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

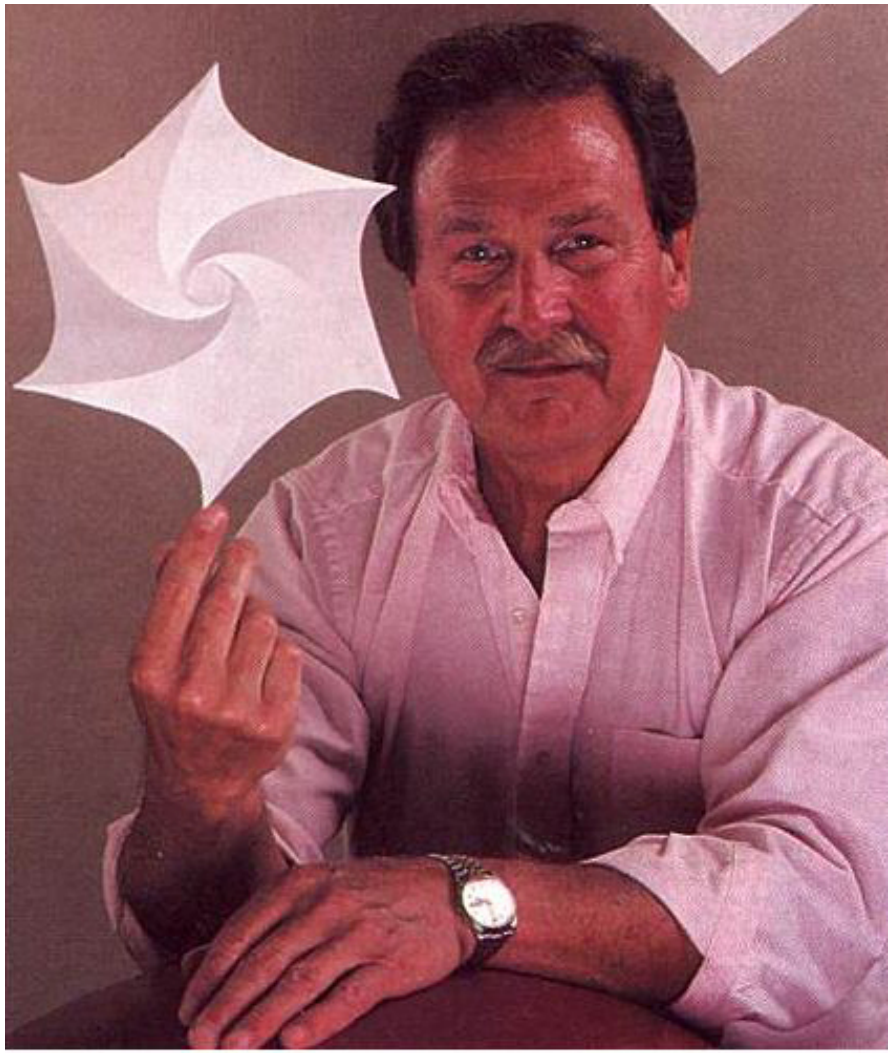
Alistair Moffat

The University of Melbourne

April 2015

Celebrating 25 Years of DCC
and 64 Years of Huffman Coding
Abbreviated version for COMP20007, May 2015

David A. Huffman (1925-1999)



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

Huffman, 1951

- The context: Huffman was an MIT graduate student, taking a Signal Processing course under Robert Fano.
- 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

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:

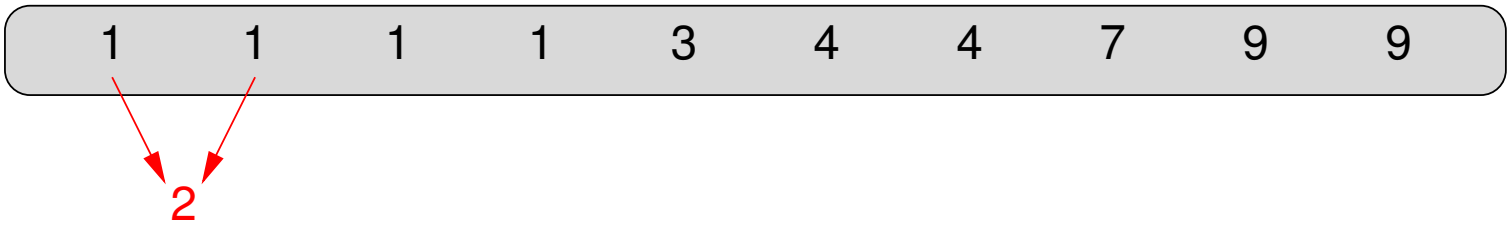
- What can be said about $\ell_{\max} = \max_i \ell_i$?
- How to implement the algorithm efficiently?
- How to implement the encoder and decoder? [not covered today]
- What if other constraints get added? [not covered today]
- What about arithmetic coding?

Huffman's Algorithm: Textbook Version

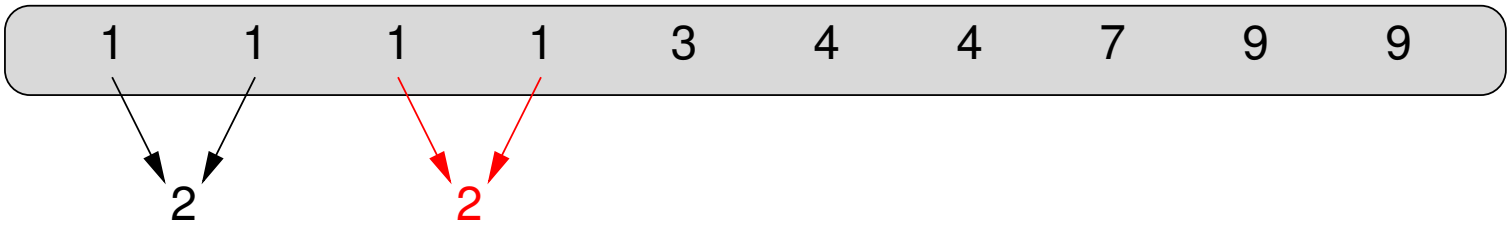

```
 0: function CalcHuffLens(⟨w_i⟩, n)
 1:     set Q ← [ ]
 2:     for each symbol s ∈ ⟨0 . . . n − 1⟩ do
 3:         set node ← new(leaf)
 4:         set node.symb ← s
 5:         set node.wght ← w_s
 6:         Insert(Q, node)
 7:     while |Q| > 1 do
 8:         set node_0 ← ExtractMin(Q)
 9:         set node_1 ← ExtractMin(Q)
10:         set node ← new(internal)
11:         set node.left ← node_0
12:         set node.rght ← node_1
13:         set node.wght ← node_0.wght + node_1.wght
14:         Insert(Q, node)
15:     set node ← ExtractMin(Q)
16:     return node, as the root of the constructed Huffman tree
```

