It's Been a 1,000,000 Years Since Huffman

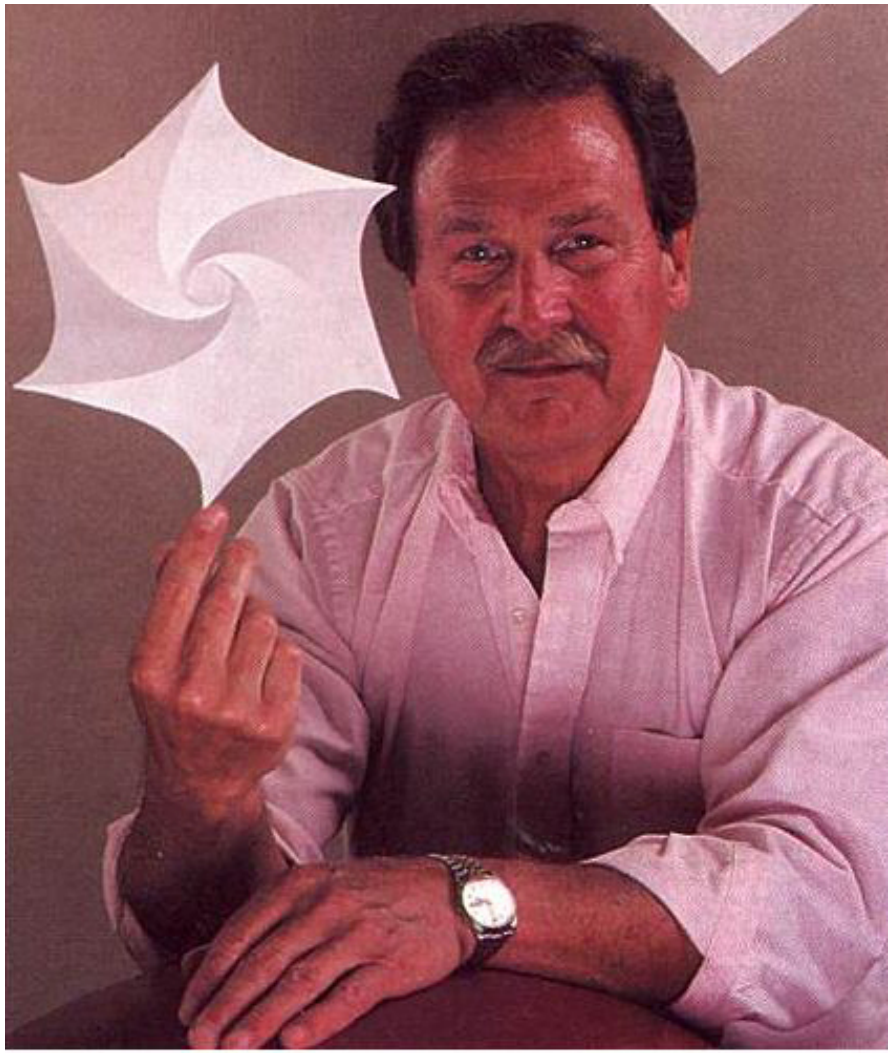
Alistair Moffat

The University of Melbourne

April 2015

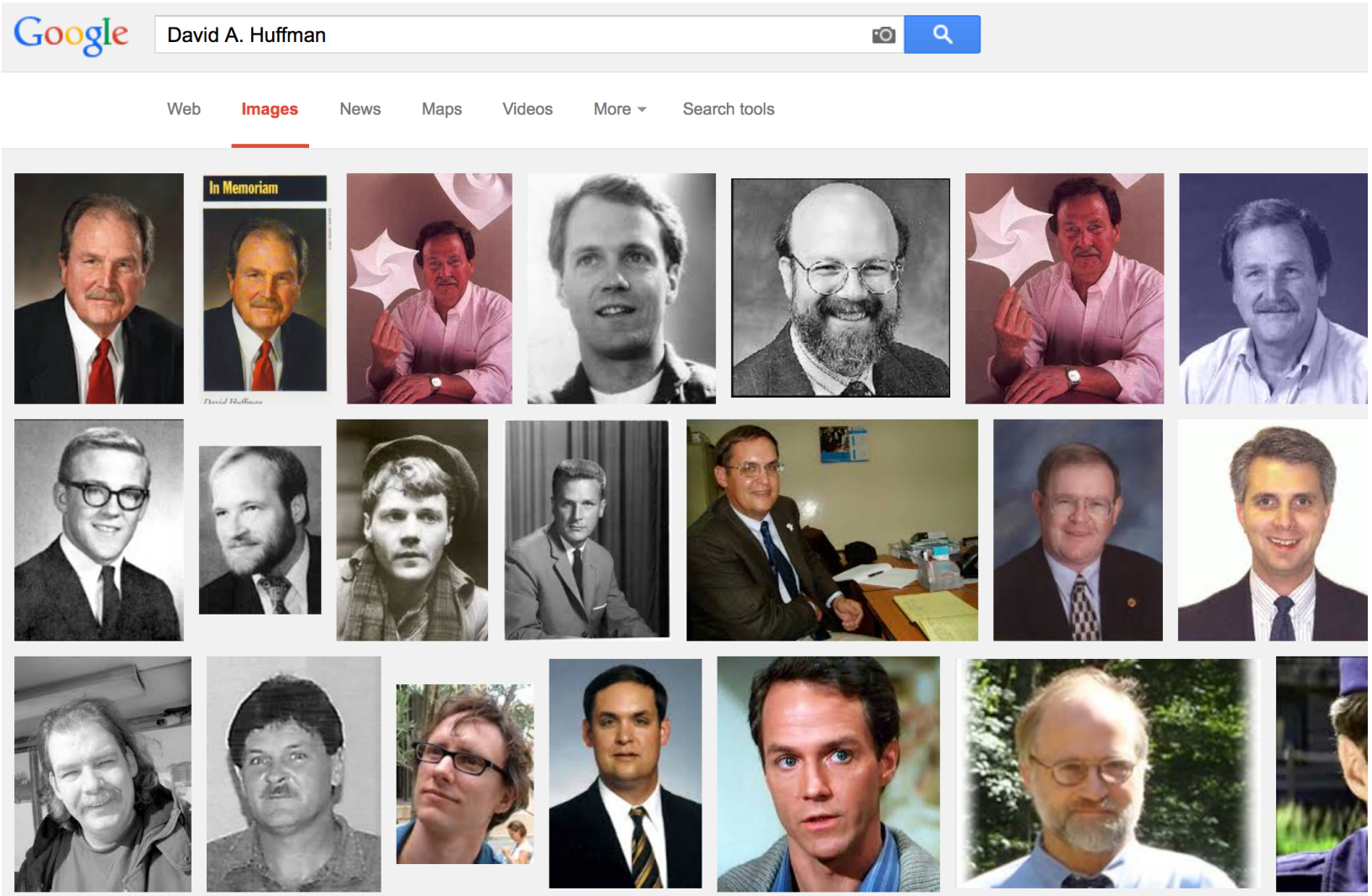
Celebrating 25 Years of DCC
and 64 Years of Huffman Coding

David A. Huffman (1925-1999)

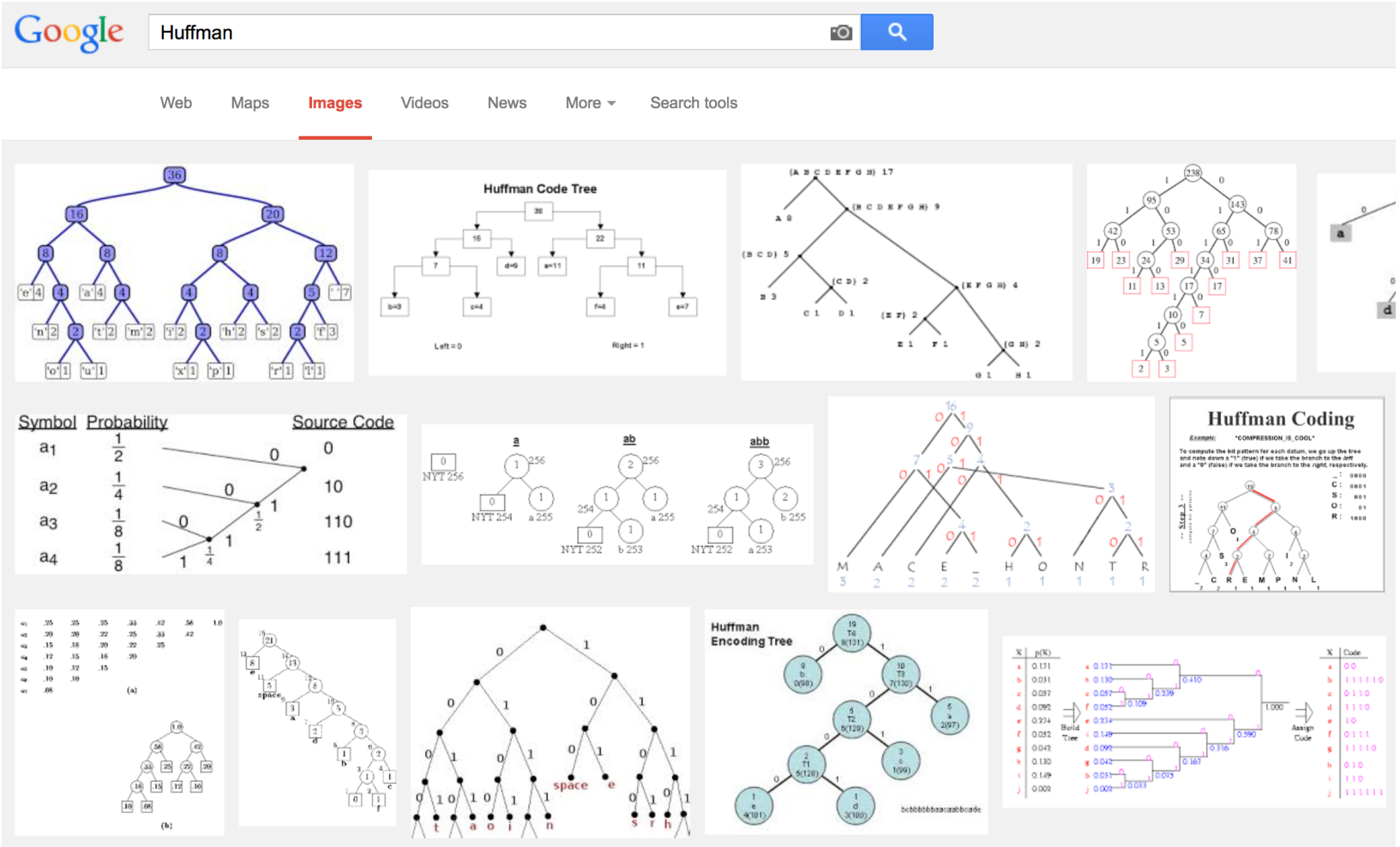


`http://www.huffmancoding.com/my-uncle/scientific-american`
(Photo by Matthew Mulbry, originally for *Scientific American*, Sep. 1991)

