

# A Crash Course on Data Compression

## 4. Statistical Compressors

**Giulio Ermanno Pibiri**

ISTI-CNR, [giulio.ermanno.pibiri@isti.cnr.it](mailto:giulio.ermanno.pibiri@isti.cnr.it)



@giulio\_pibiri



@jermmp

# Overview

- Shannon-Fano
- Huffman, Canonical Huffman
- Arithmetic
- Asymmetric Numeral Systems

# The *Statistical* Coding Problem

- **Problem.** We are given a list  $L[1..n]$  of  $n$  symbols and we are asked to compress it in as few as possible bits.
- Without loss of generality, we are going to assume that the symbols are positive integers.
- **Idea.** Build a *statistical model* for  $L$  and assign codewords *based on the model*.
- **Simplest statistical model.** Occurrences of each *distinct* symbol  $\langle s_1, \dots, s_m \rangle$  in  $L$ . We will refer to these occurrences as the *weights*  $\langle w_1, \dots, w_m \rangle$  for the symbols  $\langle s_1, \dots, s_m \rangle$ . It follows that  $\sum_{i=1}^m w_i = n$ .

Example for  $L = [1, 3, 1, 1, 1, 5, 2, 1, 7, 3, 1, 2, 1, 1, 1, 1]$ . We have  $n = 16$  and  $m = 5$ .  
The distinct symbols  $\langle s_1, \dots, s_5 \rangle$  are  $\langle 1, 2, 3, 5, 7 \rangle$  and the weights  $\langle w_1, \dots, w_5 \rangle$  are  $\langle 10, 2, 2, 1, 1 \rangle$ .

# The *Minimum-Redundancy* Coding Problem

- **Problem.** We are given a list  $L[1..n]$  of  $n$  symbols with weights  $W = \langle w_1, \dots, w_m \rangle$ , where  $m \leq n$  and  $\sum_{i=1}^m w_i = n$ . We are asked to determine a set of *codeword lengths*  $T = \langle \ell_1, \dots, \ell_m \rangle$  such that:

- (1)  $\sum_{i=1}^m 2^{-\ell_i} \leq 1$  (Kraft-McMillan inequality compliancy)
- (2) the cost  $C(W, T) = \sum_{i=1}^m (w_i \cdot \ell_i)$  bits of the coded  $L$  is *minimum*.

We will refer to the set of codeword lengths  $T = \langle \ell_i \rangle$  as the *code*. A code  $T$  satisfying **(1)** and **(2)** above is said to be an optimal or *minimum-redundancy* code for  $W$ , that is:  
 $C(W, T) \leq C(W, T')$  for any other code  $T' = \langle \ell'_i \rangle$ .

- Once the set  $T$  has been determined, it is easy to assign *prefix-free* codewords.

# Assigning *Canonical* Prefix-Free Codewords

- Given the set of codeword lengths  $\langle \ell_1, \dots, \ell_m \rangle$ , in non-decreasing order:
- Algorithm.**
  - The first codeword is  $C_1 = 0^{\ell_1}$ . Now let  $\ell = \ell_1$ .
  - For all  $i = 2, \dots, m$ :
    - Let  $C_i$  be the smallest unassigned *lexicographic* codewords of  $\ell$  bits (the one “coming after”  $C_{i-1}$ ).  
If  $\ell_i = \ell$ , then return  $C_i$ . Otherwise ( $\ell_i > \ell$ ),  $C_i$  is padded with possible 0s to the right until a codeword of  $\ell_i$  is obtained.
    - Set  $\ell = \ell_i$ .
- This assignment is also known as *canonical* because it allows fast encoding/decoding using simple tables.  
(Note that the codewords are *lexicographically* sorted.)

Example for  $\langle \ell_i \rangle = \langle 1, 2, 4, 4, 4, 4 \rangle$ .

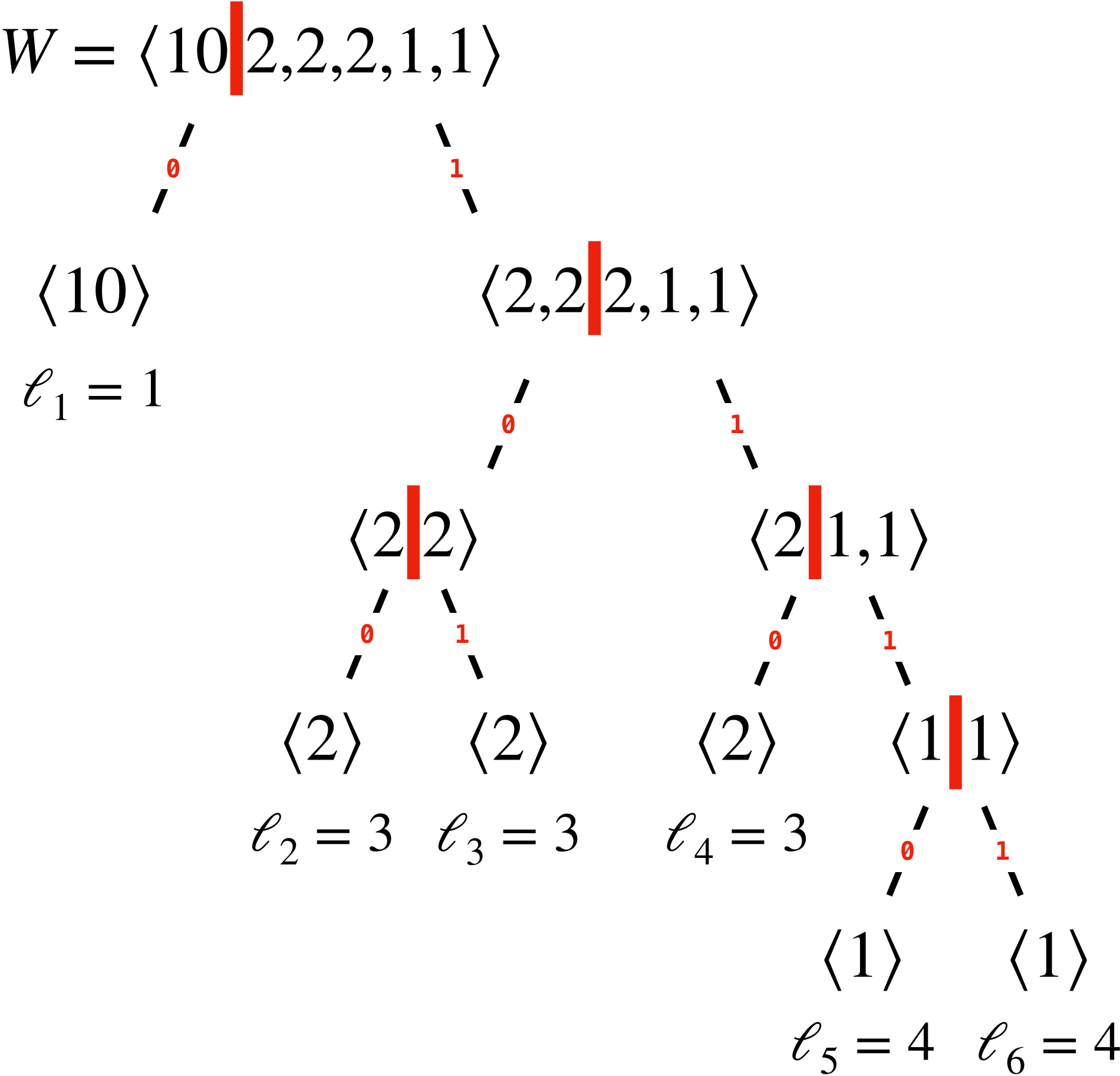
$\ell_i$	$C_i$
1	0
2	10
4	1100
4	1101
4	1110
4	1111

# Shannon-Fano

Shannon, 1949 — Fano, 1949

- A greedy, top-down, algorithm to compute a set of codeword lengths.
- It does *not always* produce an optimal prefix-free code.
- Remember the “golden rule” of data compression: assign shorter codewords to more frequent symbols.
- **Algorithm.**
  - Sort the set  $\langle w_i \rangle$  in non-increasing order and partition it into two sets  $\langle w'_i \rangle$  and  $\langle w''_i \rangle$ , of size  $m'$  and  $m - m'$  respectively, so that  $\sum_{i=1}^{m'} w'_i \approx \sum_{i=m'+1}^{m-m'} w''_i$ .
  - Proceed recursively on each of the two partitions. Stop when the partition size is 1.
- Informally: the codeword length  $\ell_i$  for the weight  $w_i$  is the number of partitions that contain  $w_i$ .

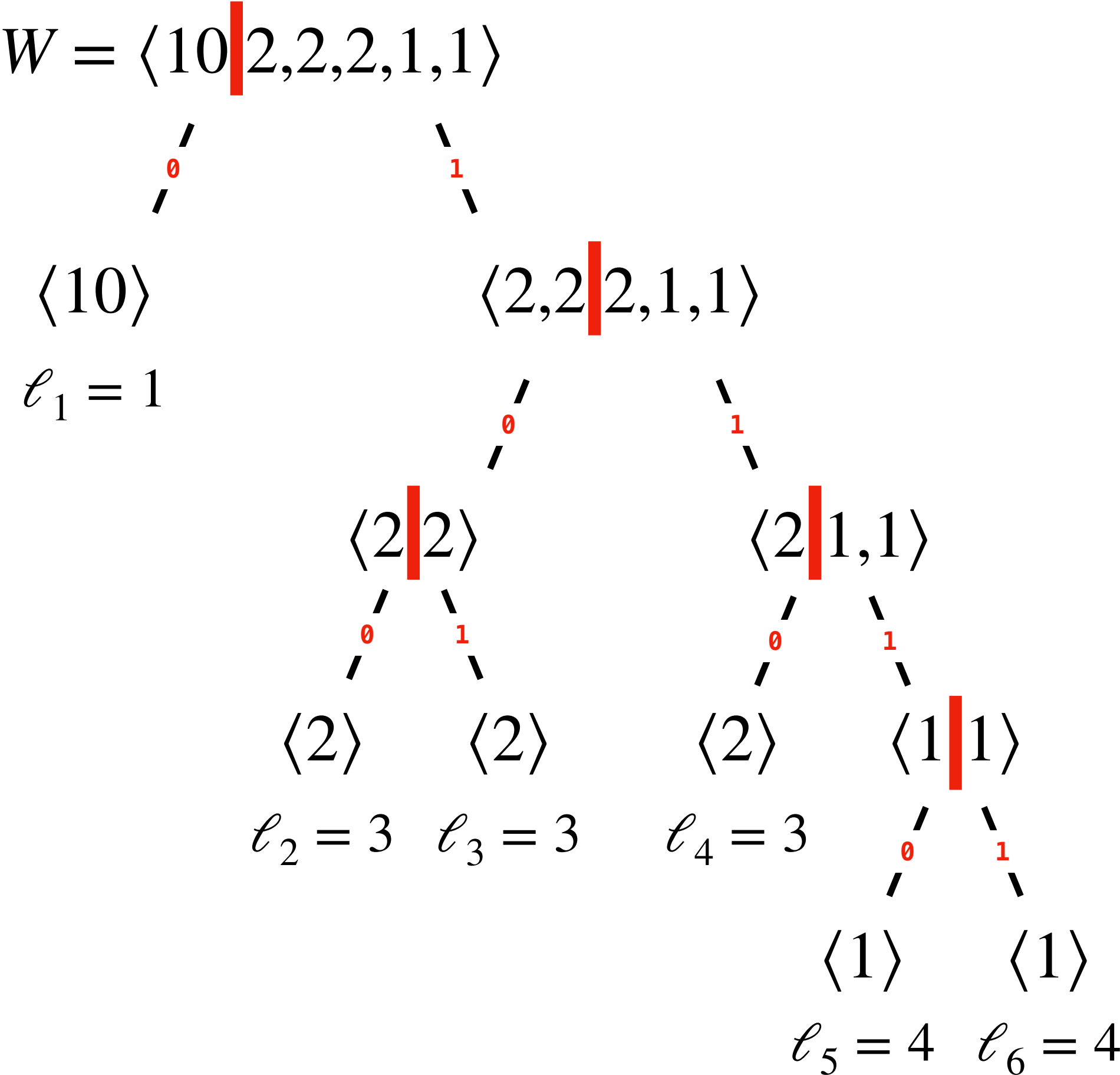
# Shannon-Fano — Example



$$\langle \ell_i \rangle = \langle 1, 3, 3, 3, 4, 4 \rangle$$

$\ell_i$	$C_i$
1	0
3	100
3	101
3	110
4	1110
4	1111

# Shannon-Fano — Example



$$\langle \ell_i \rangle = \langle 1, 3, 3, 3, 4, 4 \rangle$$

$\ell_i$	$C_i$
1	0
3	100
3	101
3	110
4	1110
4	1111

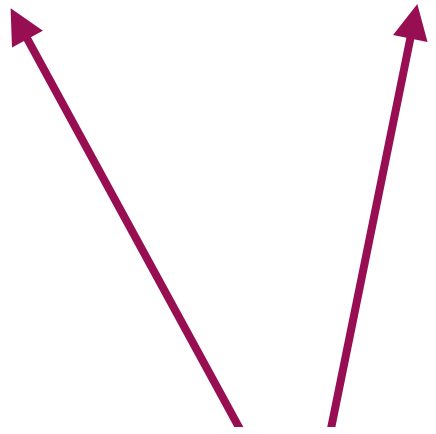
- Because of the approximate partitioning rule, the resulting code may not be optimal.
- It is easy to see why the algorithm produces a *prefix-free* code: each symbol is associated to a distinct leaf and two siblings are distinguished by two different symbols, 0 and 1.





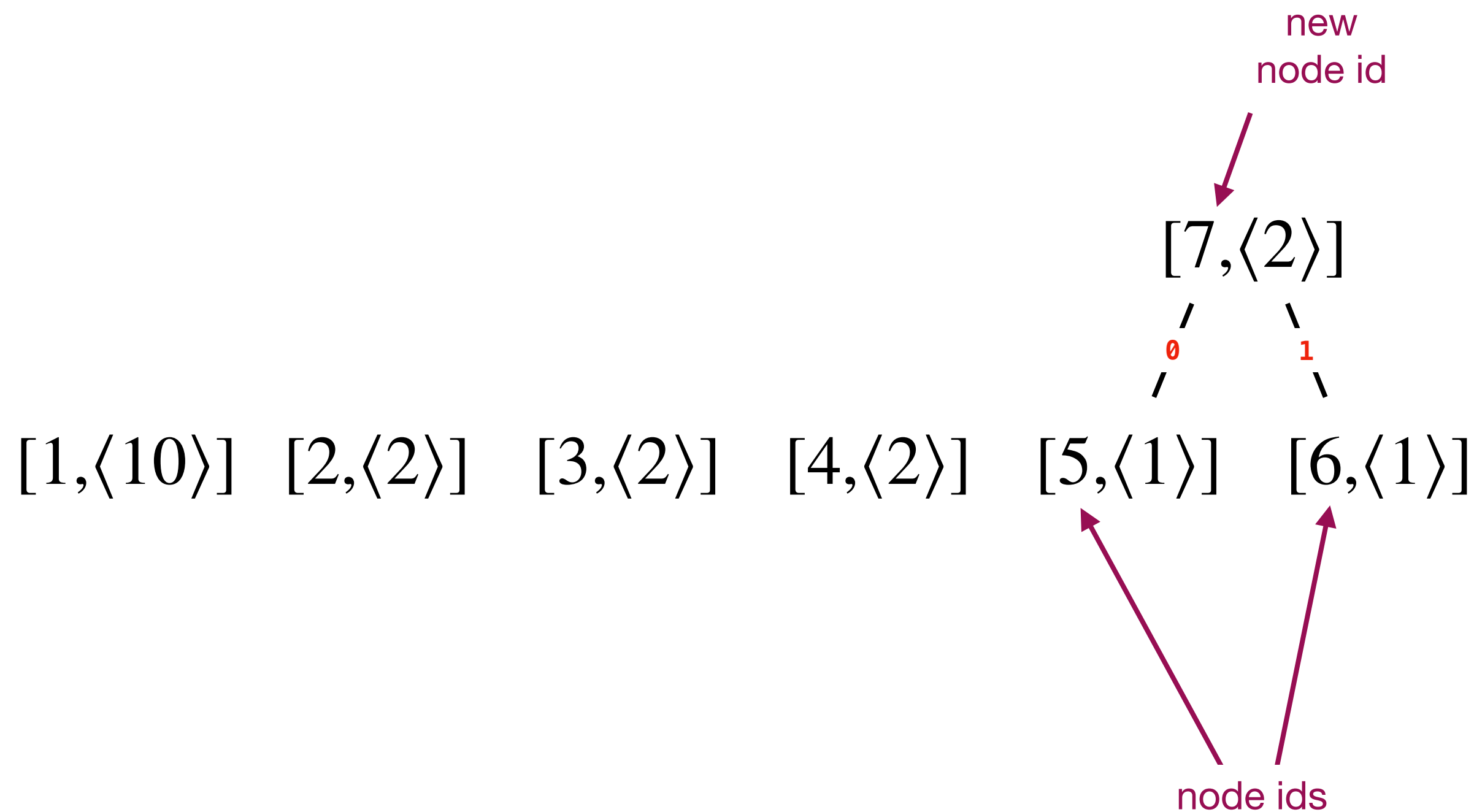
# Huffman — Demo

$[1, \langle 10 \rangle]$   $[2, \langle 2 \rangle]$   $[3, \langle 2 \rangle]$   $[4, \langle 2 \rangle]$   $[5, \langle 1 \rangle]$   $[6, \langle 1 \rangle]$   $S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]\}$



