation of Cumulative Probability *PC*(*s*) for symbols up to *s*-1 but not including *s* .

Cumulative Probability

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].

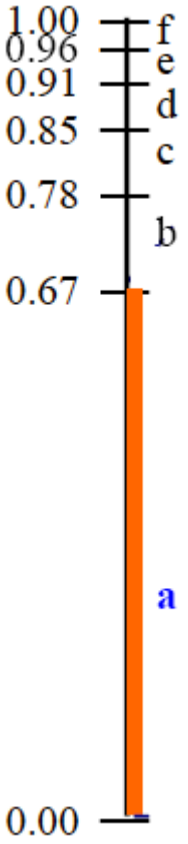
$$F_X(x) = \begin{cases} 0.00 & x \notin \Sigma \\ 0.67 & x \in \{a\} \\ 0.78 & x \in \{a, b\} \\ 0.85 & x \in \{a, b, c\} \\ 0.91 & x \in \{a, b, c, d\} \\ 0.96 & x \in \{a, b, c, d, e\} \\ 1.00 & x \in \{a, b, c, d, e, f\} \end{cases}$$

- For simplicity, we denote it by a vector PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

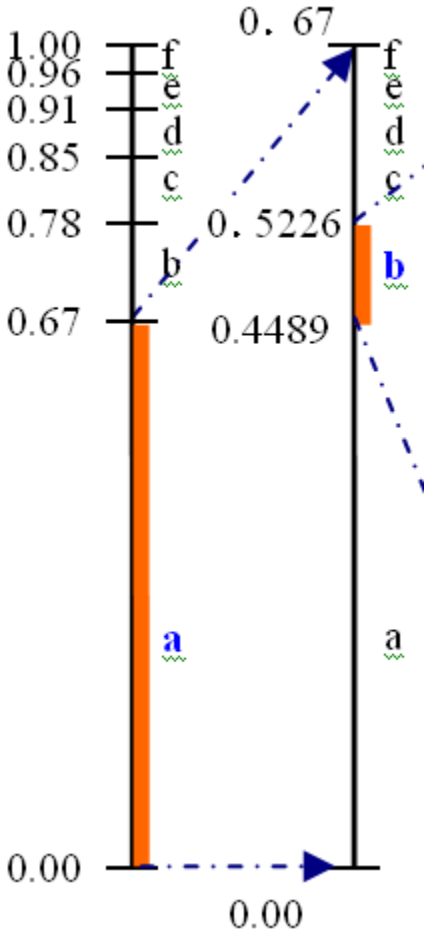
- M[1] = a,
  - LOW = 0.0
    RANGE = P[a] =0.67



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

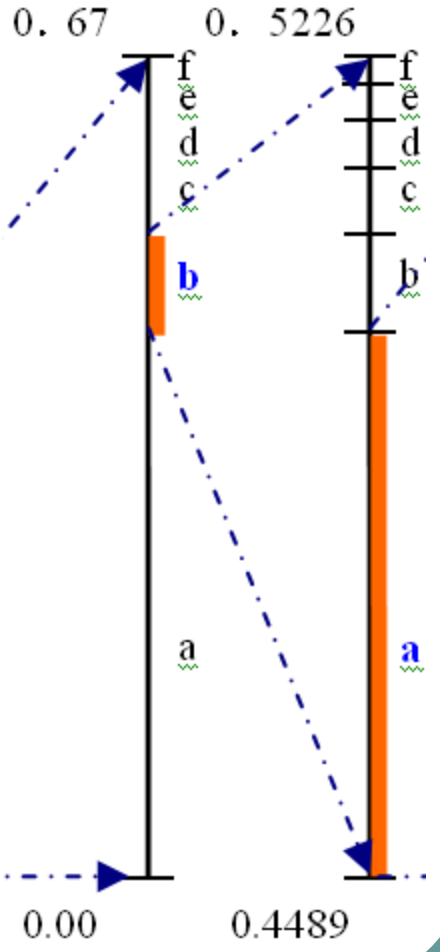
- M[2] = b,
  - LOW = LOW + PC[b] * RANGE = 0.0 + 0.67 * 0.67
    = 0.44890000000000
    RANGE = RANGE * P[b] = 0.67 * 0.11
    = 0.07370000000000



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

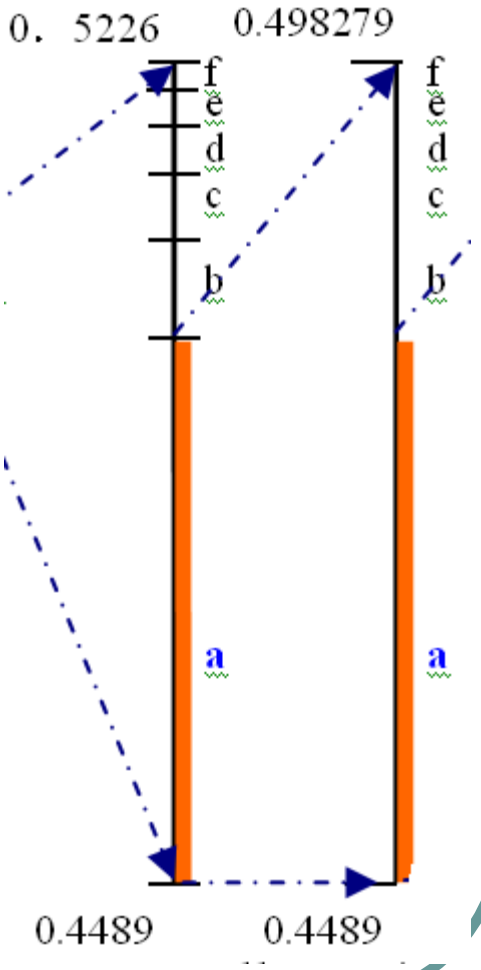
- M[3] = a,
  - LOW = LOW + PC[a] * RANGE
    = 0.44890000000000 + 0.0 * 0.07370000000000
    = 0.44890000000000
    RANGE = RANGE * P[a]
    = 0.07370000000000 * 0.67
    = 0.04937900000000



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

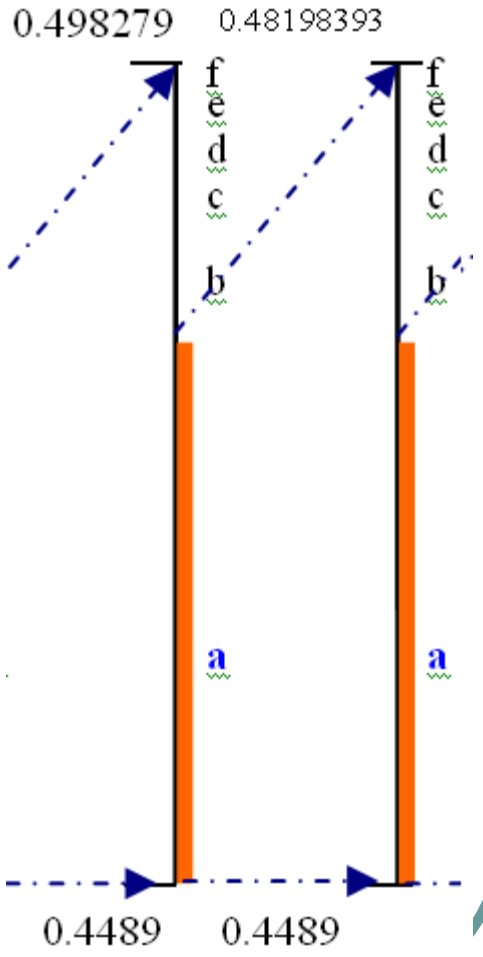
- M[4] = a,
  - LOW = LOW + PC[a] * RANGE
    = 0.44890000000000 + 0.0 * 0.04937900000000
    = 0.44890000000000
    RANGE = RANGE * P[a]
    =0.04937900000000 * 0.67
    = 0.03308393000000



