4.4. Arithmetic coding

Advantages:

- Reaches the entropy (within computing precision)
- Superior to Huffman coding for small alphabets and skewed distributions
- Clean separation of modelling and coding
- Suits well for adaptive one-pass compression
- Computationally efficient

History:

- Original ideas by Shannon and Elias
- Actually discovered in 1976 (Pasco; Rissanen)

Arithmetic coding (cont.)

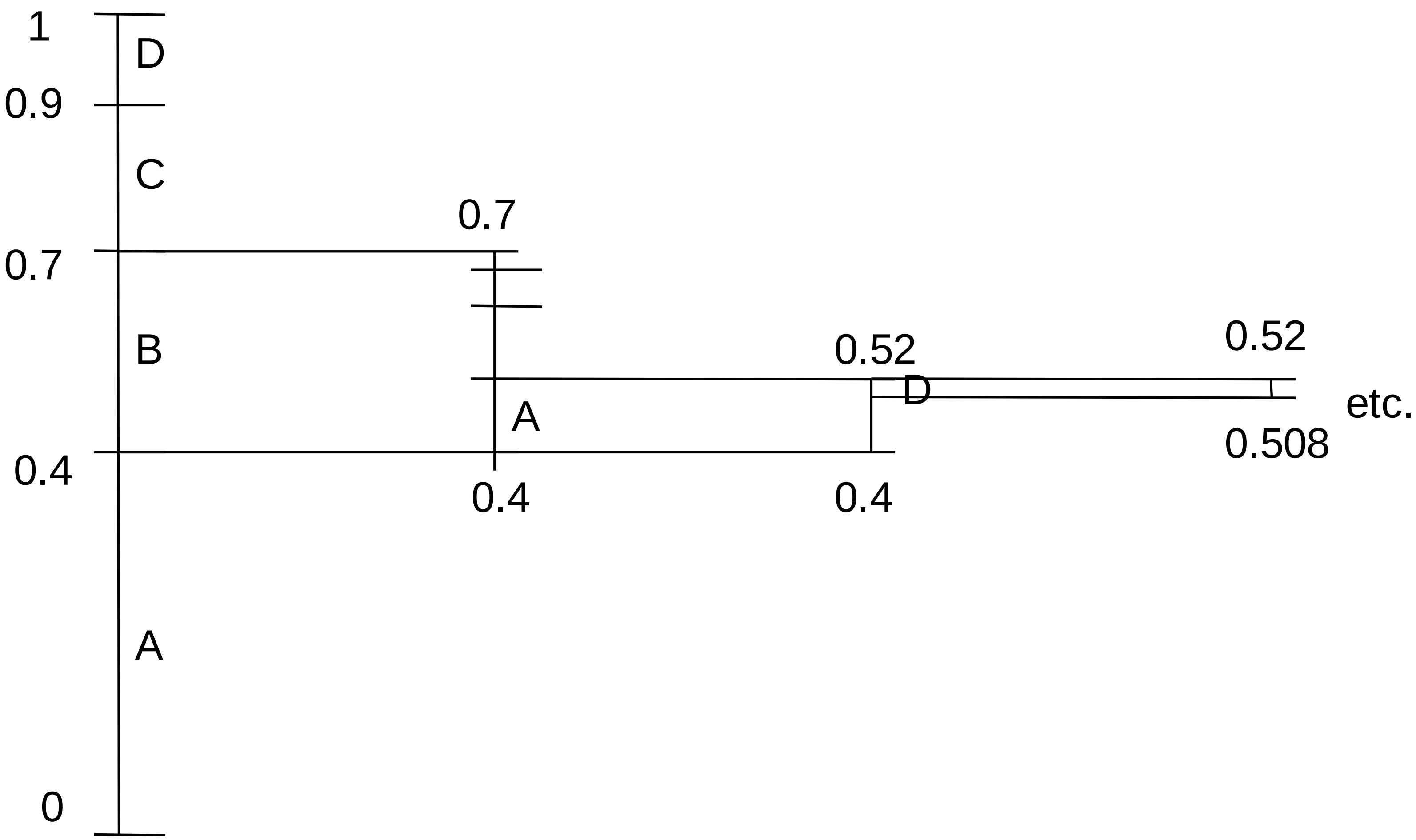
Characterization:

- One codeword for the whole message
- A kind of extreme case of extended Huffman (or Tunstall) coding
- No codebook required
- No clear correspondence between source symbols and code bits

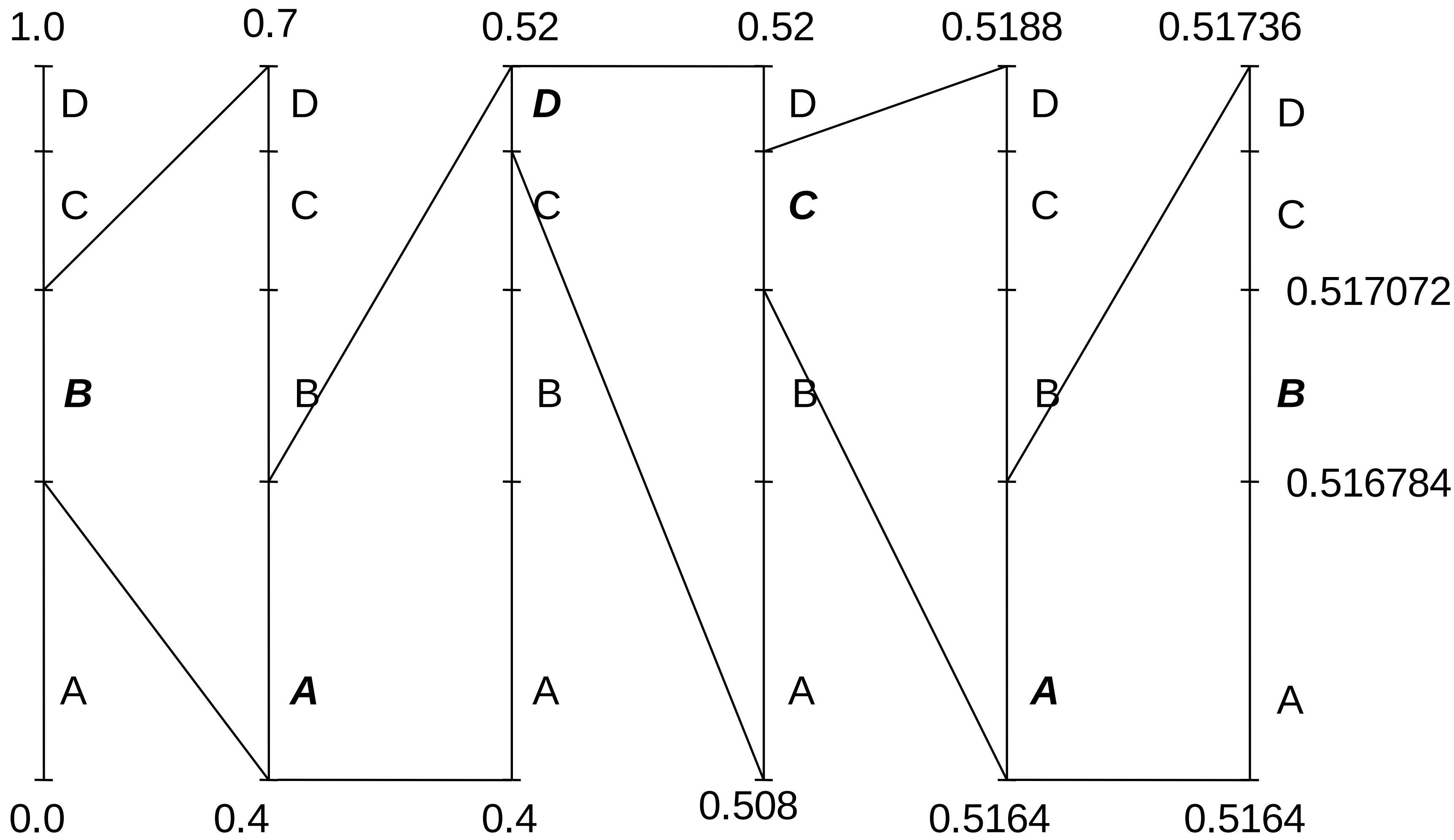
Basic ideas:

- Message is represented by a (small) interval in [0, 1)
- Each successive symbol reduces the interval size
- Interval size = product of symbol probabilities
- Prefix-free messages result in disjoint intervals
- Final code = any value from the interval
- Decoder computes the same sequence of intervals

Arithmetic coding: Encoding of ”BADCAB”



Encoding of ”BADCAB” with rescaled intervals



Algorithm: Arithmetic encoding

***Input*:** Sequence $x = x_i$, i=1, ..., n; probabilities p_1, ..., p_q of symbols 1, ..., q.
***Output*:** Real value between [0, 1) that represents X.

```
begin
    cum[0] := 0
    for i := 1 to q do cum[i] := cum[i–1] + p_i
    lower := 0.0
    upper := 1.0
    for i := 1 to n do
    begin range := upper – lower
            upper := lower + range ∗ cum[x_i]
            lower := lower + range ∗ cum[x_i–1]
    end
    return (lower + upper) / 2
end
```

Algorithm: Arithmetic decoding

Input: v: Encoded real value; n: number of symbols to be decoded; probabilities $p_1, ..., p_q$ of symbols $1, ..., q$.

Output: Decoded sequence x.

begin

$cum[1] := p1$

for $i := 2$ **to** q **do** $cum[i] := cum[i-1] + p_i$

$lower := 0.0$

$upper := 1.0$

for $i := 1$ **to** n **do**

begin $range := upper - lower$

$z := (v - lower) / range$

Find j such that $cum[j-1] \leq z < cum[j]$

$x_i := j$

$upper := lower + range * cum[j]$

$lower := lower + range * cum[j-1]$

end

return $x = x_1, ..., x_n$

end

Arithmetic coding (cont.)

Practical problems to be solved:

- Arbitrary-precision real arithmetic
- The whole message must be processed before the first bit is transferred and decoded.
- The decoder needs the length of the message

Representation of the final binary code:

- Midpoint between *lower* and *upper* ends of the final interval.
- Sufficient number of significant bits, to make a distinction from both *lower* and *upper*.
- The code is prefix-free among prefix-free messages.

Example of code length selection

midpoint ≠ lower and upper

- *upper*: 0.517072 = .100001000101***1***1101...
- *midpoint*: 0.516928 = .10000100010***10***1010...
- *lower*: 0.516784 = .10000100010***01***0111...

13 bits

range = 0.00028

$\log_2(1/\text{range}) \approx 11.76$ bits

Another source message

"ABCDABCABA"

- *Precise* probabilities:

  P(A) = 0.4, P(B) = 0.3, P(C) = 0.2, P(D) = 0.1

- Final range length:

  $0.4 \cdot 0.3 \cdot 0.2 \cdot 0.1 \cdot 0.4 \cdot 0.3 \cdot 0.2 \cdot 0.4 \cdot 0.3 \cdot 0.4 =$

  $0.4^4 \cdot 0.3^3 \cdot 0.2^2 \cdot 0.1 = 0.000002764$

  $-\log_2 0.000002764 \approx 18.46 =$ *entropy*

Arithmetic coding: Basic theorem

Theorem 4.2.
Let *range* = *upper* – *lower* be the final probability interval in Algorithm 4.8. The binary representation of *mid* = (*upper* + *lower*) / 2 truncated to $l(x) = \lceil \log_2(1/range) \rceil + 1$ bits is a uniquely decodable code for message *x* among prefix-free messages.

Proof: Skipped.