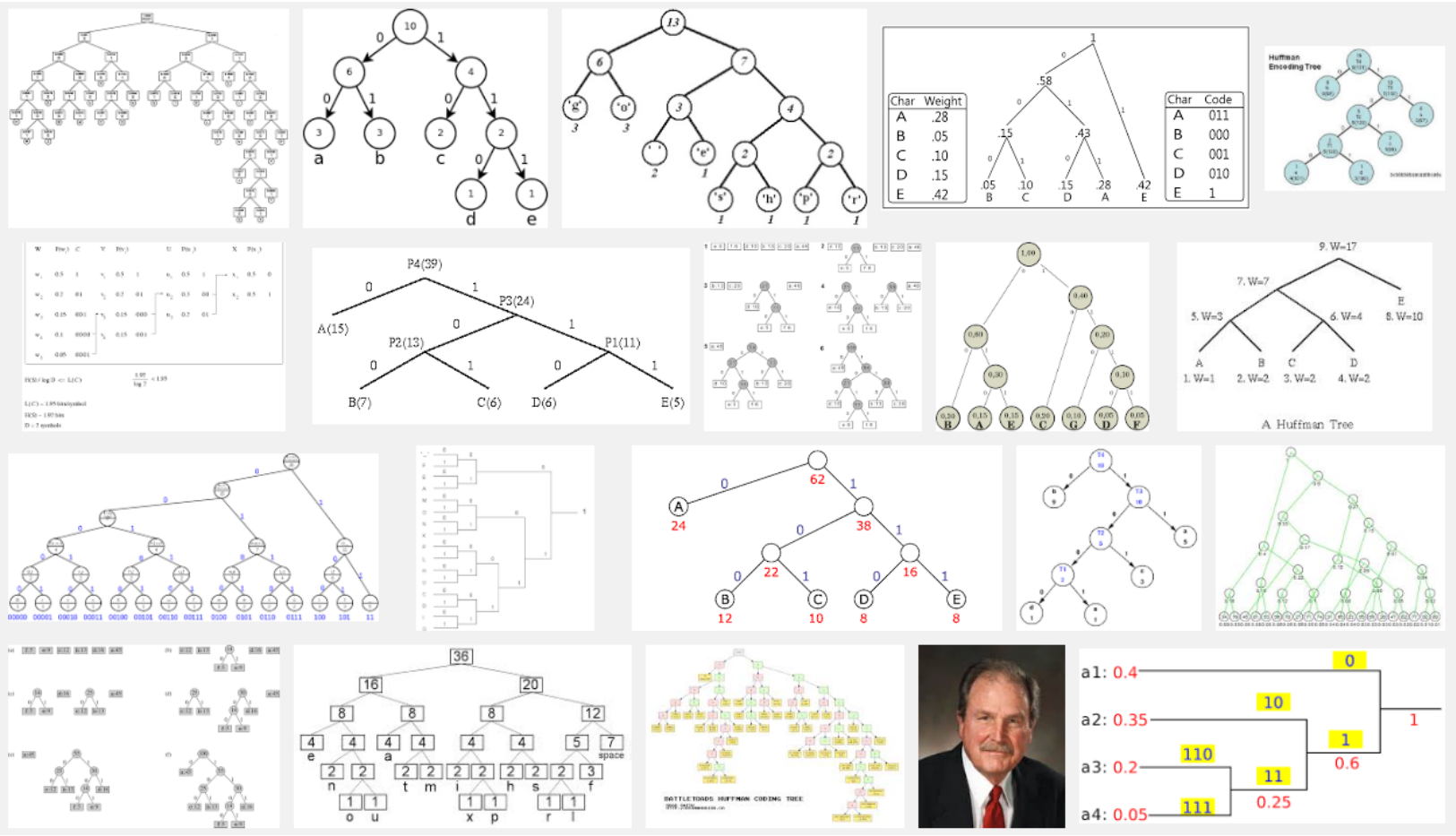
Querying "David A. Huffman"



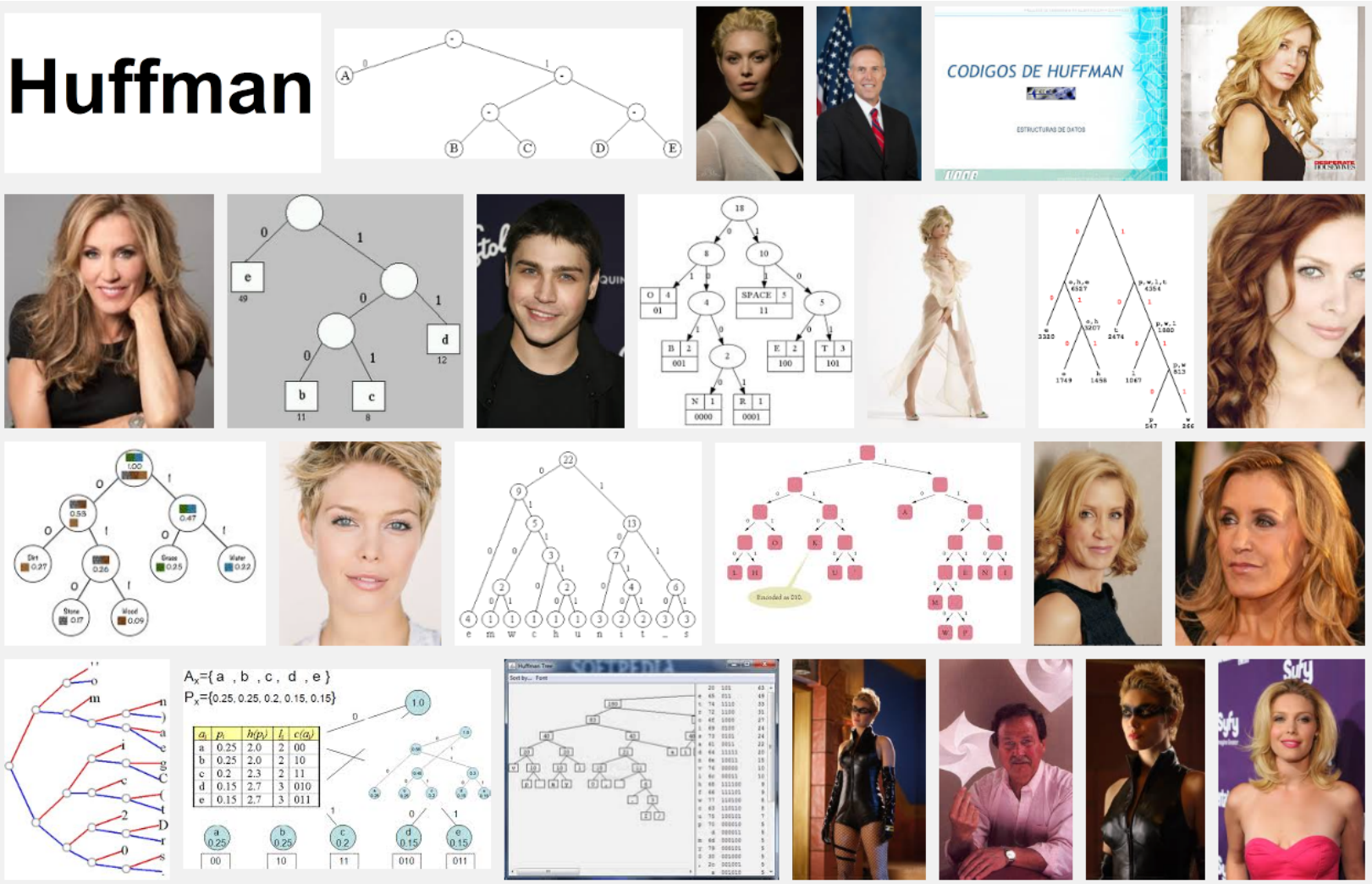
Querying “Huffman”



Querying "Huffman"



Querying "Huffman"



David A. Huffman ...

- Engineering at Ohio State University, graduated at 19.
- War service in the Pacific with US Navy.
- Masters in Electrical Engineering at OSU in 1949 as a veteran, then to MIT for doctoral studies.
- Took "Switching Theory" subject in 1951 while waiting to re-take doctoral exams.
- Graduated PhD in 1953, *The Synthesis of Sequential Switching Circuits*; faculty at MIT 1953–1967; faculty at UCSC 1967–1994; Chair 1970–1973.
- Awarded IEEE Hamming Medal, 1999.
- Principal body of work in area of flexible surfaces and folding, including a gallery show.

... And His Famous Paper

1098 PROCEEDINGS OF THE I.R.E. September

A Method for the Construction of Minimum-Redundancy Codes*

DAVID A. HUFFMAN†, ASSOCIATE, IRE

***Summary*—An optimum method of coding an ensemble of messages consisting of a finite number of members is developed. A minimum-redundancy code is one constructed in such a way that the average number of coding digits per message is minimized.**

INTRODUCTION

ONE IMPORTANT METHOD of transmitting messages is to transmit in their place sequences of symbols. If there are more messages which might be sent than there are kinds of symbols available, then some of the messages must use more than one symbol. If it is assumed that each symbol requires the same time for transmission, then the time for transmission (length) of a message is directly proportional to the number of symbols associated with it. In this paper, the symbol or sequence of symbols associated with a given message will be called the "message code." The entire number of messages which might be transmitted will be will be defined here as an ensemble code which, for a message ensemble consisting of a finite number of members, N, and for a given number of coding digits, D, yields the lowest possible average message length. In order to avoid the use of the lengthy term "minimum-redundancy," this term will be replaced here by "optimum." It will be understood then that, in this paper, "optimum code" means "minimum-redundancy code."