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

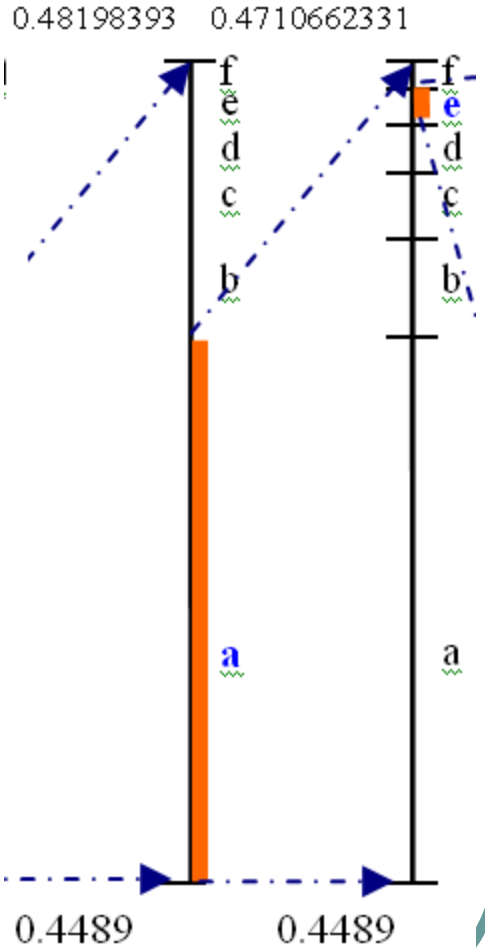
- M[5] = a,
  - LOW = LOW + PC[a] * RANGE
    = 0.44890000000000 + 0.0 * 0.04937900000000
    = 0.44890000000000
    RANGE = RANGE * P[a]
    =0.03308393000000 * 0.67
    = 0.02216623310000



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

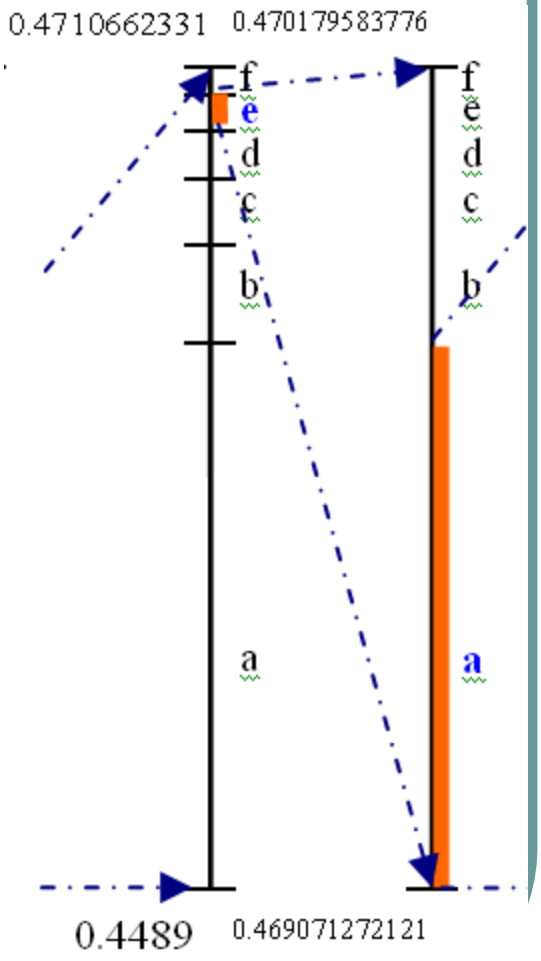
- M[6] = e,
  - LOW = LOW + PC[e] * RANGE
    = 0.44890000000000 + 0.91 * 0.02216623310000
    = 0.46907127212100
    RANGE = RANGE * P[e]
    = 0.02216623310000 * 0.05
    = 0.00110831165500



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

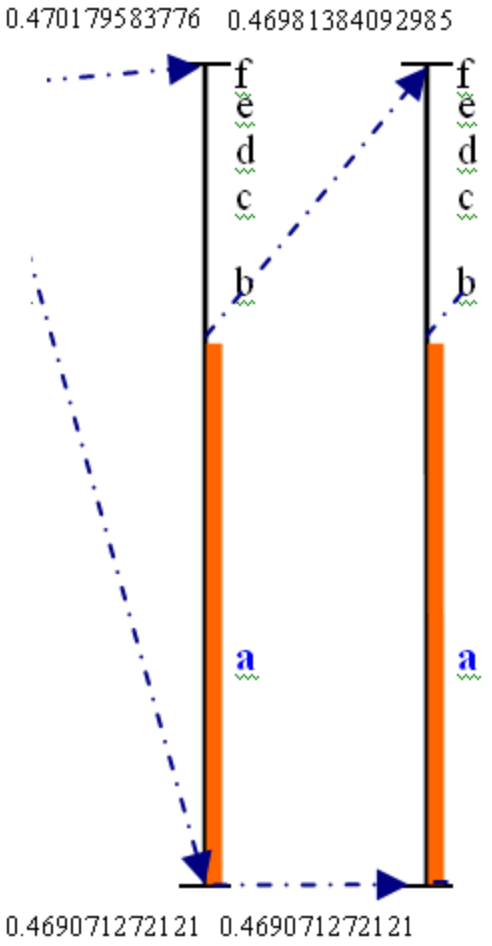
- M[7] = a,
  - LOW = LOW + PC[a] * RANGE
    = 0.46907127212100 + 0.0 * 0.00110831165500
    = 0.46907127212100
    RANGE = RANGE * P[a]
    = 0.00110831165500 * 0.67
    = 0.00074256880885



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

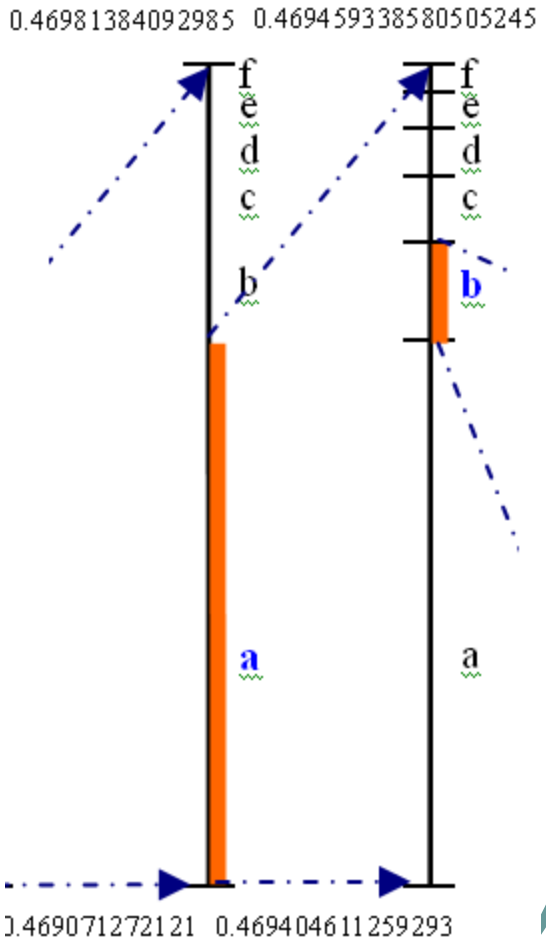
- M[8] = a,
  - LOW = LOW + PC[a] * RANGE
    = 0.46907127212100 + 0.0 * 0.00074256880885
    = 0.46907127212100
    RANGE = RANGE * P[a]
    = 0.00074256880885 * 0.67
    = 0.0004975211019295



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

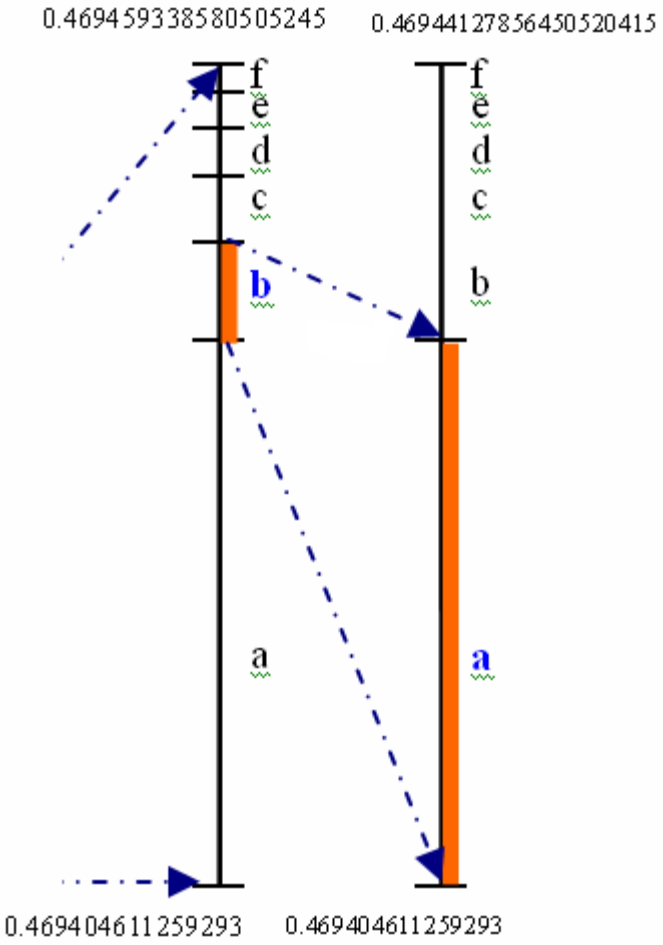
- M[9] = b,
  - LOW = LOW + PC[b] * RANGE
    = 0.469071272121 00 + 0.67 * 0.0004975211019295
    = 0.469404611259293
    RANGE = RANGE * P[b]
    = 0.0004975211019295 * 0.11
    = 0.0000547273212122 45