node ids

# Huffman — Demo

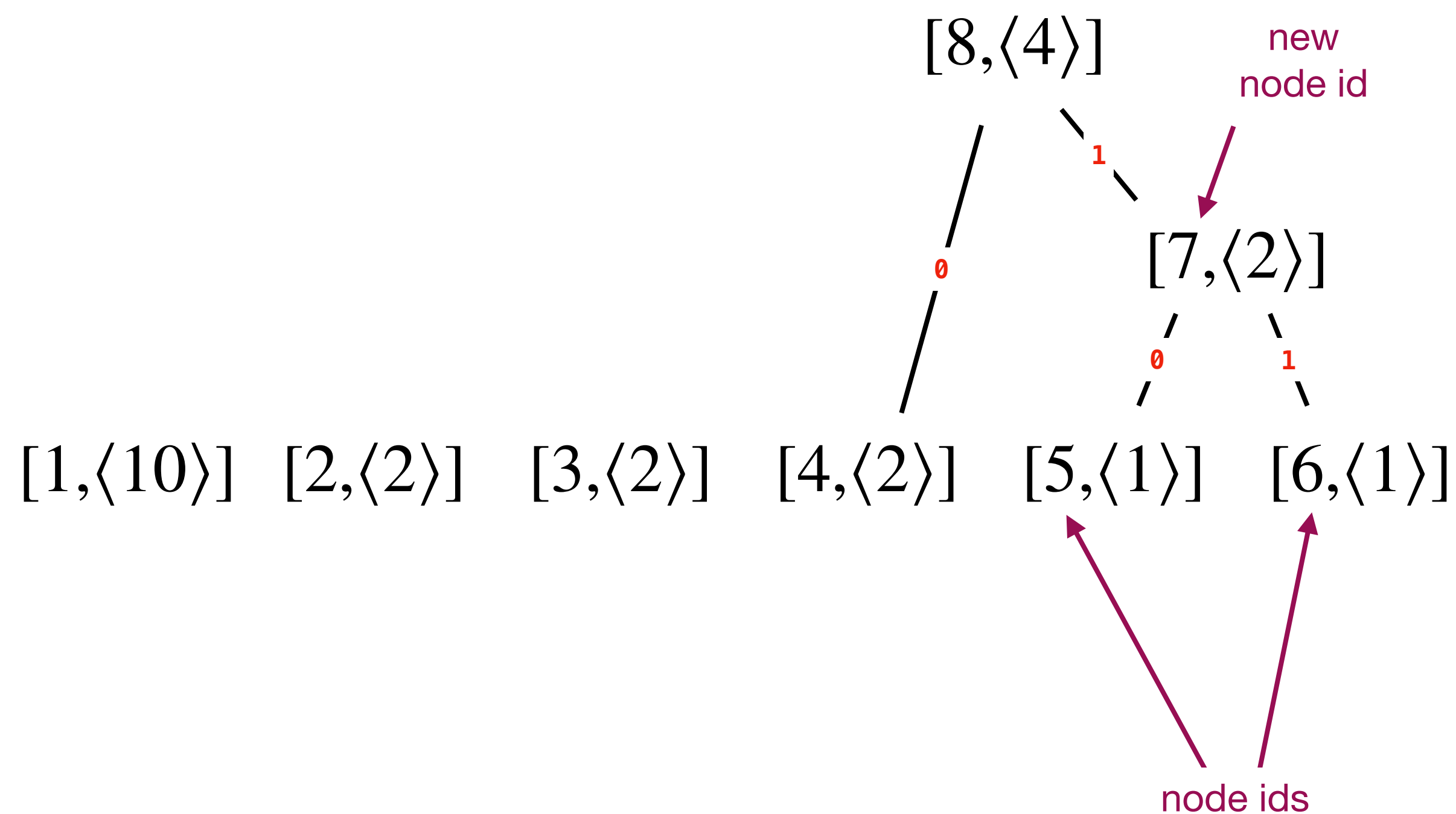


children of  
node 7

$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [7, \langle 2 \rangle], (5, 6)]\}$

$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]]\}$

# Huffman — Demo



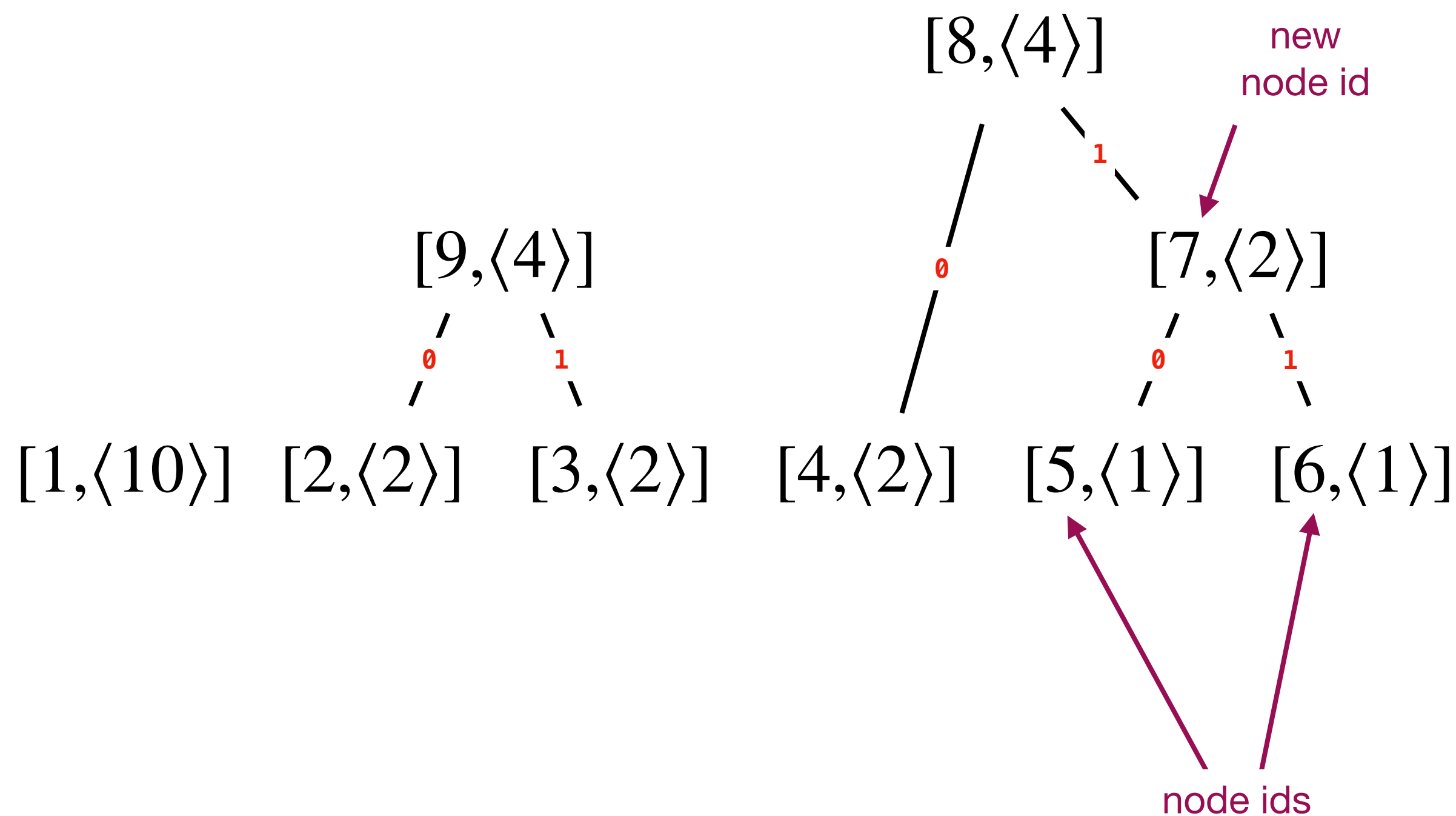
$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [8, \langle 4 \rangle], (4, 7)]\}$

$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [7, \langle 2 \rangle], (5, 6)]\}$

$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]]\}$

children of node 7

# Huffman — Demo



$S = \{[1, \langle 10 \rangle], [9, \langle 4 \rangle, (2, 3)], [8, \langle 4 \rangle, (4, 7)]\}$

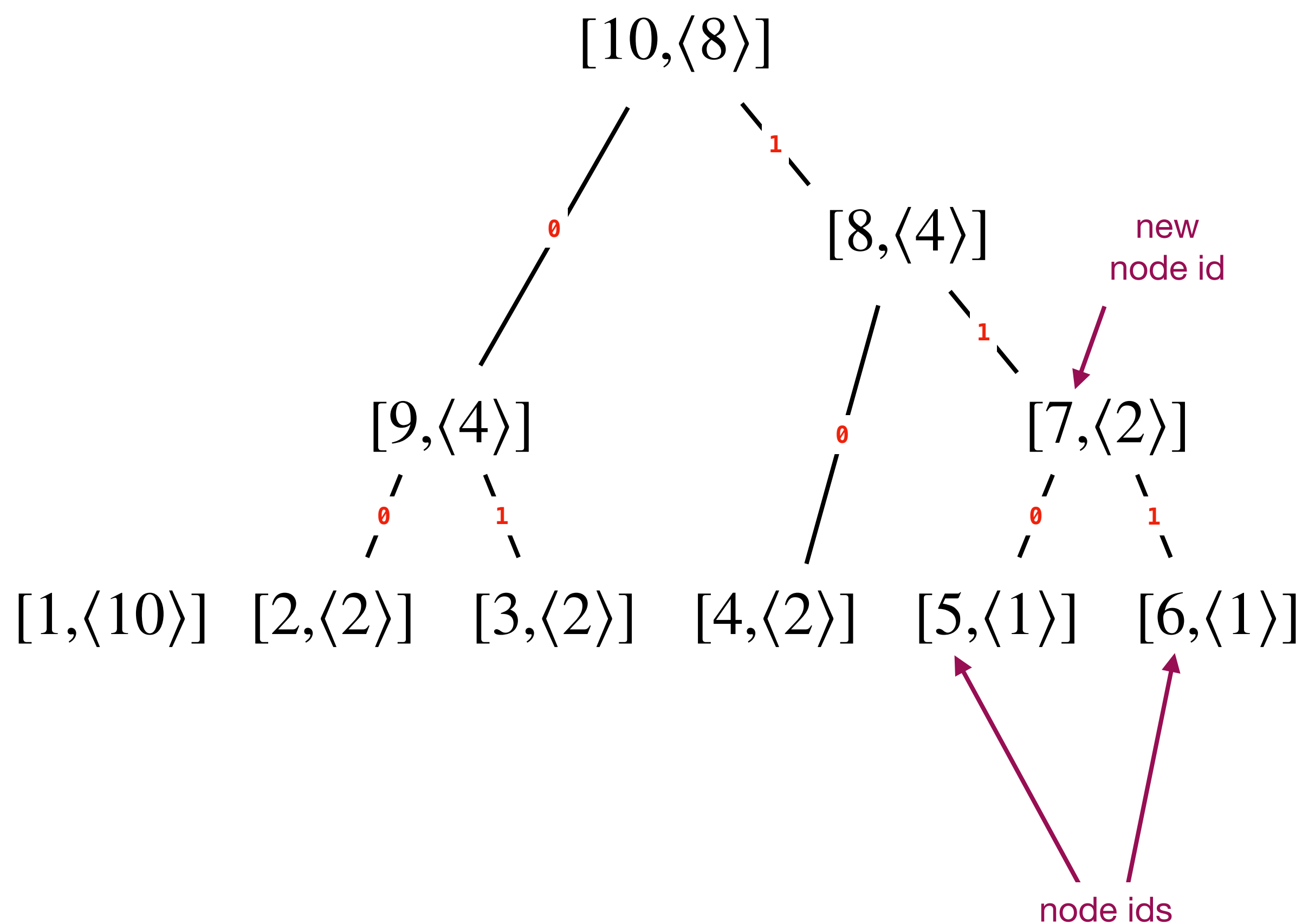
$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [8, \langle 4 \rangle, (4, 7)]\}$

children of node 7

$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [7, \langle 2 \rangle, (5, 6)]\}$

$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]\}$

# Huffman — Demo



$$S = \{[1, \langle 10 \rangle], [10, \langle 8 \rangle], (9, 8)]\}$$

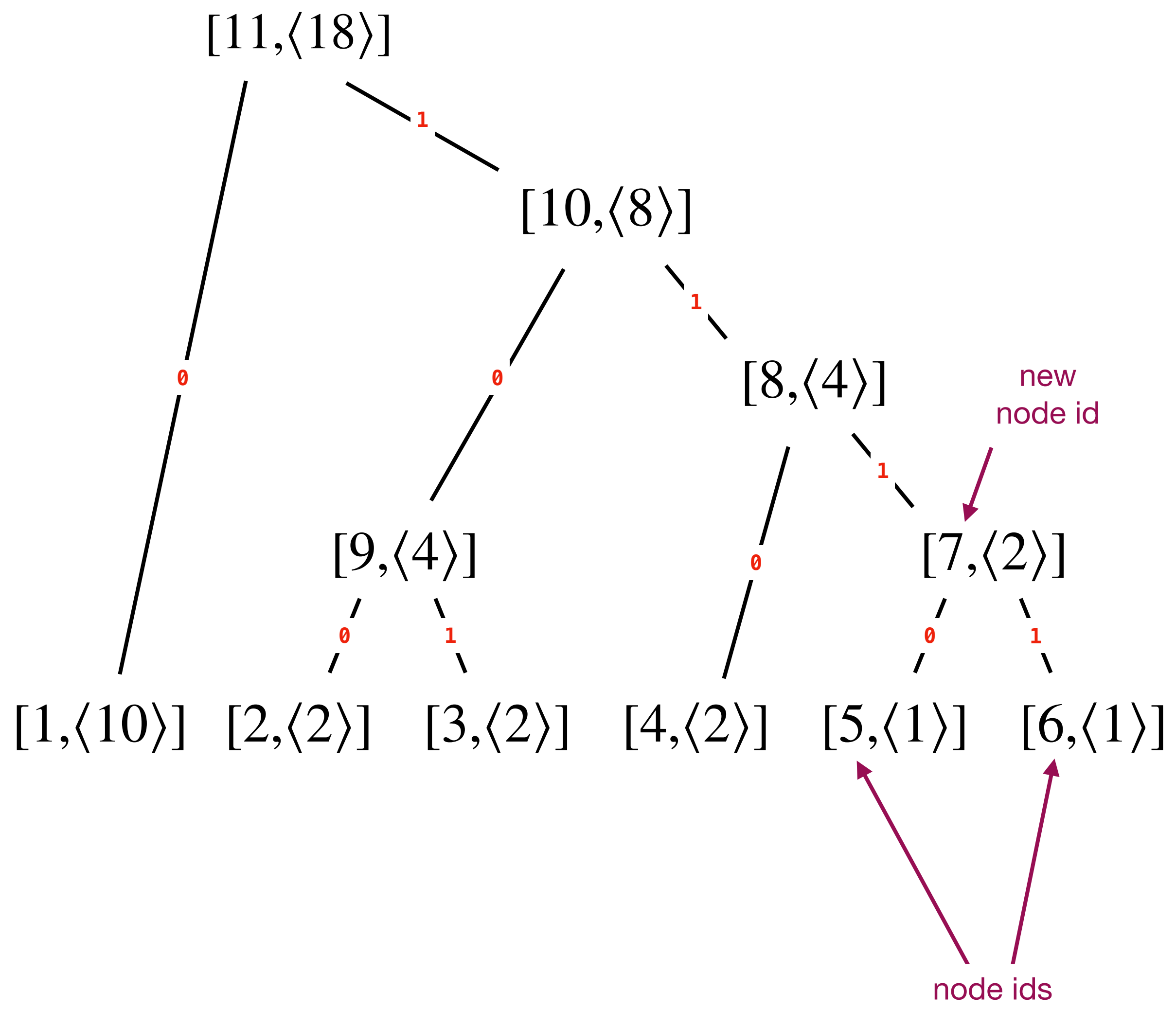
$$S = \{[1, \langle 10 \rangle], [9, \langle 4 \rangle], (2, 3)], [8, \langle 4 \rangle], (4, 7)]\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [8, \langle 4 \rangle], (4, 7)]\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [7, \langle 2 \rangle], (5, 6)]\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]]\}$$

