A Crash Course on Data Compression

3. List Compressors

Giulio Ermanno Pibiri

ISTI-CNR, giulio.ermanno.pibiri@isti.cnr.it

@giulio_pibiri

@jermp

Overview

- Blocking
- PForDelta
- Simple
- Elias-Fano
- Interpolative
- Directly-Addressable
- Hybrid approaches

The *Sorted* List Coding Problem

- **Problem.** We are given a *sorted* list $L[1..n]$ of n integers from a universe of size $U > L[n]$, and we are asked to compress it in *as few as possible bits*.
- In the following examples, we will also assume that the integers are *distinct*, thus L is *strictly* increasing. (Not a hard requirement, though.)
- **Queries.** Apart from decoding $L[1], L[2], \ldots, L[n]$ (*sequential* decoding), we are usually interested in supporting two types of queries, directly on the compressed representation of L:
  - Succ(x), the "successor" of x as the smallest integer y that is $y \geq x$.
  - Access(i), returns the i-th element of L for any random $i \in [1,n]$.

Motivations

- Search Engines
- Social Networks
- Databases
- DNA Indexes
- A fundamental building block for many other data structures!
- (…)

Combinatorial Lower Bound

- **Q.** How many bits do I need to represent the list L?

Combinatorial Lower Bound

- **Q.** How many bits do I need to represent the list L?

  A. There are $\binom{U}{n}$ ways of choosing n distinct integers from a universe of size $U \geq n$,

  thus we need at least $\left\lceil \log_2 \binom{U}{n} \right\rceil$ bits, that is (by Stirling's approximation):

  $$\left\lceil \log_2 \binom{U}{n} \right\rceil \approx n\Big(\log_2\Big(\frac{U}{n}\Big) + \log_2 e\Big) \approx n \log_2\Big(\frac{U}{n}\Big) + 1.44n \text{ bits},$$

  where $e = 2.718$ is the *Nepero* number.

Combinatorial Lower Bound

- **Q.** How many bits do I need to represent the list L?

  A. There are $\binom{U}{n}$ ways of choosing n distinct integers from a universe of size $U \geq n$,

  thus we need at least $\left\lceil \log_2 \binom{U}{n} \right\rceil$ bits, that is (by Stirling's approximation):

  $$\left\lceil \log_2 \binom{U}{n} \right\rceil \approx n\left(\log_2\left(\frac{U}{n}\right) + \log_2 e\right) \approx n \log_2\left(\frac{U}{n}\right) + 1.44n \text{ bits},$$

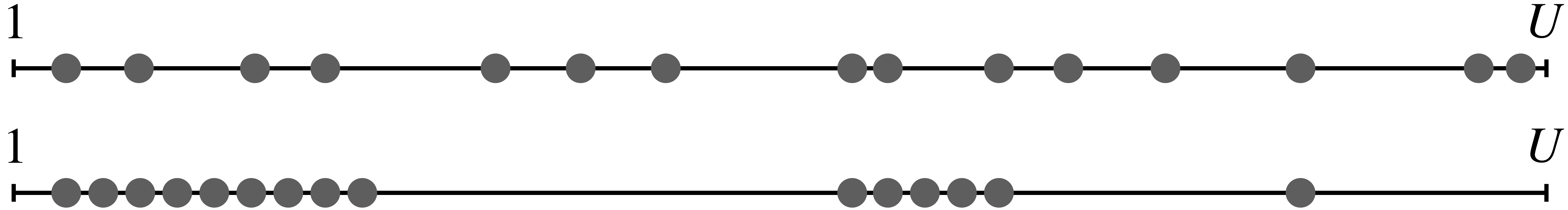
  where $e = 2.718$ is the *Nepero* number.

- **Spoiler!** Most compressors surveyed in this module beat the lower bound on *real* data. Why?

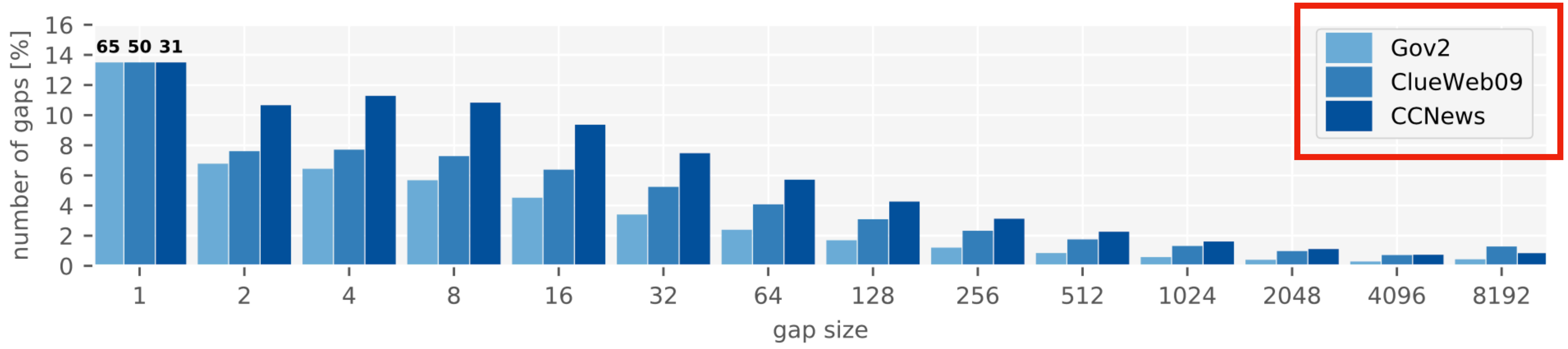
Combinatorial Lower Bound — Intuition

- **Spoiler!** Most compressors surveyed in this module beat the lower bound on *real* data.
- **Q.** Why?
  A. Because the data is not (usually) random.

  For these two lists below, the number of bits computed by the lower bound is the *same* just because they have the same length $n = 15$ and the same universe U.



Folklore Strategies



Web Search engine datasets

- Just *ignore* the property that the list is sorted, but do *exploit* the property that the *gaps* between the integers are *small* on average:
    gaps + Delta (for better space effectiveness)
    gaps + Variable-Byte (for better time efficiency)
- Use any code seen in Module 2 on the gaps.
- **Q.** Successor and Access?
  A. If only the gaps are encoded, then we are forced to *decode* the list. So both queries are answered in $O(n)$ time.
  (Not splendid!)

Example.
If $L = [1,3,13,14,15,16,20,22,23,34,35,36,40]$, taking the gaps we obtain
$L' = [1,2,10,1,1,1,4,2,1,11,1,1,4]$.

Blocking

- **General idea.** Split the list into *blocks* (either of fixed or variable length) and encode each block independently.
- Very simple but very effective strategy if the list is locally homogeneous, i.e., the gaps are *all* small inside a block.
- Different coding strategy can be used on the blocks according to their characteristics (e.g., largest integer, average gap, ecc.).

Example: A two-level representation with blocks of (at most) 5 integers.

$L = [1,3,13,14,15 \mid 16,20,22,23,34 \mid 35,36,40,48,51 \mid 52,53,54]$

$upper = [15,34,51,54]$ (encoding of the last elements of the blocks)

$lower = [1,2,10,1][1,4,2,1][1,1,4,8][1,1]$ (gaps of the integers in the blocks)

$widths = [4,3,4,1]$ (all gaps from block i can be represented in $widths[i]$ bits)

Blocking — Operations

- Succ(x):
  binary search x on the upper bounds of the blocks to locate the block where x lies in, then decode the block.

  For a block size of B integers, the complexity is $O(\log(n/B) + B)$.

- Access(i):
  the i-th integer lies in the block $\lfloor i/B \rfloor$, so we decode it.

  The complexity is $O(B)$.

PForDelta

Zukowski *et al.*, 2006

- Suppose we have the following gapped block:
  $[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,8247,1,1,1,1,1,1,1,1,1,1,1,1,1,1]$.

  The presence of just one single integer causes the binary width to be large for all the integers in the block. (In the example, we will use $\lceil \log_2 8247 \rceil = 14$ bits for all the integers, whereas all of them except one could be encoded in 1 bit.)

- The idea of PForDelta (Patched Frame of Reference, with Delta encoding) is to choose a base b and a value $k > 0$ such that *most* (e.g., 90%) of the integers in a block fall into the range $[b, b + 2^k - 1)$.

- Each integer x falling into the range can written as the delta $x - b$ in k bits.

- Each integer $x > b + 2^k$ is assigned the *exception* codeword $2^k - 1$ and coded in a separate list.

Example for $L = [3,4,7,21,9,12,5,16,6,2,34]$.
We can choose $b = 2$ and $k = 4$, and model L as $[1,2,5,*,7,10,3,*,4,0,*] - [21,16,34]$.
The first part will be coded as:
`0001.0010.0101.1111.0111.1010.0011.1111.1000.0000.1111`

Simple

Anh and Moffat, 2005

- Consider again the gapped blocks we introduced before.
- Simple4b (also called Simple9): how many gaps would fit into a 32-bit word of memory?
  (9 possible configurations.)
- Simple8b: how many gaps would fit into a 64-bit word of memory?
  (16 possible configurations.)
- It clearly depends on the (binary) magnitude of the integers to encode: specify this with a *selector* code.
  (Simple16 has 16 configurations using 32-bit words.)