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]

- M[10] = a,
  - LOW = LOW + PC[a] * RANGE
    = 0.469404611259293 + 0.0 * 0. 0737
    = 0.469404611259293
    RANGE = RANGE * P[a]
    = 0.0000547273212212245 * 0.67
    = 0.00003666730521220415



Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- M = [a, b, a, a, a, e, a, a, b, a]
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- LOW = 0.469404611259293
  RANGE = 0.00003666730521220415
- OUTPUT **0.46942**

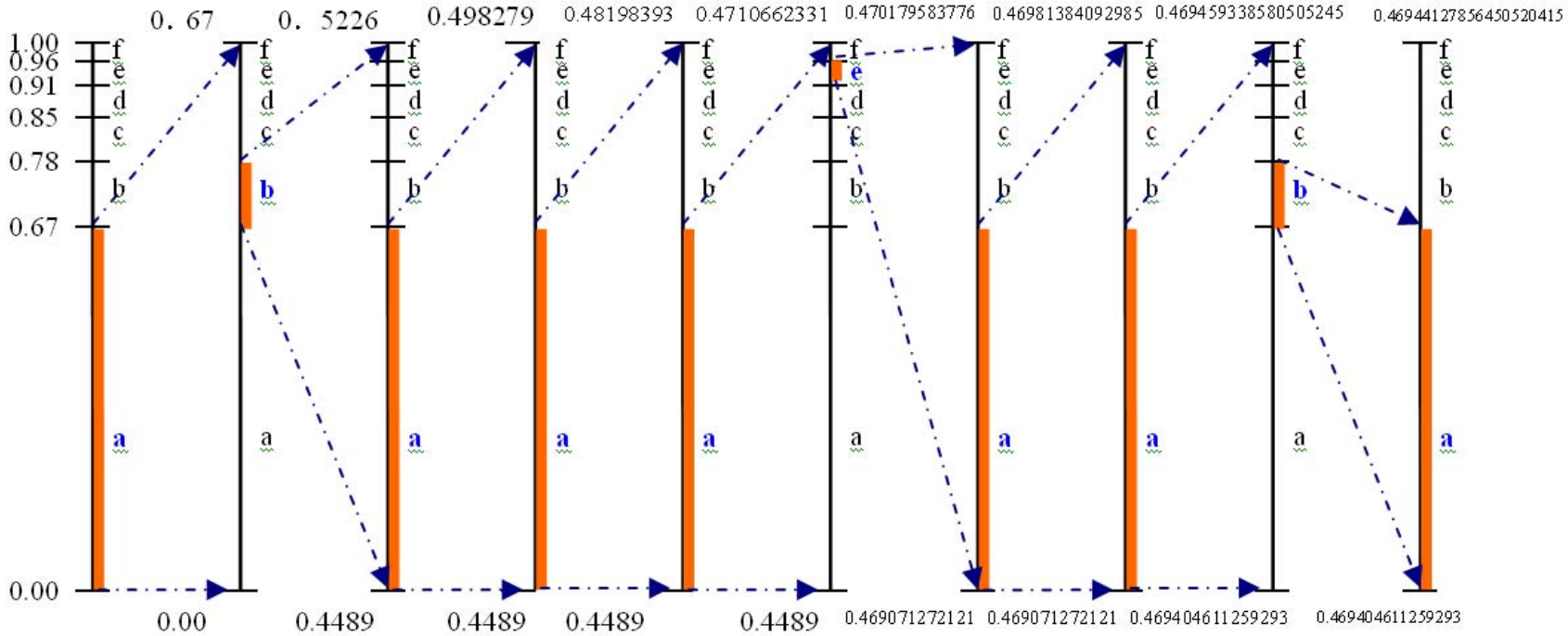


FIGURE 1. Illustration of arithmetic coding for message [abaaaeaaba]

Decode the Message --- Example

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- V = **0.46942**

- **Recover symbol #1.**
  - LOW = 0.0
  - RANGE = 1.0
  - V=0.46942, lies in the interval [0.0, 0.67)
  - Output symbol **a**
  - we have the interval [newLOW, newRANGE) = [0.0, 0.67)
  - Update the V: V = (V-newLOW) / newRANGE.
  - We have V = 0.46942/0.67 = 0.70062686567164

Decode the Message --- Example (continued)

- Alphabet $\sum = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #2.**
  - LOW = 0.0
  - RANGE = 1.0
  - V= 0.70062686567164, lies in the interval [0.67, 0.78). Output symbol **b**
  - we have the interval [newLOW, newRANGE) = [0.67, 0.11)
  - Update the V: V = (V-newLOW) / newRANGE.
  - We have V = (0.70062686567164 -0.67)/ 0.11 = 0.27842605156036

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #3.**
  - V=0.27842605156036, check the CDF, lies in the interval [0.0, 0.67), so target symbol is **a**.
  - Now, we have the interval [newLOW, newRANGE) = [0.0, 0.67).
  - We have V = (0.27842605156036 -0.0)/ 0.67 =0.41556127098561

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #4.**
  - V=0.41556127098561, check the CDF, lies in the interval [0.0, 0.67), so target symbol is **a**.
  - Now, we have the interval [newLOW, newRANGE) = [0.0, 0.67).
  - We have V = (0.41556127098561-0.0)/ 0.67 = 0.62024070296360

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #5.**
  - V=0.62024070296360, check the CDF, lies in the interval [0.0, 0.67), so target symbol is **a**.
  - Now, we have the interval [newLOW, newRANGE) = [0.0, 0.67).
  - We have V = (0.62024070296360-0.0)/ 0.67 = 0.92573239248299

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #6.**
  - V=0.92573239248299, check the CDF, lies in the interval [0.91, 0.96), so target symbol is **e**.
  - Now, we have the interval [newLOW, newRANGE) = [0.91, 0.05).
  - We have V = (0.92573239248299-0.91)/ 0.05 = 0.31464784965980

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #7.**
  - V=0.31464784965980, check the CDF, lies in the interval [0.0, 0.67), so target symbol is **a**.
  - Now, we have the interval [newLOW, newRANGE) = [0.0, 0.67).
  - We have V = (0.31464784965980-0.0)/ 0.67 = 0.46962365620866

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #8.**
  - V=0.46962365620866, check the CDF, lies in the interval [0.0, 0.67), so target symbol is **a**.
  - Now, we have the interval [newLOW, newRANGE) = [0.0, 0.67).
  - We have V = (0.46962365620866-0.0)/ 0.67 = 0.70093083016218

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #9.**
  - V=0.70093083016218, check the CDF, lies in the interval [0.67, 0.78), so target symbol is **b**.
  - Now, we have the interval [newLOW, newRANGE) = [0.67, 0.11).
  - We have V = (0.70093083016218-0.67)/ 0.11 = 0.28118936511073

Decode the Message --- Example (continued)

- Alphabet $\Sigma = \{a, b, c, d, e, f\}$
- |M| = 10
- P=[0.67, 0.11, 0.07, 0.06, 0.05, 0.04].
- PC = [0.0, 0.67, 0.78, 0.85, 0.91, 0.96, 1.00]
- **Recover symbol #10.**
  - V=0.28118936511073, check the CDF, lies in the interval [0.0, 0.67), so target symbol is **a**.
  - Now, we have recovered all the symbols. STOP decoding here.