# Huffman — Demo



$$S = \{[11, \langle 18 \rangle], (1, 10)\}$$

$$S = \{[1, \langle 10 \rangle], [10, \langle 8 \rangle], (9, 8)\}$$

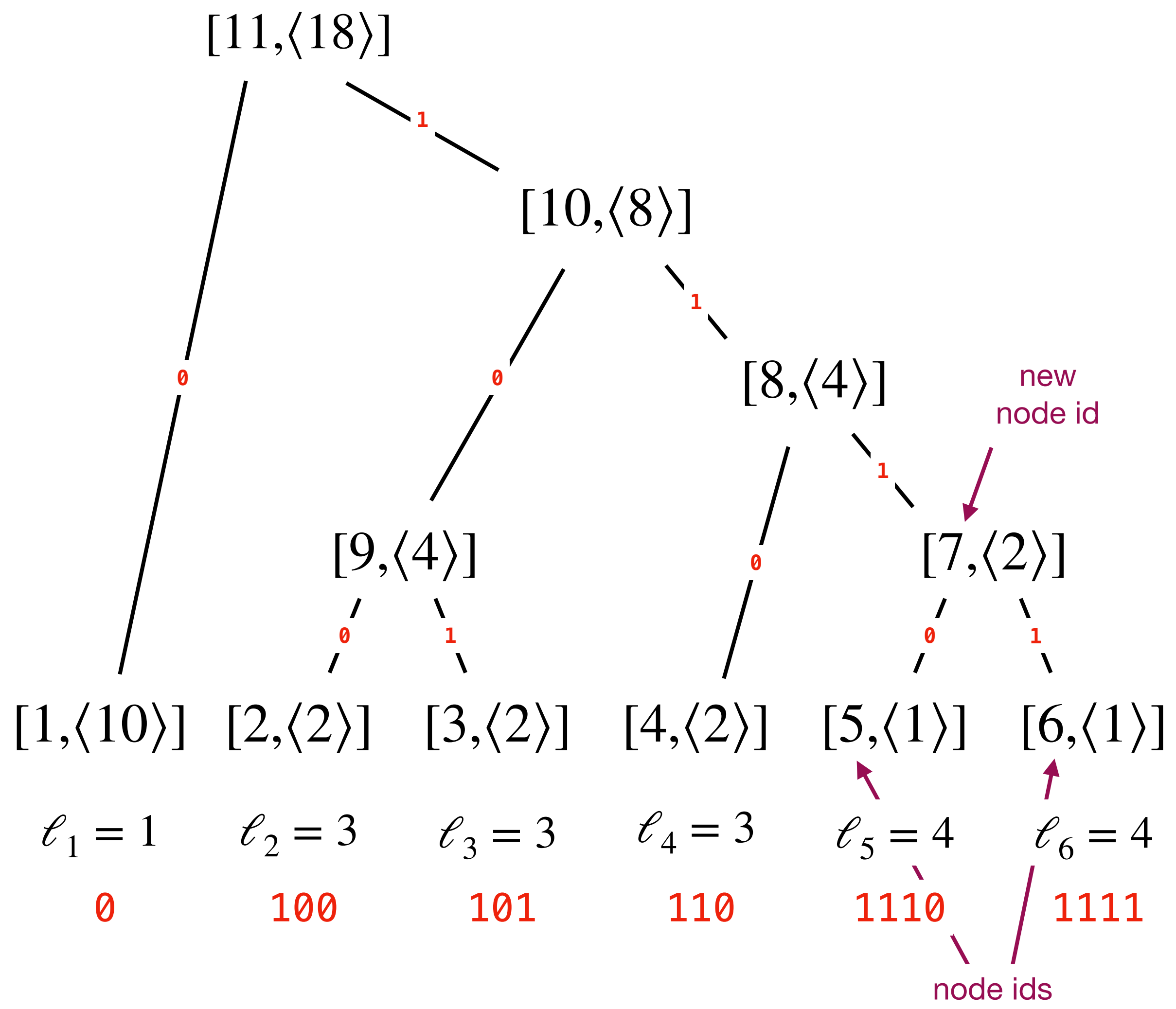
$$S = \{[1, \langle 10 \rangle], [9, \langle 4 \rangle], (2, 3), [8, \langle 4 \rangle], (4, 7)\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [8, \langle 4 \rangle], (4, 7)\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [7, \langle 2 \rangle], (5, 6)\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]\}$$

# Huffman — Demo



$$S = \{[11, \langle 18 \rangle], (1, 10)\}$$

$$S = \{[1, \langle 10 \rangle], [10, \langle 8 \rangle], (9, 8)\}$$

$$S = \{[1, \langle 10 \rangle], [9, \langle 4 \rangle], (2, 3), [8, \langle 4 \rangle], (4, 7)\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [8, \langle 4 \rangle], (4, 7)\}$$

$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [7, \langle 2 \rangle], (5, 6)\}$$

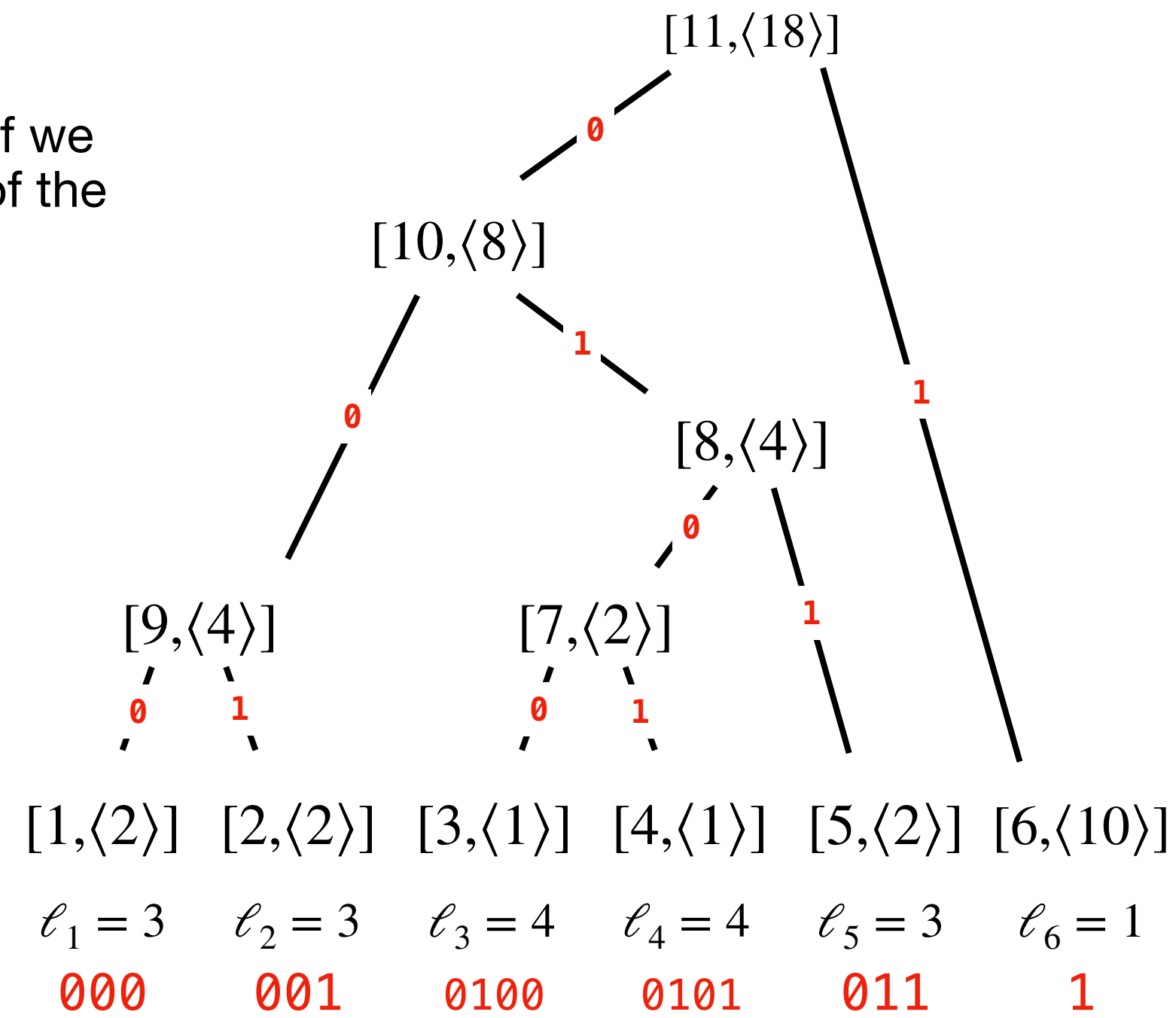
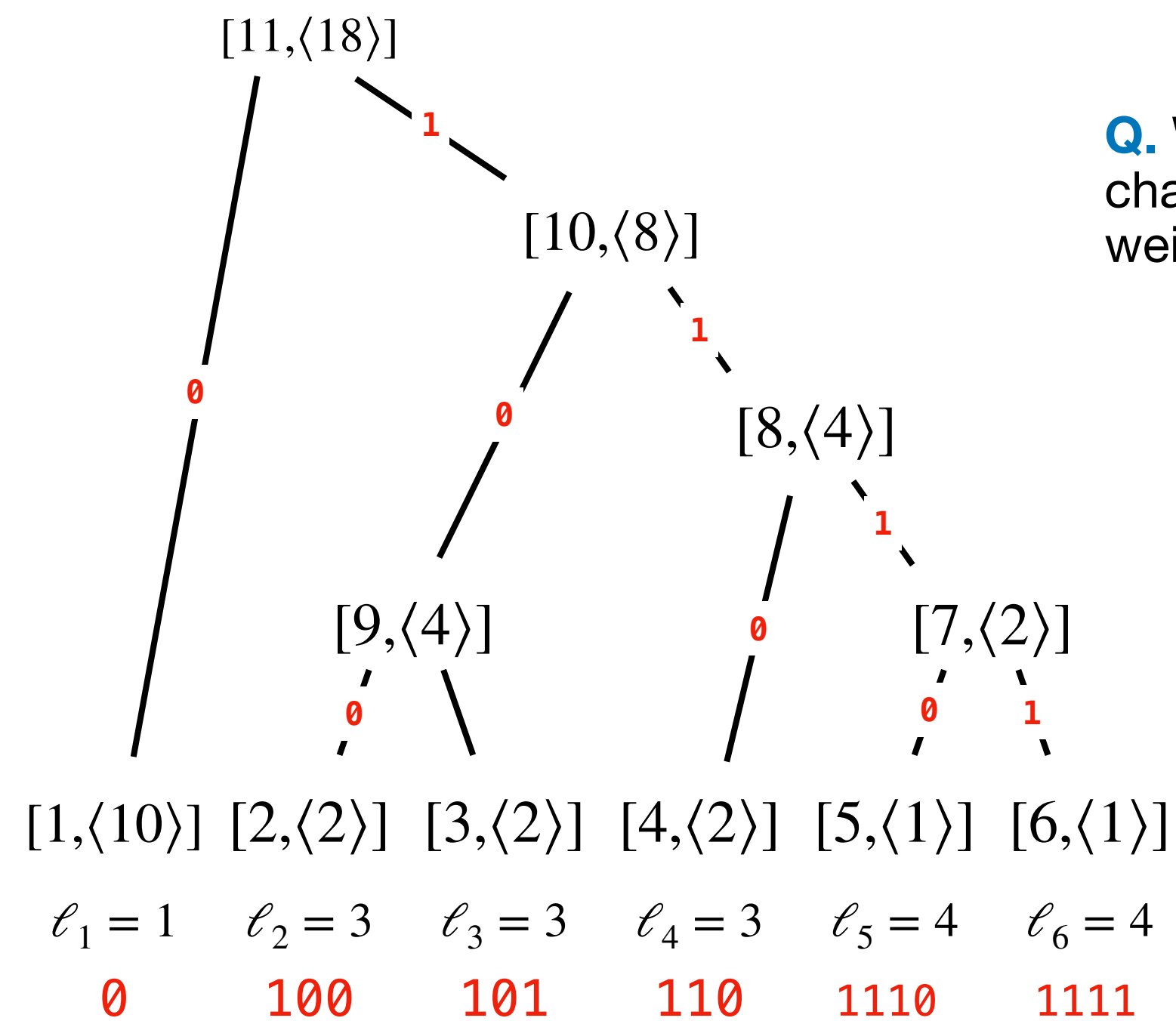
$$S = \{[1, \langle 10 \rangle], [2, \langle 2 \rangle], [3, \langle 2 \rangle], [4, \langle 2 \rangle], [5, \langle 1 \rangle], [6, \langle 1 \rangle]\}$$

← codeword lengths

← actual codewords

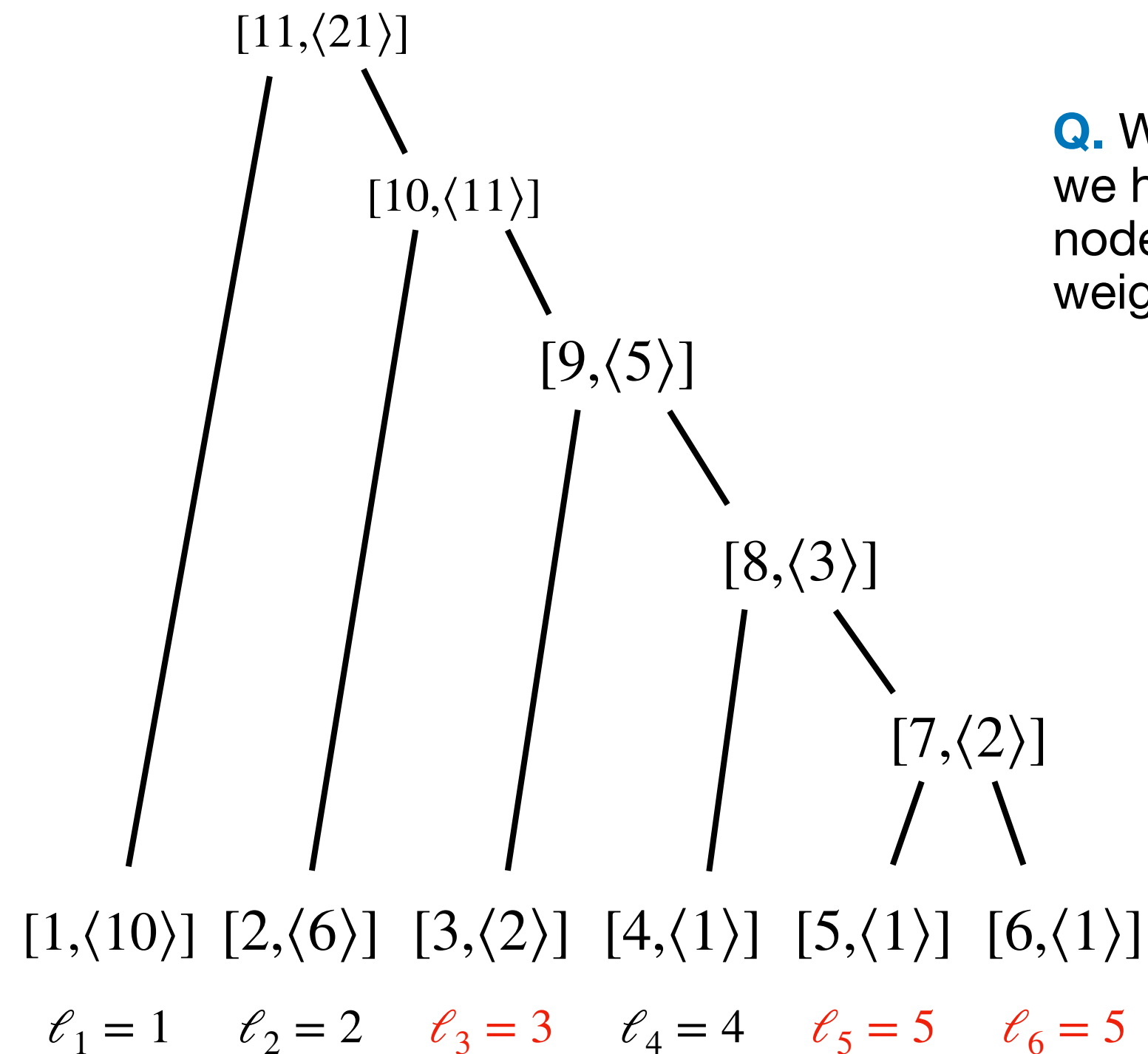


# Huffman — Different Orderings of the Weights



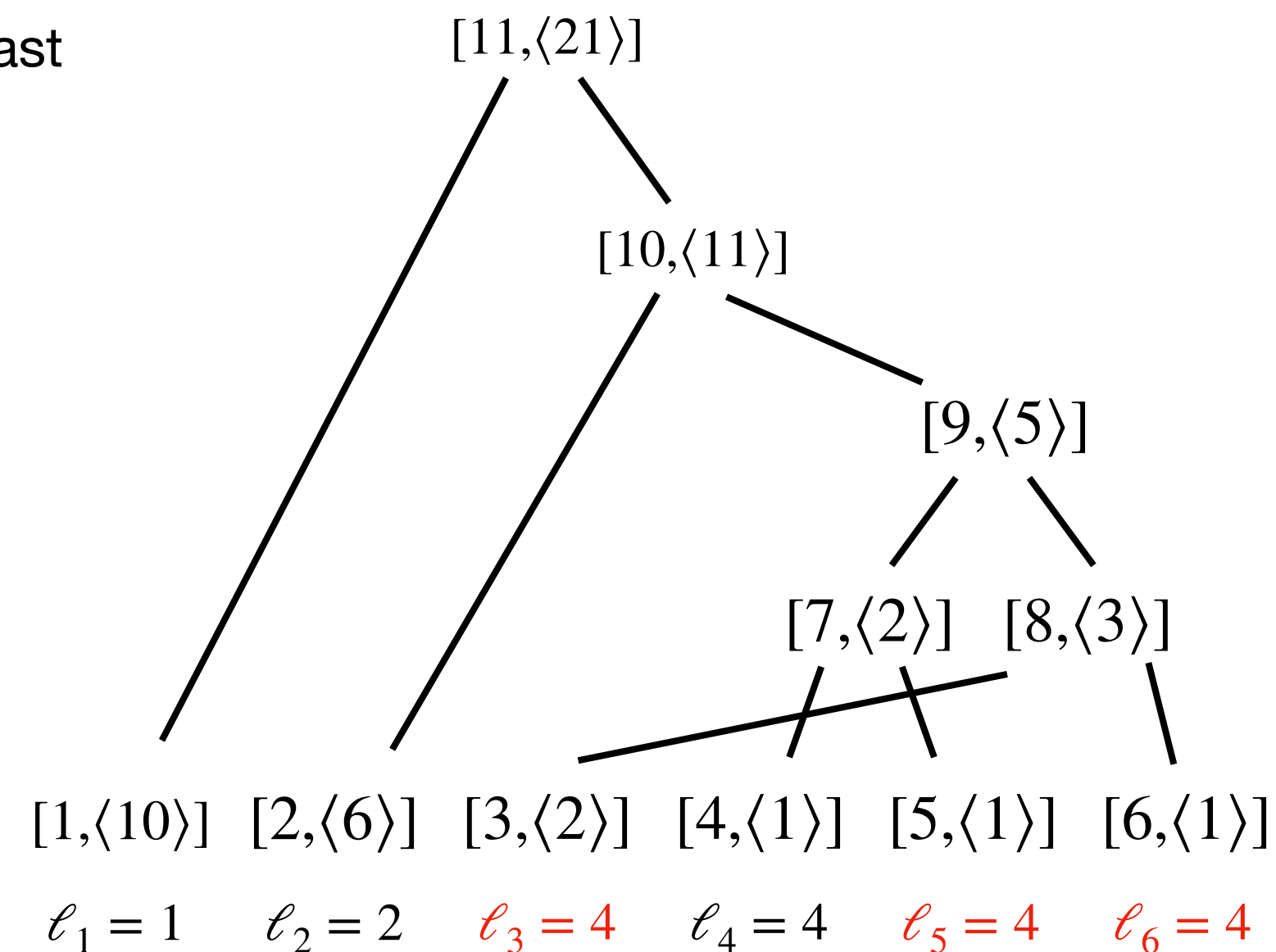
- Same codeword lengths, but different actual codewords.
- Labelling *left*-leaning edges with a 0 and *right*-leaning edges with a 1, we have that: different orderings of the weights can produce *different* sets of *valid* codewords.