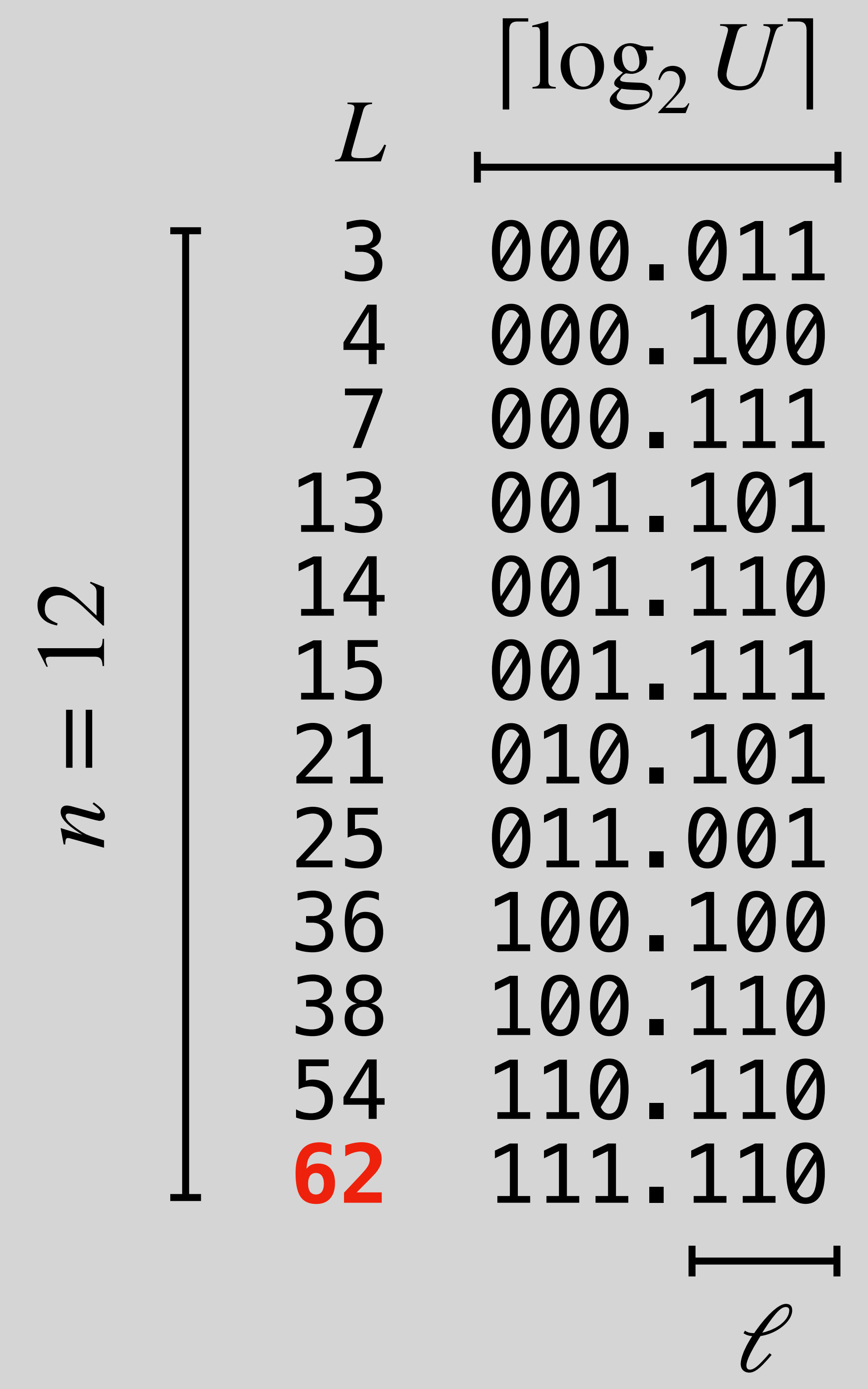
4-bit selector	integers	bits per integer	wasted bits
`0000`	28	1	0
`0001`	14	2	0
`0010`	9	3	1
`0011`	7	4	0
`0100`	5	5	3
`0101`	4	7	0
`0110`	3	9	1
`0111`	2	14	0
`1000`	1	28	0

Simple4b

Elias-Fano

Elias, 1974 — Fano, 1971

- The binary representation of each integer x in $\lceil \log_2 U \rceil$ bits is split into two parts: the least most significant $\ell = \lceil \log_2(U/n) \rceil$ bits and the remaining $(\lceil \log_2 U \rceil - \ell)$ bits — the *low* and *high* bits, respectively.
- The low bits are written in a vector, `low_bits`, of $n\ell$ bits (each integer takes ℓ bits). The high bits are *clustered together* and written in unary in another bit-vector `high_bits`.
- **Q.** How many bits for the high bits?
  A. We have a `0` bit for each possible cluster. We have $U/2^{\ell} + 1$ clusters, which are n at most.
  (If the high bits of U is the integer c, then we have $c + 1$ clusters).
  Plus 1 bit for each integer. The cost of `high_bits` is $2n$ bits at most.
- The overall space of Elias-Fano is at most $n \lceil \log_2(U/n) \rceil + 2n$ bits.
- Recall that the space lower bound is $n \log_2(U/n) + 1.44n$ bits. Thus, Elias-Fano is approximately 0.56 bits per element away from the optimum!

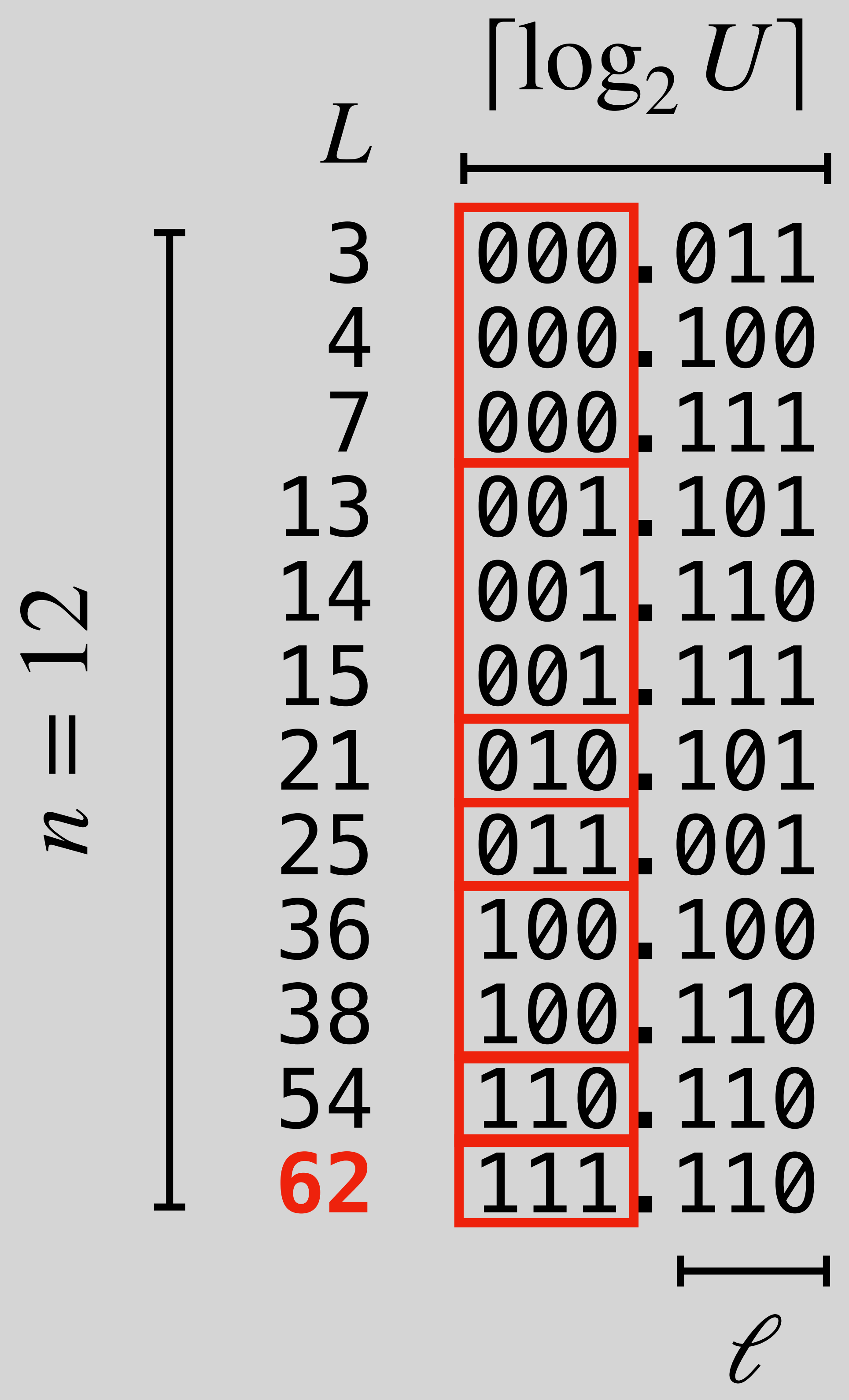


L	$\lceil \log_2 U \rceil$
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Elias-Fano

Elias, 1974 — Fano, 1971

- The binary representation of each integer x in $\lceil \log_2 U \rceil$ bits is split into two parts: the least most significant $\ell = \lceil \log_2(U/n) \rceil$ bits and the remaining $(\lceil \log_2 U \rceil - \ell)$ bits — the *low* and *high* bits, respectively.
- The low bits are written in a vector, `low_bits`, of $n\ell$ bits (each integer takes ℓ bits). The high bits are *clustered together* and written in unary in another bit-vector `high_bits`.
- **Q.** How many bits for the high bits?
  A. We have a `0` bit for each possible cluster. We have $U/2^{\ell} + 1$ clusters, which are n at most.
  (If the high bits of U is the integer c, then we have $c + 1$ clusters).
  Plus 1 bit for each integer. The cost of `high_bits` is $2n$ bits at most.
- The overall space of Elias-Fano is at most $n\lceil \log_2(U/n) \rceil + 2n$ bits.
- Recall that the space lower bound is $n \log_2(U/n) + 1.44n$ bits. Thus, Elias-Fano is approximately 0.56 bits per element away from the optimum!