Reference: Moffat and Turpin, p.95

The same process is followed for each symbol of the message *M.* **At any given point in time the internal *potential* of the coder is given by $-\log_2 R$. The potential is** a **measure of the eventual cost of coding the message, and counts bits. If R' is used to denote the new value of** *R* **after** an **execution of step 3 (Slide #16), then $R' = R \times P[s]$, and $-\log_2 R' = (-\log_2 R) + (-\log_2 P[s])$. That is, each iteration of the "for" loop increases the potential by exactly the information content of the symbol being coded.**

Reference: Moffat and Turpin, p.95

- **"At the end of the message the transmitted code is any number *V* such that**

$$L \leq V < L + R.$$

- **By this time** $R = \prod_{i=1}^{m} P[M[i]]$ **where *M[i]* is the ith of the m input symbols. The potential has thus increased to** $-\sum_{i=1}^{m} \log_2 P[M[i]]$ **, and to guarantee that the number *V* is within the specified range between *L* and *L* + *R*, it must be at least this many bits long.**
- **For example, consider the sequence of *L* and *R* values that arises when the message *M* = [1,2,1,1,1,5,1,1,2,1] is coded according to the static probability distribution *P* = [0.67,0.11,0.07,0.06,0.05,0.04] that was used in the example "**

The L and R values with binary Representations

		Decimal			Binary	
i	$M[i]$	L	R	$L+R$	L	$L+R$
–	–	0.00000000	1.00000000	1.00000000	0.00000000000000000000	1.00000000000000000000
1	1	0.00000000	0.67000000	0.67000000	0.00000000000000000000	0.10101011100001010100 0
2	2	0.44890000	0.07370000	0.52260000	0.01110010111010110001 11	0.10000101110010010001 11
3	1	0.44890000	0.04937900	0.49827900	0.01110010111010110001 11	0.01111111100011110011 10
4	1	0.44890000	0.03308393	0.48198393	0.01110010111010110001 11	0.01111011011000110100 11
5	1	0.44890000	0.02216623	0.47106623	0.01110010111010110001 11	0.01111000100101111100 11
6	5	0.46907127	0.00110831	0.47017958	0.01111000000101010001 00	0.01111000010111011011 00
7	1	0.46907127	0.00074257	0.46981384	0.01111000000101010001 00	0.01111000010001011011 10
8	1	0.46907127	0.00049752	0.46956879	0.01111000000101010001 00	0.01111000001101011010 10
9	2	0.46940461	0.00005473	0.46945934	0.01111000001010101110 10	0.01111000001011100111 11
10	1	0.46940461	0.00003667	0.46944128	0.01111000001010101110 10	0.01111000001011010100 11

Table 5.1: Example of arithmetic coding: representing the message $M = [1, 2, 1, 1, 1, 5, 1, 1, 2, 1]$ assuming the static probability distribution $P = [0.67, 0.11, 0.07, 0.06, 0.05, 0.04]$.

Obtaining the value V

- **"As each symbol is coded, *R* gets smaller, and *L* and *L* + *R* move closer together. By the time the 10 symbols of the example message *M* have been fully coded, the quantities *L* and *L* + *R* agree to four decimal digits, and to thirteen binary digits. This arrangement is shown in the last line of Table 5.1. Any quantity *V* that lies between *L* and *L* + *R* must have exactly the same prefix, so thirteen bits of the compressed representation of the message are immediately known. Moreover, three more bits must be added to *V* before a number is achieved that, irrespective of any further bits that follow in the coded bits tream, is always between *L* and *L* + *R:***

- *L + R :* ***0.0111 1000 00101*** *101010011*
- V : ***0.0111 1000 00101*** *100*
- *L :* ***0.0111 1000 00101*** *010111010*

Information content of the range

- High probability events do not decrease the interval *Range* very much, but low probability events result in a much smaller next interval requiring large number of digits.
- A large interval needs only a few digits. The number of digits required is *–log(size of interval).*
- The size of the final interval is the *product* of the probabilities of the symbols encoded. Thus a symbol *s* with probability *p*(*s*) contributes –log *p*(*s*) bits to the output which is the symbol's self-information.

Information content

- **"At the conclusion of the processing *R* has the value 3.67 x 10^{-5}, the product of the probabilities of the symbols in the message. The minimum number of bits required to separate *R* and *L* + *R* is thus given by $-\log_2 R = 14.74 = 15$, one less than the number of bits calculated above for *V*. A minimum-redundancy code for the same set of probabilities would have codeword lengths of [1,3,3,3,4,4] for a message length of 17 bits. The one bit difference between the arithmetic code and the minimum-redundancy code might seem a relatively a small amount to get excited about, but when the message is long, or when one symbol has a very high probability, an arithmetic code can be much more compact than a minimum-redundancy code.**
- **As an extreme situation, consider the case when *n* = 2, *P* = [0.999,0.001], and a message containing 999 "1 "s and one "2" is to be coded. At the end of the message *R* = 3.7 x 10^{-4}, and *V* will contain just**

$$-\log_2 3.7 \times 10^{-41} = 12 \text{ or } -\log_2 3.7 \times 10^{-41} + 1 = 13 \text{ bits},$$

far fewer than the 1,000 bits necessary with a minimum-redundancy code. On average, each symbol In this hypothetical message is coded in just 0.013 bits! "

Arithmetic Coding Advantages

- “There are workarounds to prefix codes that give improved compression effectiveness, such as grouping symbols together into blocks over a larger alphabet, in which individual probabilities are smaller and the redundancy reduced; or extracting runs of "1 " symbols and then using a Golomb code; or using the interpolative code. But they cannot compare with the sheer simplicity and elegance of arithmetic coding. As a further point in its favor, arithmetic coding is relatively unaffected by the extra demands that arise when the probability estimates are adjusted adaptively. “

Efficiency of the Ideal Arithmetic Coding

- The average length per symbol using the arithmetic coding is $H(X) <= l_A <= H(X)+2/m$,
  - where m is the length of the message.
  - Proved in the text book (Sayood, page 91).
- So, it is guaranteed that the encoding rate is close to the entropy, given a long enough message.

Compare Huffman Coding and Arithmetic Coding

- Huffman coding: Creates binary (Huffman) tree such that path lengths correspond to symbol probabilities. Uses path labels as encodings.
- Arithmetic coding: Combine probabilities of subsequent symbols into a single fixed-point high precision number. Encode that number in binary. Variable-to-fixed length encoding.
- Arithmetic coding is slower than Huffman coding.

Compare Huffman Coding and Arithmetic Coding (continued)

- Arithmetic coding efficiency:
  - $H(X) <= l_A <= H(X)+2/m$
  - m is the length of the message
- Huffman coding efficiency:
  - $H(S) <= l_H <= H(S) + 1/m$
  - m is size of the block
- Is Huffman coding more efficient than arithmetic coding?

Compare Huffman Coding and Arithmetic Coding (continued)

- It seems that Huffman coding is more efficient than the arithmetic coding.
  - However, in this case, the size of the codebook will be exponentially big, making Huffman encoding not practical.
- If the probabilities of the symbols are powers of two, Huffman coding can achieve the entropy bound. In this case, we cannot do any better with arithmetic coding, no matter how long a sequence we pick.

Compare Huffman Coding and Arithmetic Coding (continued)

- Also, Average code length for Huffman coding $l_H < H(S) + p_{max} + 0.086$
  - p_{max} is the probability of the most frequently occurring symbol.
  - If the alphabet size is relatively large and the probabilities are not too skewed, p_{max} will be generally small. In this case, the Huffman coding is better than the arithmetic coding in favor of the speed.
  - However, if the alphabet size is small, and the probabilities are highly unbalanced, arithmetic coding is generally worth the added complexity.

Compare Huffman Coding and Arithmetic Coding (continued)

- Arithmetic coding can handle adaptive coding without much increase in algorithm complexity. It calculates the probabilities on the fly and less primary memory is required for adaptation.
- Canonical Huffman is also fast but use only static or semi-static models.

Compare Huffman Coding and Arithmetic Coding (continued)

- It is not possible to start decoding in the middle of a compressed string which is possible in Huffman by indexing “starting points”.
- So, from random access point of view and from the point of view of compressed domain pattern matching, arithmetic coding is not suitable.

Compare Huffman Coding and Arithmetic Coding (continued)

- For text using static model, Huffman is almost as good as Arithmetic.
- Arithmetic is better suited for image and video compression.
- Once again, Huffman is faster than Arithmetic.
- Moffat's implementation (1998) is slightly better than Witten's (1987).

Ideal Arithmetic Coding ---- Remarks

- Theoretically, therefore, arithmetic code can achieve compression identical to the entropy bound. But, *finite precision* of computer limits the maximum compression achievable.
- Note, the algorithm does not output anything until encoding is completed.
- In practice, it is possible to output most significant digits sequentially during the execution while at the same time utilize the finite precision of the machine effectively.

Binary Implementation

- We need an arbitrary precision floating point arithmetic machine to implement the ideal schemes. In practice, we only have finite precision machines.
- Decoding cannot start until the data value *V* is communicated to the decoder.
  - Both problems will be fixed in the binary implementation.
  - As it turns out, arithmetic coding is best accomplished using standard 32 bit or 64 bit integer arithmetic. No floating point arithmetic is required, nor would it help to use it.
  - What is used instead is a an incremental transmission scheme, where an integer is formed with new bits in at the least significant end and shift them out from the most significant end, forming a single number that can be as many bits as the computer's storage will permit.

Ideal Arithmetic Coding ---- Underflow

- During the encoding, every time we read the next symbol, we scale the [L, L+R) to the new (smaller) value according to the probability of the symbol.
- Suppose RANGE is very small, such that different symbols will be mapped to the same interval [L, L+R). In this case, it is impossible for the decoder to recover the symbol correctly.
- Instead of decreasing RANGE, we can expand the interval [L, L+R).
- In the seminal implementation of arithmetic encoding, it is enforced that RANGE is always no less than 0.25 (Witten et al, 1987).
- The material here is based on Witten's implementation.