# Huffman — Different Merging Strategies



(a) **Higher node ids** first

**Q.** What happens when we have more than two nodes with the same least weight?



(b) **“Older” nodes first** (created farthest in the past)

- Same *average* codeword length, but different *maximum* codeword length.
- Different merging strategies can produce different sets of codeword lengths.
- In the example, we have  $\langle 1, 2, 3, 4, 5, 5 \rangle$  in (a) and  $\langle 1, 2, 4, 4, 4, 4 \rangle$  in (b). In both cases the average codeword length is 2.

if node x has a smaller id than node y, then it was created before y

# Canonical Huffman

- If the weights are *sorted in non-increasing order* and we label *left*-leaning edges with a 0 and *right*-leaning edges with a 1, then we always obtain *lexicographic sorted codewords*.
- We refer to this construction as the *canonical* Huffman code.
- Also, we can always merge nodes that were created farthest in the past to *minimize the maximum codeword length* — a good feature for faster decoding in practice.

# Shannon-Fano vs. Huffman

Shannon-Fano (SF) and Huffman (H) codes built for the 26-letter English alphabet. In this case, Shannon-Fano has an average codeword length of 4.16677 bits; Huffman is optimal with 4.15506 bits.

- Both algorithms build prefix-free codes in a greedy manner.
- Shannon-Fano:
  - top-down (divisive)
  - not optimal
- Huffman:
  - bottom-up (aggregative)
  - *optimal*

(a)				(b)			
symbol	probability	SF	H	symbol	probability	SF	H
A	0.08833	010	1111	N	0.06498	1000	1001
B	0.01267	111110	100000	O	0.07245	0110	1011
C	0.02081	111010	00000	P	0.02575	11010	01000
D	0.04376	10111	11101	Q	0.00080	11111111110	000101010
E	0.14878	000	110	R	0.06872	0111	1010
F	0.02455	11011	00011	S	0.05537	10110	0101
G	0.01521	111100	100001	T	0.09351	001	001
H	0.05831	1001	0111	U	0.02762	11001	01001
I	0.05644	1010	0110	V	0.01160	1111110	000100
J	0.00080	1111111110	0001010111	W	0.01868	111011	100011
K	0.00867	11111110	0001011	X	0.00146	111111110	00010100
L	0.04124	11000	11100	Y	0.01521	111101	100010
M	0.02361	11100	00001	Z	0.00053	11111111111	0001010110

Probabilities taken from page 174 of:  
David Salomon. 2007. *Variable-Length Codes for Data Compression*.  
Springer Science & Business Media, ISBN 978-1-84628-959-0.

# Huffman — Optimality

- **Optimality.** The Huffman algorithm produces a minimum-redundancy prefix-free code  $Huf_m = \langle \ell_i \rangle_m$  for the weights  $W_m = \langle w_i \rangle_m$ .
- The proof builds on two ingredients:
  1. **Max-Depth Leaves Property.** If a code  $T$  is optimal, then its binary tree must contain two least-weight leaves that are children of the same parent (siblings), hence at the same maximum depth  $D$  in the tree.
  2. Inductive process on the number  $m$  of weights.
    - 2.1 Base case for  $m = 2$  is always valid since the trivial optimal code is  $\langle \ell_1 = 1, \ell_2 = 1 \rangle$  (and the two codewords will always be 0 and 1).
    - 2.2 Assume  $Huf_{m-1}$  is optimal for  $W_{m-1}$  and prove  $Huf_m$  is optimal for  $W_m$ .



# Huffman — Max-Depth Leaves Property

- **Max-Depth Leaves Property.** If the code  $T$  is optimal, then its binary tree must contain two least-weight leaves that are children of the same parent (siblings), hence at the same maximum depth  $D$  in the tree.

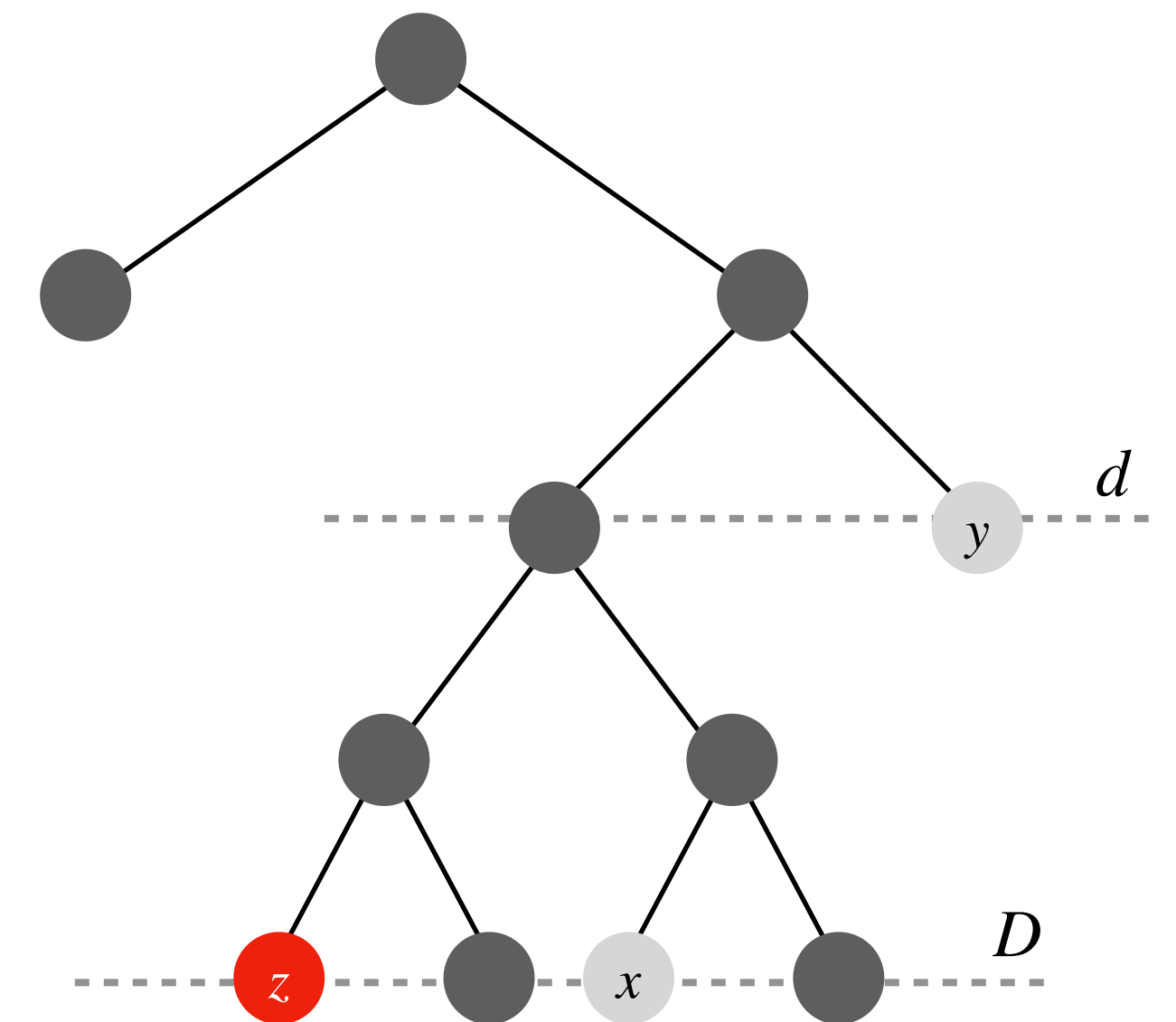
*Proof.*

- Suppose that the two least-weight leaves, say for symbols  $x$  and  $y$ , are *not* at the same maximum depth  $D$ . Let  $x$  be at depth  $D$  and  $y$  at some other depth  $d < D$ . Since each internal node must have two children, there must be another **leaf  $z$**  at depth  $D$ . The weight of  $z$  is then  $w_z \geq w_y$  because  $y$  is a least-weight leaf by assumption.
- If now we exchange the codeword lengths assigned to  $z$  and  $y$ , the resulting code  $T'$  will have a cost  $C(T') \geq C(T)$  because  $T$  is optimal by assumption. Since the two codes only differ in the assignment for  $z$  and  $y$ :

$$C(T') - C(T) = (w_y D + w_z d) - (w_y d + w_z D) = (w_y - w_z) \cdot (D - d) \geq 0$$

and since  $D - d > 0$ , then it must be  $w_y \geq w_z$ . Therefore we conclude that  $w_z = w_y$  and that  $z$  is indeed a least-weight leaf at maximum depth  $D$ .

- Now that we are sure the two least-weight leaves are both at depth  $D$ , it could be that they are not children of the same parent. If that is the case, just “relabel” the leaves by sorting the weights. ■



# Huffman — Induction

- Suppose that  $Huf_{m-1}$  is an optimal code for the set  $W_{m-1}$  of  $m - 1$  weights, for  $m > 2$ . (The case for  $m = 2$  is the base case.) And let  $Opt_m$  be an optimal code for  $W_m$  instead. Therefore, we have

$$C(Opt_m) \leq C(Huf_m). \quad (1)$$

- Since  $Opt_m$  is optimal, for the *Max-Depth Leaves Property* there must be two leaves,  $x$  and  $y$ , at the same depth  $D_m$  of least-weight, so:

$$C(Opt_m) = C(Opt_{m-1}) + (w_x + w_y) \cdot D_m \quad (2)$$

where  $Opt_{m-1}$  is an optimal code on the “reduced” weights  $W_{m-1} = W_m \setminus \{w_x, w_y\} \cup \{w_x + w_y\}$ .

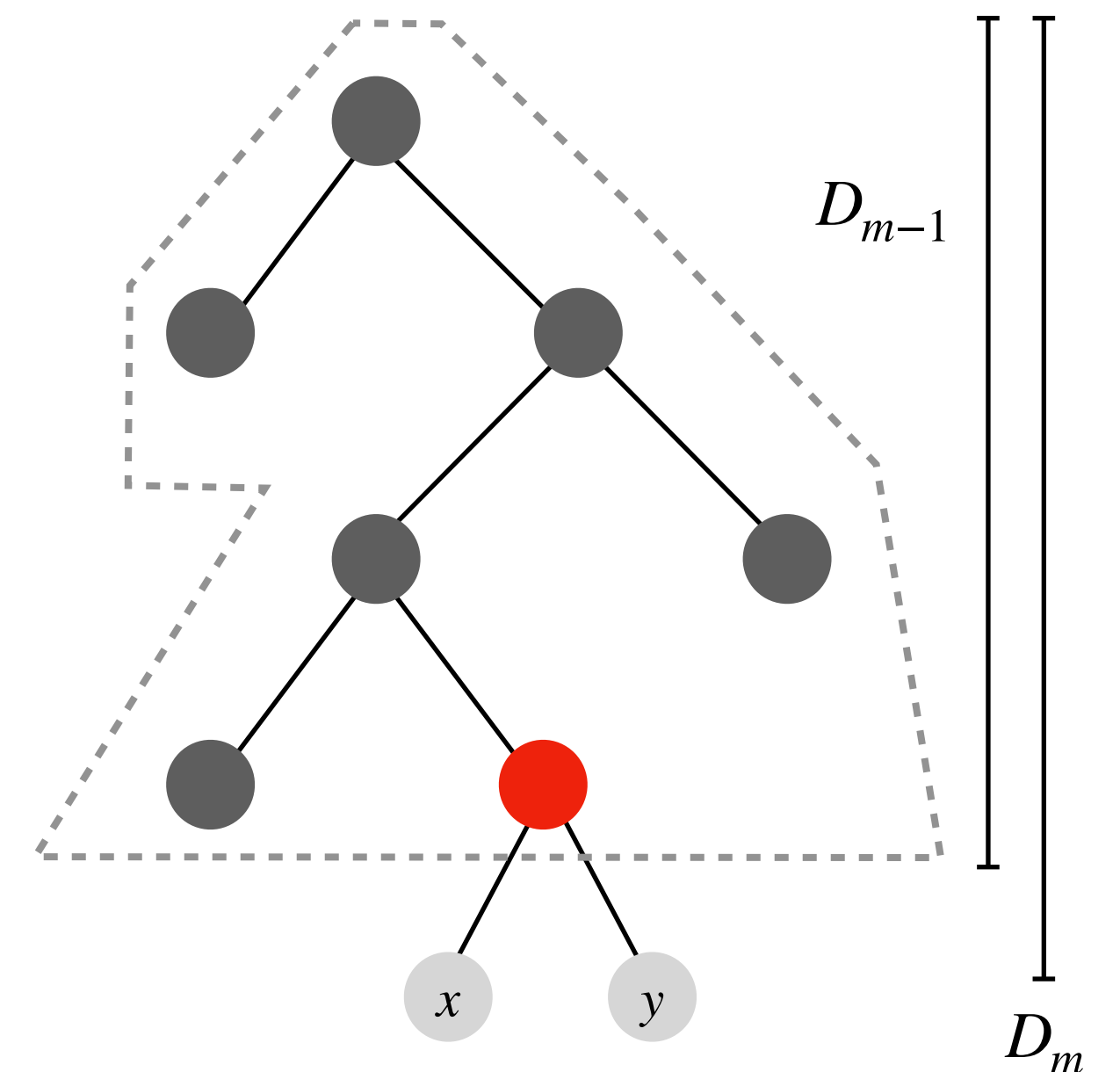
- But for the inductive hypothesis, also  $Huf_{m-1}$  is optimal on  $W_{m-1}$ , thus it must be

$$C(Huf_{m-1}) \leq C(Opt_{m-1}). \quad (3)$$

- Therefore, combining **(2)** with **(3)** we have:

$$C(Opt_m) = C(Opt_{m-1}) + (w_x + w_y) \cdot D_m \geq C(Huf_{m-1}) + (w_x + w_y) \cdot D_m = C(Huf_m).$$

Combining the latter inequality with **(1)**, we obtain that it must be  $C(Opt_m) = C(Huf_m)$ , hence  $Huf_m$  is an optimal code for  $W_m$ . ■



# Huffman — Relation to Entropy

- Recall the entropy for the probability distribution  $P$ :

$H(P) = \sum_x P(x) \log_2(1/P(x))$  is the *expected* codeword length for an optimal code for the distribution  $P$ . In this module, we model  $P(x) = w_x/n$  using the “weights”  $W = \langle w_x \rangle$ , so we write

$$H = H(W) = (1/n) \cdot \sum_x w_x \log_2(n/w_x) \text{ bits.}$$

- The average codeword length of an Huffman code  $Huf$  for  $W$  is  $L_{Huf} = C(W, Huf)/n$ .
- A fundamental result established that  $H \leq L_{Huf} < H + 1$ , meaning that Huffman can loose **up to 1 bit per symbol** compared to the entropy. **Q.** Is this bad or good?