Optimality

Expected length of an *n*-symbol message *x*:

$$L^{(n)} = \sum P(x) l(x)$$
$$= \sum P(x)\left[\left\lceil \log_2 \frac{1}{P(x)} \right\rceil + 1\right]$$
$$\le \sum P(x)\left[\log_2 \frac{1}{P(x)} + 2\right]$$
$$= \sum P(x) \log_2 \frac{1}{P(x)} + 2\sum P(x)$$
$$= H(S^{(n)}) + 2$$

Bits per symbol:

$$\frac{H(x^{(n)})}{n} \le L \le \frac{H(x^{(n)})}{n} + \frac{2}{n}$$
$$H(S) \le L \le H(S) + \frac{2}{n}$$

Ending problem

- The above theorem holds only for prefix-free messages.
- The ranges of a message and its prefix overlap, and may result in the same code value.
- How to distinguish between “VIRTA” and “VIRTANEN”?
- Solutions:
  - Transmit the *length* of the message before the message itself: “5VIRTA” and “8VIRTANEN”.
    This is not good for online applications.
  - Use a special end-of-message symbol, with prob = 1/*n* where *n* is an *estimated* length of the message.
    Good solution unless *n* is totally wrong.

Arithmetic coding: Incremental transmission

- Bits are sent as soon as they are known.
- Decoder can start well before the encoder has finished.
- The interval is *scaled* (*zoomed*) for each output bit: Multiplication by 2 means shifting the binary point one position to the right:

$$\begin{array}{ll} \textit{upper}: & 0.\boxed{0}11010\ldots \\ \textit{lower}: & 0.\boxed{0}01101\ldots \end{array} \quad \Rightarrow \quad \begin{array}{l} 0.11010\ldots \\ 0.01101\ldots \end{array} \quad \text{and transmit 0}$$

$$\begin{array}{ll} \textit{upper}: & 0.\boxed{1}10100\ldots \\ \textit{lower}: & 0.\boxed{1}00011\ldots \end{array} \quad \Rightarrow \quad \begin{array}{l} 0.10100\ldots \\ 0.00011\ldots \end{array} \quad \text{and transmit 1}$$

- **Note:** The common bit also in midpoint value.

Arithmetic coding: Scaling situations

// Number p of pending bits initialized to 0

***upper* < 0.5:**

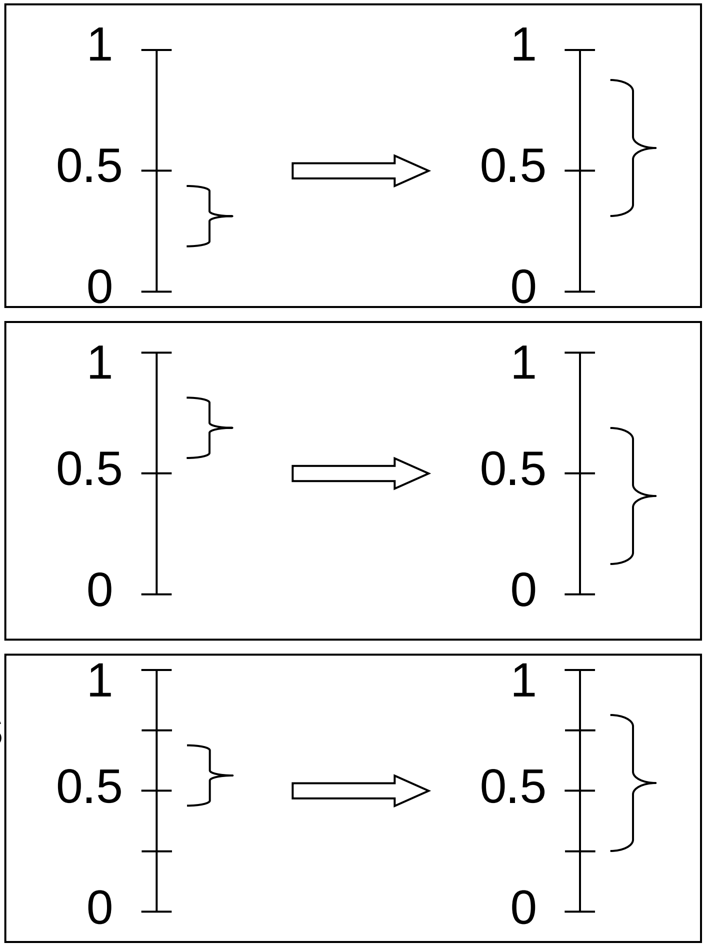
- transmit bit 0 (plus *p* pending 1's)
- *lower* := 2 · *lower*
- *upper* := 2 · *upper*