The following basic restrictions will be imposed on an ensemble code:

(a) No two messages will consist of identical arrangements of coding digits.

(b) The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of messages is known.

Huffman, 1951

- The challenge: find a way of computing a minimum-redundancy code, given a set of symbol weights. The previous Shannon-Fano code was not optimal.
- The promise: be exempted from the final examination.
- The find: a week before the exam, gave up, and threw papers in bin. But then realized that he had developed an approach that worked.
- The famous paper: Submitted December 1951 (64 years ago), published September 1952.

Huffman, 1951

This newly created ensemble contains one less message than the original. Its members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was. The codes for each of the two least probable messages in this new ensemble are required to be identical except in their final digits; 0 and 1 are assigned as these digits, one for each of the two messages. Each new auxiliary ensemble contains one less message than the preceding ensemble. Each auxiliary ensemble represents the original ensemble with full use made of the accumulated necessary coding requirements.

The procedure is applied again and again until the number of members in the most recently formed auxiliary message ensemble is reduced to two. One of each of

Huffman, 1951

TABLE I

OPTIMUM BINARY CODING PROCEDURE

Message Probabilities												
Original Message Ensemble	*Auxiliary Message Ensembles*											
	1	2	3	4	5	6	7	8	9	10	11	12
0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	→0.24	→0.36	→0.40	→0.60	→1.00
0.18	0.18	0.18	0.18	0.18	0.18	0.18	→0.20	0.20	0.24	0.36	0.40	
0.10	0.10	0.10	0.10	0.10	→0.14	→0.18	0.18	0.18	0.20	0.24		
0.10	0.10	0.10	0.10	0.10	0.10	0.14	0.14	0.20				
0.10	0.10	0.10	0.10	→0.10	0.10	0.10	0.10					
		→0.08	0.08	0.08	0.10	0.10						
0.06	0.06	0.06	→0.08	0.08	0.08							
0.06	0.06	0.06	0.06	0.06								
0.04	0.04	0.04	0.04									
*0.04	0.04	0.04										
0.04	0.04											
0.04	0.04											
	→0.04											
0.03												
0.01												

Huffman, 1951

Given: a set of n positive weights, w_i.

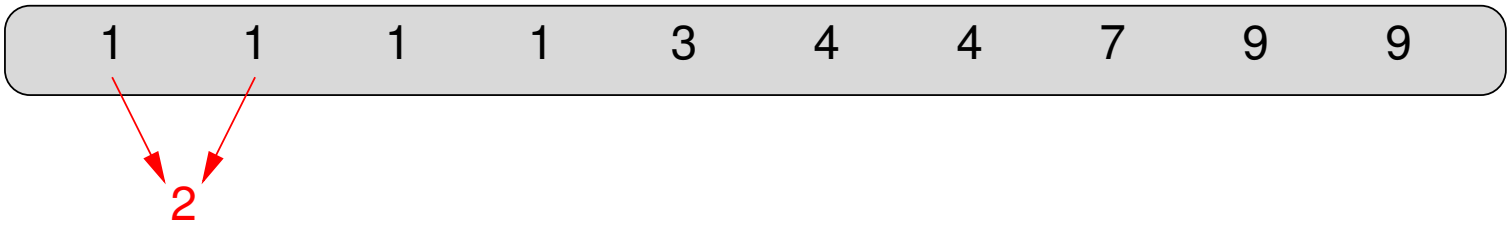
Compute a set of n corresponding codeword lengths, ℓ_i, such that $\sum_i 2^{-\ell_i} \leq 1$ and $\sum_i w_i \cdot \ell_i$ is minimal.

Huffman gave us the algorithm. Questions to consider:

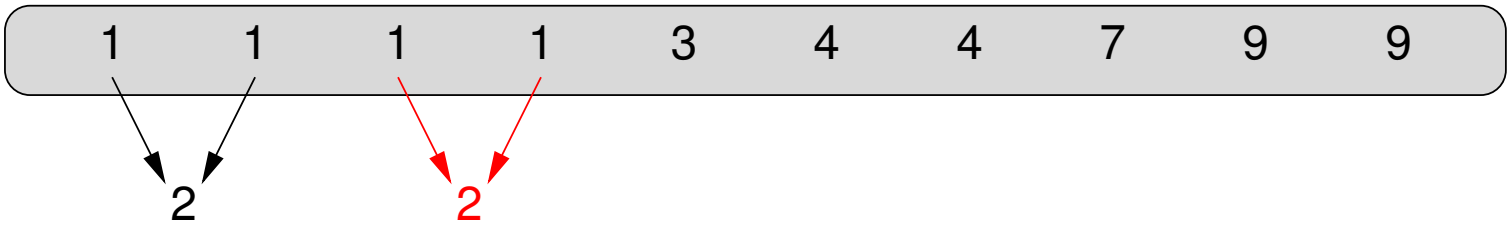
- What can be said about $\ell_{\max} = \max_i \ell_i$?
- How to implement the algorithm efficiently?
- How to implement the encoder and decoder?
- What if other constraints get added?
- What about arithmetic coding?