# Huffman — Relation to Entropy

- Recall the entropy for the probability distribution  $P$ :

$H(P) = \sum_x P(x) \log_2(1/P(x))$  is the *expected* codeword length for an optimal code for the distribution  $P$ . In this module, we model  $P(x) = w_x/n$  using the “weights”  $W = \langle w_x \rangle$ , so we write

$$H = H(W) = (1/n) \cdot \sum_x w_x \log_2(n/w_x) \text{ bits.}$$

- The average codeword length of an Huffman code  $Huf$  for  $W$  is  $L_{Huf} = C(W, Huf)/n$ .
- A fundamental result established that  $H \leq L_{Huf} < H + 1$ , meaning that Huffman can loose **up to 1 bit per symbol** compared to the entropy. **Q.** Is this bad or good?
- A.** It depends. If  $H$  is high, say  $H \gg 1$ , then the extra bit is *negligible* and Huffman reaches the entropy limit. Otherwise, if  $H \ll 1$ , the extra bit lost is not negligible and Huffman is *not effective*. Note that this limit is intrinsic of prefix-free codes as they must assign codewords of length  $\geq 1$ .

# Huffman — Limitation

Example for  $W = \langle 999, 1 \rangle$ , where one symbol is very, very, frequent. Then the entropy is

$$H = (999/1000) \cdot \log_2(1000/999) + (1/1000) \cdot \log_2(1000) \simeq 0.01141 \text{ bits},$$

thus, much less than 1 bit because of the skewed distribution.

# Huffman — Limitation

Example for  $W = \langle 999, 1 \rangle$ , where one symbol is very, very, frequent. Then the entropy is

$$H = (999/1000) \cdot \log_2(1000/999) + (1/1000) \cdot \log_2(1000) \simeq 0.01141 \text{ bits},$$

thus, much less than 1 bit because of the skewed distribution.

- **Q.** How to overcome this limitation?  
**A.** Associate a codeword to *more than one symbol*.

# Huffman on Blocks

- **Idea.** Compute the Huffman code on *blocks* of  $L$ .  
If  $B$  is the block size, then the extra bit lost compared to the entropy is amortized among the  $B$  symbols, thus the loss is just  $1/B$  bits.
- The redundancy becomes smaller for larger  $B$ .
- But  $B$  cannot grow as much as we want!  
For large  $B$  we would incur in high space for the encoding of the Huffman tree and high decoding time.

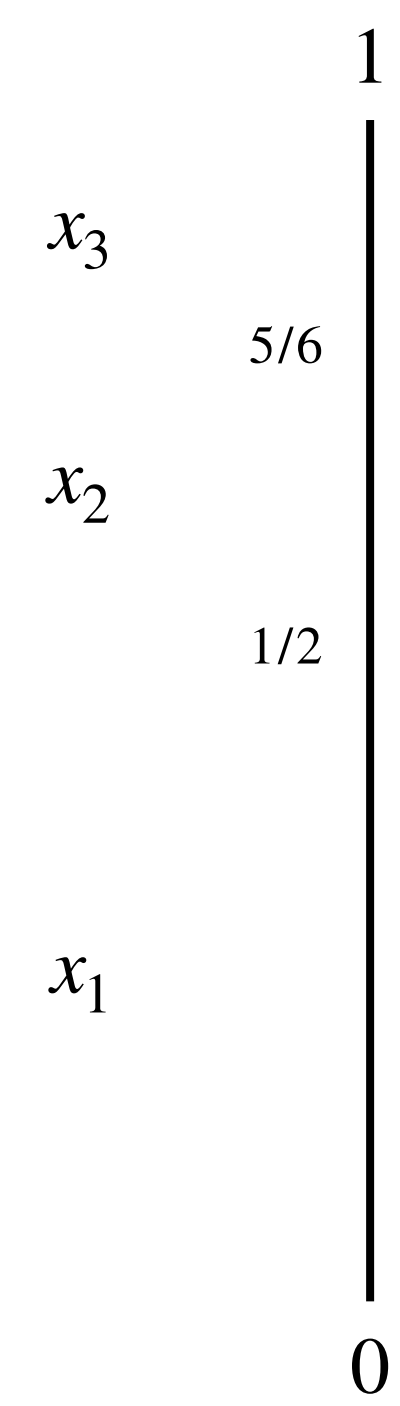
# Arithmetic Coding

Elias, ~1960

- This method offers higher compression ratios than Huffman because it is *not* a prefix-free code.
- This means that we cannot decode *individual* symbols, rather the whole message.
- In practice:
  - slower than (canonical) Huffman to decode;
  - no need to transmit the model (e.g., the Huffman tree).
- **Algorithm.**
  - The real interval  $[0.0, 1.0)$  is partitioned into segments of length *proportional* to the probabilities  $P(x) = w_x/n$  in  $L$  and the segment  $[l_1, r_1) \subset [0.0, 1.0]$  associated to the symbol  $L[1]$  is considered.
  - The same partitioning step is applied to  $[l_1, r_1)$  and the segment  $[l_2, r_2) \subset [l_1, r_1)$  associated to the symbol  $L[2]$  is considered.
  - Repeat for all the  $n$  symbols and emit a single real number  $y \in [l_n, r_n)$ .
- The pair  $\langle y, n \rangle$  is sufficient to decode the original list  $L$ .

# Arithmetic Coding — Encoding Demo

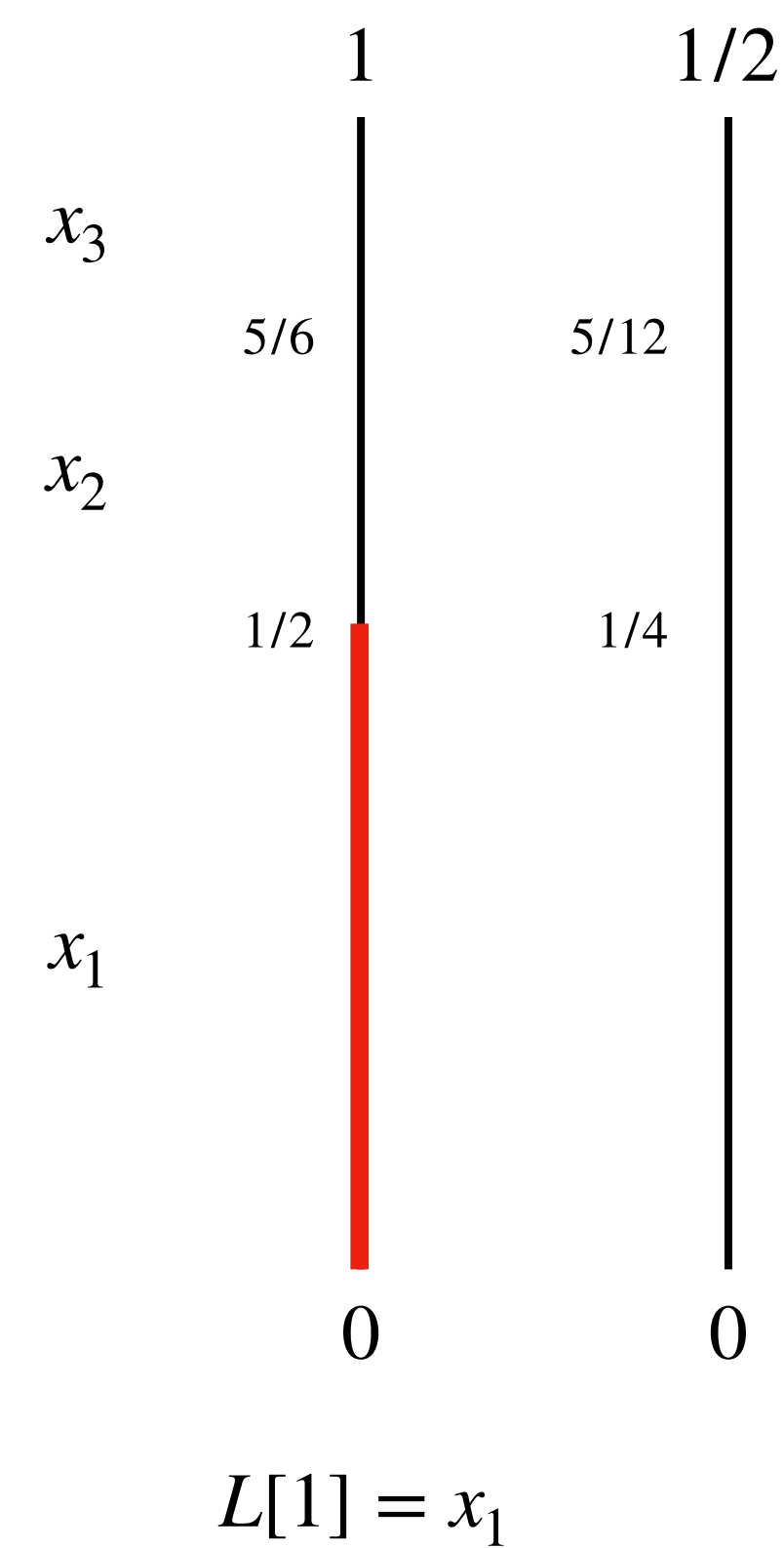
Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1		
2		
3		
4		
5		
6		

# Arithmetic Coding — Encoding Demo

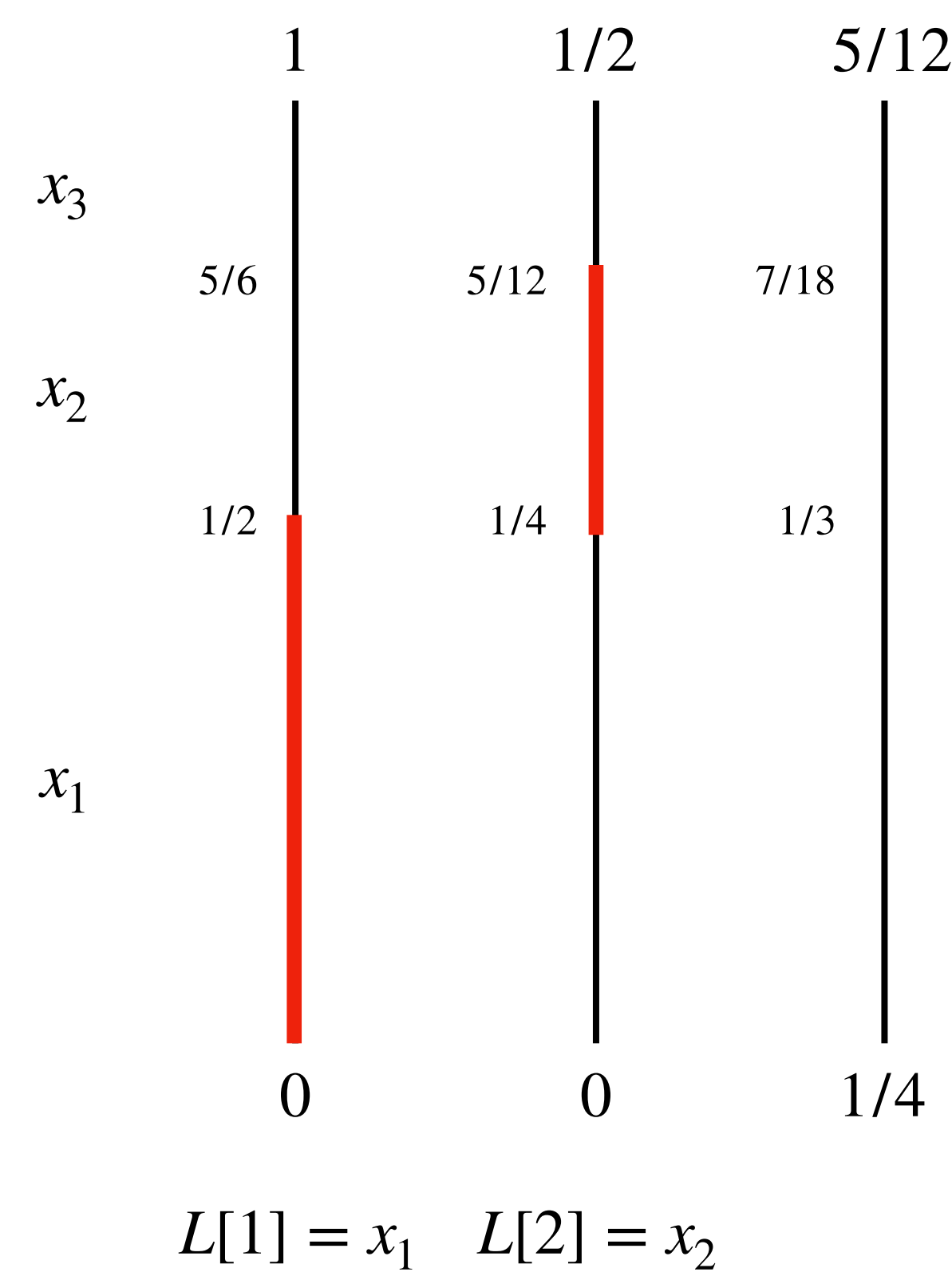
Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2		
3		
4		
5		
6		