***lower* > 0.5**

- transmit bit 1 (plus *p* pending 0's)
- *lower* := 2 · (*lower* – 0.5)
- *upper* := 2 · (*upper* – 0.5)

***lower* > 0.25 and *upper* < 0.75:**

- Add one to the number *p* of pending bits
- *lower* = 2 · (*lower* – 0.25)
- *upper* = 2 · (*upper* – 0.25)



Decoder operation

- Reads a sufficient number of bits to determine the first symbol (unique interval of cumulative probabilities).
- Imitates the encoder: performs the same scalings, after the symbol is determined
- Scalings drop the ‘used’ bits, and new ones are read in.
- No pending bits.

Implementation with integer arithmetic

- Use symbol frequencies instead of probabilities
- Replace [0, 1) by $[0, 2^{k}-1)$
- Replace 0.5 by $2^{k\text{-}1}-1$
- Replace 0.25 by $2^{k\text{-}2}-1$
- Replace 0.75 by $3\cdot 2^{k\text{-}2}-1$

Formulas for computing the next interval:

- *upper* := *lower* + (*range* · *cum*[*symbol*] / *total_freq*) – 1
- *lower* := *lower* + (*range* · *cum*[*symbol*–1] / *total_freq*)

Avoidance of overflow: *range* · *cum*() < $2^{wordsize}$

Avoidance of underflow: *range* > *total_frequency*

Solution to avoiding over-/underflow

- Due to scaling, *range* is always > $2^{k\text{-}2}$
- Both overflow and underflow are avoided, if *total_freq* < $2^{k\text{-}2}$, and $2k–2 \leq w$ = machine word

Suggestion:

- Present *total_freq* with max 14 bits, *range* with 16 bits

Formula for decoding a symbol *x* from a *k*-bit *value*:

$$cum(x-1) \leq \left\lfloor \frac{(value - lower + 1)\cdot total\_freq - 1}{upper - lower + 1} \right\rfloor < cum(x)$$

4.4.1. Adaptive arithmetic coding

Advantage of arithmetic coding:

- Used probability distribution can be changed at any time, but synchronously in the encoder and decoder.

Adaptation:

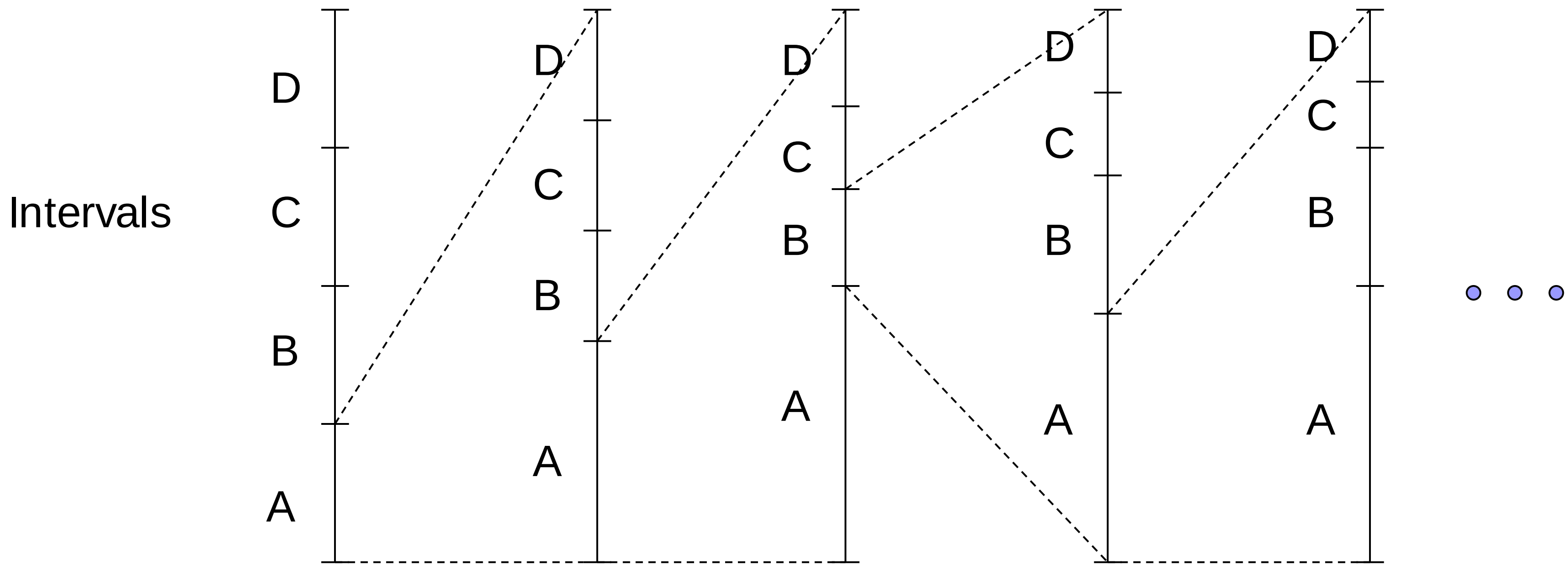
- Maintain frequencies of symbols during the coding
- Use the current frequencies in reducing the interval