$\lceil \log_2 U \rceil$

L	
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

$n = 12$

ℓ

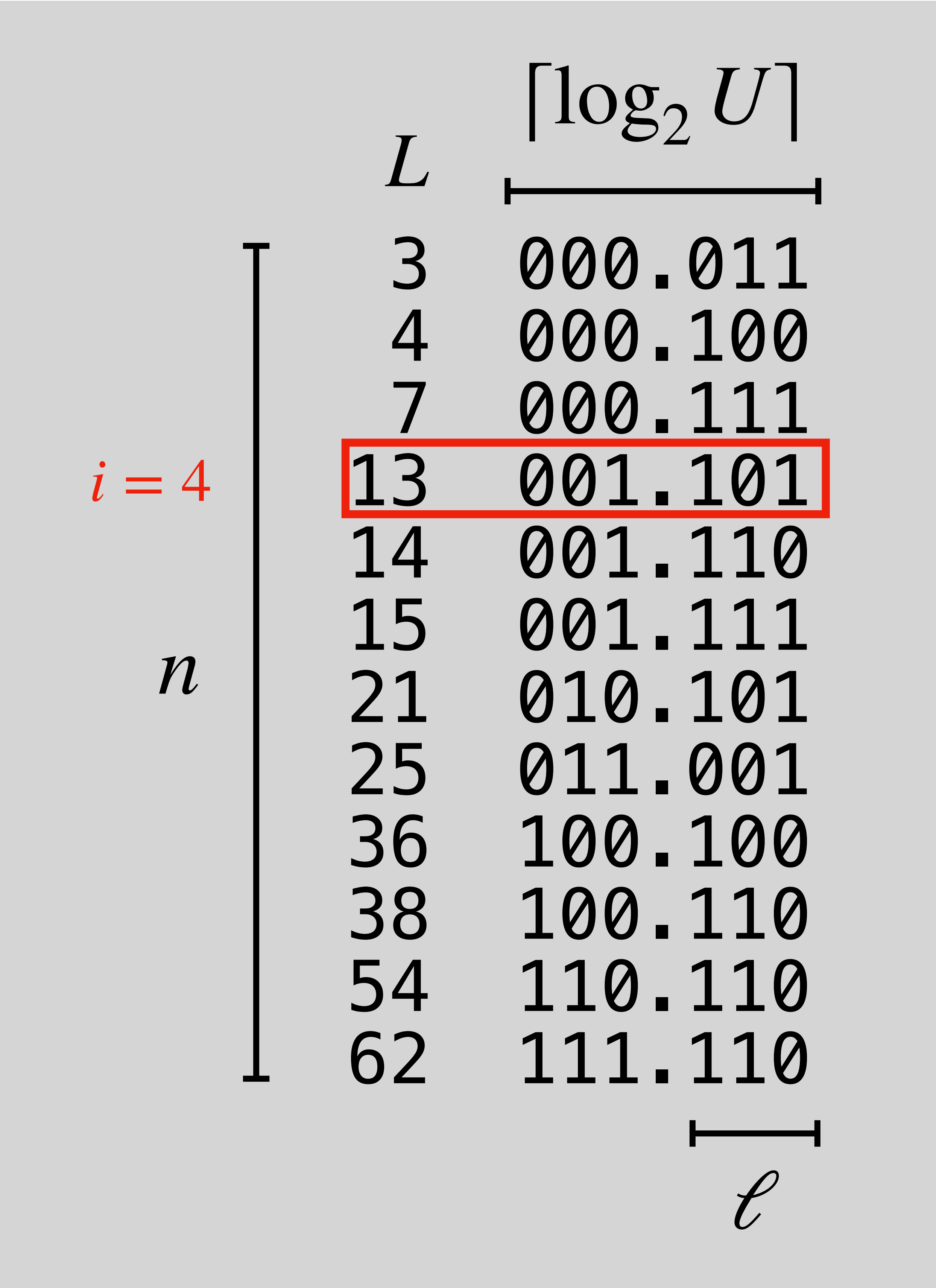
```
high_bits =
1110.1110.10.10.110.0.10.10
low_bits =
011.100.111.101.110.111.101.00
1.100.110.110.110
```

Elias-Fano — Random Access

- One of the most important properties of Elias-Fano is that it supports *random* Access in $O(1)$ time.
- Now that we have the list L encoded as

`high_bits` = `1110.1110.10.10.110.0.10.10`
`low_bits` = `011.100.111.101.110.111.101.001.100.110.110.110`

how to retrieve $L[i]$ given a random $i \in [1,n]$?



L	$\lceil \log_2 U \rceil$
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Elias-Fano — Random Access

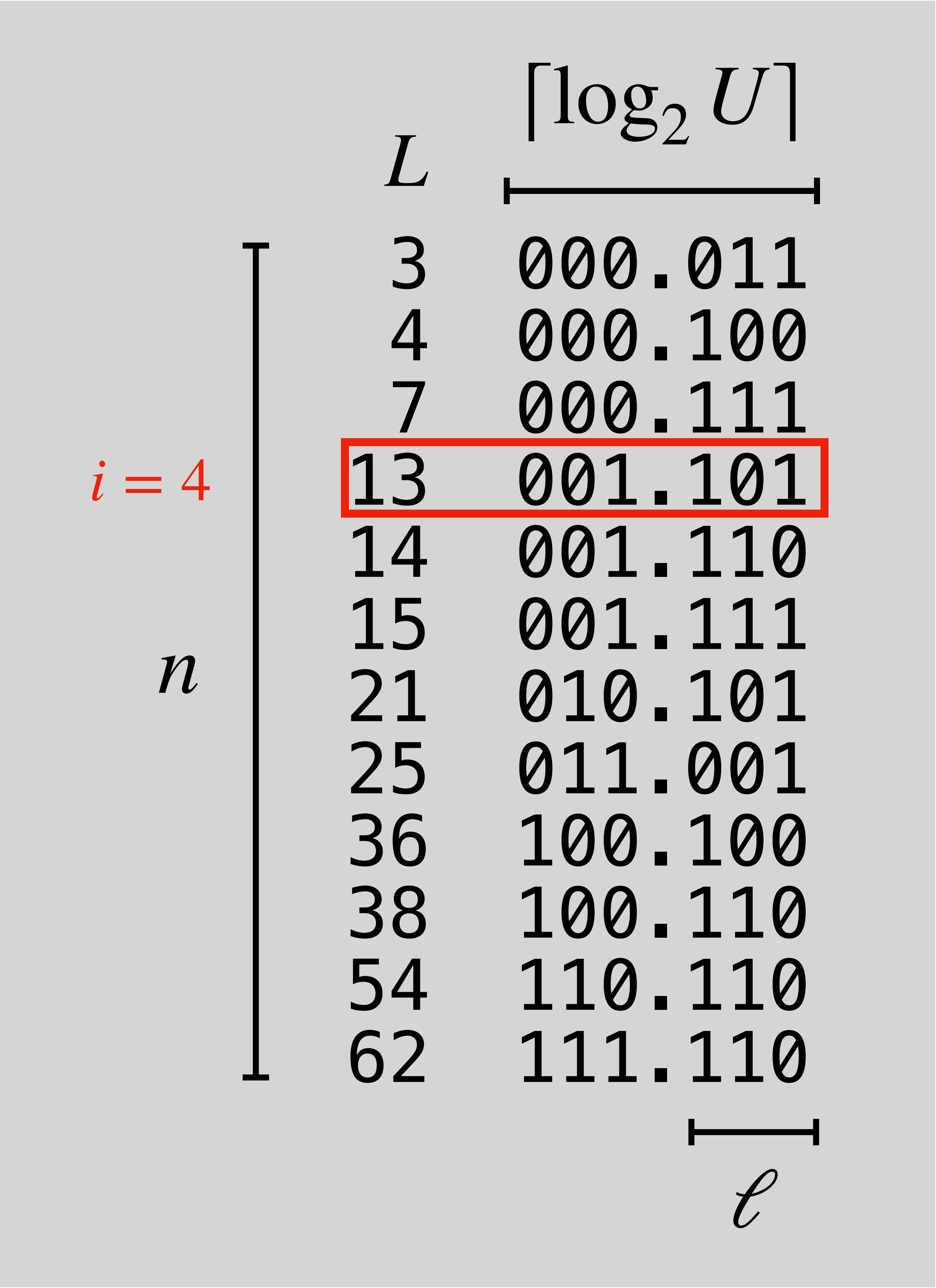
- One of the most important properties of Elias-Fano is that it supports *random* Access in $O(1)$ time.

- Now that we have the list L encoded as

  `high_bits` =
  `1110.1110.10.10.110.0.10.10`
  `low_bits` =
  `011.100.111.101.110.111.101.001.100.110.110.110`

  how to retrieve $L[i]$ given a random $i \in [1,n]$?

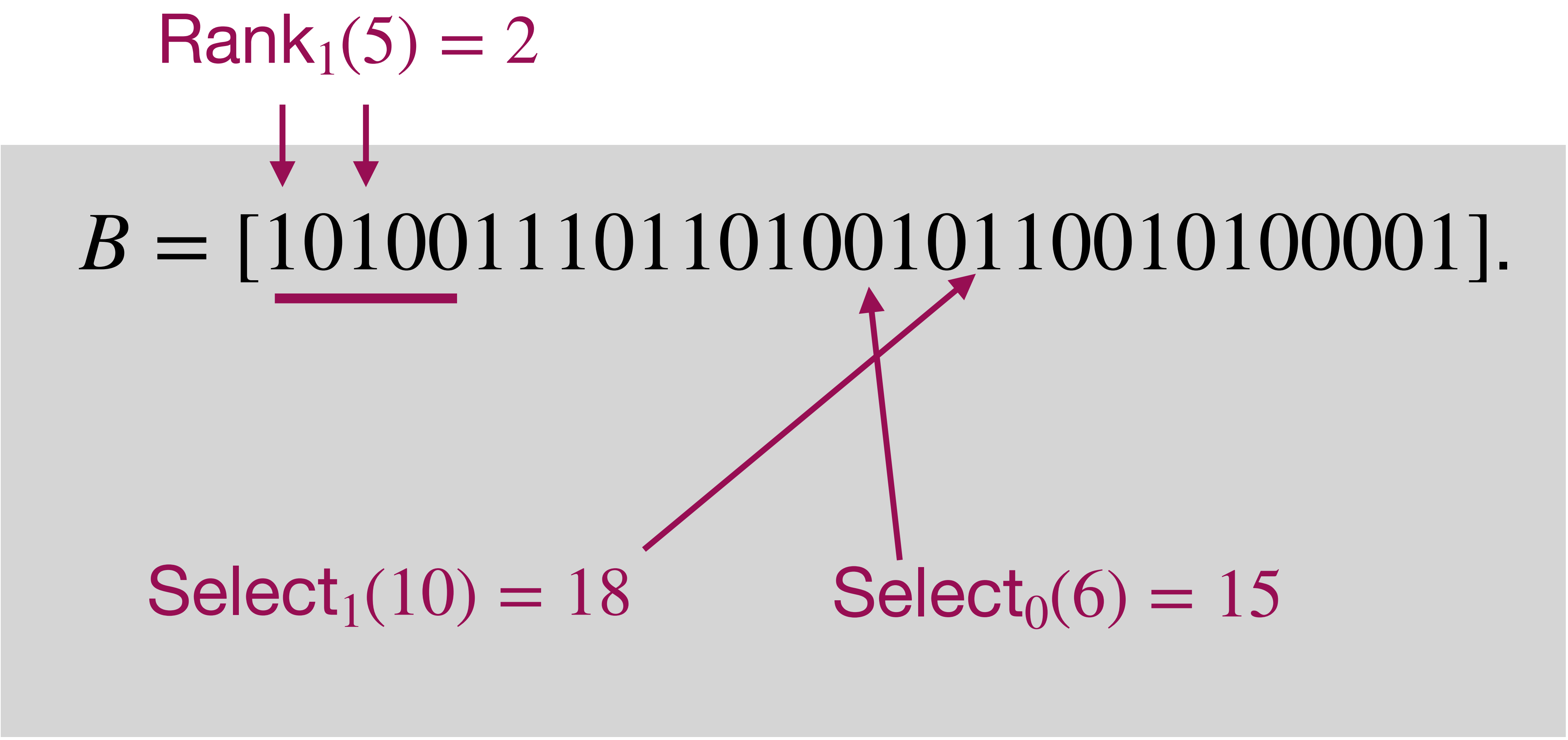
- Before illustrating the Access algorithm, we need to introduce one more tool: *Rank & Select* queries on bit-vectors.