Binary Arithmetic Coding ---- Scaling

- Why do scaling?
  - Cope with the limited-precision of integer operations.
- When?
  - Whenever R<0.25
- How?
  - Shift the common prefix of {L, L+R} left
  - In each shifting we only process one bit.

The Common Prefix of {L, L+R}

- Use binary representation.
- Some facts:
  - All the values in interval [0, 0.01) has common prefix: 0.00
  - All the values in interval [0.01, 0.10) has common prefix: 0.01
  - All the values in interval [0.10, 0.11) has common prefix: 0.10
  - All the values in interval [0.11, 1.00) has common prefix: 0.11

Shifting one bit left

- If [L, L+R) is in [0.00, 0.10),
  - we have the common prefix 0.0
  - Shift a 0 left
  - E1 mapping
- If [L, L+R) is in [0.10, 1.00),
  - we have the common prefix 0.1
  - Shift a 1 left
  - E2 mapping
- If [L, L+R) straddles point 0.10
  - we may not have the common prefix
  - E3 mapping

Straddling Midpoint 0.10

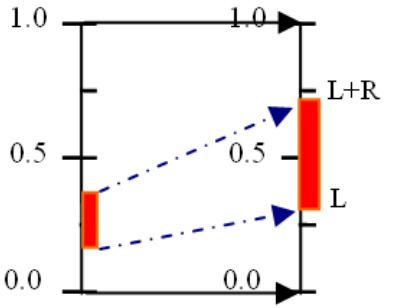
- If [L, L+R) straddles point 0.10
  - we may not have the common prefix
- However, any value bigger than 0.10 and close to 0.10 has form $0.1\{0\}^n$, where parameter *n* gives the precision.
- Similarly, any value smaller than 0.10 and close to 0.10 has form $0.0\{1\}^n$, where parameter *n* gives the precision.

Straddling Midpoint 0.10

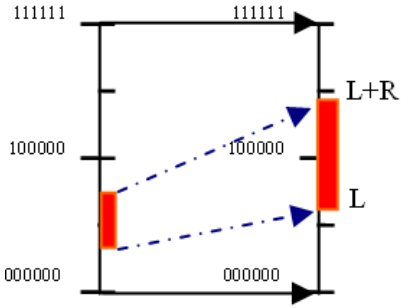
- After certain steps, E3 mapping will either be reduced to E1 mapping or E2 mapping, definitely.
- If it is reduced to E1 mapping, L and L+R have the common prefix $0.0\{1\}^n$, the output is a 0 followed by *n* 1s.
- Similarly, If E3 mapping is reduced to E2 mapping, L and L+R have the common prefix $0.1\{0\}^n$, the output is a 1 followed by *n* 0s.

Binary Arithmetic Coding
---- **E1** Mapping

- When the interval [L, L+R) lies in the lower half [0.0, 0.5), we can expand this lower half to make it occupy the full interval [0.0, 1.0)
- And adjust
  - LOW = 2 * LOW
  - RANGE = 2 * RANGE.

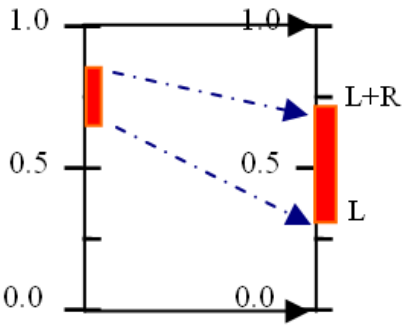


a)

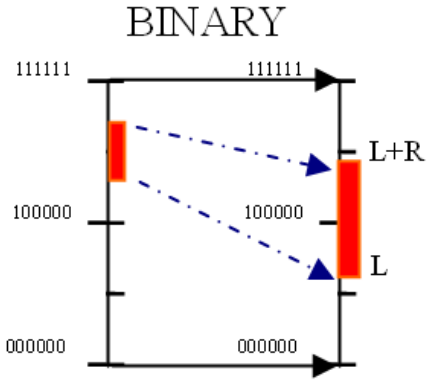


Binary Arithmetic Coding
---- **E2** Mapping

- When the interval [L, L+R) lies in the upper half, we can expand this upper half to make it occupy the full interval [0.0, 1.0),
- and adjust
  - LOW = 2 * (LOW – 0.5)
  - RANGE = 2 * RANGE.

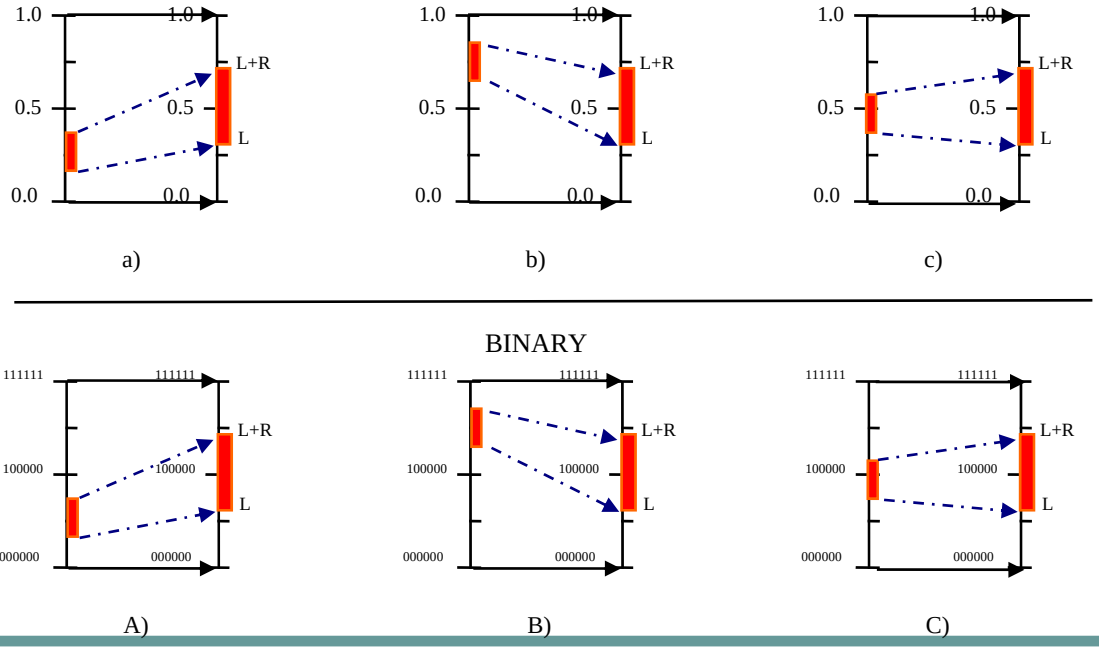


b)



Binary Arithmetic Coding ---- Basic Idea

- Shift the L and R left whenever L and L+R have the same prefix.
- When we shift out (and output) the prefix, the range should be re-normalized by shifting the LOW and double the RANGE which must straddle the midpoint.



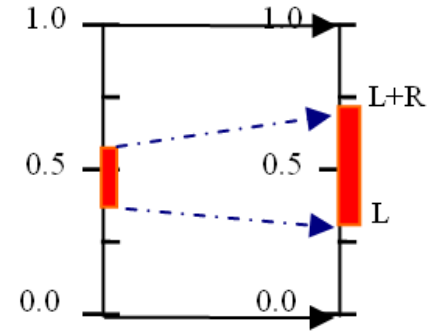
E3 Mapping

- When the interval [L, L+R) straddles the middle point 0.5, we cannot decide which bit (0, or 1) should be output only based on the current information. “The polarity of the immediately next output bit cannot be known, as it depends upon future symbols that have not yet been coded. What is known is that the bit after that immediately next bit will be of opposite polarity to the next bit, because all binary numbers in the range (0.25, 0.75) start either with [01] or [10]. Hence, in this case, the renormalization can still take place, provided a note is made using the variable *bits_outstanding* to output an additional opposite bit the next time a bit of unambiguous polarity is produced. In this case, L is translated by 0.25 before L and R are doubled. So, we adjust LOW = 2 * (LOW – 0.25) and RANGE = 2 * RANGE.”

For example, if the next bit turns out to be zero (i.e., the subinterval [LOW, LOW+RANGE) lies in [0.0, 0.5) --- [0.25, 0.5), more specifically --- and [0, 0.5) is expanded to [0.0, 1)), the bit after that will be one, since the subinterval has to be above the midpoint of the expanded interval. Conversely, if the next bit happens to be one, the one after that will be zero. Therefore the interval can safely be expanded right now, if only we remember that, whatever bit actually comes next, its opposite must be transmitted afterwards as well. Variable bits_outstanding is used to denote that the bit that is output next must be followed by an opposite bit.