Initial model; alternative choices:

- All symbols have an initial frequency = 1.
- Use a placeholder (NYT = Not Yet Transmitted) for the unseen symbols, move symbols to active alphabet at the first occurrence.

Basic idea of adaptive arithmetic coding

- Alphabet: {A, B, C, D}
- Message to be coded: “AABAAB …”



Frequencies	{1,1,1,1}	{2,1,1,1}	{3,1,1,1}	{3,2,1,1}	{4,2,1,1}
Interval size	1	1/4	1/10	1/60	3/420

Adaptive arithmetic coding (cont.)

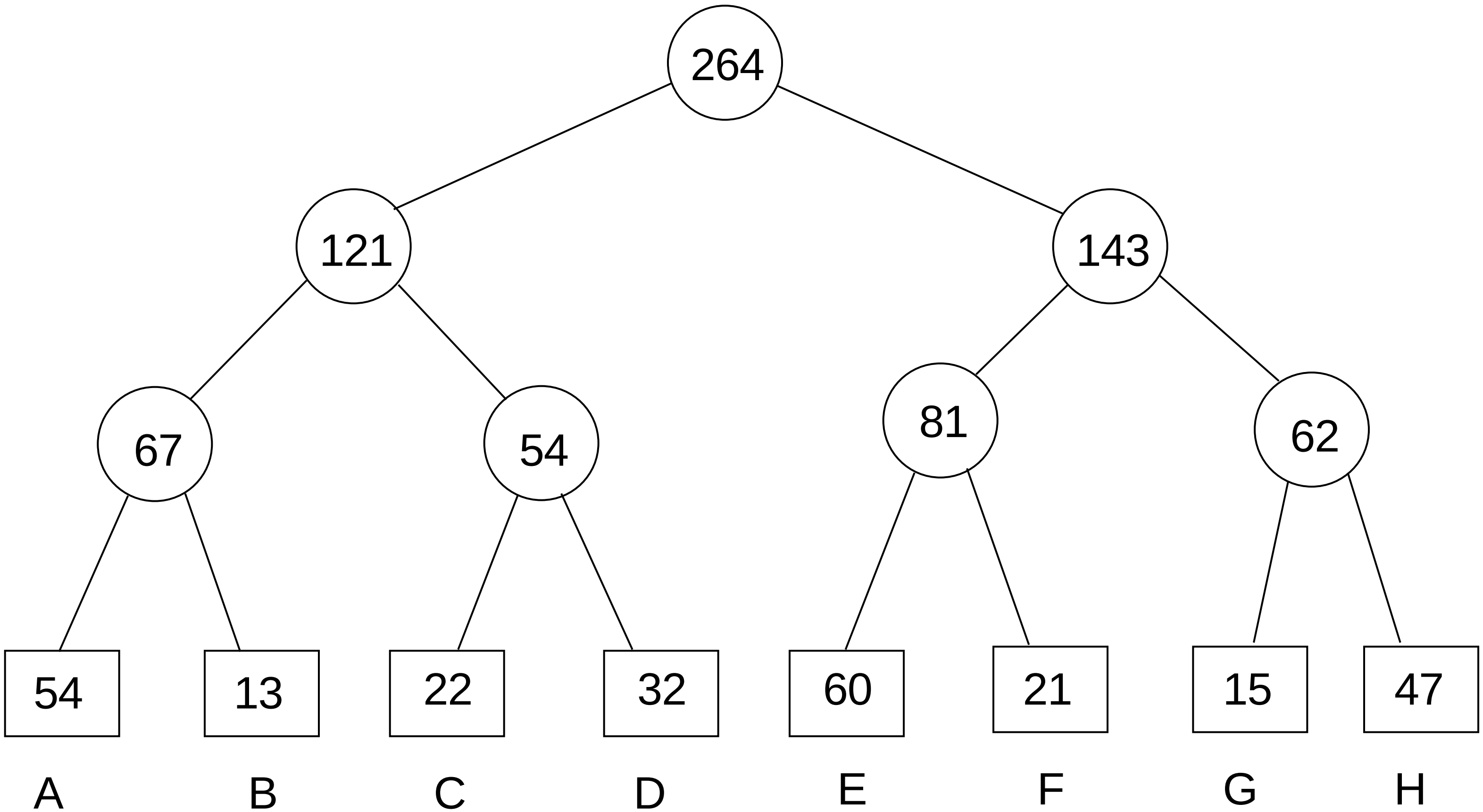
Biggest problem:

- Maintenance of cumulative frequencies; simple vector implementation has complexity $O(q)$ for q symbols.

General solution:

- Maintain partial sums in an explicit or implicit binary tree structure.
- Complexity is $O(\log_2 q)$ for both search and update

Tree of partial sums



Implicit tree of partial sums

1	2	3	4	5	6	7	8
f	*f*1+*f*2	*f*3	*f*1+...+*f*4	*f*5	*f*5+*f*6	*f*7	*f*1+...+*f*8

9	10	11	12	13	14	15	16
*f*9	*f*9+*f*10	*f*11	*f*9+...+*f*12	*f*13	*f*13+*f*14	*f*15	*f*1+...+*f*16

Correct indices are obtained by bit-level operations.

4.4.2. Arithmetic coding for a binary alphabet

Observations:

- Arithmetic coding works as well for any size of alphabet, contrary to Huffman coding.
- Binary alphabet is especially easy: *No cumulative probability table*.

Applications:

- Compression of black-and-white images
- Any source, interpreted bitwise

Speed enhancement:

- Avoid multiplications
- Approximations cause additional redundancy

Arithmetic coding for binary alphabet (cont.)

Note:

- Scaling operations need only multiplication by two, implemented as shift-left.
- Multiplications appearing in reducing the intervals are the problem.

Convention:

- **MPS** = More Probable Symbol
- **LPS** = Less Probable Symbol
- The correspondence to actual symbols may change locally during the coding.

Skew coder (Langdon & Rissanen)

- **Idea**: approximate the probability p of LPS by $1/2^Q$ for some integer $Q > 0$.
- Choose LPS to be the first symbol of the alphabet (can be done without restriction)
- Calculating the new *range*:
  - For LPS: $range \leftarrow range >> Q;$
  - For MPS: $range \leftarrow range - (range >> Q);$
- Approximation causes some redundancy
- Average number of bits per symbol (p = exact prob):

$$pQ - (1-p)\log_2(1 - \frac{1}{2^Q})$$

Solving the 'breakpoint' probability $\hat{p}$

- Choose Q to be either r or r+1, where $r = \lfloor -\log_2 p \rfloor$
- Equate the bit counts for rounding down and up:

$$\hat{p}r - (1-\hat{p})\log_2(1-\frac{1}{2^r}) = \hat{p}(r+1) - (1-\hat{p})\log_2(1-\frac{1}{2^{r+1}})$$

which gives

$$\hat{p} = \frac{z}{1+z} \quad \text{where} \quad z = \log_2 \frac{1-1/2^{r+1}}{1-1/2^r}$$

Skew coder (cont.)