Huffman's Algorithm: Example

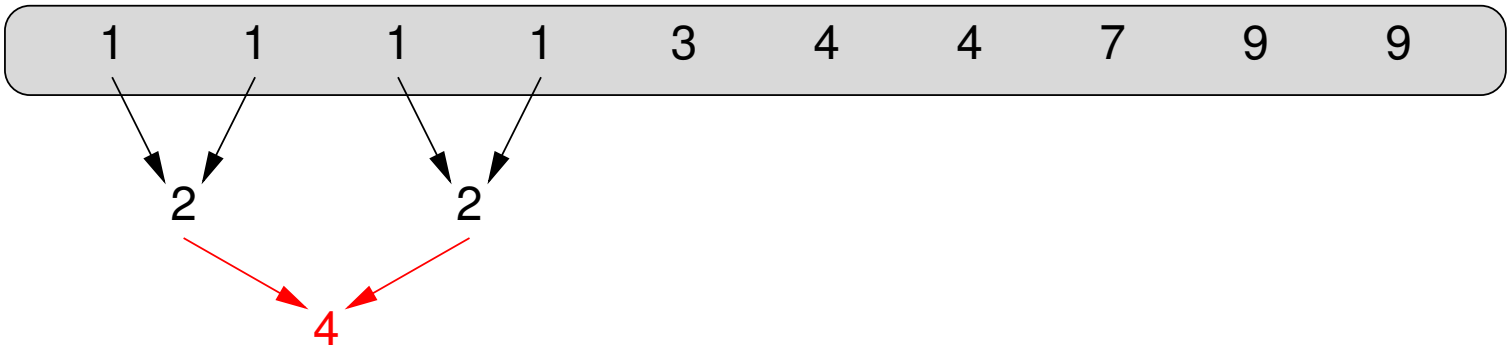
Huffman's Algorithm: Example



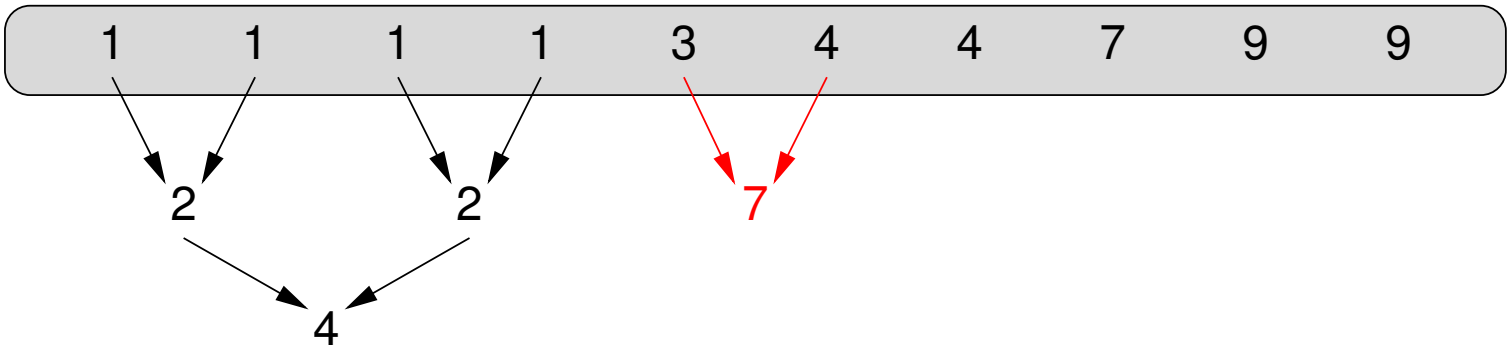
Huffman's Algorithm: Example



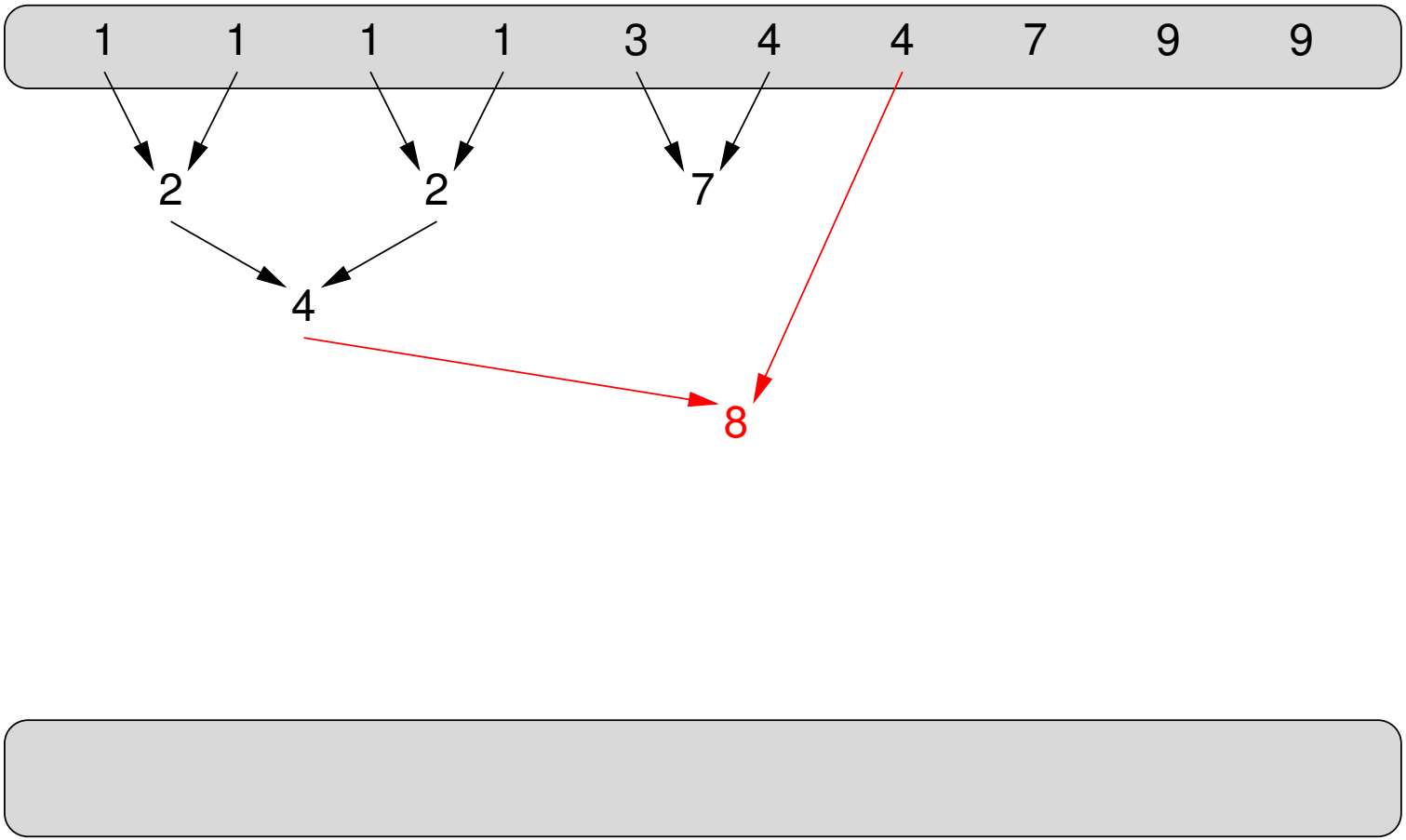
Huffman's Algorithm: Example



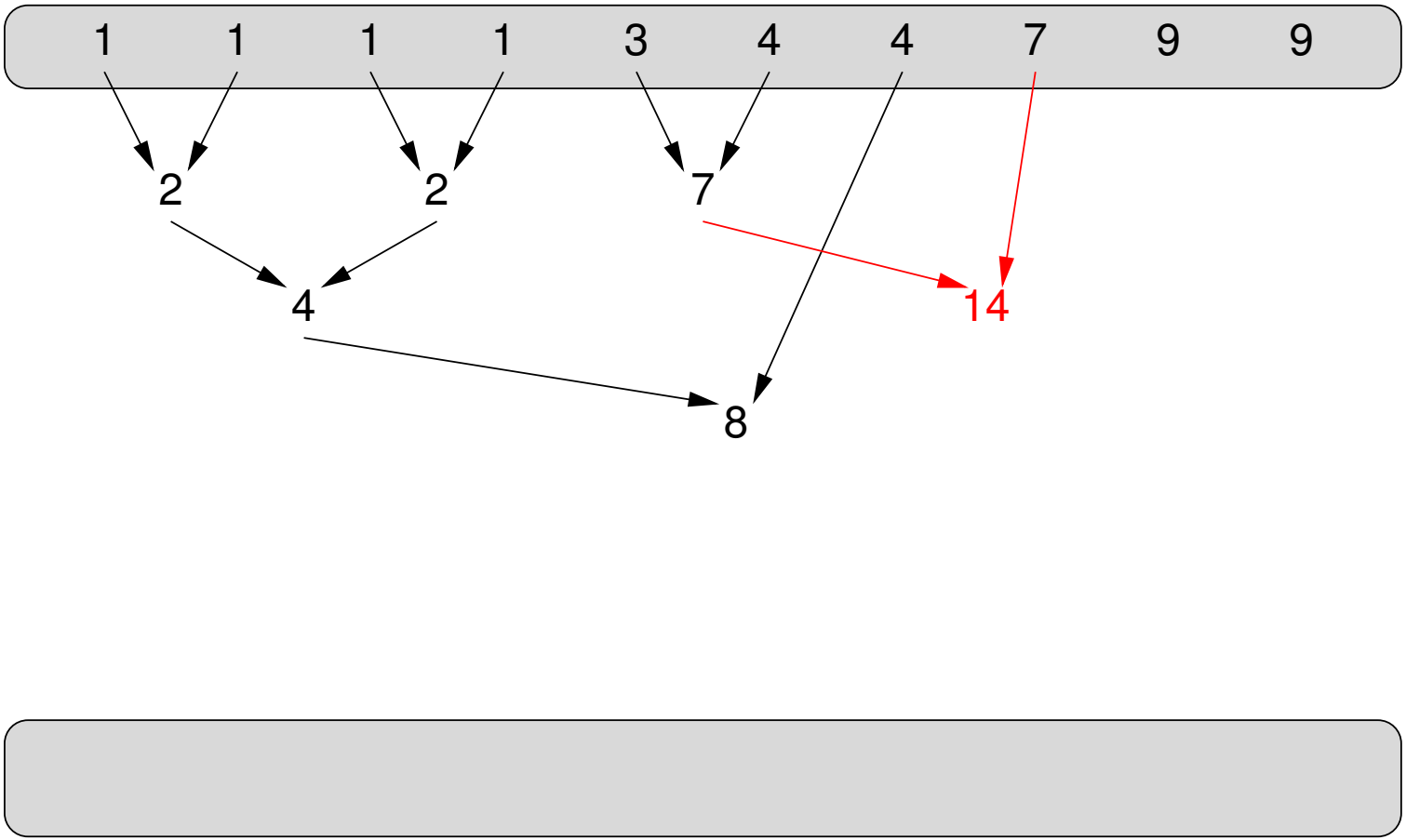
Huffman's Algorithm: Example



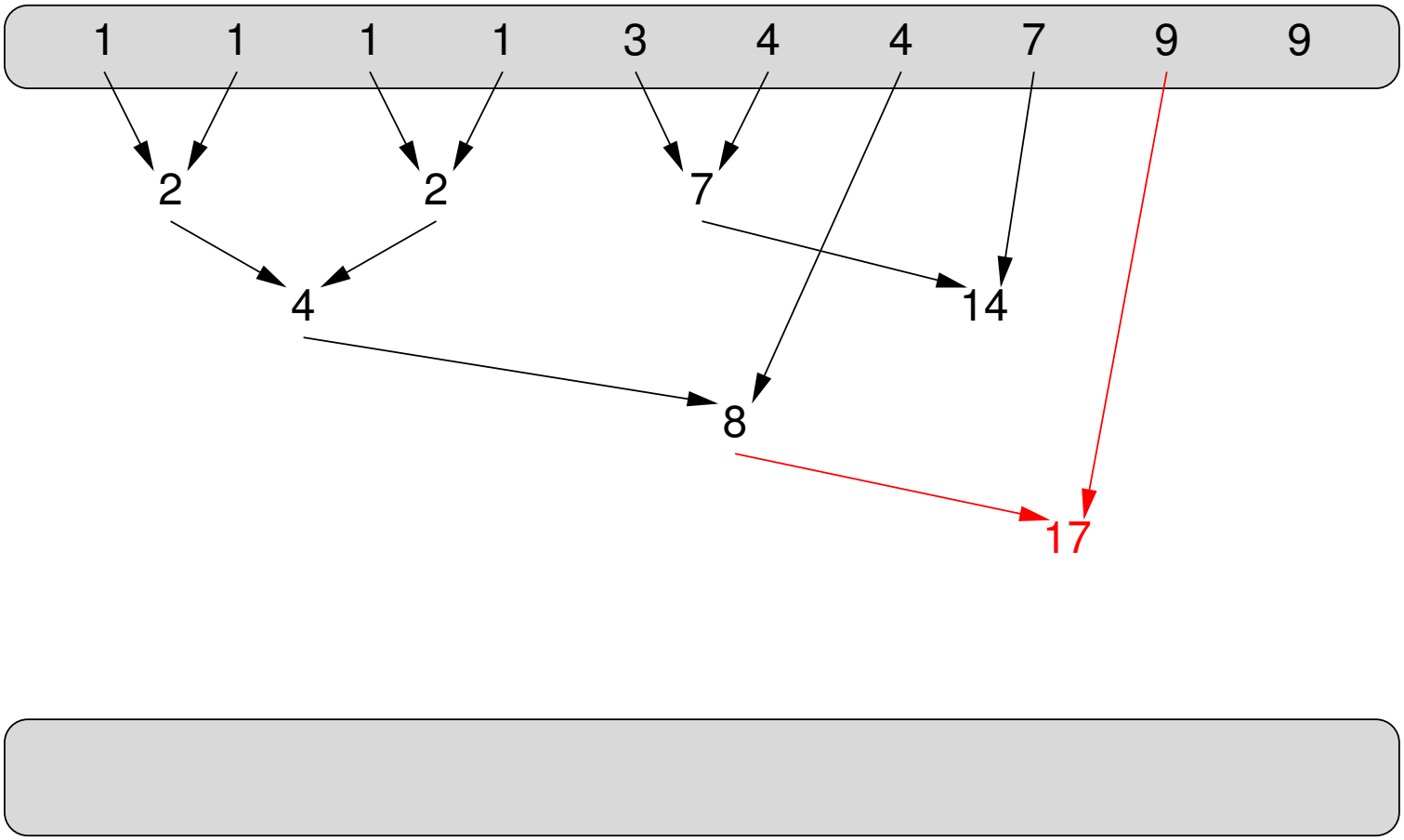
Huffman's Algorithm: Example



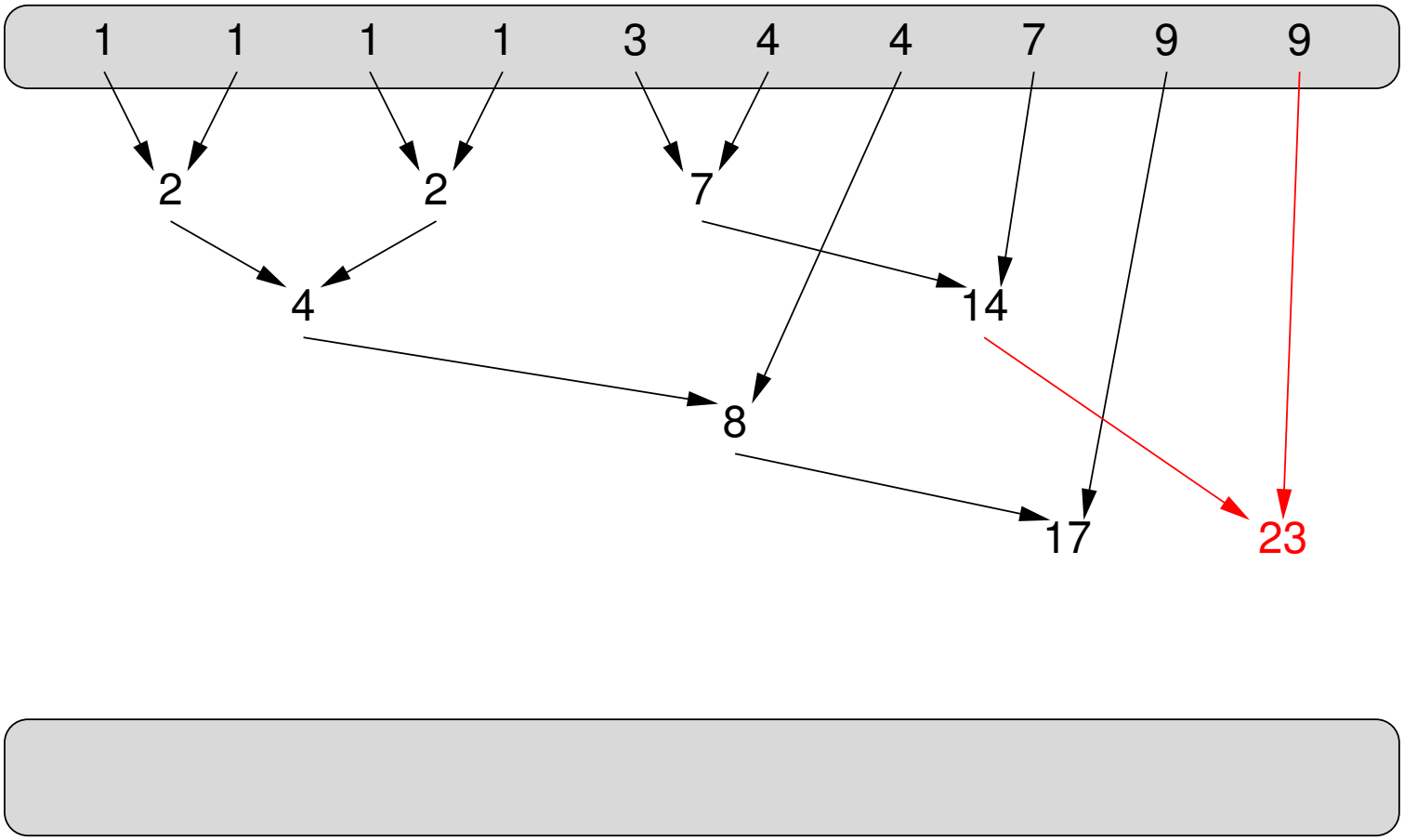
Huffman's Algorithm: Example



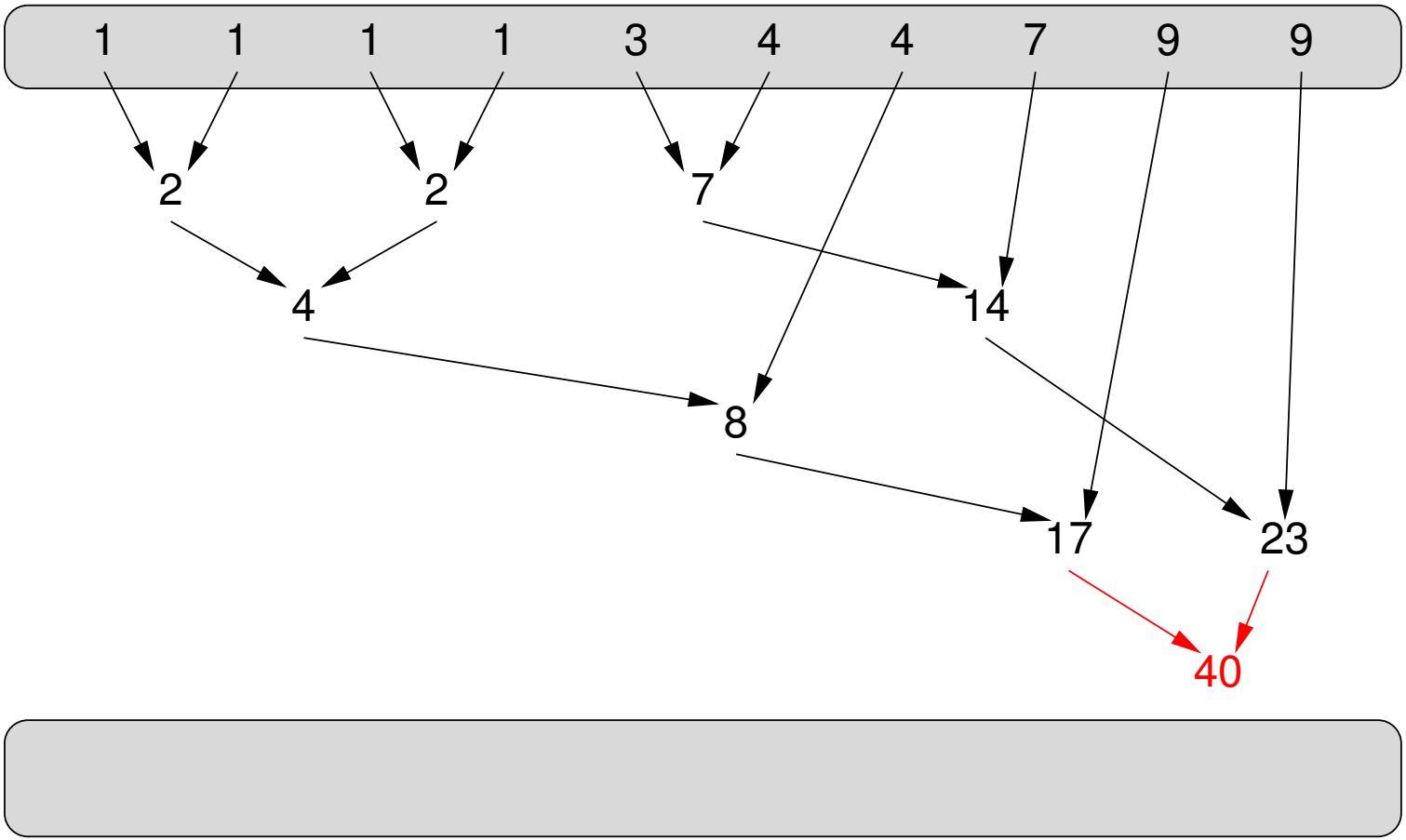
Huffman's Algorithm: Example



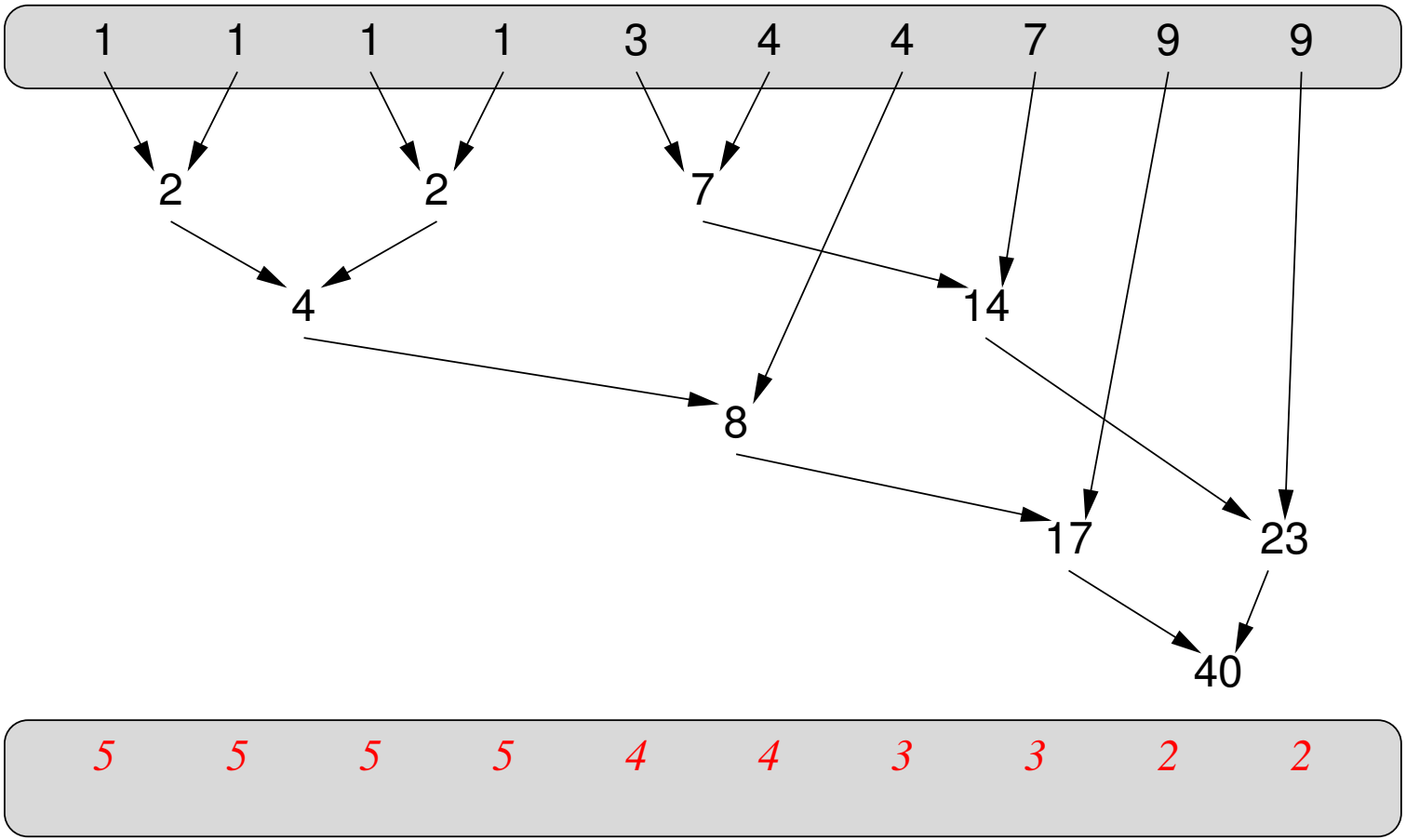
Huffman's Algorithm: Example



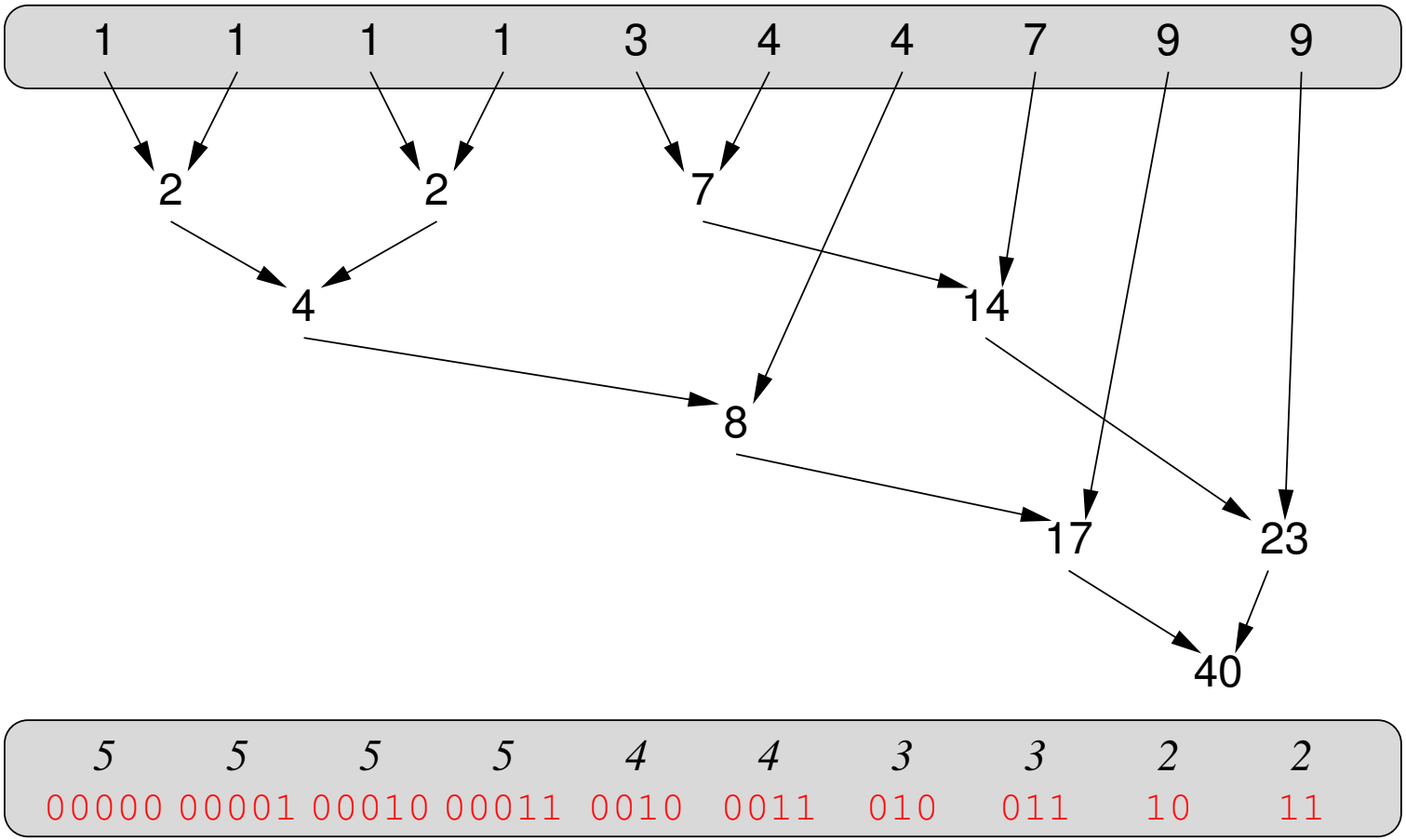