3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Rank & Select Queries on Bit-Vectors

- We are given a bit-vector B of m bits:
  - Rank$_b(i)$ returns the number of b bits in $B[1,i]$, where b is either a 0 or 1 bit.
  - Select$_b(i)$ returns the position of the i-th b bit.
- By definition, we have:
  - Rank$_b($Select$_b(i)) = i$,
  - Rank$_1($Select$_0(i)) =$ Select$_0(i) - i$, and
  - Rank$_0($Select$_1(i)) =$ Select$_1(i) - i$.
- Both queries can be trivially implemented in $O(m)$ time and no extra space, or in practically $O(1)$ time using as little as $o(m)$ extra bits ($\approx 3 - 10\,\%$ extra bits in practice).



Rank$_1(5) = 2$

$$B = [10100111011010010110010100001].$$

Select$_1(10) = 18$

Select$_0(6) = 15$

Elias-Fano — Random Access

- Access(i):

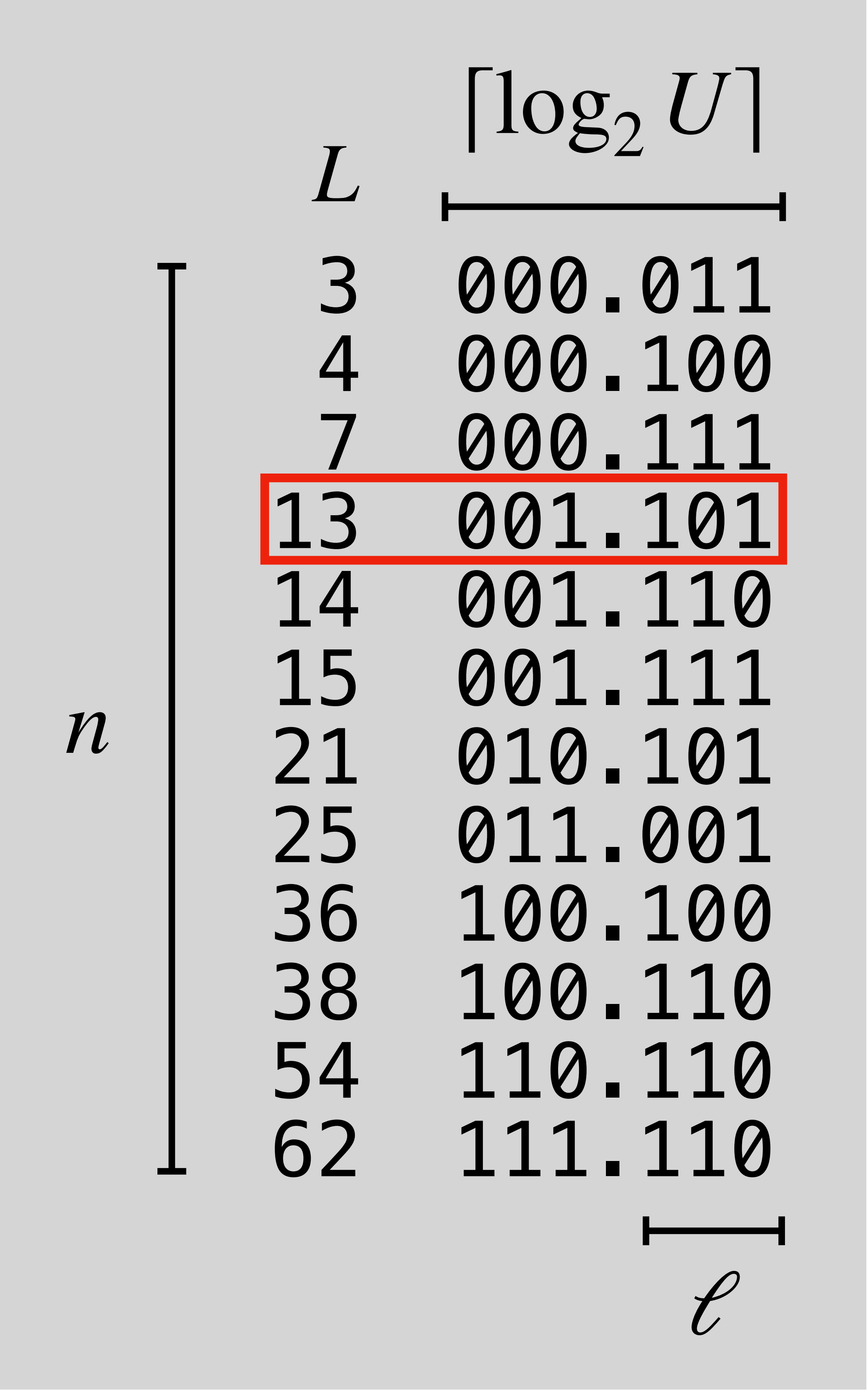
```
    $l = low\_bits[i]$
    $h = \text{Rank}_0(\text{Select}_1(i))$
    return $(h \ll \ell) \mid l$
```

$\text{Select}_1(i) - i$

- Example for $i = 4$. L is encoded as:

```
high_bits =
1110.1110.10.10.110.0.10.10
   0    1  2  3   4 5  6  7

low_bits =
011.100.111.101.110.111.101.001.100.110.110.110
```



L	
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Elias-Fano — Random Access

$\text{Select}_1(i) - i$

- Access(i):

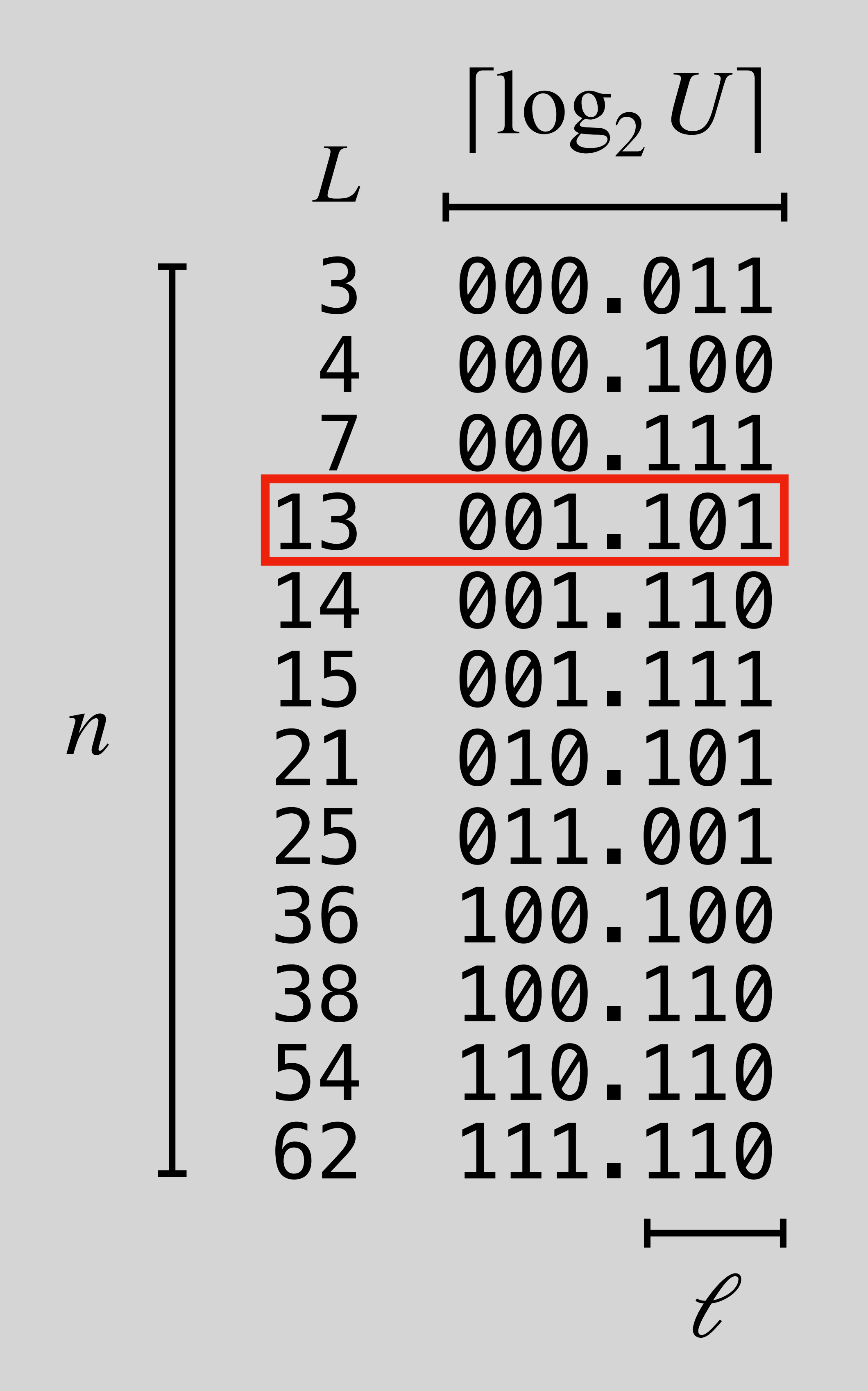
```
    $l = low\_bits[i]$
    $h = \text{Rank}_0(\text{Select}_1(i))$
    return $(h \ll \ell) \,|\, l$
```