Huffman's Algorithm: Example

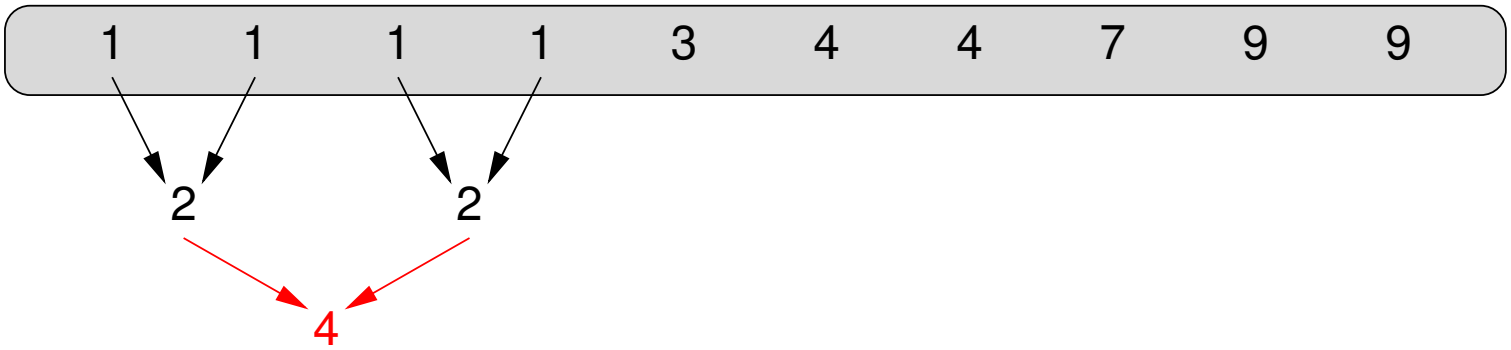
Huffman's Algorithm: Example



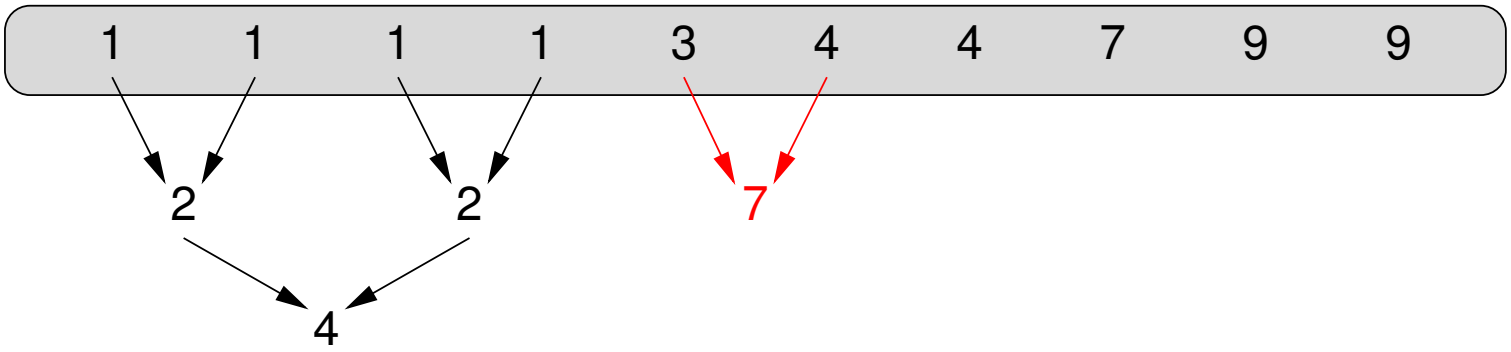
Huffman's Algorithm: Example



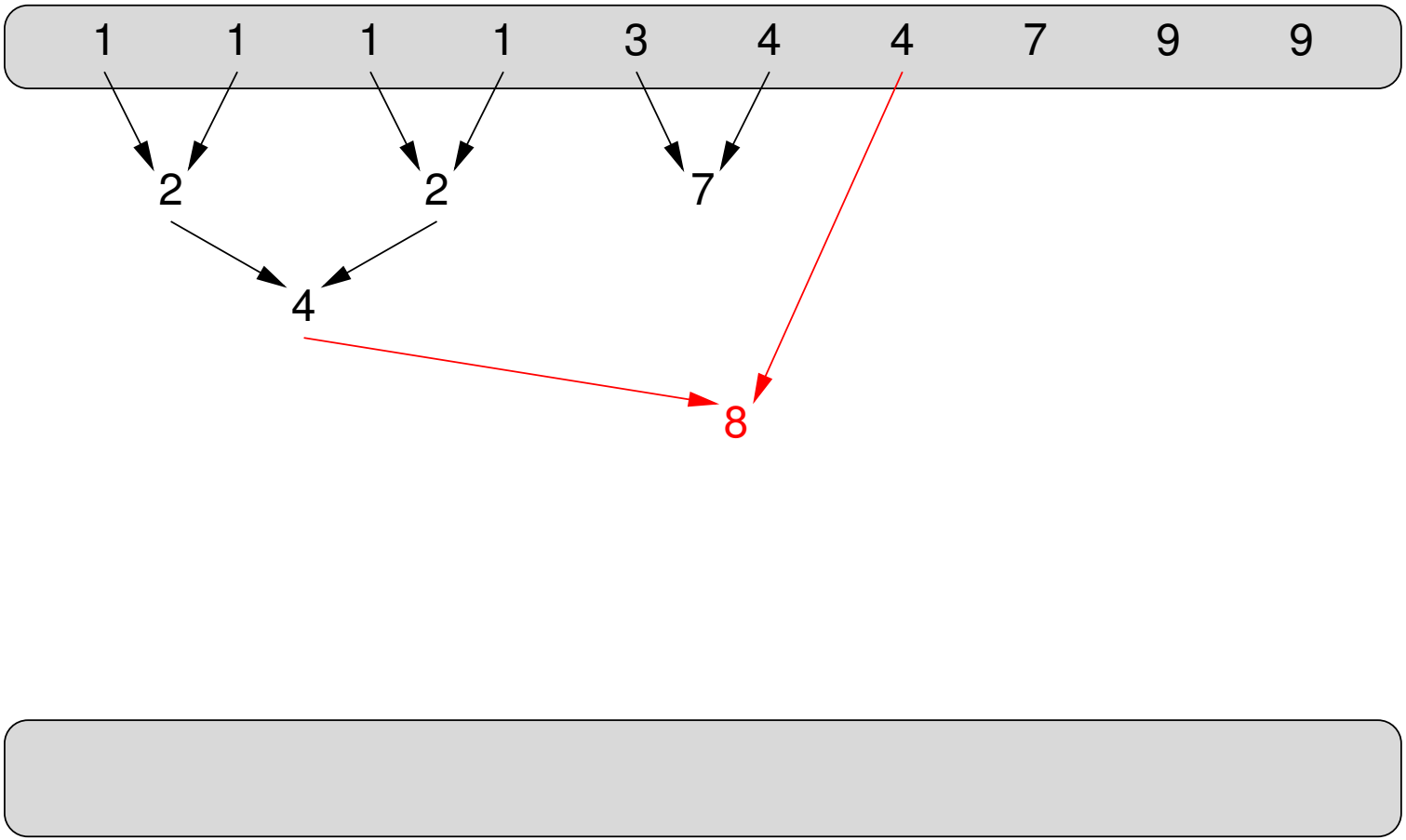
Huffman's Algorithm: Example



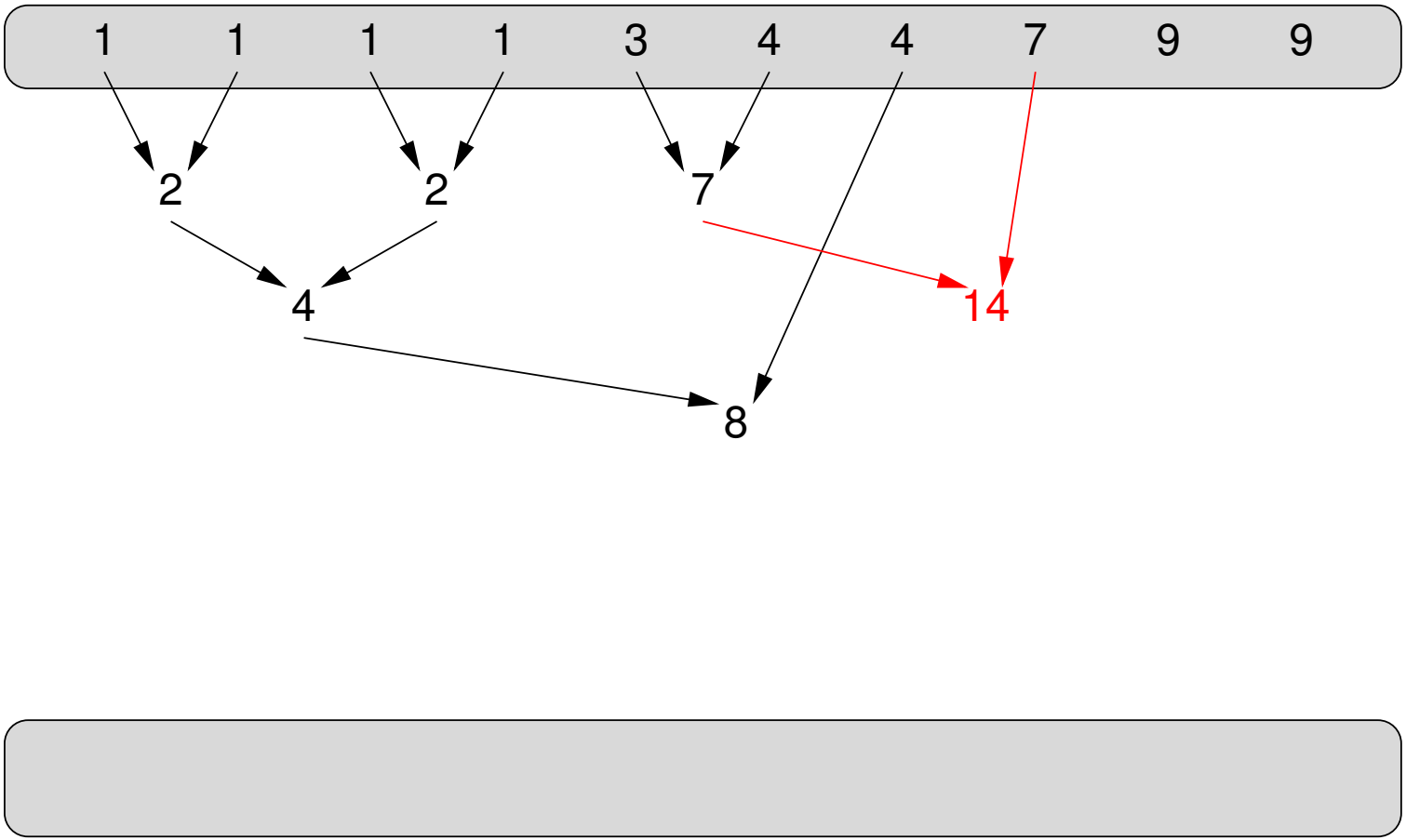
Huffman's Algorithm: Example