Huffman's Algorithm: Example



Huffman's Algorithm: Example



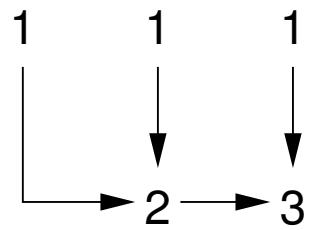
Huffman's Algorithm: Example



Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

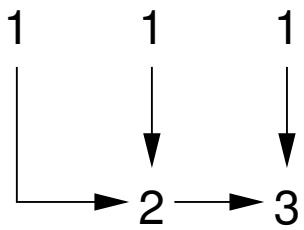
But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:



Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

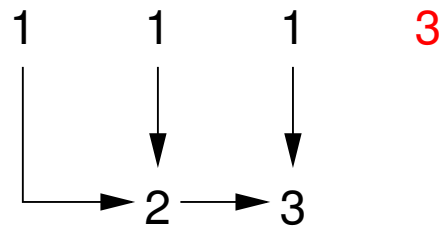


$F'(1) = 2,\ F'(2) = 3$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

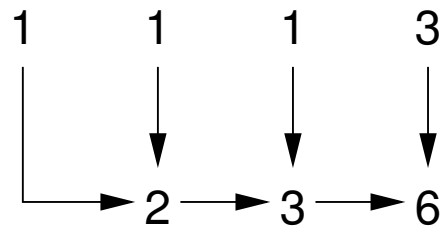


$F'(1) = 2,\ F'(2) = 3$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

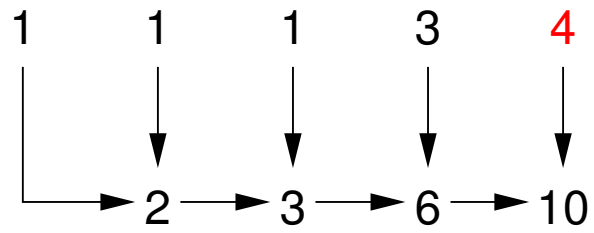