$l = low\_bits[4] = 5$ // 101

- Example for $i = 4$. L is encoded as:

```
high_bits =
1110.1110.10.10.110.0.10.10
   0    1  2  3   4 5  6  7
```

```
low_bits =
011.100.111.101.110.111.101.001.100.110.110.110
```



L	$\lceil \log_2 U \rceil$
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Elias-Fano — Random Access

- Access(i):

```
    $l = low\_bits[i]$
    $h = \text{Rank}_0(\text{Select}_1(i))$
    return $(h \ll \ell) \,|\, l$
```

$\text{Select}_1(i) - i$

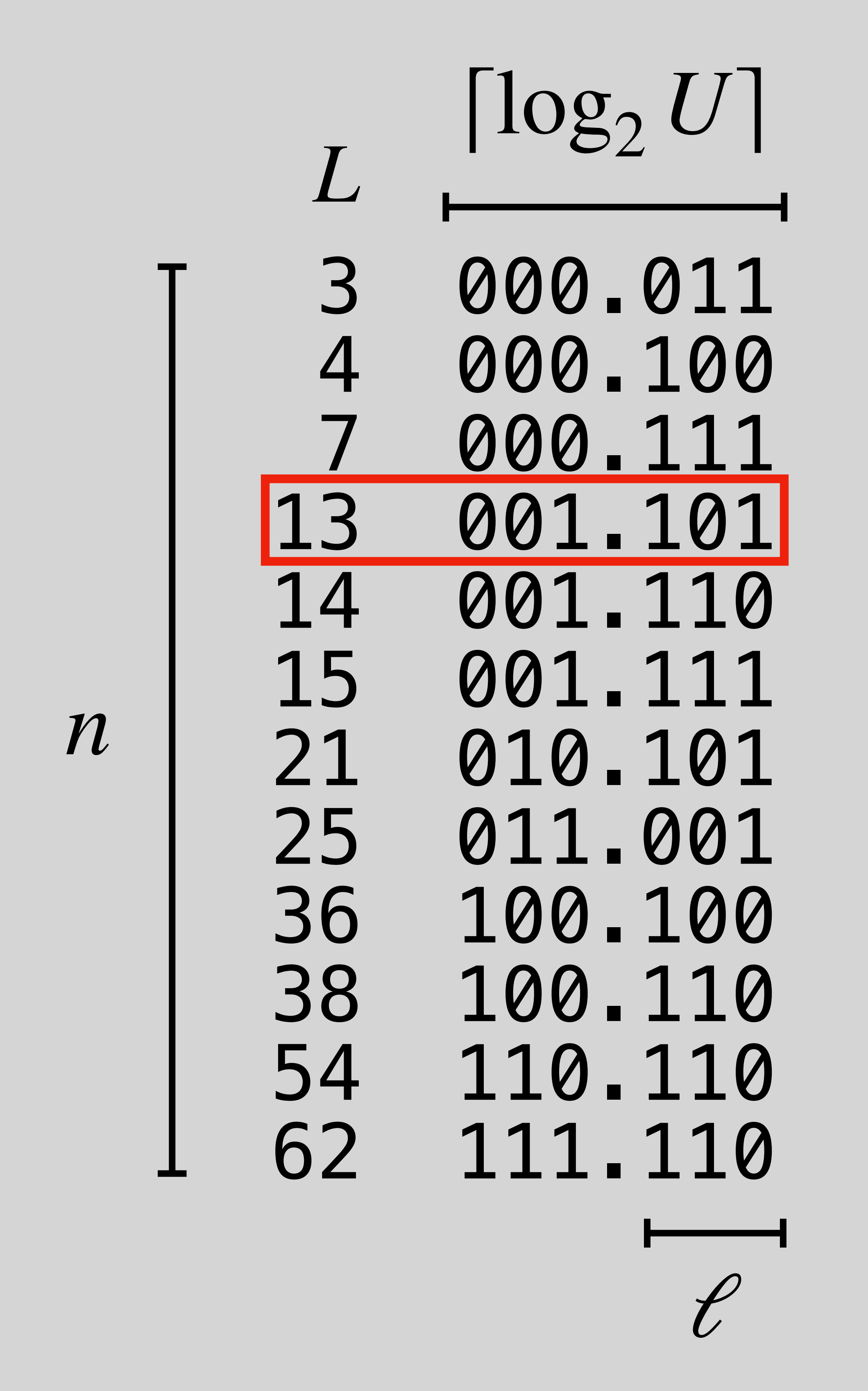
$l = low\_bits[4] = 5$ // 101

$h = \text{Select}_1(4) - 4 = 5 - 4 = 1$

- Example for $i = 4$. L is encoded as:

```
high_bits =
1110.1110.10.10.110.0.10.10
   0    1  2  3   4 5  6  7

low_bits =
011.100.111.101.110.111.101.001.100.110.110.110
```



L	$\lceil \log_2 U \rceil$
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Elias-Fano — Random Access

Select$_1(i) - i$

- Access(i):
  - $l = low\_bits[i]$ ← $l = low\_bits[4] = 5$ // `101`
  - $h =$ Rank$_0$(Select$_1(i)$) ← $h =$ Select$_1(4) - 4 = 5 - 4 = 1$
  - return $(h \ll \ell) \,|\, l$ ← $(1 \ll 3) \,|\, 5 = 13$ // `001.101`

- Example for $i = 4$. L is encoded as:

```
high_bits =
1110.1110.10.10.110.0.10.10
   0     1  2  3    4 5  6  7

low_bits =
011.100.111.101.110.111.101.001.100.110.110.110
```

n

L	$\lceil \log_2 U \rceil$
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

ℓ

Elias-Fano — Partitioned by Cardinality

Ottaviano and Venturini, 2014

- Note that the space of Elias-Fano, $n\lceil \log_2(U/n)\rceil + 2n$, just depends on n and U: it does *not* take advantage of any clustering between the integers of L.

- Suppose we have $L = [2,3,4,5,6,7,10,11,13]$, then Elias-Fano would take $9\lceil \log_2(13/9)\rceil + 18 = 27$ bits, whereas we can always represent L with a bit-vector of $U = 13$ bits (the so-called *characteristic bit-vector* of L):

```
 0  1  1  1  1  1  1  0  0  1  1  0  1
 1  2  3  4  5  6  7  8  9 10 11 12 13
```

- By doing $n\lceil \log_2(U/n)\rceil + 2n < U$, we see that Elias-Fano is convenient only if $n < U/4$, thus if the block is sufficiently *sparse*.

Elias-Fano — Partitioned by Cardinality

Ottaviano and Venturini, 2014

- Note that the space of Elias-Fano, $n\lceil \log_2(U/n)\rceil + 2n$, just depends on n and U:
  it does *not* take advantage of any clustering between the integers of L.

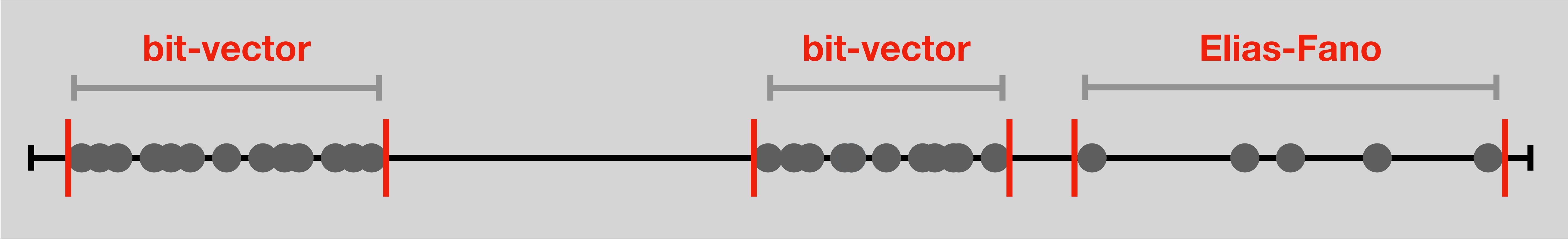
- Suppose we have $L = [2,3,4,5,6,7,10,11,13]$, then Elias-Fano would take $9\lceil \log_2(13/9)\rceil + 18 = 27$ bits, whereas we can always represent L with a bit-vector of $U = 13$ bits (the so-called *characteristic bit-vector* of L):

0	1	1	1	1	1	1	0	0	1	1	0	1
1	2	3	4	5	6	7	8	9	10	11	12	13

- By doing $n\lceil \log_2(U/n)\rceil + 2n < U$, we see that Elias-Fano is convenient only if $n < U/4$, thus if the block is sufficiently *sparse*.

- **Idea.** Use Elias-Fano on sparse blocks and the characteristic bit-vector representation on the dense blocks.

- Consider the following example:



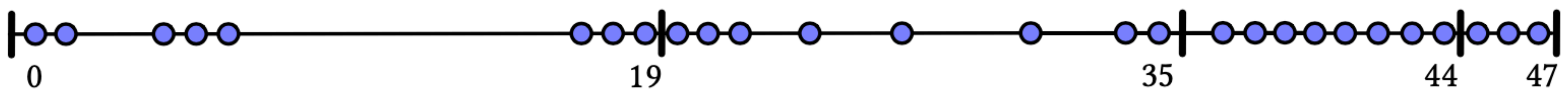
Elias-Fano — Partitioned by Universe

Chambi *et al.*, 2016 — Lemire *et al.*, 2018

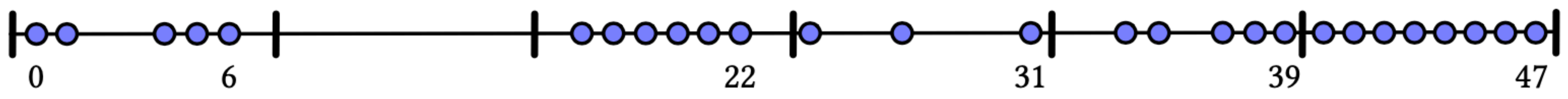
- Assume $U = 2^{32}$. Instead of splitting the 32-bit integers *parametrically* into $\ell = \lceil \log_2(2^{32}/n) \rceil$ low bits and remaining $32 - \ell$ high bits, we fix $\ell = 16$ and partition U into 2^{16} chunks: all integers falling into a chunk are encoded with either a bit-vector (dense case) or a sorted list (sparse case).