# Arithmetic Coding — Encoding Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .

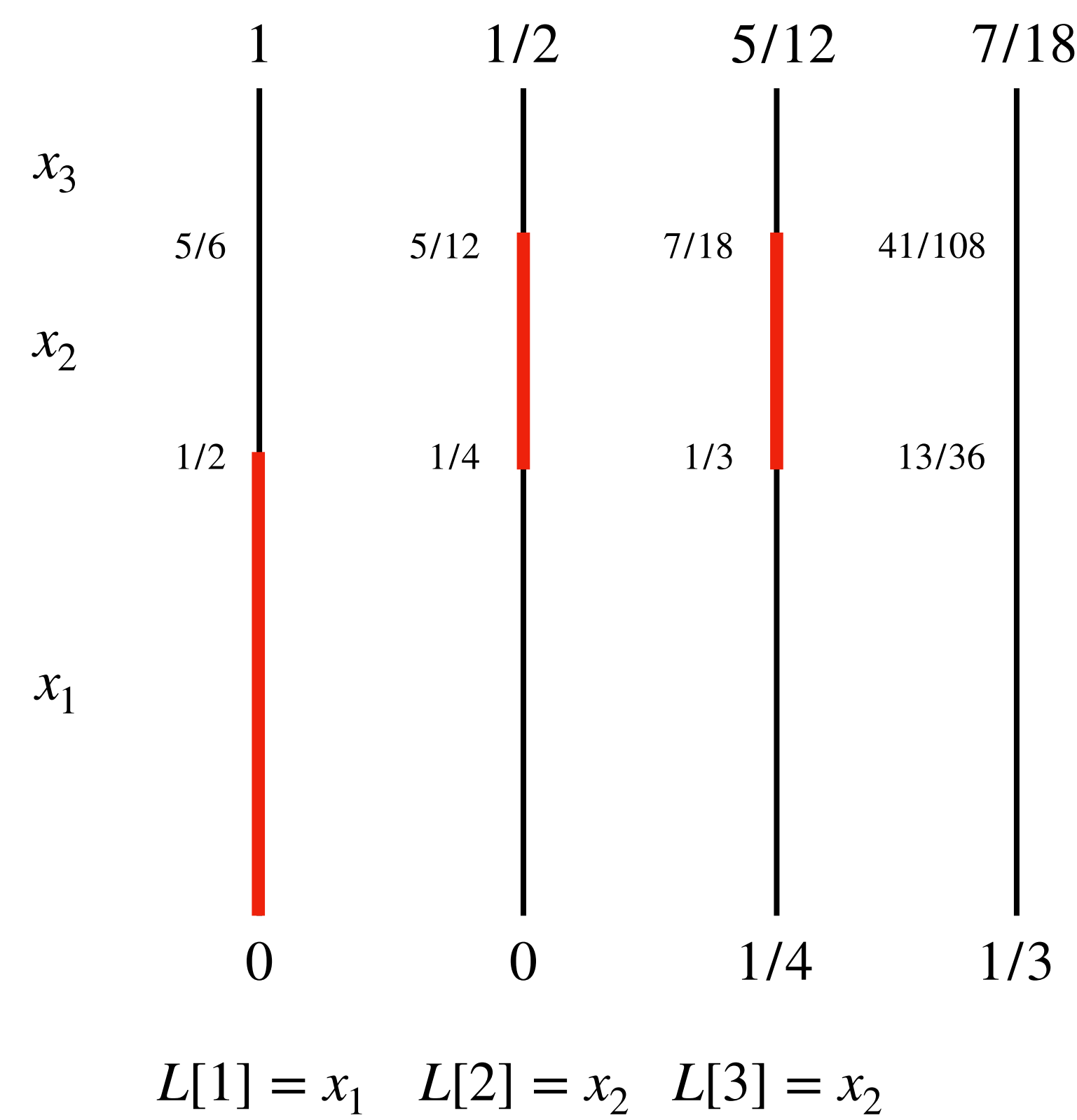


$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3		
4		
5		
6		



# Arithmetic Coding — Encoding Demo

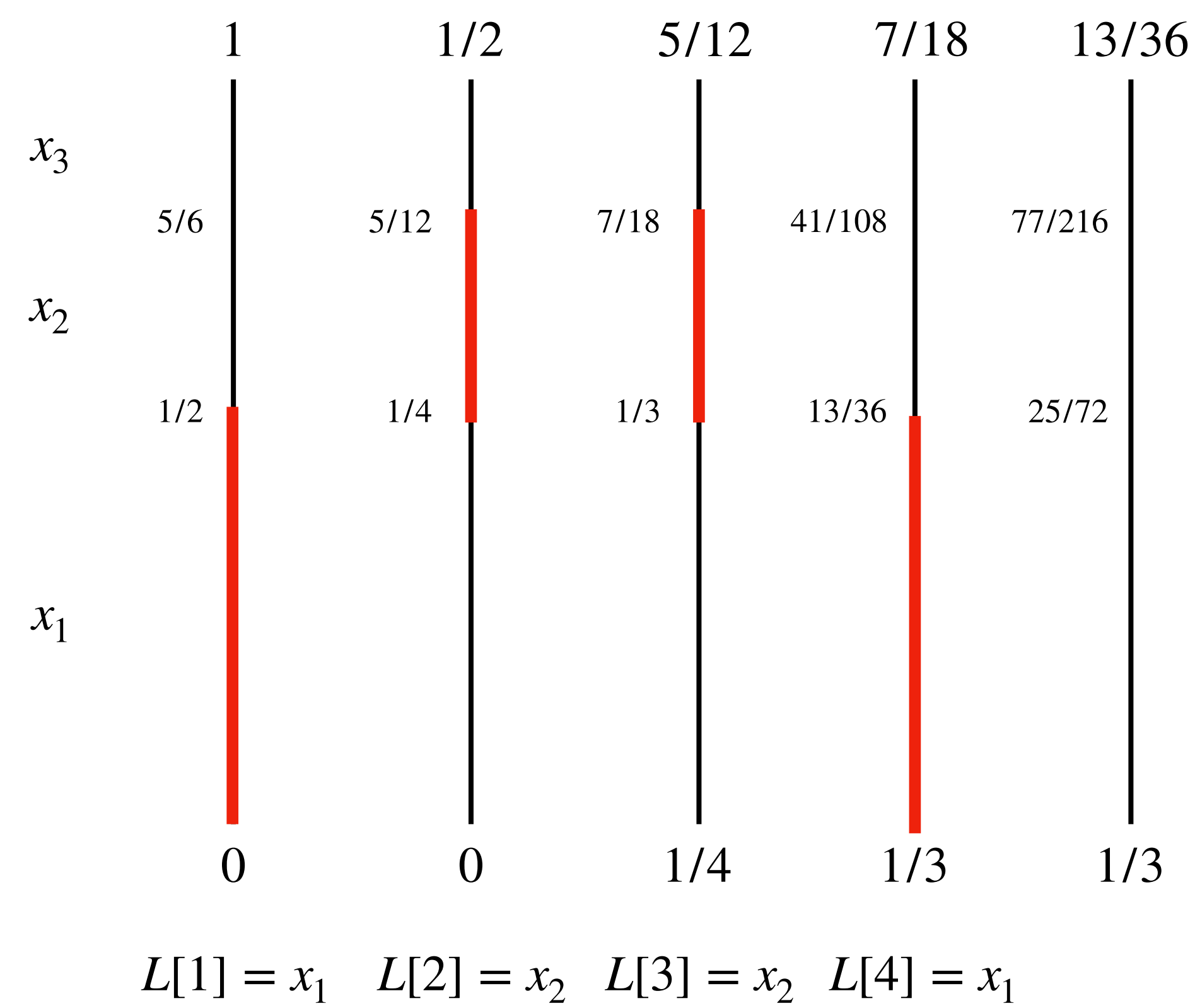
Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4		
5		
6		

# Arithmetic Coding — Encoding Demo

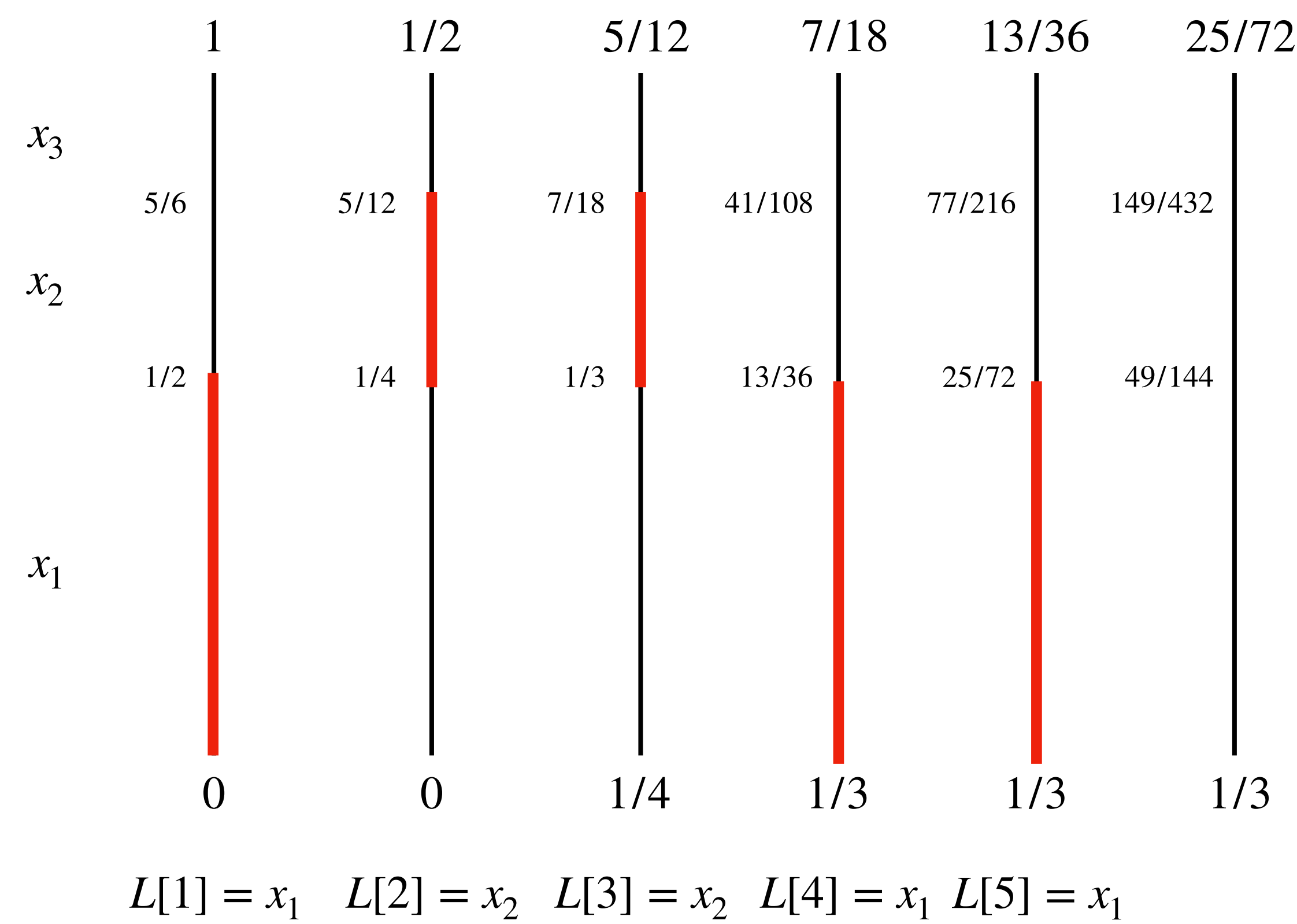
Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5		
6		

# Arithmetic Coding — Encoding Demo

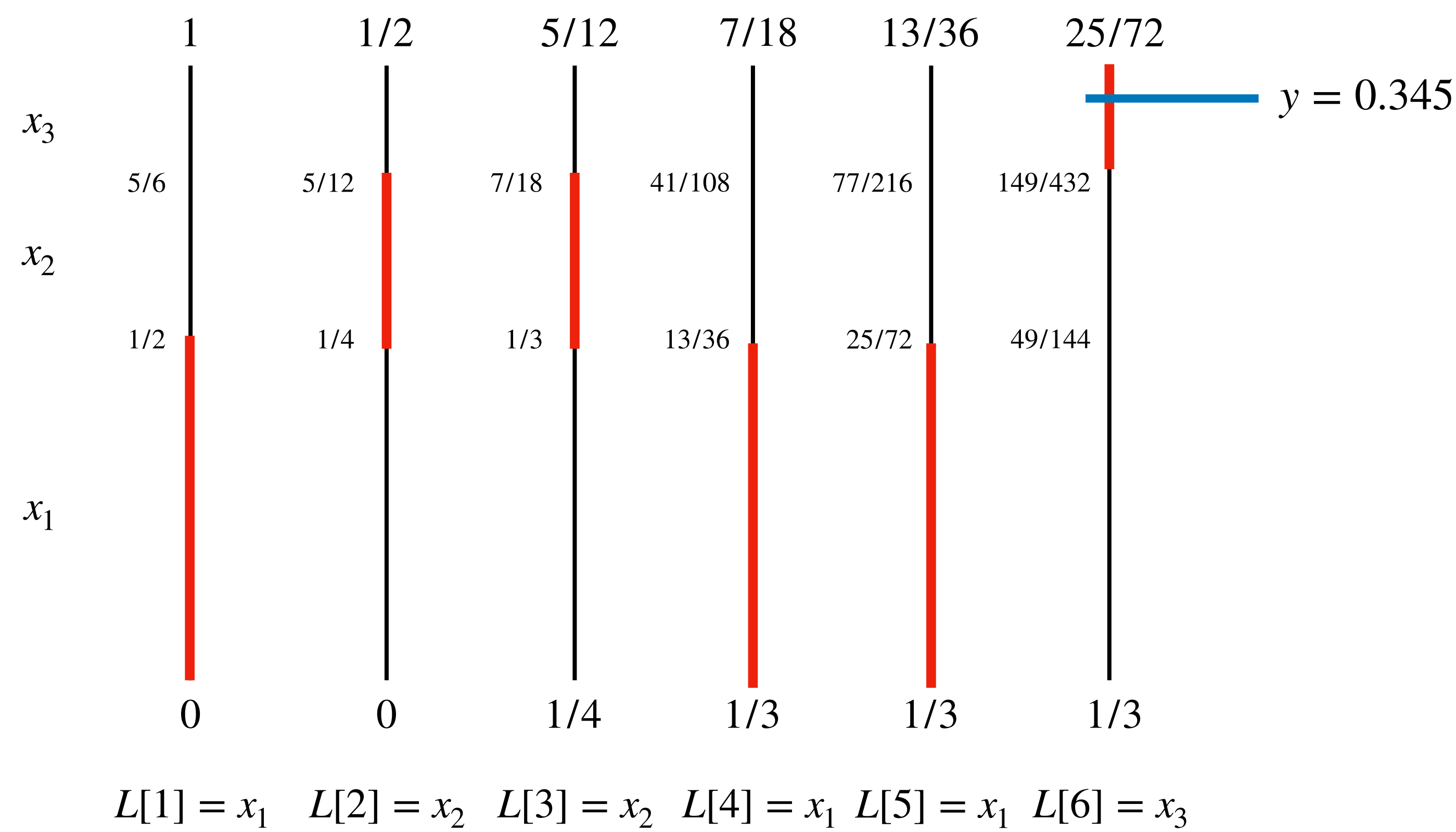
Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5	0.333333	0.347222
6		

# Arithmetic Coding — Encoding Demo

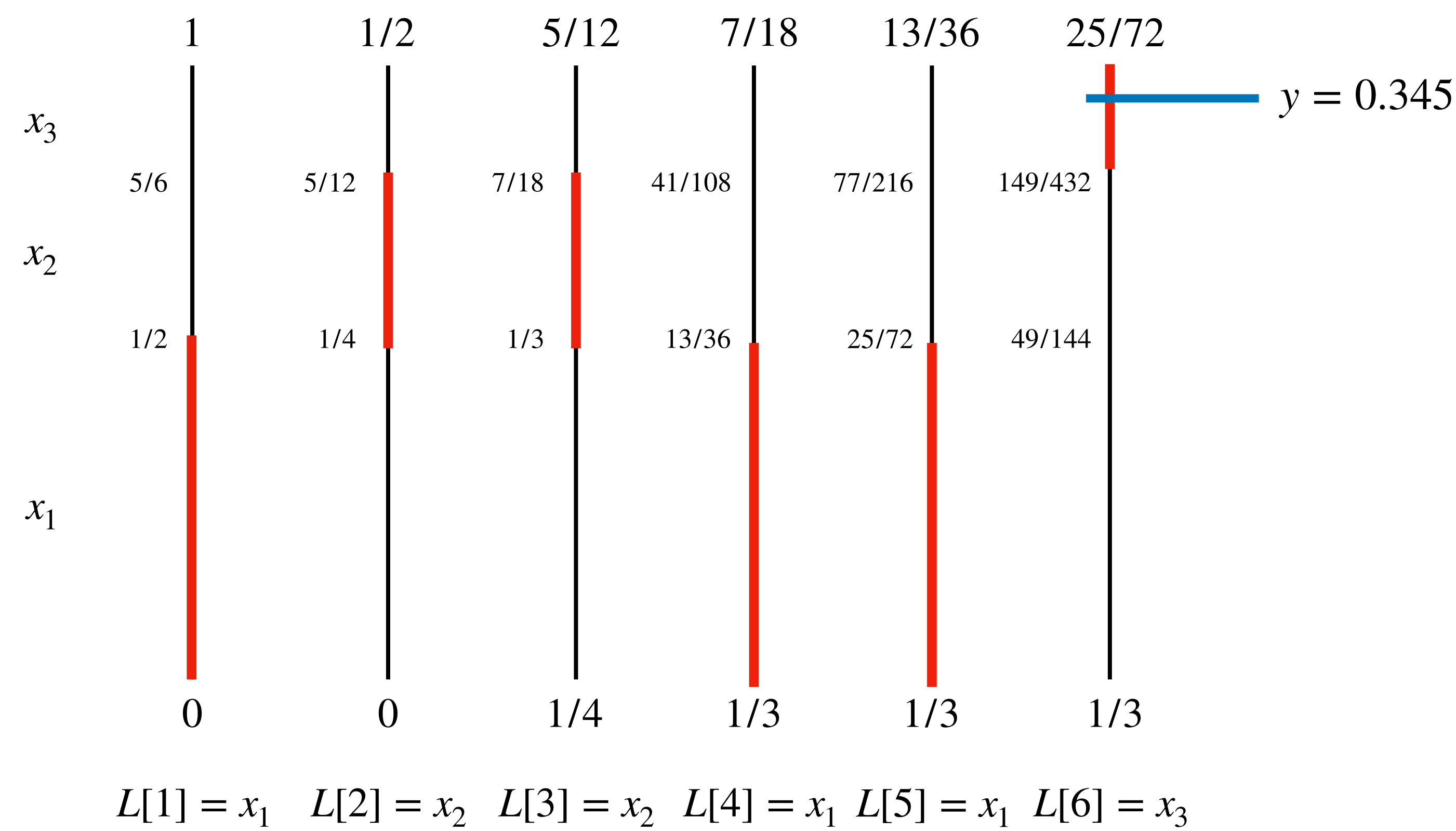
Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5	0.333333	0.347222
6	0.344907	0.347222

# Arithmetic Coding — Encoding Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $P(x_1) = 1/2$ ,  $P(x_2) = 1/3$ , and  $P(x_3) = 1/6$ .

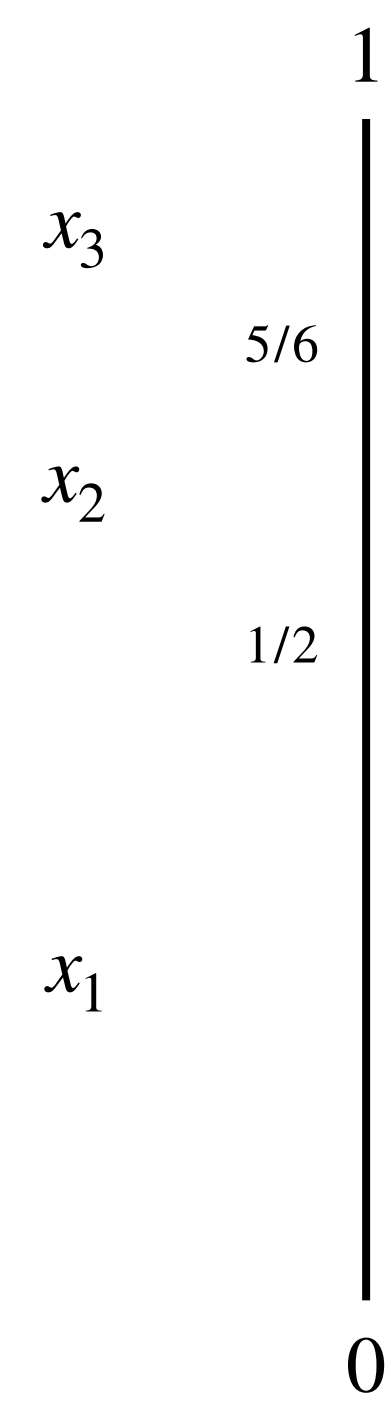


<i>i</i>	<i>l<sub>i</sub></i>	<i>r<sub>i</sub></i>
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5	0.333333	0.347222
6	0.344907	0.347222

Note that, by construction, the length  $(r_n - l_n)$  of the final interval  $[l_n, r_n)$  is  $\prod_{i=1}^n P(L[i])$ .