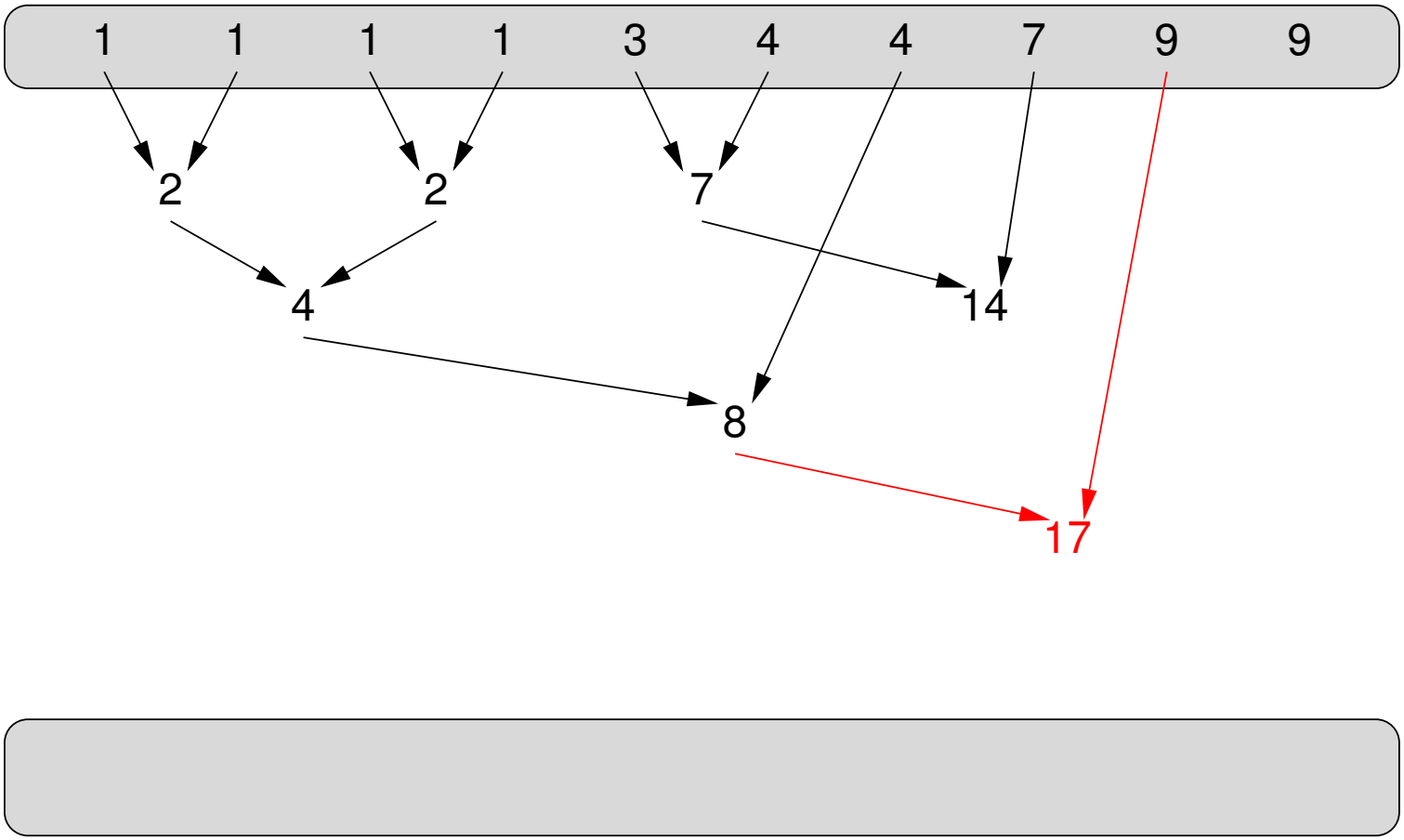
Huffman's Algorithm: Example



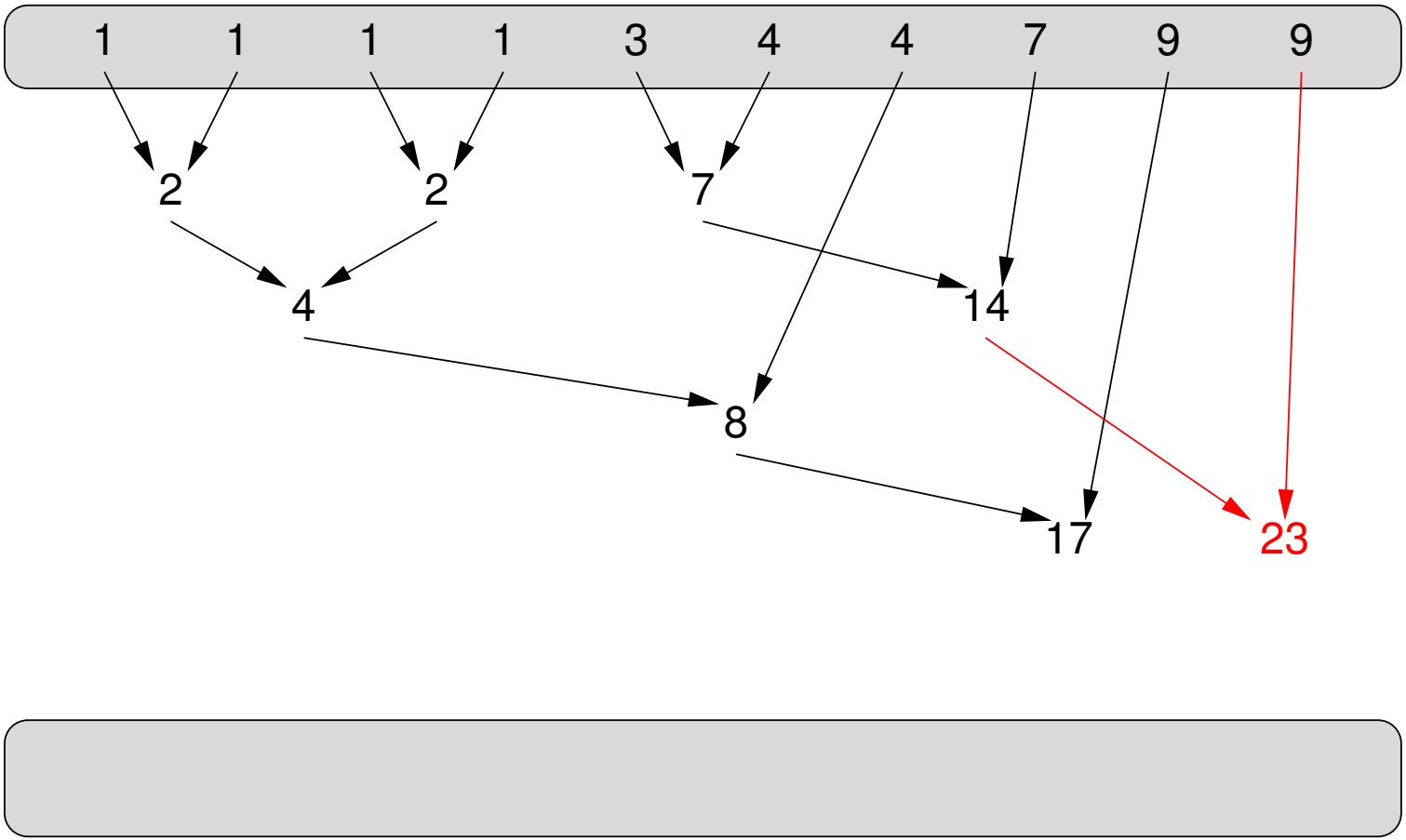
Huffman's Algorithm: Example



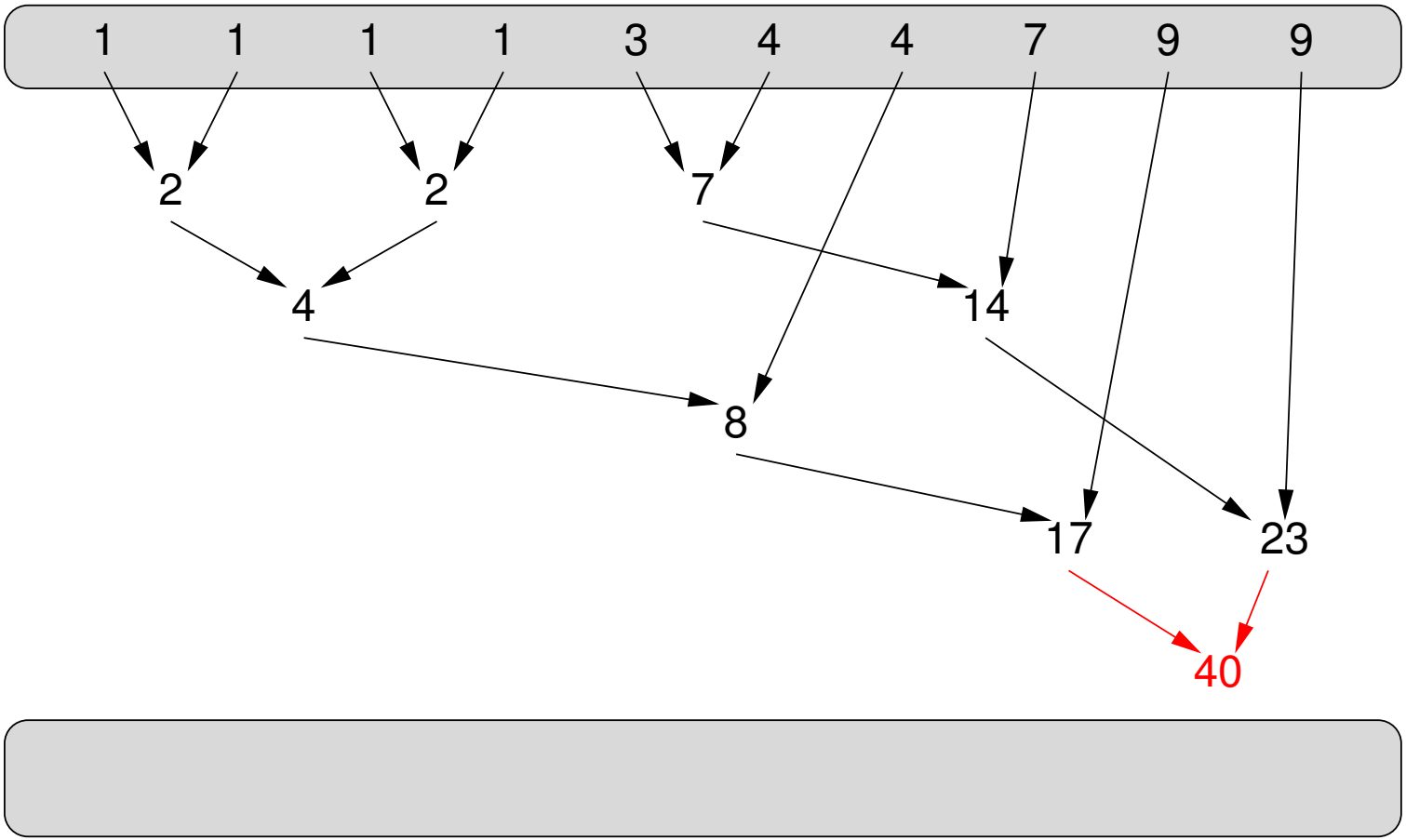
Huffman's Algorithm: Example



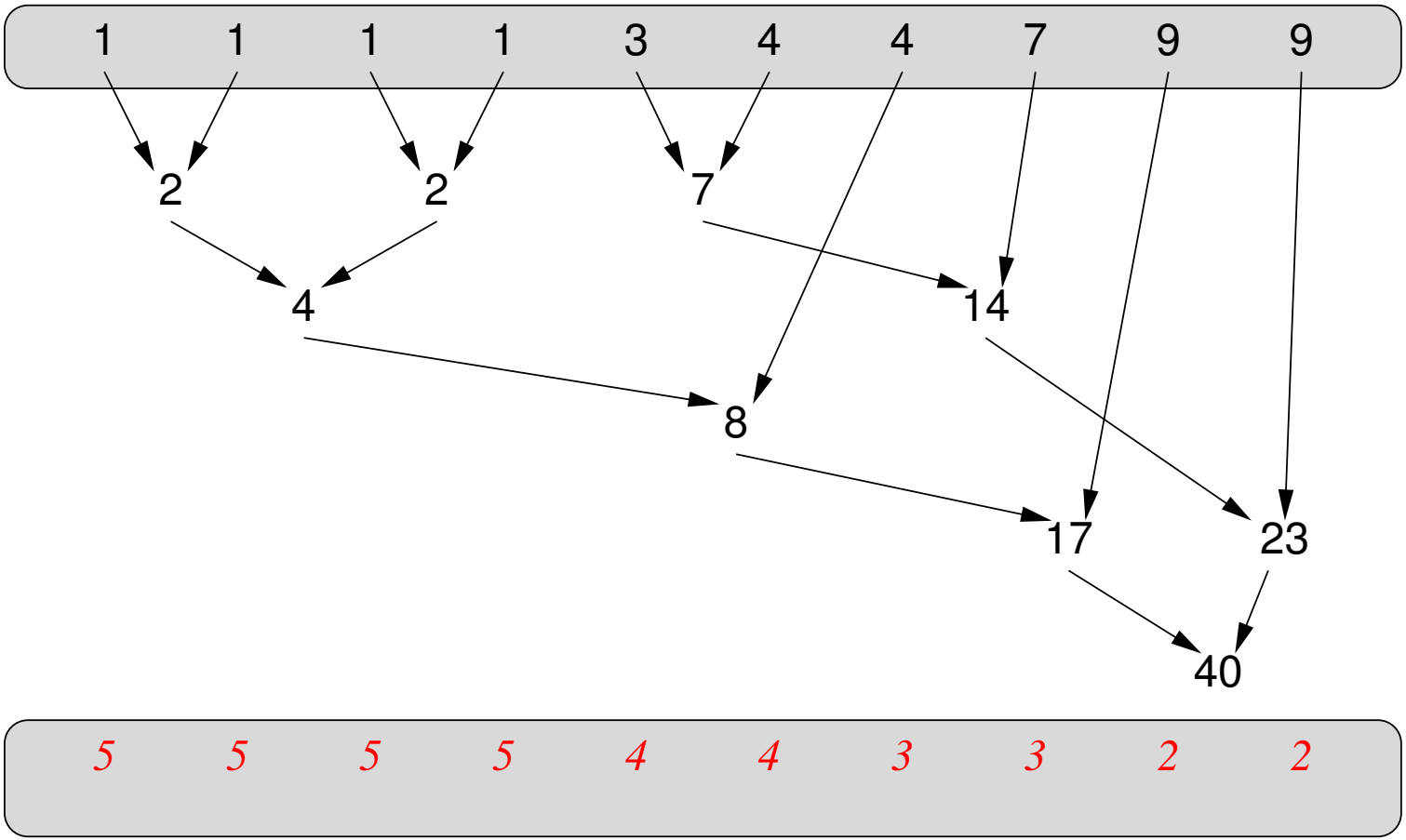
Huffman's Algorithm: Example



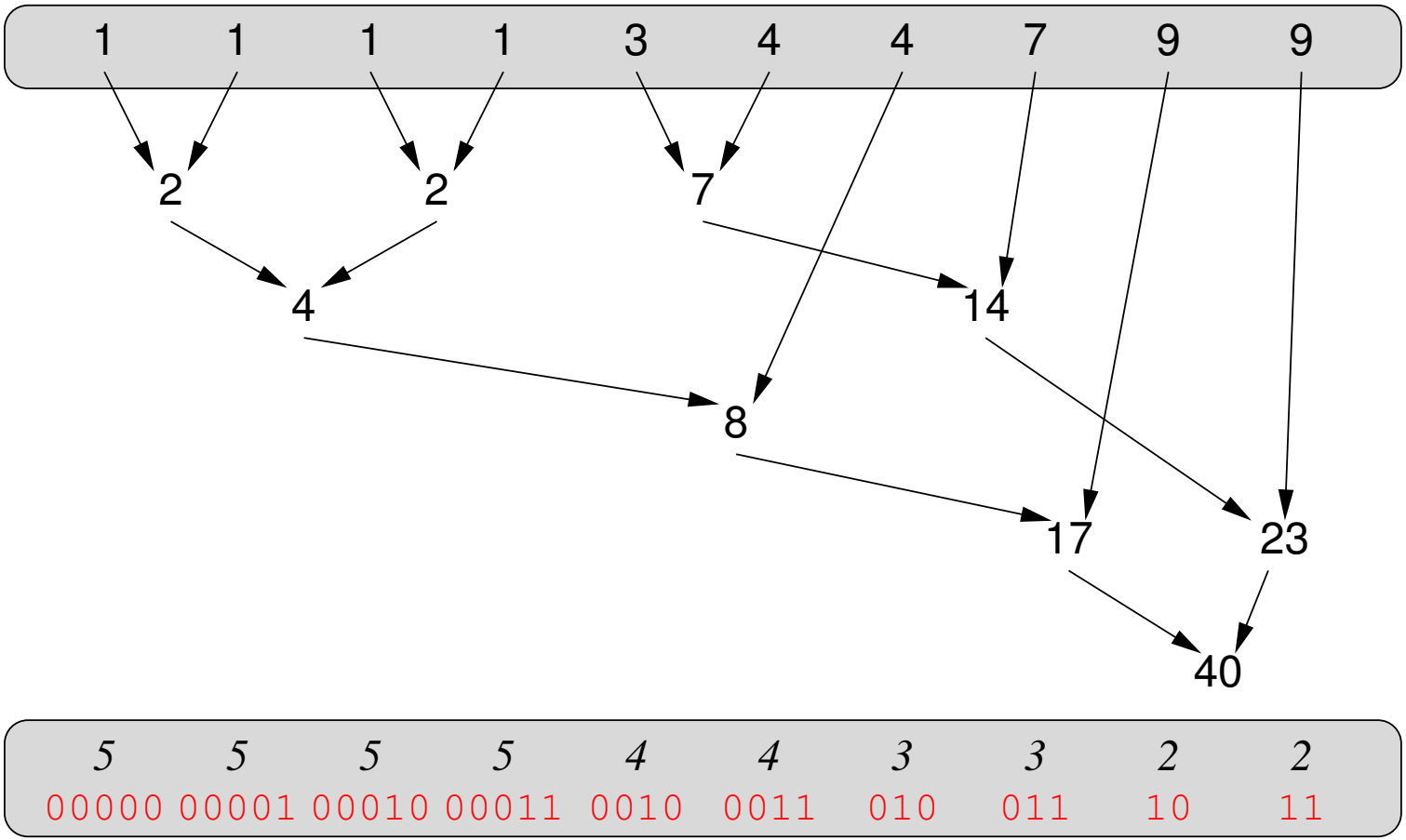