- The sparse chunk, now having a reduced universe $U' = 2^{16}$, can be further split into 2^8 chunks.

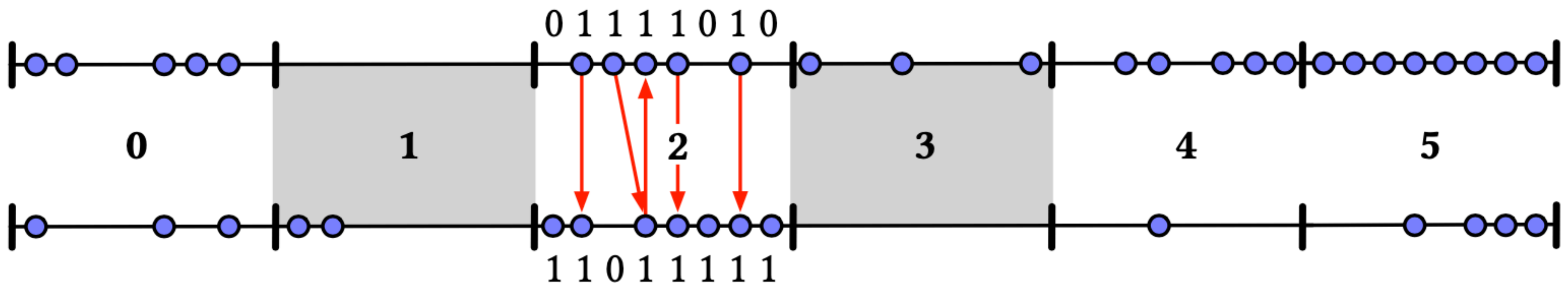
- This strategy is (usually) less effective than partitioning by cardinality, but allows faster operations, especially faster intersection/union of lists.



(a) partitioned by cardinality



(b) partitioned by universe



Binary Interpolative Coding

Moffat and Stuiver, 1996

- Suppose we have specified to quantities $l \leq L[1]$ and $h \geq L[n]$, then we can encode the element in the middle of the sequence, i.e., $L[m]$ with $m = \lceil n/2 \rceil$, exploiting the knowledge that $l \leq L[m] \leq h$.

- For example, we can write $L[m] - l$ using $\lceil \log_2(h - l) \rceil$ bits.

- Then apply the same encoding step to the halves $L[1..m-1]$ and $L[m+1..n]$ with *updated knowledge* of lower and upper bounds (l, h) that are set to, respectively, $(l, L[m] - 1)$ and $(L[m] + 1, h)$.

- The crucial property of the method is that whenever the length of the interval, say r, is equal to $h - l$, then a *run* of r consecutive integers is detected $(l, l+1, \ldots, l+r-1 = h)$ and *no bits* are necessary. *This is very effective on highly clustered sequences.*

Binary Interpolative Coding — Example

$$L = [3,4,7,13,14,15,21,25,36,38,54,62]$$

$$(n = 12, m = 6, l = 0, h = 62)$$

Binary Interpolative Coding — Example

$$L = [3,4,7,13,14,15,21,25,36,38,54,62]$$

$$(n = 12, m = 6, l = 0, h = 62)$$

Binary Interpolative Coding — Example

$$L = [3,4,7,13,14,15,21,25,36,38,54,62]$$

$$(n = 12, m = 6, l = 0, h = 62)$$

$$[3,4,7,13,14]$$

$$(n = 5, m = 3, l = 0, h = 14)$$

Binary Interpolative Coding — Example

$$L = [3,4,7,13,14,15,21,25,36,38,54,62]$$

$$(n = 12, m = 6, l = 0, h = 62)$$

$$[3,4,7,13,14]$$

$$(n = 5, m = 3, l = 0, h = 14)$$

Binary Interpolative Coding — Example

$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

Binary Interpolative Coding — Example

$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

Binary Interpolative Coding — Example

$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

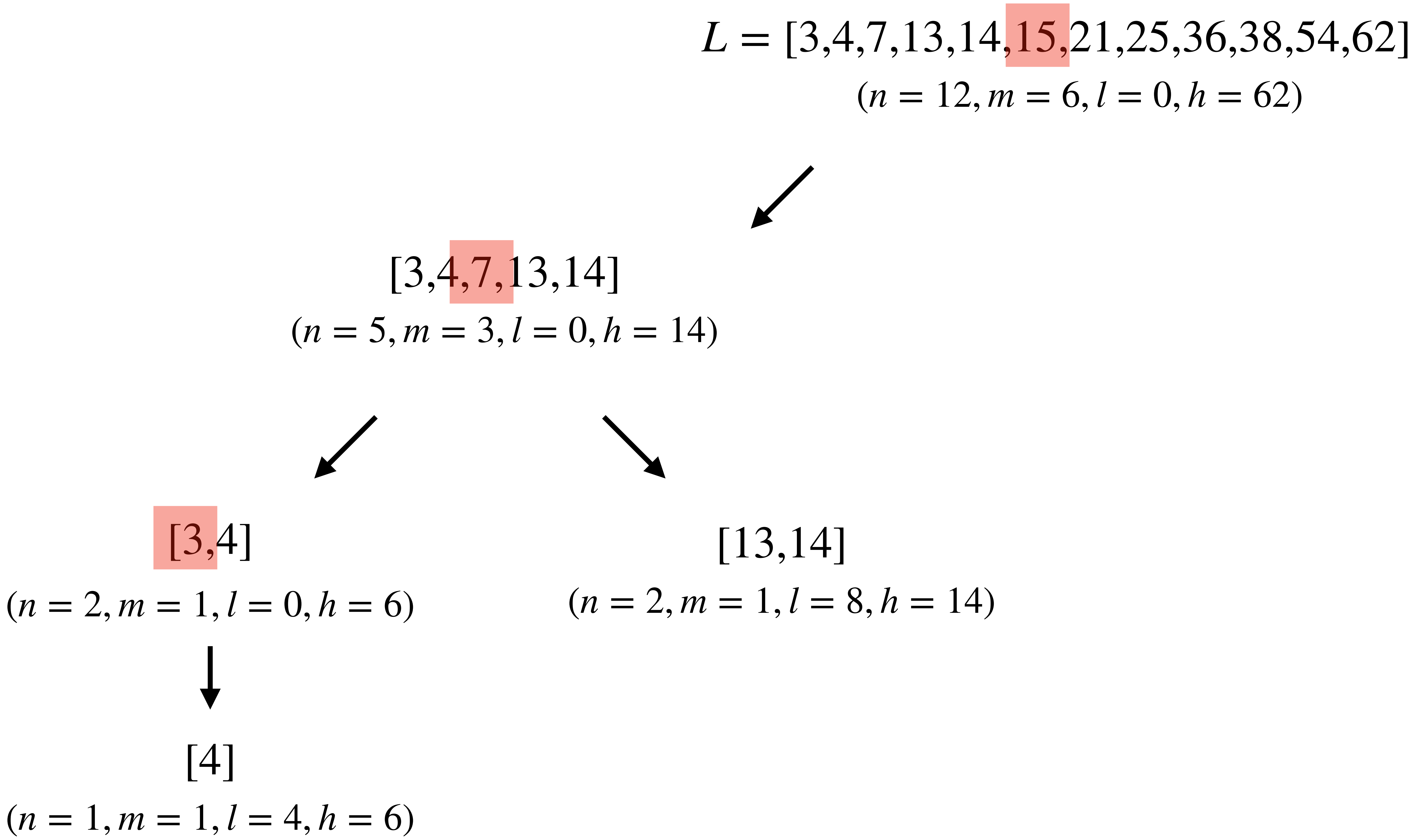
$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

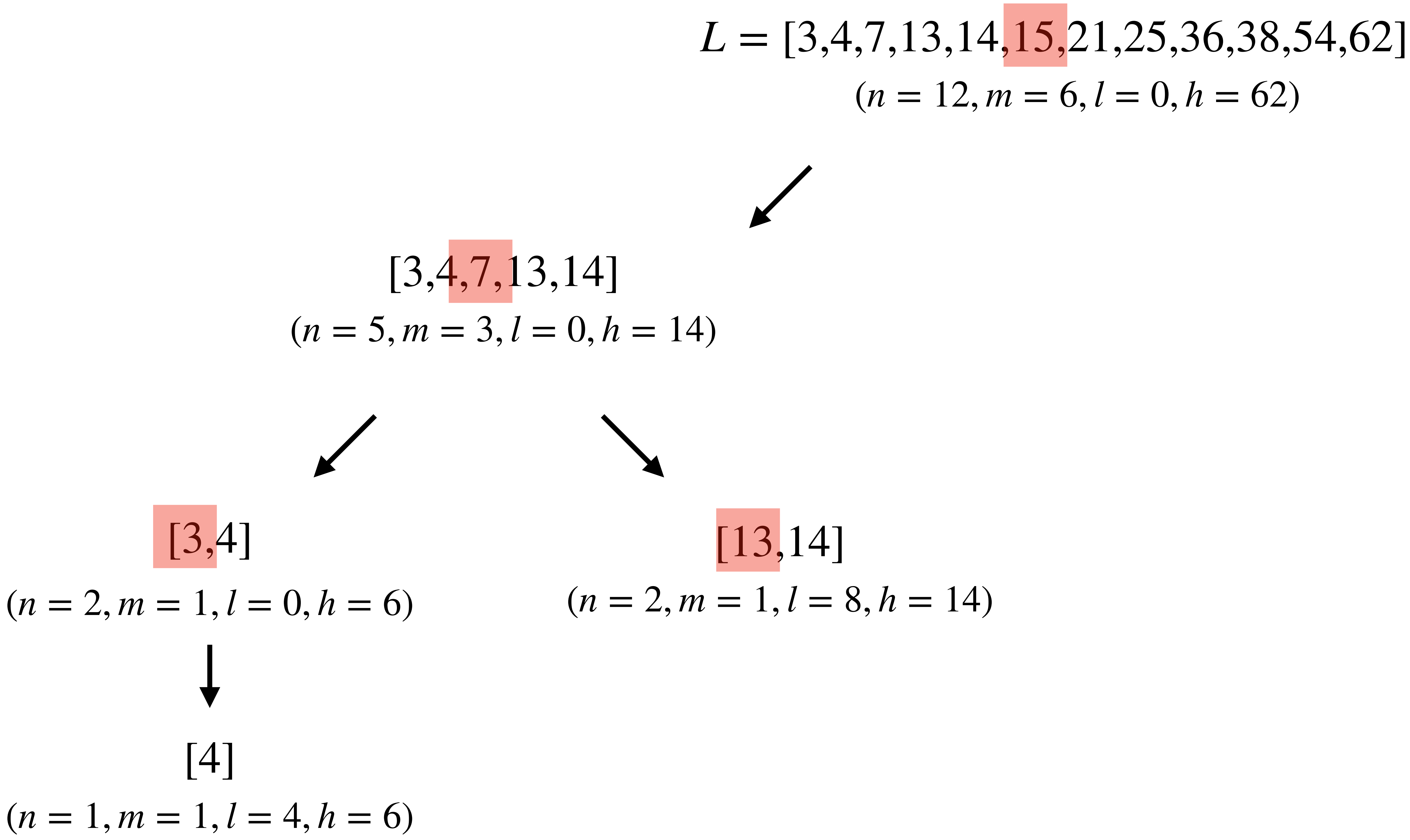
$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