# Arithmetic Coding — Decoding Demo

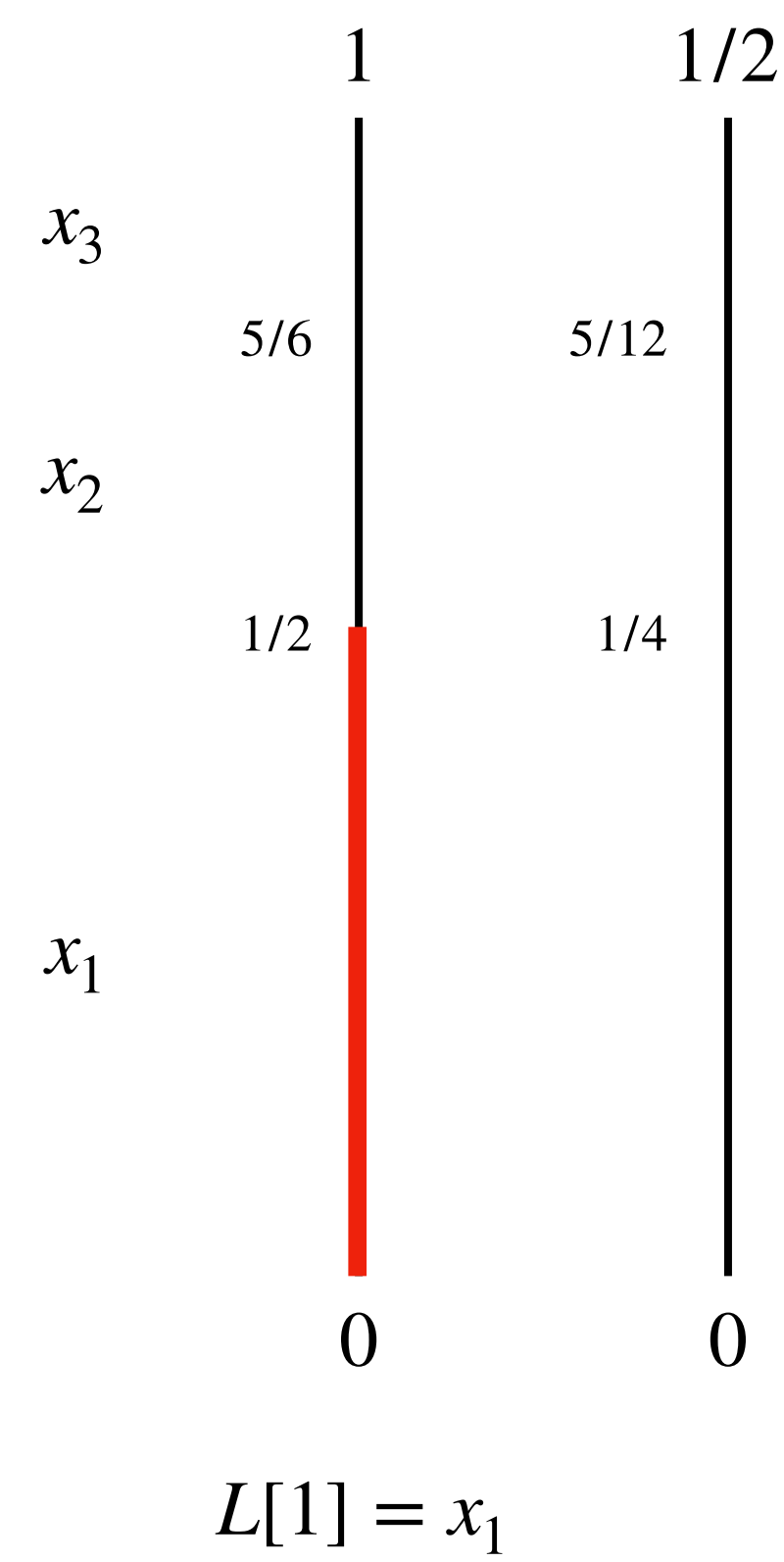
**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1		
2		
3		
4		
5		
6		

# Arithmetic Coding — Decoding Demo

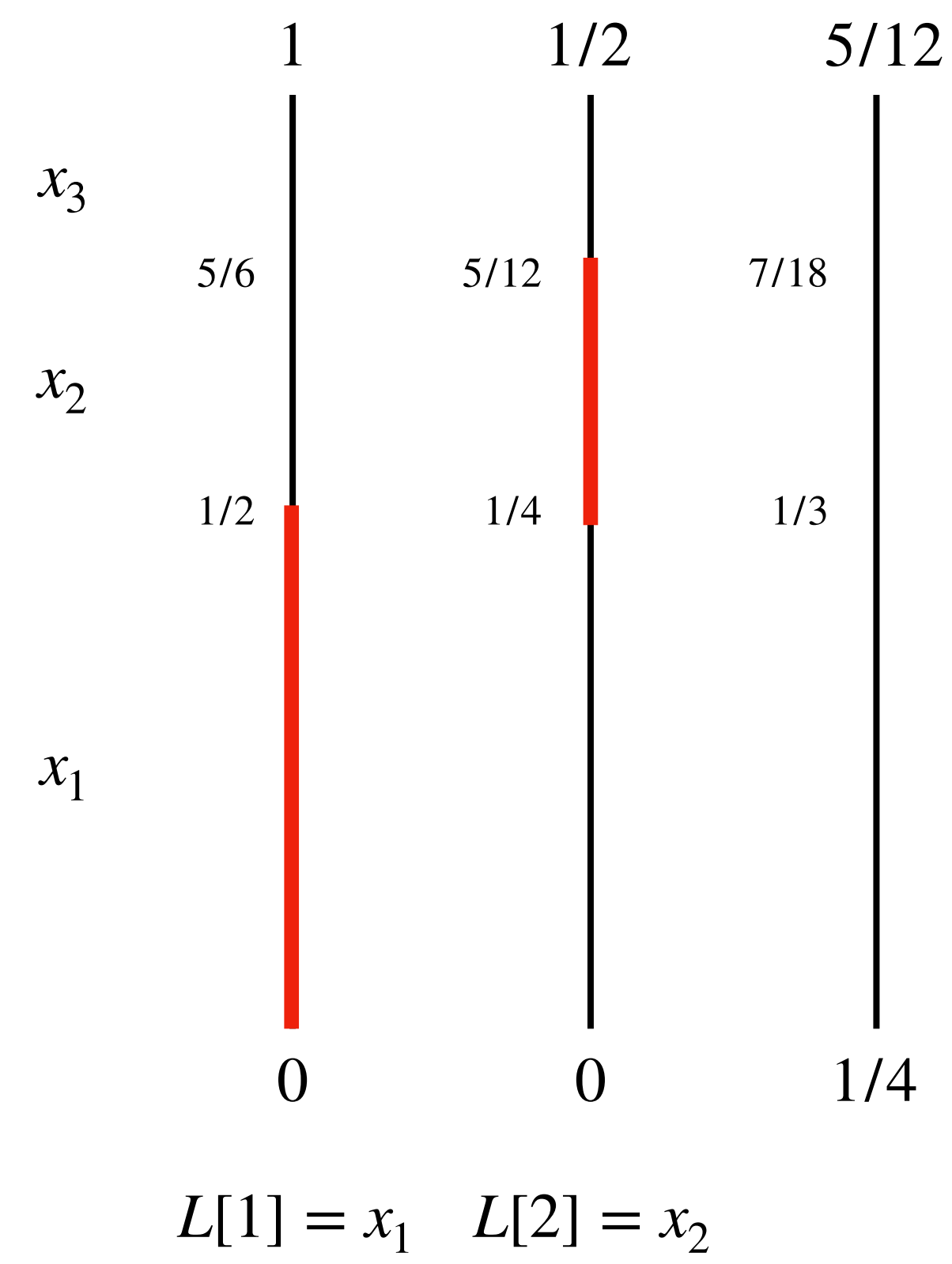
**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2		
3		
4		
5		
6		

# Arithmetic Coding — Decoding Demo

**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .

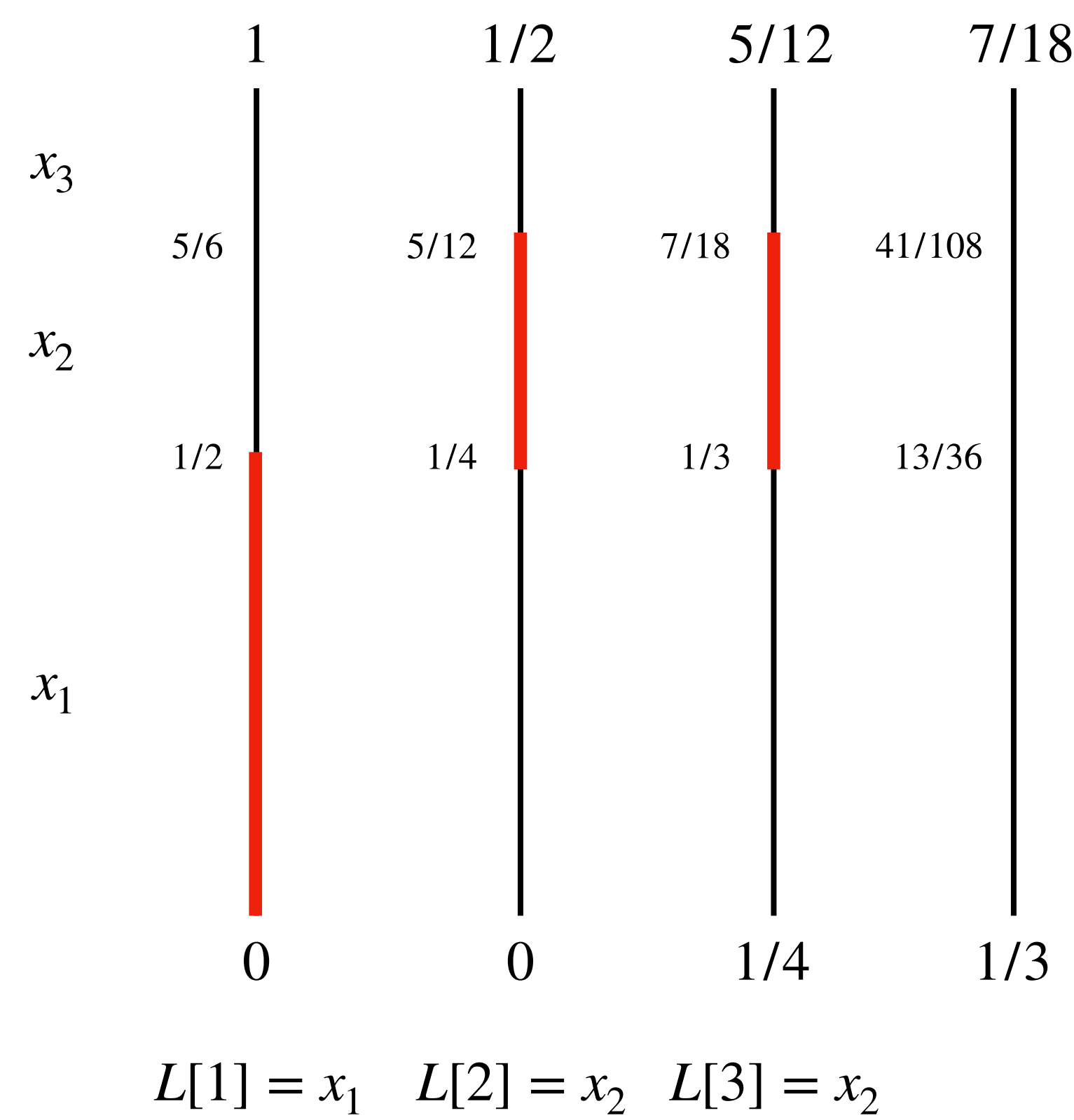


$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3		
4		
5		
6		



# Arithmetic Coding — Decoding Demo

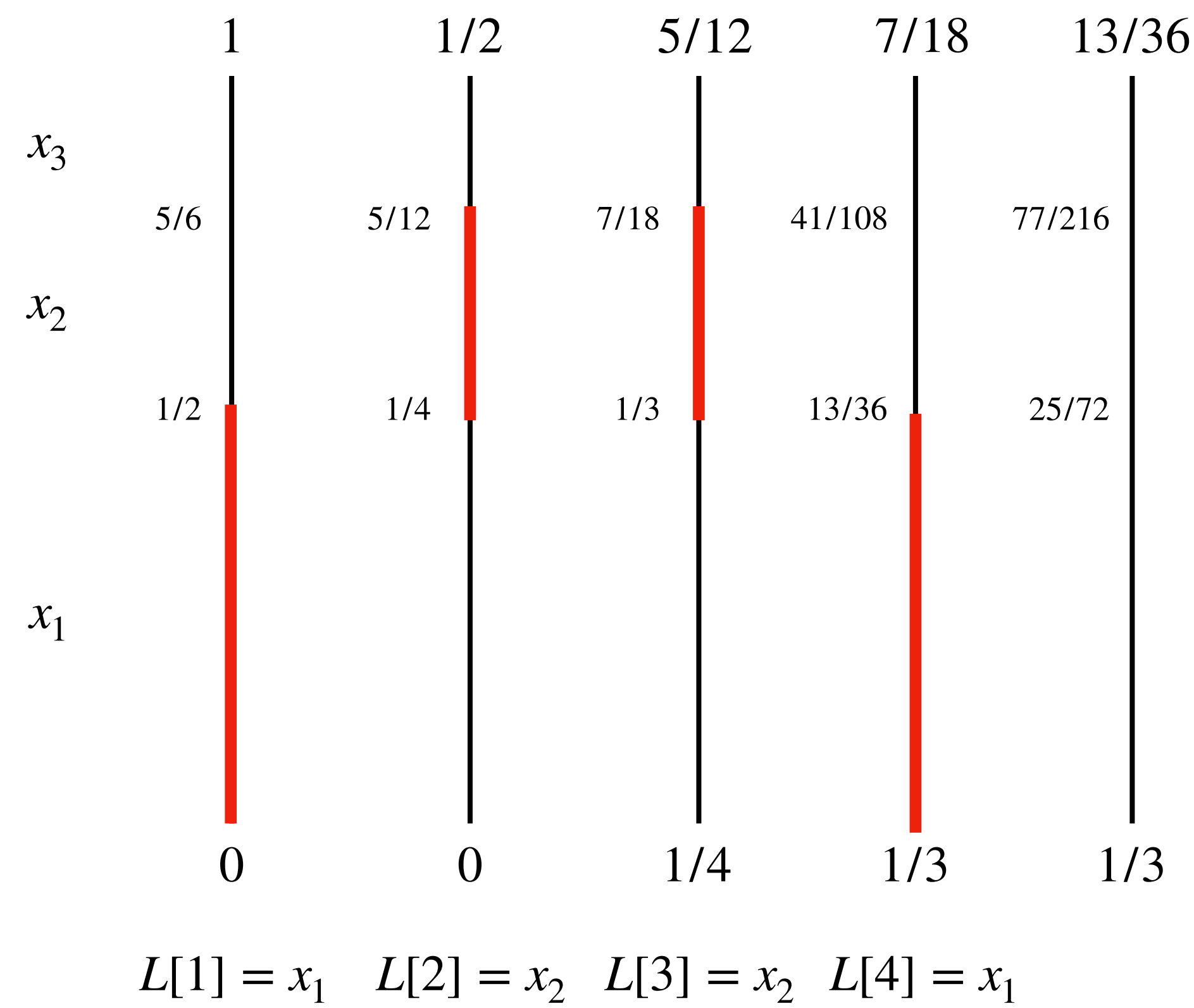
**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4		
5		
6		

# Arithmetic Coding — Decoding Demo

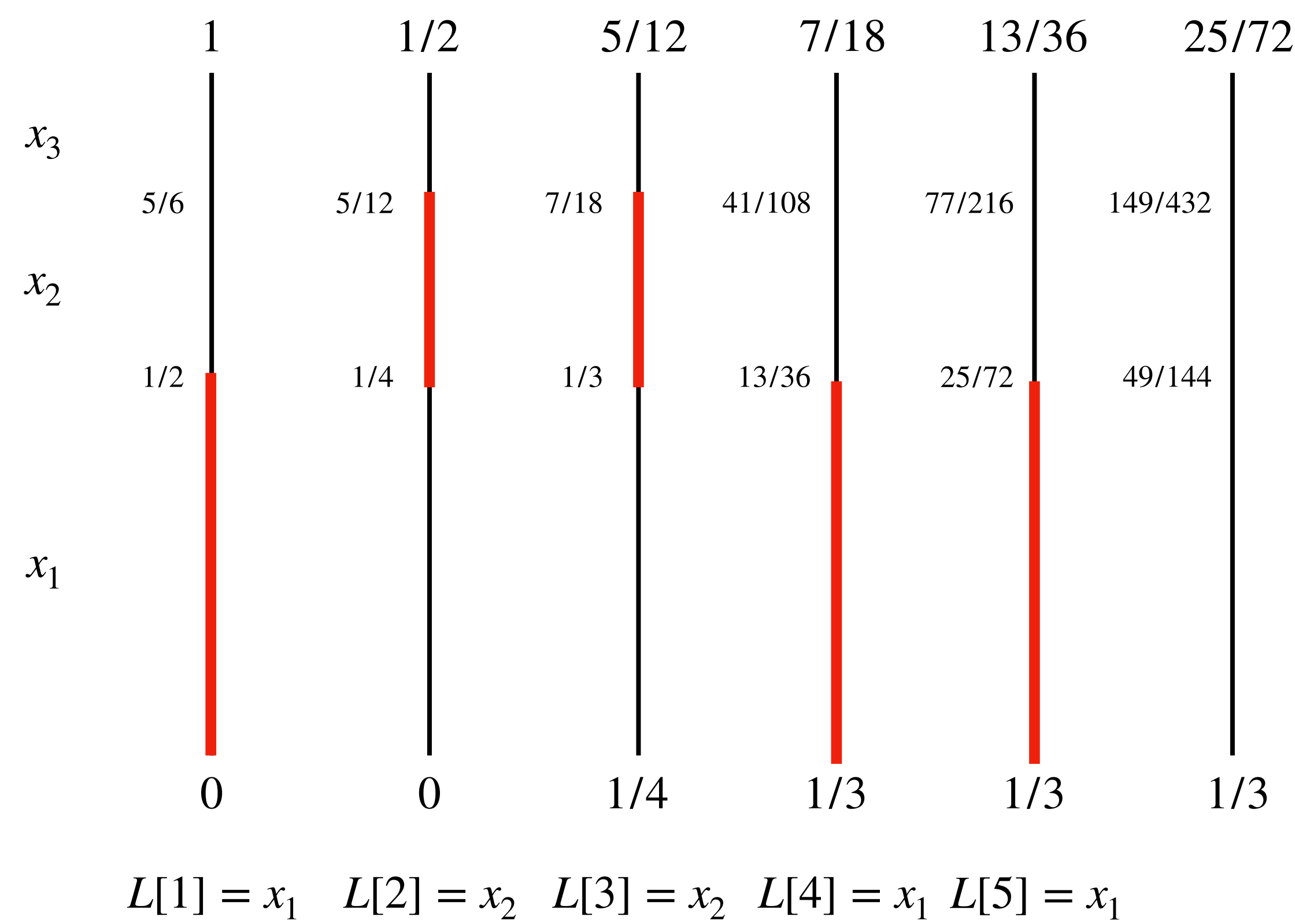
**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5		
6		