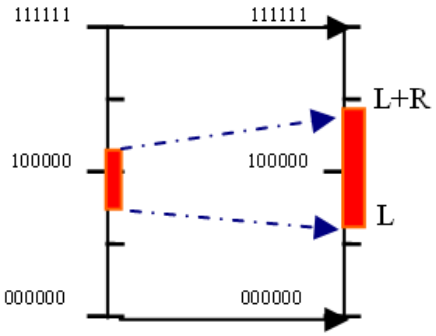
Binary Arithmetic Coding
---- **E3** Mapping

- When the interval [L, L+R) straddles the middle point 0.5, L is translated by 0.25 before L and R are doubled.
- So, we adjust
  - LOW = 2 * (LOW – 0.25)
  - RANGE = 2 * RANGE.



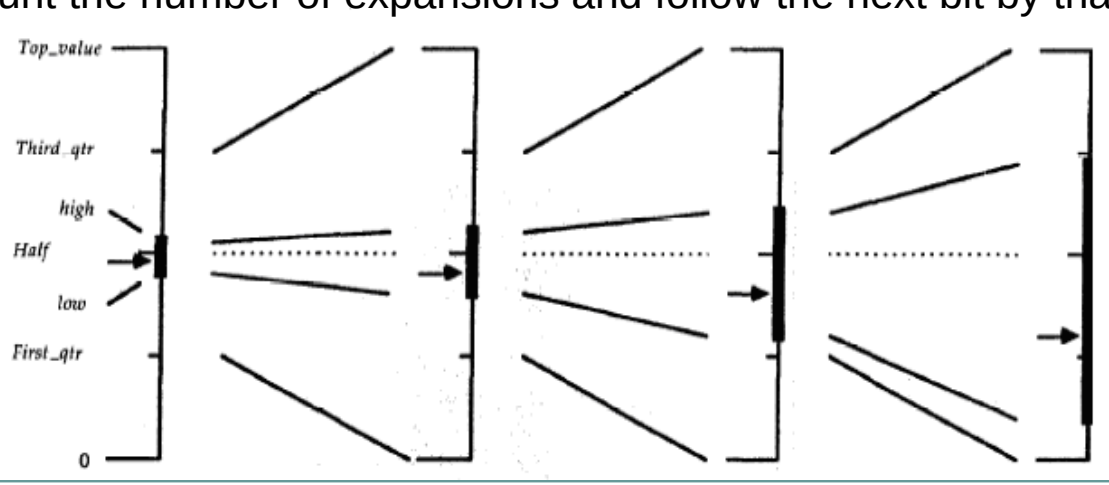
c)

- But, what to output?
- For example,
  - Output 0[1]^n, if in interval [0.25, 0.50)
  - Output 1[0]^n, if in interval [0.5, 0.75)
  - But, the new scaled interval may straddle 0.5 again?



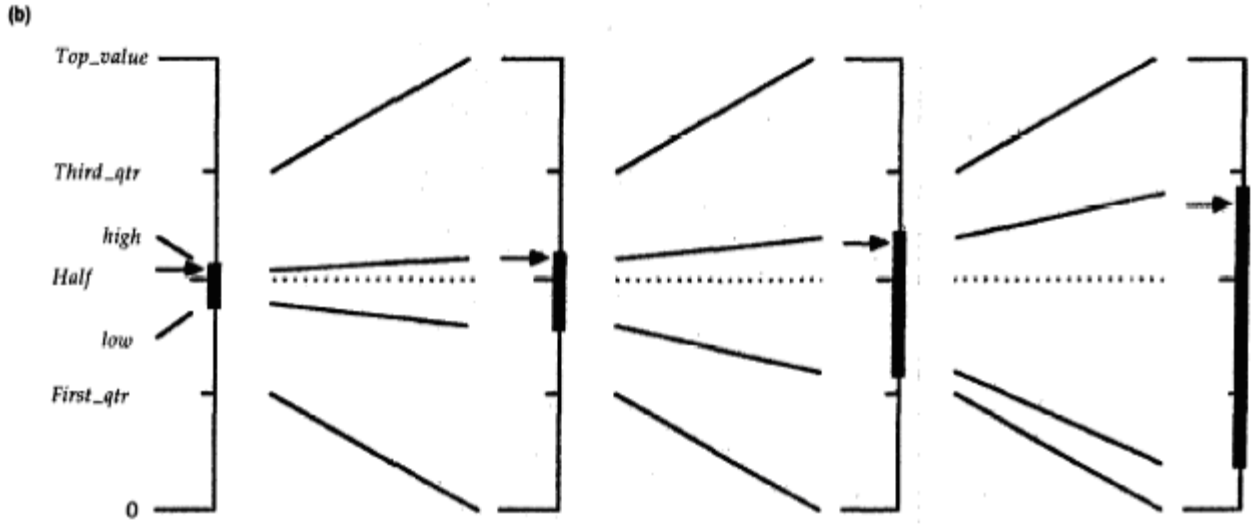
E3 Mapping, new scaled interval straddles 0.5 again

- One bit is output for each scaling
- But what if, after this operation, it is still true that the interval [L, L+R) straddles the middle point 0.5 again? Suppose the current subinterval has been expanded a total of three times. Suppose the next bit turns out to be zero, which means it lies in [0.0, 0.5). Then the next three bits will be ones, since the arrow is not only in the top half of the bottom half of the original range --- [0.25, 0.5), with binary encoding starting with 01, more specifically ---, but in the top quarter--- [0.375, 0.5), with binary encoding starting with 011, more specifically ---, and moreover the top eighth--- [0.4375, 0.5), with binary encoding starting with 0111, more specifically ---, of that half-this is why the expansion can occur three times. Similarly, as Figure (b) shows, if the next bit turns out to be a one, it will be followed by three zeros. Consequently, we need only count the number of expansions and follow the next bit by that number of opposites.



E3 Mapping, new scaled interval straddles 0.5 again

- Suppose the current subinterval has been expanded a total of three times
- Similarly, if the next bit turns out to be a one, it will be followed by three zeros. Consequently, we need only count the number of expansions and follow the next bit by that number of opposites.



Implementation

Both *L* and R are taken to be integers of some fixed number of bits *b.* Typical values of *b* are 32 or 64 and $0<=L,R<2^{b.}$ The actual values are assumed to be fractions normalized by 2^{b} so that they lie in the range 0 and 1.The next slide shows the integer values and the fractions. The algorithm maintains a loop whose loop invariant is $R>2^{b-2}$ *which corresponds to fraction 0.25* in scaled terms.

Binary Arithmetic Coding -----Pseudo Code

Both *L* and R are taken to be integers of some fixed number of bits *b.* Typical values of *b* are 32 or 64 and $0<=L,R<2^b$. The actual values are assumed to be fractions normalized by 2b so that they lie in the range 0and 1.The next slide shows the integer values and the fractions. The algorithm maintains a loop whose loop invariant is $R>2^{b-2}$ *which corresponds to fraction 0.25* in scaled terms. The corresponding values for arithmetic coding, real-number interpretation and scaled integer interpretations.

Decimal Encoding	1.00	0.50	0.25	0.0
FULL Binary Encoding	$0.\overbrace{1111\cdots111}^{32\ 1s}$	$0.\overbrace{1000\cdots000}^{31\ 0s}$	$0.\overbrace{010000\cdots000}^{30\ 0s}$	$0.\overbrace{0000\cdots000}^{32\ 0s}$
SHORT-FORM Binary Encoding	$\overbrace{1111\cdots111}^{32\ 1s}$	$\overbrace{1000\cdots000}^{31\ 0s}$	$\overbrace{010000\cdots000}^{30\ 0s}$	$\overbrace{0000\cdots000}^{32\ 0s}$

Encoding One Symbol

- The algorithm given in next page encodes one symbol *s.* It passes three parameters to the program l, h and t which define the position of the symbol in the probability range. As a pre-processing step, the probabilities are computed by pre-scanning the message *M* and accumulating the frequency count of each symbol (the un-normalized self-frequencies in *M* of the jth symbol *P*[*j*], for all j). More formally,

$$l=\sum_{j=1}^{s-1} P[j], \quad h=l+P[s], \quad t=\sum_{j=1}^{n} p[j]=m$$

- For example, given the message *M* = [*a, b, a, a, a, e, a, a, a, a*] (*m*=|*M*|=10) on alphabet (*a,b,c,d,e,f*) with size *n* = 6. We collect the frequency distribution P = [8, 1, 0, 0, 1, 0], and PC = [0, 8, 9, 9, 9, 10, 10] (notice that PC[0] is not used). When encoding the first symbol '*a*', we pass *l*=0, *h*=8 and *t*=10. When encoding the second symbol, we pass *l*=8, *h*=9 and *t*=10 and so on.
- Note the algorithm also computes the variable value *bits_outstanding* which is used by the program in the following page to generate the actual bits.