Huffman's Algorithm: Example



Huffman's Algorithm: Example



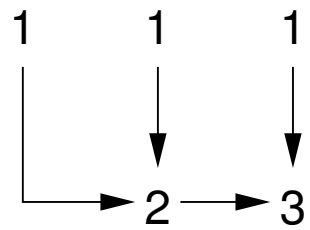
Huffman's Algorithm: Example



Bounds

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

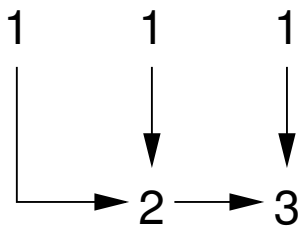
But if the weights are frequency counts, to get a codeword of length ℓ, some number $F'(\ell)$ symbols must be present:



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:

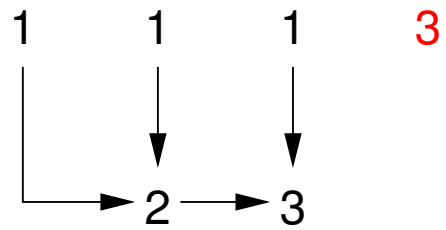


$F'(1) = 2,\ F'(2) = 3$

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:

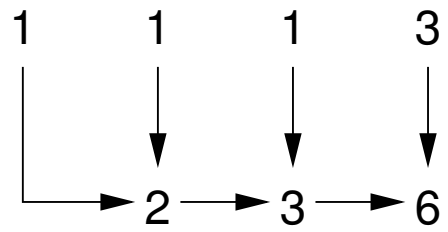


$F'(1) = 2,\ F'(2) = 3$

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:

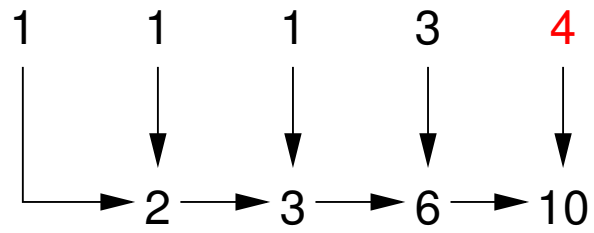


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6$

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:

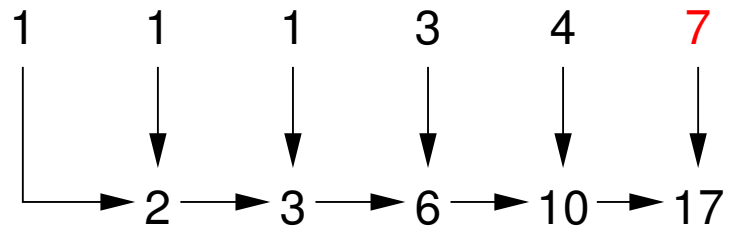


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10$

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:

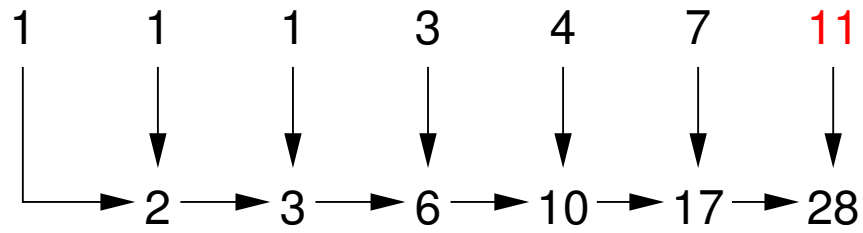


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

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:

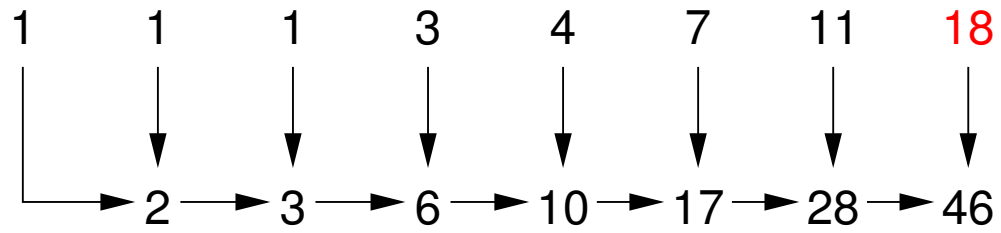


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

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:

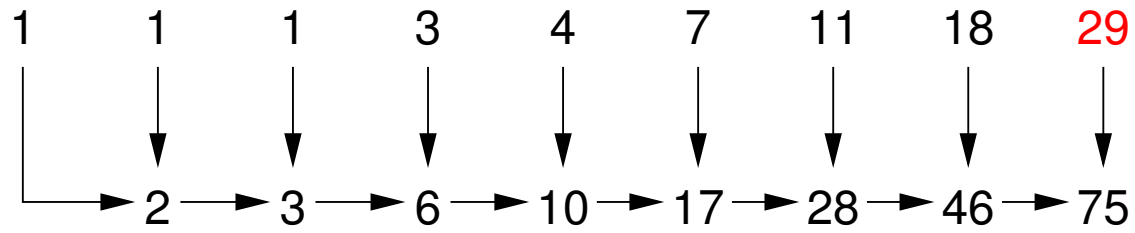


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

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:

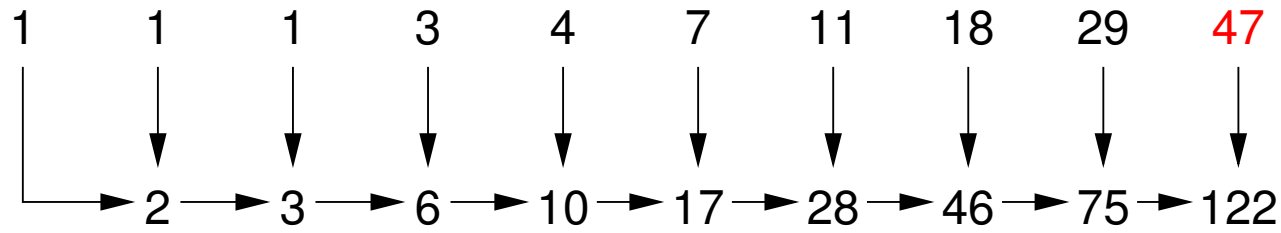


$F'(1) = 2,\ F'(2) = 3, F'(3) = 6, F'(4) = 10, F'(5) = 17$

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.

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?

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.

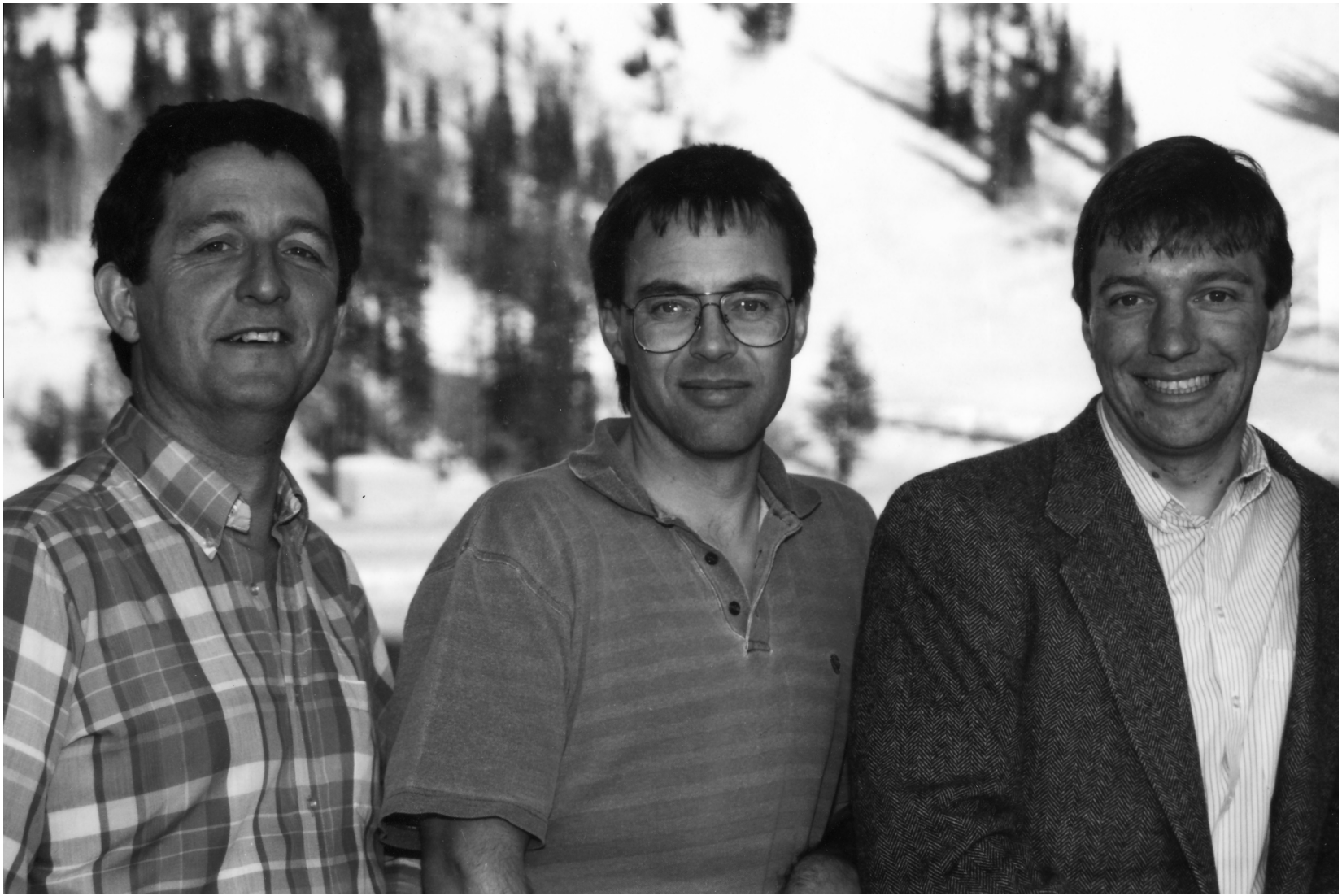
PPM is the perfect application for arithmetic coding [Cleary & Witten, 1984; Moffat, 1990; Bunton, 1997].

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

Bottom line: Huffman codes remain important.

Snowbird: 1991

Snowbird: 1994



Snowbird: 1996