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

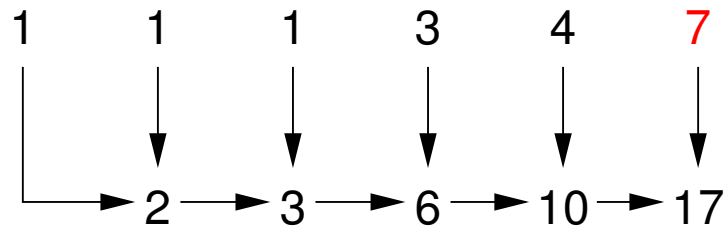


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

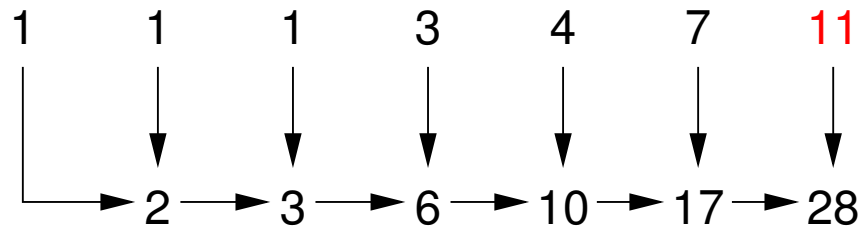


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

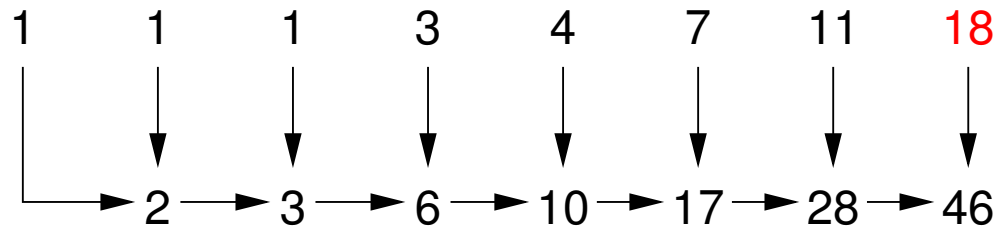


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

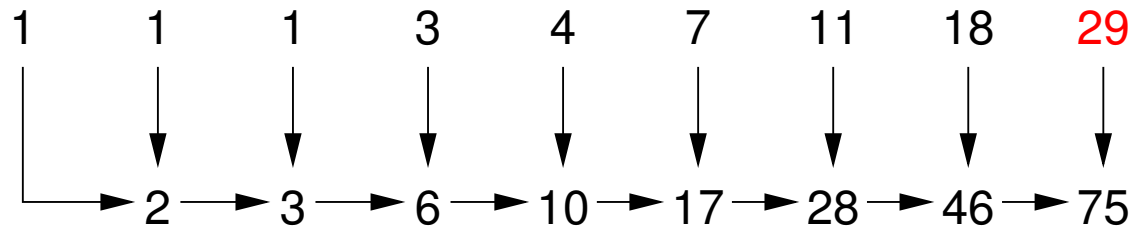


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n - 1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:

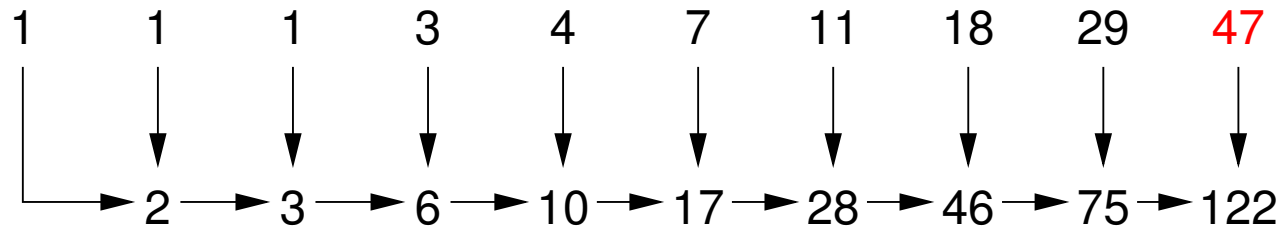


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

Bounds

For an n-symbol alphabet, the codewords can be as long as $n-1$ bits.