Probability approximation table:

Probability range	Q	Effective probability
0.3690 – 0.5000	1	0.5
0.1820 – 0.3690	2	0.25
0.0905 – 0.1820	3	0.125
0.0452 – 0.0905	4	0.0625
0.0226 – 0.0452	5	0.03125
0.0113 – 0.0226	6	0.015625

Proportional compression efficiency:

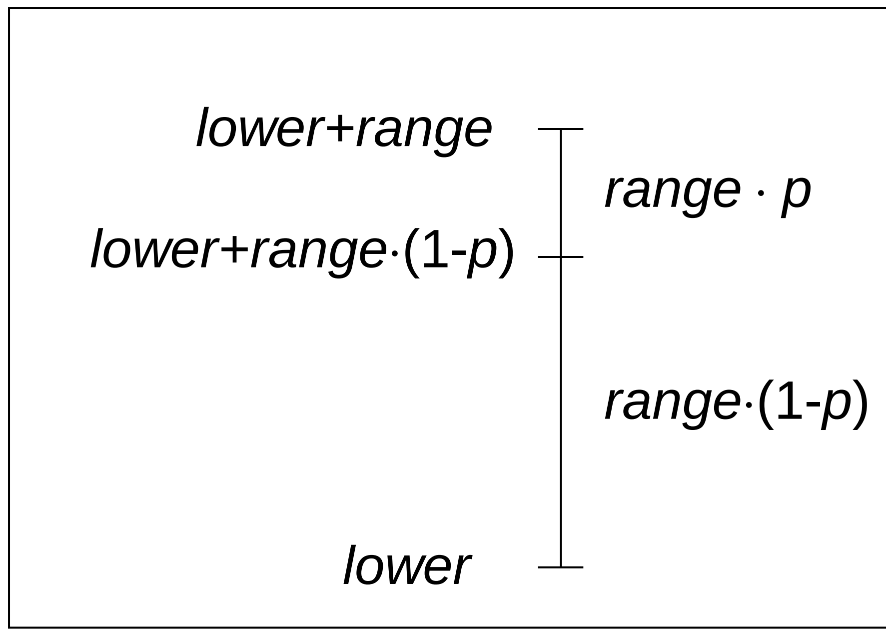
$$\frac{entropy}{averageLength} = \frac{-p\log p - (1-p)\log(1-p)}{-pQ - (1-p)\log(1-1/2^Q)}$$

QM-coder

- One of the methods for e.g. black-and-white images
- Others:
  - *Q-coder* (predecessor of QM, tailored to hardware impl. / IBM)
  - *MQ-coder* (in JBIG2; Joint Bi-Level Image Compression Group)
  - *M-coder* (in H.264/AVC video compression standard)
- Tuned Markov model (finite-state automaton) for adapting probabilities.

Interval setting:

- MPS is the ‘first’ symbol
- Maintain *lower* and *range*:



QM-coder (cont.)

Key ideas:

- Operate within interval [0, 1.5)
- Rescale when *range* < 0.75
- Approximate *range* by 1 in multiplications

  $$range \cdot p \approx p$$
  $$range \cdot (1-p) \approx range - p$$

- No pending bits, but a ‘carry’ bit can propagate to the output bits, which must be buffered. Unlimited propagation is prevented by ‘stuffing’ 0-bits after bytes containing only 1’s (small redundancy).
- Practical implementation is done using integers within [0, 65536).

4.4.3. Practical problems with arithmetic coding

- *Not partially decodable* nor *indexable*:
  Start decoding always from the beginning even to recover a small section in the middle.
- *Vulnerable*: Bit errors result in a totally scrambled message
- Not *self-synchronizable*, contrary to Huffman code

Solution for static distributions: *Arithmetic Block Coding*

- Applies the idea of arithmetic coding within machine words
- Restarts a new coding loop when the word bits are ‘used’.
- Resembles Tunstall code, but no explicit codebook.
- Fast, because avoids the scalings and bit-level operations.
- Non-optimal code length, but rather close