Binary Arithmetic Coding ----Encoding One Symbol

```
/*Arithmetically encode the range [l/t, h/t) using fixed-precision integer arithmetic. The state variables L
and R are modified to reflect the new range, and then renormalized to store the initial and final invariants
2^(b-2)<R<=2^(b-1), 0<=L<2^b-2^(b-2), L+R<=2^b (or, 0.25<realR<=0.50, 0<= realL<0.75, realL+realR<=1.00)*/

ALGORITHM ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t)
    1.  set  L = L + R * l / t
    2.  set  R = R * h / t – R * l / t
    3.  WHILE R <= 2^(b-2) DO   //Normalization is done only if range falls below 0.25. Otherwise do nothing//
    4.      /* renormalize R, adjust L, and output one bit */
    5.      IF L + R <= 2^(b-1) THEN   //Range is in lower half (0-0.5); output '0'//
    6.          bit_plus_follow(0)
    7.      ELSE IF 2^(b-1) <= L THEN
    8.          bit_plus_follow(1)   // Range is in upper half; output '1'; Shift to lower half//
    9.          set L = L – 2^(b-1)   /* clear the leftmost one */
    10.     ELSE
    11.         set bits_outstanding = bits_outstanding + 1
    12.         set L = L – 2^(b-2)   //Range straddles the midpoint; Shift by 0.25 //
    13.     END IF
    14.     set L = 2 * L and R = 2 * R   /* shift L and R left by one bit */
    15. END WHILE
END ALGORITHM
```

Binary Arithmetic Coding ----E3, Output Outstanding Bits

*/*Write the bit x (value 0 or 1) to the output bit stream, plus any outstanding following bits, which are known to be of opposite polarity.*/*

```
ALGORITHM bit_plus_follow(x)
    1.  put_one_bit(x)
    2.  WHILE bits_outstanding > 0 DO
    3.          put_one_bit(1-x)
    4.          set bits_outstanding = bits_outstanding -1
    5.  END WHILE
END ALGORITHM
```

Encoding the Message

In order to encode a message, the encoding one symbol routine has to be called *m* times. Before that the encoder must have all the necessary 'prelude' viz. symbol frequency count, alphabet size, the message size , the frequency distribution and cumulative frequency distribution . The code for all these tasks are rather straightforward.

The next two algorithms shown in succeeding pages does the reverse operations - decoding one symbol and decoding the entire message given the value *V* received by the decoder from the encoder. The concatenation of the bits generated by the routine *bit_plus_follow(x)* in the encoder is creating this value.

Binary Arithmetic Coding ----Encoding the Message

```
/*Use an arithmetic code to represent the n-symbol message M, where 1 <=M[i] <= n for 1<=i <=|M|*/

ALGORITHM ARITHMETIC_ENCODE_SEQUENCE(M)
    1.  /* 1-14: calculate the cumulative frequency */
    2.  FOR s = 0 to n DO
    3.      set PC[s] = 0;
    4.  END FOR
    5.  FOR i = 1 to |M| DO
    6.      set s = M[i]
    7.      PC[s]++;
    8.  END FOR
    9.  encode and transmit |M| and n
    10. FOR s = 1 to n DO
    11.     encode and transmit (1+PC[s]);
    12.     PC[s] = PC[s-1] + PC[s]
    13. END FOR
    14.
    15. /* 16: init for encoding */
    16. set L = 0, R = 2^(b-1) (corresponding to 0.5), bits_outstanding = 0
    17.
    18. /* 19-23: encode the message */
    19. FOR i = 1 to |M| DO
    20.     set s = M[i]
    21.     l = PC[s-1], h = PC[s], t = |M|
    22.     ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t)
    23. END FOR
    24.
    25. encode and transmit L as one integer (b bits long) using bit_plus_follow
END ALGORITHM
```

Binary Arithmetic Coding ----Decoding One Symbol

```
/* Adjust the decoder's state variable L and R to reflect the changes made in the encoder during t
corresponding call to ARITHMETIC_ENCODE_SEQUENCE( )*/

ALGORITHM ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t)
1.    set L = L + R * l / t
2.    set   R = R * h / t – R * l / t
3.    WHILE R <= 2^(b-2) DO
4.          /* Renormalize R, adjust L and V, and shift the next input bit into V */
5.          IF L + R <= 2^(b-1) THEN
6.                /* do nothing */
7.          ELSE IF 2^(b-1) <= L THEN
8.                set L = L – 2^(b-1)    /* clear the leftmost one */
9.                set V= V – 2^(b-1)     /* clear the leftmost one */
10.         ELSE
11.               set L = L – 2^(b-2)
12.               set V = V- 2^(b-2)
13.         END IF
14.         set L = 2 * L, R = 2 * R and V = 2 * V + get_one_bit()
15.   END WHILE
END ALGORITHM
```

//The three components of the "if" statement are for three cases: range in lower, Upper halves or straddling the midpoint. The code maps the range and the value V //

Binary Arithmetic Coding ----Decoding the Message

```
/*Decode and return an m-symbol message M using an arithmetic code.*/

ALGORITHM ARITHMETIC_DECODE_SEQUENCE( )
1.  /* 1-8: receive and decode the cumulative frequency */
2.  receive and decode m=|M| and n
3.  set PC[0] = 0
4.  FOR s = 1 to n DO
5.      receive and decode value pc
6.      set PC[s] = pc – 1;
7.      set PC[s] = PC[s-1] + PC[S]
8.  END FOR
9.
10. /* init for decoding */
11. set R = 2^(b-1)
12. set L = 0
13. set V = get_one_integer(b)
14.
15. /* receive and decode the message */
16. FOR i = 1 to |M| DO
17.     set target = decode_target(m);
18.     determine s such that PC[s-1] <= target < PC[s]
19.     l = PC[s-1], h = PC[s], t = m
20.     ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t)
21.     set M[i] = s
22. END FOR
23. RETURN M
END ALGORITHM
```

```
ALGORITHM decode_target(t)
1.  RETURN (((V – L + 1) * t) – 1) / R
END ALGORITHM
```

// Target gets the cumulative frequency of the symbol

// Finds the location of the symbol in in PC(s) domain //

//Update V //

// decoded output symbol is retrieved //

Binary Arithmetic Coding ----Example

- Now, we go through one example step-by-step to illustrate how to encode and decode a message.
- Given the message M = [a, b, a, a, a, e, a, a, a, a] (|M|=10) on alphabet , n = 6.
- Suppose b = 8, that is, the size of the integer is 8.
- **Encode the message**
- First, we call ALGORITHM ARITHMETIC_ENCODE_SEQUENCE(M). We collect the frequency distribution P = [8, 1, 0, 0, 1, 0], and PC = [0, 8, 9, 9, 9, 10, 10] (notice that PC[0] is not used).
- Next, we need to encode and transmit |M| and n as the bit stream: [0000 1010, 0000 0110]
- Next, we encode and transmit PCs, as the bit stream: [0000 1001, 0000 1010, 0000 1010, 0000 1010, 0000 1011, 0000 1011]
- Next, we set L = 0, R = 128, bits_outstanding = 0
- Next, we encode the message M.

Example (cntd.)

- i = 1, s = M[i] =a, l = 0, h = 8, t = 10. L = 0, R = 128. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 0, R = 102. Since R is not small enough (greater than 64), so no scaling adjust is necessary.
- i = 2, s = M[i] =b, l = 8, h = 9, t = 10. L = 0, R = 102. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 81, R = 10. Since R is small enough (less than 64), scaling is performed. First, the range lies in the lower part [0.0, 0.5), we output bit [0] to the output stream, shift L and R left by one bit, and have L = 162, R = 20. Now the range lies in the upper part [0.5, 1.0), we output bit [1] to the output stream, clear the leftmost bit of L, and have L = 34, shift L and R left by one bit, and have L = 68, R = 40. We do another scaling, the range in the lower part [0.0, 0.5), we output bit [0] to the output stream, shift L and R left by one bit, and have L = 136, R = 80.

Example (cntd.)

i = 3, s = M[i] =a, l = 0, h = 8, t = 10. L = 136, R = 80. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 136, R = 64. Since R is small enough (less equal to 64), scaling is performed. First, the range lies in the upper part [0.5, 1.0), we output bit [1] to the output stream, clear the leftmost bit of L, and have L = 8, shift L and R left by one bit, and have L = 16, R = 128. i = 3, s = M[i] =a, l = 0, h = 8, t = 10. L = 136, R = 80. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 136, R = 64. Since R is small enough (less equal to 64), scaling is performed. First, the range lies in the upper part [0.5, 1.0), we output bit [1] to the output stream, clear the leftmost bit of L, and have L = 8, shift L and R left by one bit, and have L = 16, R = 128.