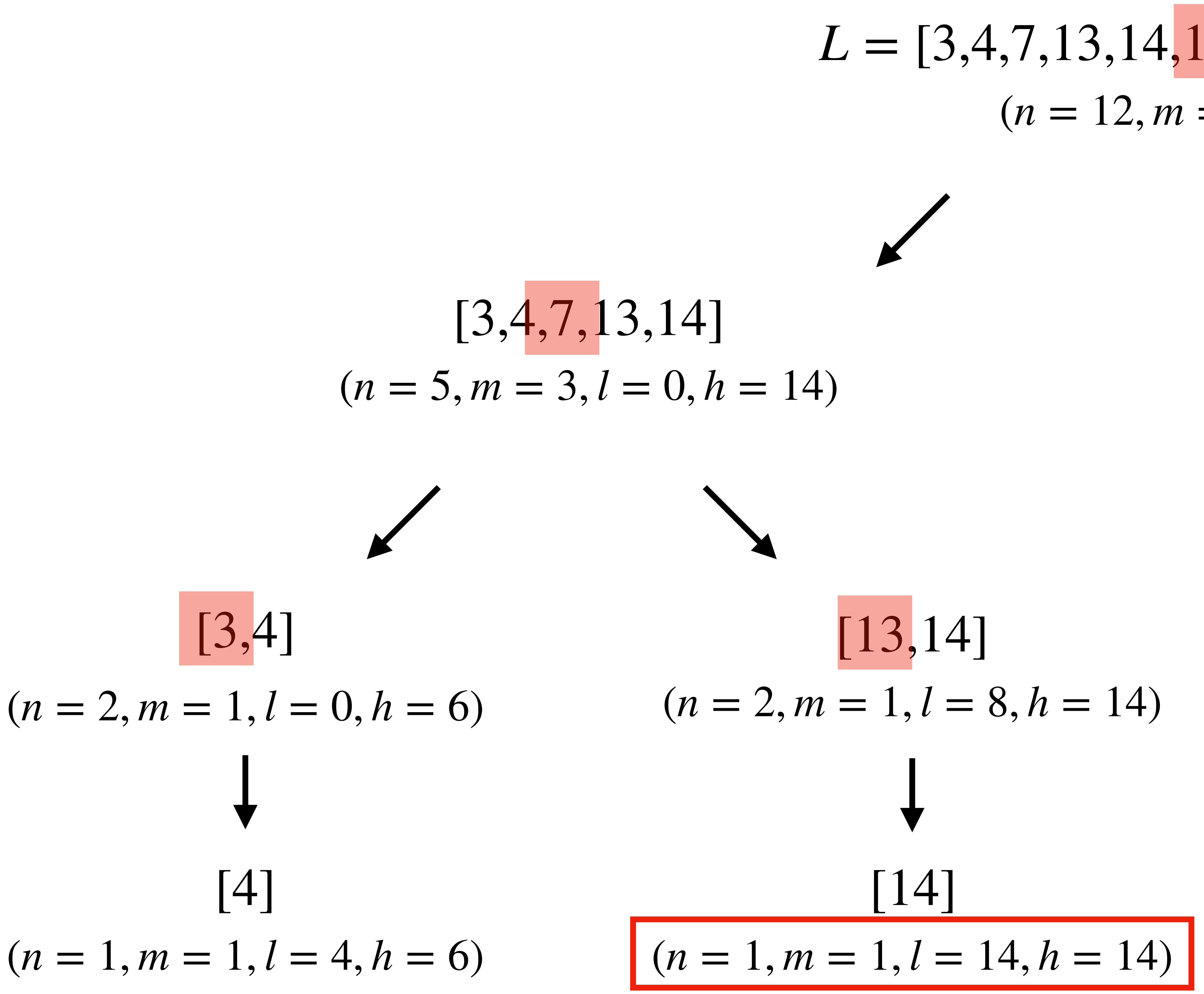
$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[4]$

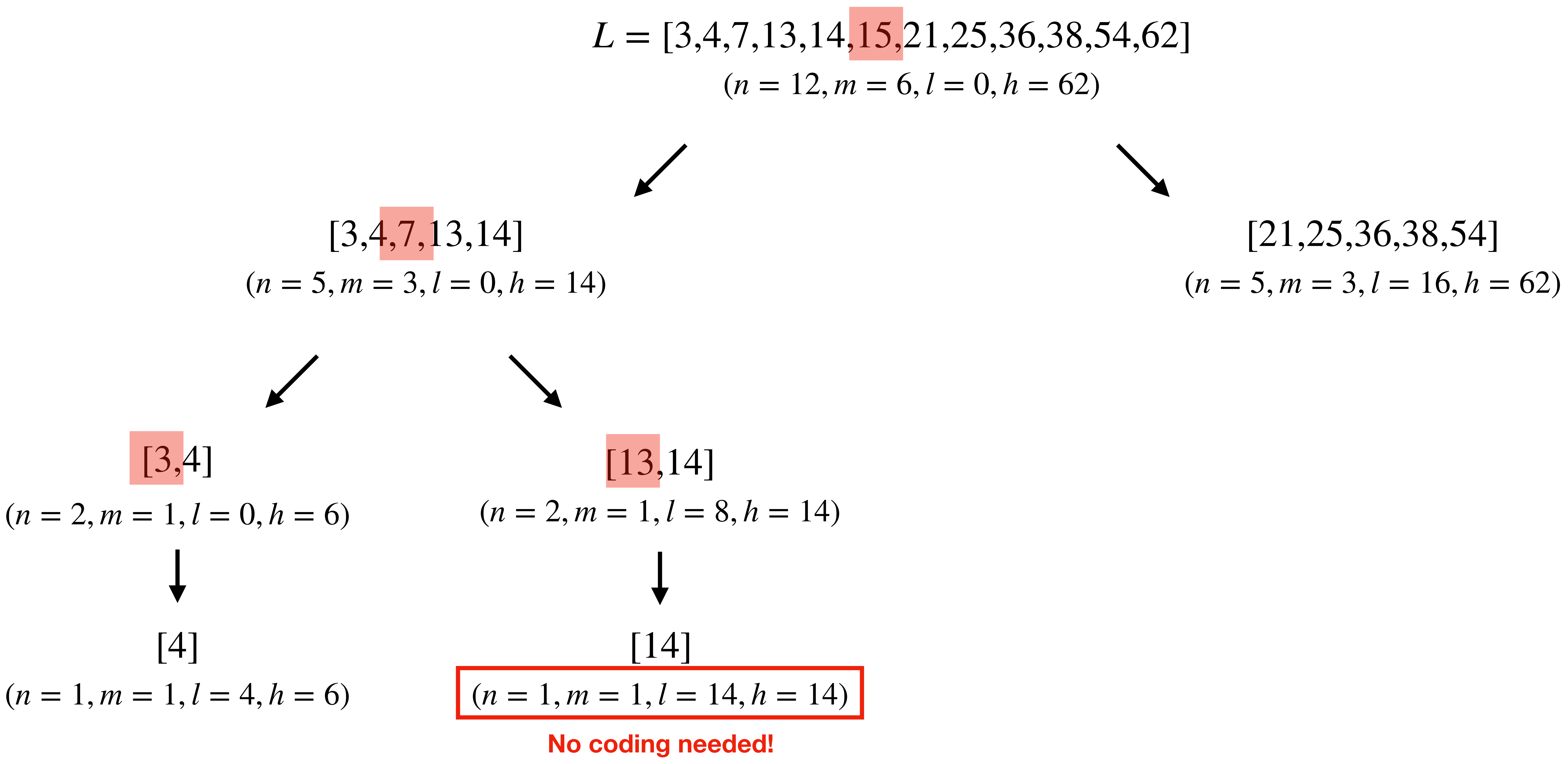
$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

$(n = 1, m = 1, l = 14, h = 14)$

No coding needed!