But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:



$F'(1) = 2,\ F'(2) = 3$

$F'(k) = F'(k-1) + F'(k-2) + 1.$

Bounds

The standard Fibonacci sequence has base cases $F(1) = F(2) = 1$, and recursive rule $F(k) = F(k-1) + F(k-2)$; with closed form

$$F(k) = \left\lfloor \frac{\phi^k}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

where ϕ is the root of $x^2 - x - 1 = 0$, approximately 1.618

Easy to show that $F'(k) = F(k+2) + F(k) - 1$. That is,

$$F'(k) \approx \frac{\phi^{k+2} + \phi^k}{\sqrt{5}} - 1 \approx \phi^{k+1} - 1\,.$$

Bounds

Hence, for an N symbol message, the codewords can be at most $(\log_\phi N) - 1 \approx (1.44 \log_2 N) - 1$ bits long.

If p bits of precision are available for frequency counts, then the codewords can be at most $\ell_{\max} \leq \lfloor 1.44p \rfloor - 1$ bits long. [Katona & Nemetz, 1976; Buro, 1993].

To represent a codeword length requires $\lceil \log_2 \ell_{\max} \rceil \approx \lceil \log_2 p + 0.53 \rceil$ bits. If $p = 32$, then $\ell_{\max} \leq 45$, and six bits suffice.

If $p = 64$, then $\ell_{\max} \leq 91$ and at most seven bits are needed. A codeword length will never take more than one byte.

Implementation (1)

Textbooks describe heap-based methods, building explicit binary trees with left/right edges labeled with a 0/1.

Encoding: start at the symbol's leaf, stack up the labels on the edges on path to root, and emit them in reverse order.

Decoding: start at the root, follow left/right edge for 0/1 bits, emit the corresponding leaf symbol label.

Resources: $O(n)$ space ($4n$ to $6n$ words) and $O(n \log n)$ time to compute. En/decoding requires bit-by-bit processing.

Implementation (2)

If the n input weights are sorted, then at each min-finding step take the smaller of next leaf, and front item in a queue of internal nodes that has been formed. New internal nodes are appended at the end of the same queue when they are formed.

Resources: $O(n)$ time and $O(n)$ space, once the weights are sorted [van Leeuwen, 1976].

Pre-sorting takes $O(n \log n)$ time, so overall is the same as heap-based textbook approach.

Implementation (3)

Input: array A of the n symbol weights w_i, sorted so that $A[i] \leq A[i+1]$.

Output: element $A[i]$ is now the length ℓ_i of the corresponding i th codeword, with $A[i] \geq A[i+1]$.

Resources: the transformation requires $O(n)$ time and $O(1)$ further space [Moffat & Katajainen, 1995].

If symbols are not naturally sorted, pre-sort can be done by Quicksort, and a second array of n words used to store a permutation vector.

Implementation (3)

Pass One: Turn n weights into $n-1$ internal node weights and then $n-2$ internal parent pointers.

Pass Two: Turn $n-2$ internal parent pointers in to $n-1$ internal node depths, using $A[i] \leftarrow A[A[i]]+1$.

Pass Three: turn $n-1$ internal node depths in to n leaf depths.

Implementation (3)

leaf weights

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**

Implementation (3)

leaf weights **internal weights**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, first	**2**	–	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**

Implementation (3)

leaf weights **internal weights**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	–	–	**3**	**4**	**4**	**7**	**9**	**9**

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	**4**	–	**3**	**4**	**4**	**7**	**9**	**9**

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	**4**	**7**	–	–	**4**	**7**	**9**	**9**

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	**4**	**7**	**8**	–	–	**7**	**9**	**9**

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	**4**	**5**	**8**	**14**	–	–	**9**	**9**

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	**4**	**5**	**6**	**14**	**17**	–	–	**9**

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, part	**2**	**2**	**4**	**5**	**6**	**7**	**17**	**23**	–	–

Implementation (3)

leaf weights **internal weights** **internal parents**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, first	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**0**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, part	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**1**	**0**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, part	**2**	**2**	**4**	**5**	**6**	**7**	**1**	**1**	**0**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, part	**2**	**2**	**4**	**5**	**6**	**2**	**1**	**1**	**0**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, part	**2**	**2**	**4**	**5**	**2**	**2**	**1**	**1**	**0**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, done	**4**	**4**	**3**	**3**	**2**	**2**	**1**	**1**	**0**	–

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths **leaf depths**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, done	**4**	**4**	**3**	**3**	**2**	**2**	**1**	**1**	**0**	–
P3, first	**4**	**4**	**3**	**3**	–	–	–	–	**2**	**2**

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths **leaf depths**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, done	**4**	**4**	**3**	**3**	**2**	**2**	**1**	**1**	**0**	–
P3, part	**4**	**4**	–	–	–	–	**3**	**3**	**2**	**2**

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths **leaf depths**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, done	**4**	**4**	**3**	**3**	**2**	**2**	**1**	**1**	**0**	–
P3, part	–	–	–	–	**4**	**4**	**3**	**3**	**2**	**2**

Implementation (3)

leaf weights **internal weights** **internal parents**
internal depths **leaf depths**

	i									
	0	1	2	3	4	5	6	7	8	9
P1, start	**1**	**1**	**1**	**1**	**3**	**4**	**4**	**7**	**9**	**9**
P1, done	**2**	**2**	**4**	**5**	**6**	**7**	**8**	**8**	**40**	–
P2, done	**4**	**4**	**3**	**3**	**2**	**2**	**1**	**1**	**0**	–
P3, done	**5**	**5**	**5**	**5**	**4**	**4**	**3**	**3**	**2**	**2**

Implementation (3)


```
0: function calc_huff_lens(A, n)                ▷ Input: A[i − 1] ≤ A[i] for 0 < i < n
1:     // Phase 1
2:     set leaf ← 0 and root ← 0
3:     for next ← 0 to n − 2 do
4:         if leaf ≥ n or (root < next and A[root] < A[leaf]) then
5:             set A[next] ← A[root] and A[root] ← next and root ← root + 1      ▷ Use internal node
6:         else
7:             set A[next] ← A[leaf] and leaf ← leaf + 1                         ▷ Use leaf node
8:         end if
9:         repeat steps 4–8, but adding to A[next] rather than assigning to it   ▷ Find second child
10:    end for
11:    // Phase 2
12:    set A[n − 2] ← 0
13:    for next ← n − 3 downto 0 do
14:        set A[next] ← A[A[next]] + 1                  ▷ Compute depths of internal nodes
15:    end for
16:    // Phase 3
17:    set avail ← 1 and used ← 0 and depth ← 0 and root ← n − 2 and next ← n − 1
18:    while avail > 0 do
19:        while root ≥ 0 and A[root] = depth do   ▷ Count internal nodes used at depth depth
20:            set used ← used + 1 and root ← root − 1
21:        end while
22:        while avail > used do                   ▷ Assign as leaves any nodes that are not internal
23:            set A[next] ← d and next ← next − 1 and avail ← avail − 1
24:        end while
25:        set avail ← 2 · used and depth ← depth + 1 and used ← 0          ▷ Move to next depth
26:    end while
27:    return A                                  ▷ Output: A[i] is the length ℓ_i of the i th codeword
28: end function
```

Implementation (4)

A different input format: $\langle w_i, n_i \rangle$, where $\sum_i n_i = n$ and $\sum_i w_i n_i = N$, and where same-weight symbols are aggregated. Output is tuples $\langle \ell_i, n'_i \rangle$, with $\sum_i n'_i = n$.

For the example:

$$\text{input} = \langle 1, 4 \rangle, \langle 3, 1 \rangle, \langle 4, 2 \rangle, \langle 7, 1 \rangle, \langle 9, 2 \rangle$$
$$\text{output} = \langle 5, 4 \rangle, \langle 4, 2 \rangle, \langle 3, 2 \rangle, \langle 2, 2 \rangle$$

If there are r distinct symbol weights, can compute Huffman code in $O(r + r \log(n/r))$ time and space. If $r = o(n)$, then time is $o(n)$. [Moffat & Turpin, 1998].

Implementation (5), (6)

If input is sorted weights, and non-overwrite construction is required, can compute in $O(n)$ time and $O(\ell_{\max})$ space [Milidiú, Pessoa & Laber, 2001].

Algorithm is complex, and it isn't clear that an implementation will execute quickly.

In recent work, can compute in $O(nk)$ time, where k is number of distinct codeword lengths [Belal, Elmasry, 2006]. Again, algorithm is complex, and implementation has not been provided.

Encoding: Canonical Codes

i	$\ell[i]$	codeword
0	5	00000
1	5	00001
2	5	00010
3	5	00011
4	4	0010
5	4	0011
6	3	010
7	3	011
8	2	10
9	2	11

ℓ	$fst[\ell]$	$base[\ell]$
2	8	2
3	6	2
4	4	2
5	0	0

Encoding: Canonical Codes

i	$\ell[i]$	codeword
0	5	00000
1	5	00001
2	5	00010
3	5	00011
4	4	0010
5	4	0011
6	3	010
7	3	011
8	2	10
9	2	11

ℓ	$fst[\ell]$	$base[\ell]$
2	8	2
3	6	2
4	4	2
5	0	0

encode(s):

$\ell_s \leftarrow \ell[s]$

$code \leftarrow base[\ell_s] + (s - fst[\ell_s])$

$putbits(code, \ell_s)$

[Schwartz & Kallick, 1964; Connell, 1973; Zobel & Moffat, 1995].

Encoding: Canonical Codes

For weight-ordered n symbol alphabet, and an N symbol message, $n(1 + \log\log N) + O(\log^2 N)$ bits; and (making certain assumptions) $O(1)$ time per symbol coded.