- i = 4, s = M[i] =a, l = 0, h = 8, t = 10. L = 16, R = 128. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 16, R = 102.
- i = 5, s = M[i] =a, l = 0, h = 8, t = 10. L = 16, R = 102. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 16, R = 81.

Example (cntd.)

i = 6, s = M[i] =e, l = 9, h = 10, t = 10. L = 16, R = 81. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 88, R = 9. Since R is small enough (less than 64), scaling is performed. First, the range lies in the lower part [0.0, 0.5), we output bit [0] to the output stream, shift L and R left by one bit, and have L = 176, R = 18. Now the range lies in the upper part [0.5, 1.0), we output bit [1] to the output stream, clear the leftmost bit of L, and have L = 48, shift L and R left by one bit, and have L = 96, R = 36. The polarity of the immediately next output bit cannot be known, as it depends upon future symbols that have not yet been coded. What is known is that the bit after that immediately next bit will be of opposite polarity to the next bit, because all binary numbers in the range (0.25, 0.75) start either with [01] or [10]. Hence, in this case, the renormalization can still take place, provided a note is made using the variable *bits_outstanding* to output an additional opposite bit the next time a bit of unambiguous polarity is produced. In this case, L is translated by 0.25 before L and R are doubled. So, bits_outstanding = 1, L = 32, L = 64, R = 72.

Example (cntd.)

- i = 7, s = M[i] =a, l = 0, h = 8, t = 10. L = 64, R = 72. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 64, R = 57. Since R is small enough (less than 64), scaling is performed. First, the range lies in the lower part [0.0, 0.5), we output bit [0] to the output stream, - Because of the bits_outstanding, bit [1] is also output -shift L and R left by one bit, and have L = 128, R = 114.
- i = 8, s = M[i] =a, l = 0, h = 8, t = 10. L = 128, R = 114. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 128, R = 91.
- i = 9, s = M[i] =a, l = 0, h = 8, t = 10. L = 128, R = 91. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 128, R = 72.
- i = 10, s = M[i] =a, l = 0, h = 8, t = 10. L = 128, R = 72. We call ARITHMETIC_ENCODE_ONE_SYMBOL(l, h, t), L = 128, R = 57. Since R is small enough (less than 64), scaling is performed. First, the range lies in the upper part [0.5, 1.0), we output bit [1] to the output stream, clear the leftmost bit of L, and have L = 0, shift L and R left by one bit, and have L = 0, R = 114.

Example (cntd.)

- The last step, we need to encode and transmit L as one integer. We output bit stream [0000 0000].
- We output the bit stream:
- |M| and n as the bit stream: [0000 1010, 0000 0110]
- PCs, as the bit stream: [0000 1001, 0000 1010, 0000 1010, 0000 1010, 0000 1011, 0000 1011]
- i = 2, output [010]
- i = 3, output [1]
- i = 6, output [01]
- i = 7, output [01]
- i = 10, output [1]
- transmit L, output bit stream [0000 0000].
- So, the output bit stream is:
- [0000 1010, 0000 0110 0000 1001, 0000 1010, 0000 1010, 0000 1010, 0000 1011, 0000 1011 010 1 01 01 1 0000 0000].

Example (cntd.)

- **Decode the message**
- We have the bit stream [0000 1010, 0000 0110 0000 1001, 0000 1010, 0000 1010, 0000 1010, 0000 1011, 0000 1011 010 1 01 01 1 0000 0000].
- First, receive and decode m and n, m = 0x 0000 1010 = 10, n = 0x 0000 0110 = 6.
- Now, the bit stream is [0000 1001, 0000 1010, 0000 1010, 0000 1010, 0000 1011, 0000 1011 010 1 01 01 1 0000 0000].
- Next, receive and decode PCs, input stream is [0000 1001, 0000 1010, 0000 1010, 0000 1010, 0000 1011, 0000 1011], restored PC = [0, 8, 9, 9, 9, 10, 10]
- Now, the bit stream is [010 1 01 01 1 0000 0000].
- Next, we set R = 128, L = 0, get bit stream [010 1 01 01], V = 85
- Now, the bit stream is [1 0000 0000].
- Next, we decode the message M.

Example (cntd.)

- i = 1, R = 128, L = 0, V = 85, target = (((V – L + 1) * m) – 1) / R = 6, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =0, R = R * h / t – R * l / t = 102.
- Recover M[1]= a.
- i = 2, R = 102, L = 0, V = 85, target = (((V – L + 1) * m) – 1) / R = 8, s = 2, l = 8, h = 9, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =81, R = R * h / t – R * l / t = 10. Since R is small enough (less than 64), scaling is performed. First, the range lies in the lower part [0.0, 0.5), shift L and R left by one bit, and have L = 162, R = 20. Read one bit [1] from the stream, Now, the bit stream is [0000 0000]. V = 171. Now the range lies in the upper part [0.5, 1.0), we clear the leftmost bit of L and V, and have L = 34, V = 43, shift L and R left by one bit, and have L = 68, R = 40. Read one bit [0] from the stream, Now, the bit stream is [000 0000]. V = 86. We do another scaling, the range in the lower part [0.0, 0.5), shift L and R left by one bit, and have L = 136, R = 80. Read one bit [0] from the stream, Now, the bit stream is [00 0000]. V = 172.
- Recover M[2]= b.

Example (cntd.)

- i = 3, R = 80, L = 136, V = 172, target = (((V – L + 1) * m) – 1) / R = 4, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =136, R = R * h / t – R * l / t = 64. Since R is small enough (less equal to 64), scaling is performed. First, the range lies in the upper part [0.5, 1.0), we clear the leftmost bit of L and V, and have L = 8, V = 44, shift L and R left by one bit, and have L = 16, R = 128. Read one bit [0] from the stream, Now, the bit stream is [0 0000]. V = 88.
- Recover M[3]= a.
- i = 4, R = 128, L = 16, V = 88, target = (((V – L + 1) * m) – 1) / R = 5, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =16, R = R * h / t – R * l / t = 102.
- Recover M[4]= a.
- i = 5, R = 102, L = 16, V = 88, target = (((V – L + 1) * m) – 1) / R = 7, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =16, R = R * h / t – R * l / t = 81.
- Recover M[5]= a.

Example (cntd.)

- i = 6, R = 81, L = 16, V = 88, target = (((V – L + 1) * m) – 1) / R = 9, s = 5, l = 9, h = 10, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =88, R = R * h / t – R * l / t = 9. Since R is small enough (less equal to 64), scaling is performed. First, the range lies in the lower part [0.0, 0.5), shift L and R left by one bit, and have L = 176, R = 18. Read one bit [0] from the stream, Now, the bit stream is [0000]. V = 176. Since R is small enough (less than 64), scaling is performed. First, the range lies in the upper part [0.5, 1.0), we clear the leftmost bit of L and V, and have L = 48, V = 48, shift L and R left by one bit, and have L = 96, R = 32. Read one bit [0] from the stream, Now, the bit stream is [000]. V = 96. Since R is small enough (less than 64), scaling is performed. **In this case, the interval straddles 0.5.** We adjust L and V, and have L = 32, V = 32, shift L and R left by one bit, and have L = 64, R = 64. Read one bit [0] from the stream, Now, the bit stream is [00]. V = 64. Since R is small enough (equal to 64), scaling is performed. First, the range lies in the lower part [0.0, 0.5), shift L and R left by one bit, and have L = 128, R = 128. Read one bit [0] from the stream, Now, the bit stream is [0]. V = 128
- Recover M[6]= e.

Example (cntd.)

- i = 7, R = 128, L = 128, V = 128, target = (((V – L + 1) * m) – 1) / R = 0, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =128, R = R * h / t – R * l / t = 102.
- Recover M[7]= a.
- i = 8, R = 102, L = 128, V = 128, target = (((V – L + 1) * m) – 1) / R = 0, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =128, R = R * h / t – R * l / t = 81.
- Recover M[8]= a.
- i = 9, R = 81, L = 128, V = 128, target = (((V – L + 1) * m) – 1) / R = 0, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =128, R = R * h / t – R * l / t = 64.
- Recover M[9]= a.
- i = 10, R = 64, L = 128, V = 128, target = (((V – L + 1) * m) – 1) / R = 0, s = 1, l = 0, h = 8, t = 10, call ARITHMETIC_DECODE_ONE_SYMBOL(l, h, t), L = L + R * l / t =128, R = R * h / t – R * l / t = 51.
- Recover M[10]= a.
- So, we have reconstructed the message M= abaaaeaaaa
- One limitation of this arithmetic coding scheme presented here is that it is static. We must collect the frequency information before the coding. The algorithm cannot adjust the frequency information on the fly.

Application Arithmetic Coding

- Image compression
- Video compression
- Lossless/lossy
- Why?
  - The size of the alphabet is small, and the probabilities are highly unbalanced.