# Arithmetic Coding — Decoding Demo

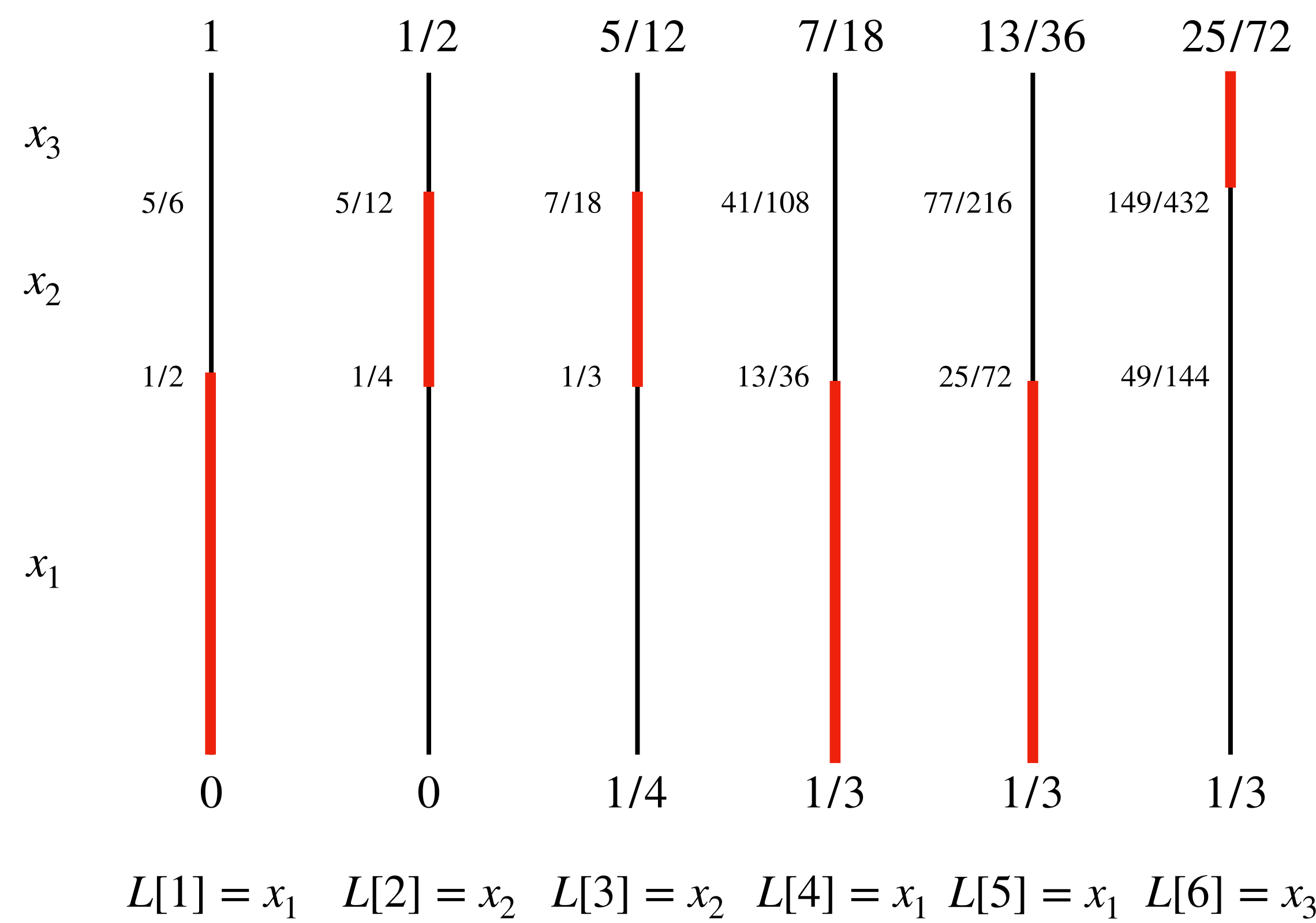
**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5	0.333333	0.347222
6		

# Arithmetic Coding — Decoding Demo

**Decoding.** Given  $\langle y, n \rangle$ , where does  $y$  lies in at each of the  $n$  steps?  
In our example:  $y = 0.345$ , and  $n = 6$ .



$i$	$l_i$	$r_i$
0	0.000000	1.000000
1	0.000000	0.500000
2	0.250000	0.416667
3	0.333333	0.388889
4	0.333333	0.361111
5	0.333333	0.347222
6	0.344907	0.347222

# Arithmetic Coding

- **Q.** How to choose  $y \in [l_n, r_n)$  and how to represent it?
- **Dyadic fraction.** In general, a binary string  $B[1..k]$  can be used to represent the real value

$$y = \frac{[B[1] \dots B[k]]_{10}}{2^k} = \frac{\sum_{i=1}^k (B[i] \cdot 2^{k-i})}{2^k} = \sum_{i=1}^k (B[i] \cdot 2^{-i}).$$

- The larger  $k$ , the better the approximation.

For example,  $B[1..6] = [001011]$  corresponds to  $y = [001011]_{10}/2^6 = 11/64 = 0.171875$  and  $B[1..7] = [0100111]$  to  $y = [0100111]_{10}/2^7 = 39/128 = 0.3046875$ .

# Arithmetic Coding

- **Finite-precision arithmetic.** Given the binary string  $B[1..\infty]$  representing the infinite-precision real number  $y$ ,  $B[1..k]$  represents a “truncated” value  $\hat{y} \in [y - 2^{-k}, y]$ .

*Proof.* Clearly we have  $\hat{y} < y$  because  $\hat{y}$  is a truncation.

Geometric series:  $\sum_{i=0}^{\infty} \frac{1}{2^i} = 1/(1 - 1/2) = 2 = \frac{1}{2^0} + \sum_{i=1}^{\infty} \frac{1}{2^i}$

Therefore:  $y - \hat{y} = \sum_{i=1}^{\infty} B[k+i]2^{-(k+i)} \leq 2^{-k} \sum_{i=1}^{\infty} \frac{1}{2^i} = 2^{-k}$ , that is  $\hat{y} \geq y - 2^{-k}$ . ■

- **Arithmetic Coding output bits.** Given  $[l_n, r_n)$ , the real value  $y = (l_n + r_n)/2$  truncated to its first  $\left\lceil \log_2 \left( \frac{2}{r_n - l_n} \right) \right\rceil$  bits,  $\hat{y}$ , belongs to the interval  $[l_n, r_n)$ .

*Proof.* We know that  $y - 2^{-k} \leq \hat{y} < y$ . Setting  $k = \lceil \log_2(2/(r_n - l_n)) \rceil$  and  $y = (l_n + r_n)/2$ , we have that:

$$y - 2^{-k} = (l_n + r_n)/2 - 2^{-\lceil \log_2(2/(r_n - l_n)) \rceil} \geq (l_n + r_n)/2 - (r_n - l_n)/2 = l_n \text{ and therefore}$$

$$l_n \leq y - 2^{-k} \leq \hat{y} < y = (l_n + r_n)/2 < r_n. \quad \blacksquare$$

# Arithmetic Coding — Optimality

- We have proved that:

Arithmetic Coding emits  $\lceil \log_2(2/(r_n - l_n)) \rceil$  bits, where  $r_n - l_n = \prod_{i=1}^n P(L[i])$ .

- **Optimality.** The number of bits emitted by Arithmetic Coding for encoding  $L[1..n]$  is at most  $nH + 2$ , where  $H$  is the entropy of  $L$ .

*Proof.*

$$\begin{aligned} \lceil \log_2(2/(r_n - l_n)) \rceil &< \log_2 2 - \log_2(r_n - l_n) + 1 = 2 - \log_2\left(\prod_{i=1}^n P(L[i])\right) = \\ &= 2 - \sum_{i=1}^n \log_2 P(L[i]) = \\ &= 2 + \sum_{i=1}^n \log_2(1/P(L[i])) = 2 + nH. \blacksquare \end{aligned}$$

The entropy is independent from the **position** of the symbol occurrences, so we can sum over groups of occurrences of the same symbol.

- This means that Arithmetic Coding loses  $2/n$  bits per symbol compared to the entropy, which is *negligible* for all practical values of  $n$ . Much *better* than Huffman on skewed distributions.

# Asymmetric Numeral Systems

Duda, 2009

- Asymmetric Numeral Systems (ANS) approaches the compression ratio of Arithmetic Coding and the speed of Huffman coding. Like Arithmetic Coding, it is *not* a prefix-free code.
- **Idea.** Represent  $L[1..i]$  with an *integer state* variable  $s_i$ ,  $i = 1, \dots, n$ .
- A coding table  $T$  is used to generate the next state  $s_i$  from state  $s_{i-1}$ , after the processing of symbol  $L[i]$ :  $s_i = T[L[i], s_{i-1}]$ .
- The row  $T[i]$  contains integers values assigned in increasing order and in “strides” of length  $w_i$ .
- **Algorithm.**
  - At the beginning, set  $s_0 = 0$ .
  - For each symbol  $L[i]$ ,  $i = 1, \dots, n$ , generate a new state  $s_i = T[L[i], s_{i-1}]$ .
  - Emit  $s_n > 0$  encoded with  $\lceil \log_2 s_n \rceil$  bits.



# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
	0	1	2	3	4	5	6	7	8	9
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60

...

$s_0 = 0$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

...

$s_0 = 0$   
 $s_1 = T[L[1],0] = 1$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

...

$s_0 = 0$

$s_1 = T[L[1],0] = 1$

$s_2 = T[L[2],1] = 5$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

...

$s_0 = 0$   
 $s_1 = T[L[1], 0] = 1$   
 $s_2 = T[L[2], 1] = 5$   
 $s_3 = T[L[3], 5] = 17$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

...

$s_0 = 0$   
 $s_1 = T[L[1], 0] = 1$   
 $s_2 = T[L[2], 1] = 5$   
 $s_3 = T[L[3], 5] = 17$   
 $s_4 = T[L[4], 17] = 33$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

...

$s_0 = 0$   
 $s_1 = T[L[1], 0] = 1$   
 $s_2 = T[L[2], 1] = 5$   
 $s_3 = T[L[3], 5] = 17$   
 $s_4 = T[L[4], 17] = 33$   
 $s_5 = T[L[5], 33] = 67$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

...

$s_0 = 0$   
 $s_1 = T[L[1], 0] = 1$   
 $s_2 = T[L[2], 1] = 5$   
 $s_3 = T[L[3], 5] = 17$   
 $s_4 = T[L[4], 17] = 33$   
 $s_5 = T[L[5], 33] = 67$   
 $s_6 = T[L[6], 67] = 408$

# Asymmetric Numeral Systems — Demo

Example for  $L = [x_1, x_2, x_2, x_1, x_1, x_3]$  with  $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 2, 1 \rangle$  and  $n = 6$ .

$W$	$T$									
$w_1$	1	2	3	7	8	9	13	14	15	19
$w_2$	4	5	10	11	16	17	22	23	28	29
$w_3$	6	12	18	24	30	36	42	48	54	60
	0	1	2	3	4	5	6	7	8	9

$$s_0 = 0$$

$$s_1 = T[L[1], 0] = 1$$

$$s_2 = T[L[2], 1] = 5$$

$$s_3 = T[L[3], 5] = 17$$

$$s_4 = T[L[4], 17] = 33$$

$$s_5 = T[L[5], 33] = 67$$

$$s_6 = T[L[6], 67] = 408$$

Actually, no table  $T$  is needed:

$$s_i = T[L[i], s_{i-1}] = r_i + n \lfloor s_{i-1} / w_i \rfloor + s_{i-1} \bmod w_i$$

where  $(x \bmod y) = x - \lfloor x/y \rfloor y \leq y - 1$  is the rest of the integer division (the “modulo”) between  $x$  and  $y$ ,  $r_i = 1 + \sum_{j=1}^i w_j$  for  $i > 1$  and  $r_1 = 1$ . We just need the array  $R[1..m] = [r_1, \dots, r_m]$ .



# Asymmetric Numeral Systems

- Actually, no table  $T$  is needed:

$$s_i = T[L[i], s_{i-1}] = r_i + n \lfloor s_{i-1} / w_i \rfloor + s_{i-1} \bmod w_i, \text{ where } r_i = 1 + \sum_{j=1}^i w_j \text{ for } i > 1 \text{ and } r_1 = 1.$$

- Note that  $s_i = T[L[i], s_{i-1}] = r_i + n \lfloor s_{i-1} / w_i \rfloor + s_{i-1} \bmod w_i \leq r_i + s_{i-1} / P(L[i]) + w_i - 1$  and since  $r_i + w_i - 1 \leq n$ , we have  $s_i \leq n + s_{i-1} / P(L[i])$ .

- **Lemma.**

$$\text{The } i\text{-th state } s_i \text{ is such that } s_i < i \cdot \frac{n}{\prod_{j=1}^i P(L[j])}.$$

*Proof.* Proceed by induction. For the base case  $i = 1$  we have  $s_0 = 0$  and  $s_1 \leq n + 0 / P(L[1]) = n < n / P(L[1])$ .

Now assume  $s_{i-1} < (i-1) \cdot \frac{n}{\prod_{j=1}^{i-1} P(L[j])}$  is true.

$$\text{Then, } s_i \leq n + s_{i-1} / P(L[i]) < n + (i-1) \cdot \frac{n}{P(L[i]) \cdot \prod_{j=1}^{i-1} P(L[j])} < i \cdot \frac{n}{\prod_{j=1}^i P(L[j])}, \text{ since } n < \frac{n}{\prod_{j=1}^i P(L[j])}. \blacksquare$$

# Asymmetric Numeral Systems — Optimality

- **Optimality.** The number of bits emitted by ANS for encoding  $L[1..n]$  is at most  $nH + 2 \log_2 n + 1$ , where  $H$  is the entropy of  $L$ .

*Proof.* For the previous lemma, we have that  $s_n < \frac{n^2}{\prod_{i=1}^n P(L[i])}$ .

Therefore

$$\begin{aligned} \lceil \log_2 s_n \rceil &< \log_2 s_n + 1 < \log_2 \frac{n^2}{\prod_{i=1}^n P(L[i])} + 1 = 2 \log_2 n - \log_2 \left( \prod_{i=1}^n P(L[i]) \right) + 1 = \\ &= 2 \log_2 n + \sum_{i=1}^n \log_2 (1/P(L[i])) + 1 = 2 \log_2 n + nH + 1. \blacksquare \end{aligned}$$

- Worse than Arithmetic Coding ( $2/n$  bits/symb vs.  $(2 \log_2 n)/n$  bits/symb), but the loss vanishes for growing values of  $n$ . Still much better than Huffman.

# Further Readings

- Sections 5.1, 5.2, 5.3 of:  
Alistair Moffat and Andrew Turpin. 2002. *Compression and coding algorithms*. Springer Science & Business Media, ISBN 978-1-4615-0935-6.
- Sections 2.1, 2.2, 2.5, 2.6, 5.1, 5.2, 6 of:  
Alistair Moffat. 2019. *Huffman Coding*. ACM Computing Surveys. 52, 4, Article 85 (July 2020), 35 pages. <https://doi.org/10.1145/3342555>
- Chapter 5.5 (pages 826-838) of:  
Robert Sedgewick and Kevin Wayne. 2011. *Algorithms*. 4-th Edition. Addison-Wesley Professional, ISBN 0-321-57351-X.
- Section 2.6 (about Huffman coding) of:  
Gonzalo Navarro. 2016. *Compact Data Structures*. Cambridge University Press, ISBN 978-1-107-15238-0.
- Section 3.8 of:  
G. E. P. and Rossano Venturini. 2020. *Techniques for Inverted Index Compression*. ACM Computing Surveys. 53, 6, Article 125 (November 2021), 36 pages. <https://doi.org/10.1145/3415148>