Or, using a linear search in *fst*, $O(\log^2 N)$ bits, and $O(\ell_s)$ time per symbol coded. Or binary search in *fst* in $O(\log(\ell_{\max} - \ell_{\min})) = O(\log\log N)$ time per symbol coded.

If alphabet is not weight-ordered, either add an $n \log n$-bit permutation vector; or keep ℓ as an $n(1 + \log\log N)$-bit vector augmented with rank-support, and compute $rank(\ell, \ell_s)$ instead of $(s - fst[\ell_s])$.

Decoding: Canonical Codes

i	codeword
0	00000
1	00001
2	00010
3	00011
4	0010
5	0011
6	010
7	011
8	10
9	11

ℓ	$fst[\ell]$	$base[\ell]$	$ljb[\ell]$
2	8	2	16
3	6	2	8
4	4	2	4
5	0	0	0

Decoding: Canonical Codes

i	codeword
0	00000
1	00001
2	00010
3	00011
4	0010
5	0011
6	010
7	011
8	10
9	11

ℓ	$fst[\ell]$	$base[\ell]$	$ljb[\ell]$
2	8	2	16
3	6	2	8
4	4	2	4
5	0	0	0

decode():

$bits \leftarrow nextbits(\ell_{\max})$

$\ell_s \leftarrow \min_\ell\{ljb[\ell] \leq bits < ljb[\ell-1]\}$

$s \leftarrow fst[\ell_s] +$
$\quad (bits - ljb[\ell_s]) \gg (\ell_{\max} - \ell_s)$

$shiftbits(\ell_s)$

return s

Decoding: Canonical Codes

For sorted alphabet, requires $O(\log^2 N)$ bits; search process can be done by

- linear search in $O(\ell_s - \ell_{\min})$ (or $O(\ell_{\max} - \ell_s)$) time
- binary search in $O(\log \log N)$ time
- table lookup using extra space.

Plus, for non-sorted alphabets, either add select-support to the $n(1 + \log \log N)$-bit vector ℓ and compute

$$s \leftarrow \mathit{select}(\ell, \ell_s, (\mathit{bits} - \mathit{ljb}[\ell_s]) \gg (\ell_{\max} - \ell_s)),$$

or add an $n \log n$-bit inverse permutation vector.

Decoding Tables

If a table of $2^{\ell_{\max}}$ entries can be allocated, each $1 + \log\log N$ bits, then decoding is $O(1)$ time per symbol.

If further space per entry is used, multiple symbols might be emitted out of some combinations of the $\ell_{\max}$ bits in *bits* [Liddell & Moffat, 2006].

Or, if even that much space is problematic, a smaller table can accelerate the linear search in *ljb*, based on a prefix of *bits* [Moffat & Turpin, 1997].

$bits \gg 3$	00	01	10	11
l_s	4+	3	2	2

Variants

- Non-binary channel alphabets
- Length-limited codes
- Unequal letter costs
- Dynamic/adaptive codes
- Plus more...

Non-Binary Channel Alphabets

Huffman also described a variant for r-ary alphabets.

TABLE III

OPTIMUM CODING PROCEDURE FOR $D=4$

Message Probabilities					
Original Message Ensemble	*Auxiliary Ensembles*			*L(i)*	*Code*
			→1.00		
		→0.40			
0.22	0.22	0.22		1	1
0.20	0.20	0.20		1	2
0.18	0.18	0.18		1	3
0.15	0.15			2	00
0.10	0.10			2	01
0.08	0.08			2	02
	→0.07				
0.05				3	030
0.02				3	031

Each step replaces r leaves by a single internal node. [Huffman, 1952].

Length-Limited Codes

Add constraint that $\ell_i \leq L$ for some $L \geq \lceil \log_2 n \rceil$.

The package-merge method has parallels with Huffman's algorithm. Requires $O(nL)$ time, and either $O(nL)$ space, or $O(n)$ space with controlled re-computation [Larmore & Hirshberg, 1990].

Can be improved to $O(n(L - \log n))$ time and $O(L^2)$ space. [Katajainen, Moffat & Turpin, 1995].

Approximation methods iteratively adjust the weights, using Huffman's algorithm until length limit is satisfied [Milidiú & Laber, 2001].

Unequal Letter-Costs

Suppose that each symbol of the output alphabet has a different cost, for example:

- "dots" take $c_0 = 1 + 1 = 2$ units of time
- "dashes" take $c_1 = 3 + 1 = 4$ four units.

Example: "▪▪▪▬▬▬▪▪▪", the maritime SOS message, costs $(6 \times 2) + (3 \times 4) = 24$ units of time.

Generally: given the cost c_i of each of r output symbols, how to construct a minimal-cost code for the n weights w_i?

With discrete codewords: complex! [Bradford et al., 2002].

Unequal Letter-Costs

(1) Solve for t:

$$t^{c_0} + t^{c_1} + \cdots + t^{c_{r-1}} = 1$$

Then set $p_i = t^{c_i}$. Example: for $c_0 = 2$ and $c_1 = 4$, $t \approx 0.7862$; and hence $p_0 = 0.618$ and $p_1 = 0.382$.

(2) Arithmetic encode the source message using the symbol weights w_i to get a stream of equi-probable zeros and ones.

(3) "Decode" that bitstream using the probability distribution p to get an output stream in which channel symbol i occurs with probability p_i.

Unequal Letter-Costs

Output symbol i costs c_i dollars to transmit; carries information $-\log_2 p_i = -c_i \log_2 t$; and hence carries information at the rate of $-\log_2 t$ bits per dollar.

Each symbol in the channel alphabet carries information at the same optimal unit rate.

To decode: "encode" the compressed message using the probabilities p_i to get a bitstream, and then decode that bitstream using the original weights w_i.

Dynamic/Adaptive Coding

Several versions, primarily differentiated by how they cater for novel symbols as they appear, and whether symbol weights are required to be incrementing integers.

Encoding and decoding, plus code rebuilding costs, $O(\ell_s)$ time per symbol – linear in total number of output bits.

Several words per symbol required for data structures, and encoding and decoding are substantially slower than for static canonical codes.

[Faller, 1973; Gallager, 1978; Cormack & Horspool, 1984; Knuth, 1985; Vitter, 1989].

Plus More...

- Infinite codes: Golomb, Elias, etc
- Alphabetic restrictions
- Redundancy bounds for each variant
- Use in wavelet trees.

Arithmetic Coding

Binary arithmetic coding based on relatively early work [Pasco, 1976; Rissanen, 1976; Rissanen and Langdon, 1979].

Multi-symbol arithmetic coding popularized in 1987 with publication of a description – and complete C source code – in CACM [Witten, Neal, & Cleary, 1987]. No `ftp` service to NZ at that time!

Faster variants followed, including byte at a time output; and data structures to support efficient frequency update and cumulative rank operations [Moffat, 1990; Howard & Vitter, 1992, 1994; Fenwick, 1994, 1996; Schindler, 1998; Moffat, Neal, & Witten, 1998; Moffat, 1999].

Arithmetic Coding

Output length: very close to optimal, that is, $\ell_s \approx -\log_2(w_i/N)$, even when $N/k < w_i < N$.

Encoding, decoding: static or dynamic, n words of memory (that is, $n \log N$ bits), and $O(\ell_s)$ time per symbol.

Cake and icing both!

Huffman Coding vs Arithmetic Coding

Is Huffman Coding Dead?

A. Bookstein[1] and *S.T. Klein*[2]

[1]Center for Information and Language Studies, University of Chicago, Chicago IL 60637, USA
[2]Dept. of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel

Recent publications about data compression suggest that Huffman coding is out of fashion. These publications stress the suboptimality of Huffman codes. Indeed, the "optimality" of Huffman codes has often been overemphasized in the past and it is not always mentioned that Huffman codes are optimal only if:

1. the set of elements to be encoded is fixed throughout the file; and
2. each codeword is constructed to consist of an integral number of bits.

Arithmetic codes overcome the above constraints. In fact, these codes have had a long history, but became especially popular after Witten, Neal and Cleary's paper in 1987, which claim arithmetic codes are *superior in most respects to the better known Huffman method*.

[Bookstein & Klein, 1993].

Huffman Coding vs Arithmetic Coding?

If:

- individual weights are small, $w_i/N < 1/k$, and

Huffman Coding vs Arithmetic Coding?

If:

- individual weights are small, $w_i/N < 1/k$, and
- adaptive coding is not required, and

Huffman Coding vs Arithmetic Coding?

If:

- individual weights are small, $w_i/N < 1/k$, and
- adaptive coding is not required, and
- only one, or a small number of coding contexts are active at any given time,

Huffman Coding vs Arithmetic Coding?

If:

- individual weights are small, $w_i/N < 1/k$, and
- adaptive coding is not required, and
- only one, or a small number of coding contexts are active at any given time,

then canonical Huffman codes are much faster than arithmetic codes, and the effectiveness loss is small.

Huffman Coding vs Arithmetic Coding?

Conversely, use arithmetic coding when

- interleaving symbols from multiple contexts, or
- when model is adaptive, or
- when individual events have high probability.

PPM is the perfect application for arithmetic coding [Cleary & Witten, 1984; Moffat, 1990; Bunton, 1997].

But `gzip`, `bzip2`, `xz` and etc – "real" compression programs used millions of times every day – use Huffman coding.

Huffman Coding vs Arithmetic Coding?

Conversely, use arithmetic coding when

- interleaving symbols from multiple contexts, or
- when model is adaptive, or
- when individual events have high probability.

PPM is the perfect application for arithmetic coding [Cleary & Witten, 1984; Moffat, 1990; Bunton, 1997].

But `gzip`, `bzip2`, `xz` and etc – "real" compression programs used millions of times every day – use Huffman coding.

Bottom line: Huffman codes remain important.

Gallery (Rogues)

Many thanks to students and other collaborators over the last 25 years:

Andrew Turpin, Anh Ngoc Vo, Ian Witten, Jesper Larsson, Justin Zobel, Jyrki Katajainen, Lang Stuiver, Linh Huynh, Mike Liddell, Neil Sharman, Radford Neal, Shane Culpepper, Timothy C. Bell, Tony Wirth, Yugo Kartona Isal.

And apologies to the many people that I haven't cited here in this whirl-wind tour.

Snowbird: 1991

Snowbird: 1991-2015