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[4]$

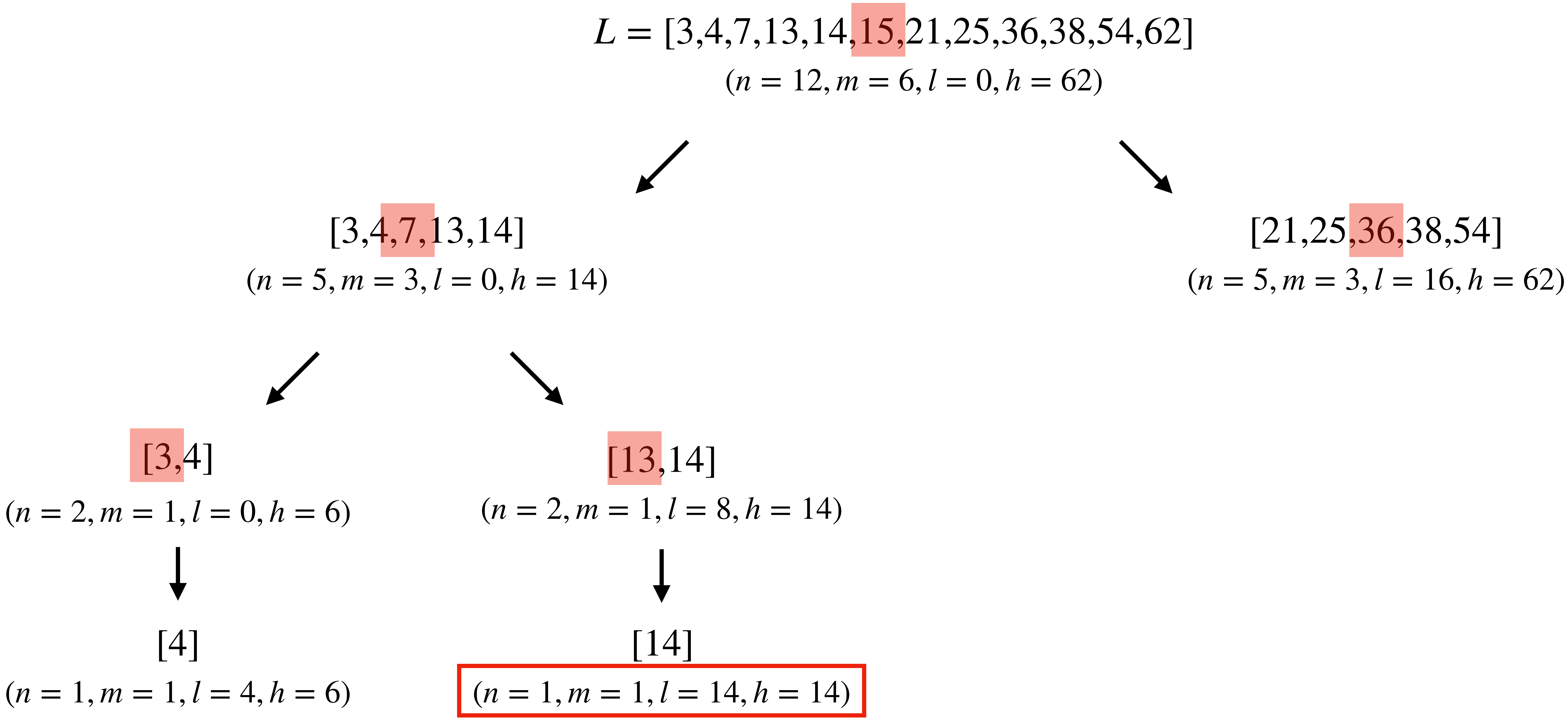
$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[4]$

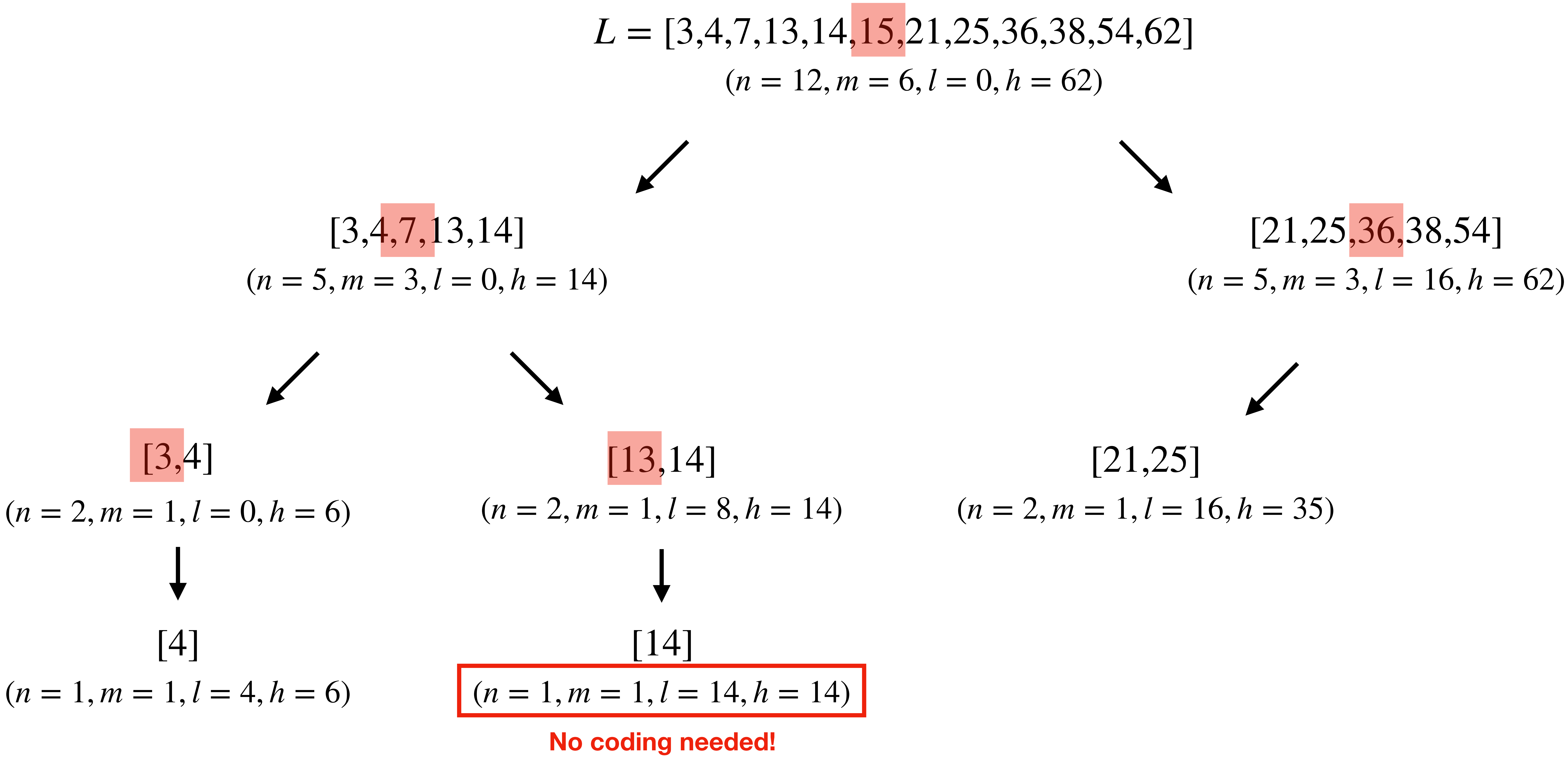
$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[21,25]$

$(n = 2, m = 1, l = 16, h = 35)$

$[4]$

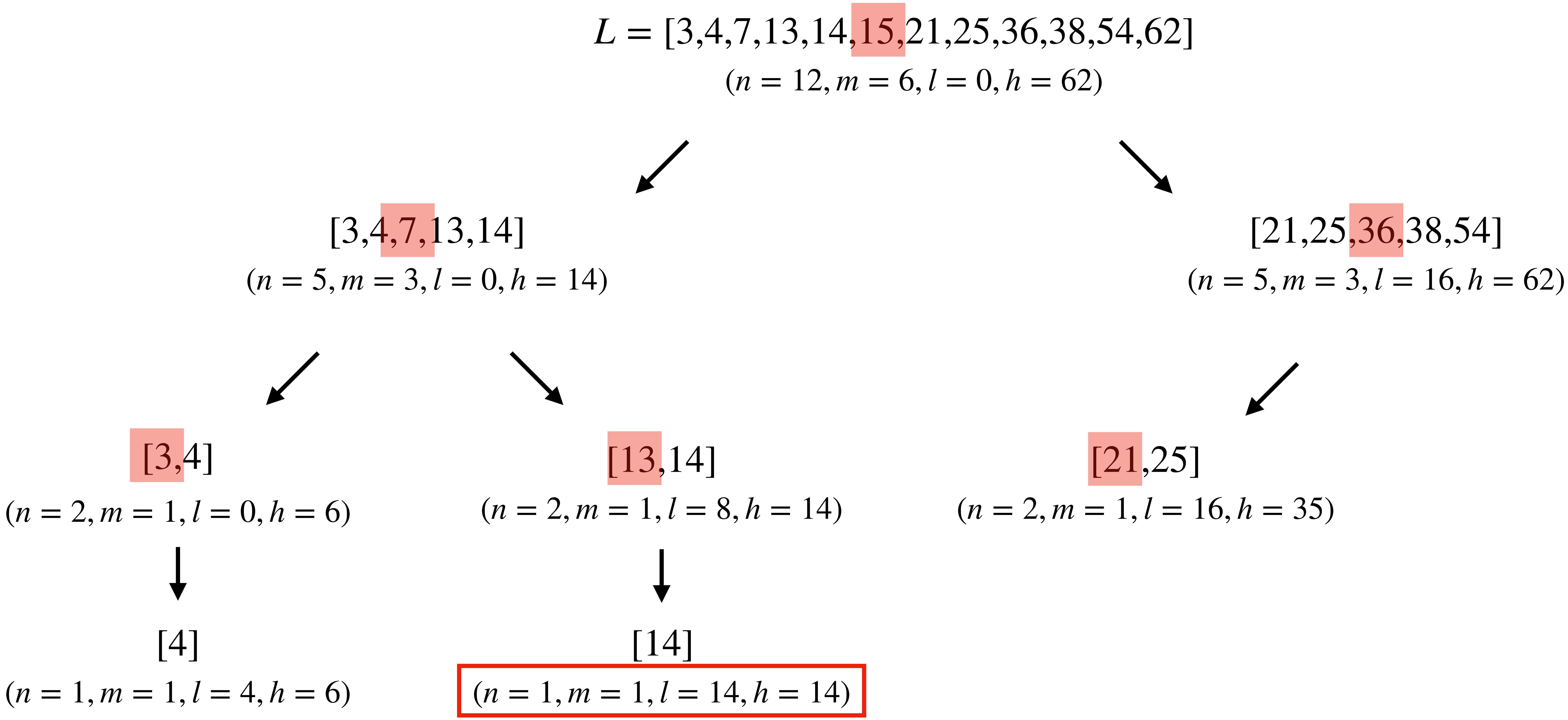
$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[21,25]$

$(n = 2, m = 1, l = 16, h = 35)$

$[4]$

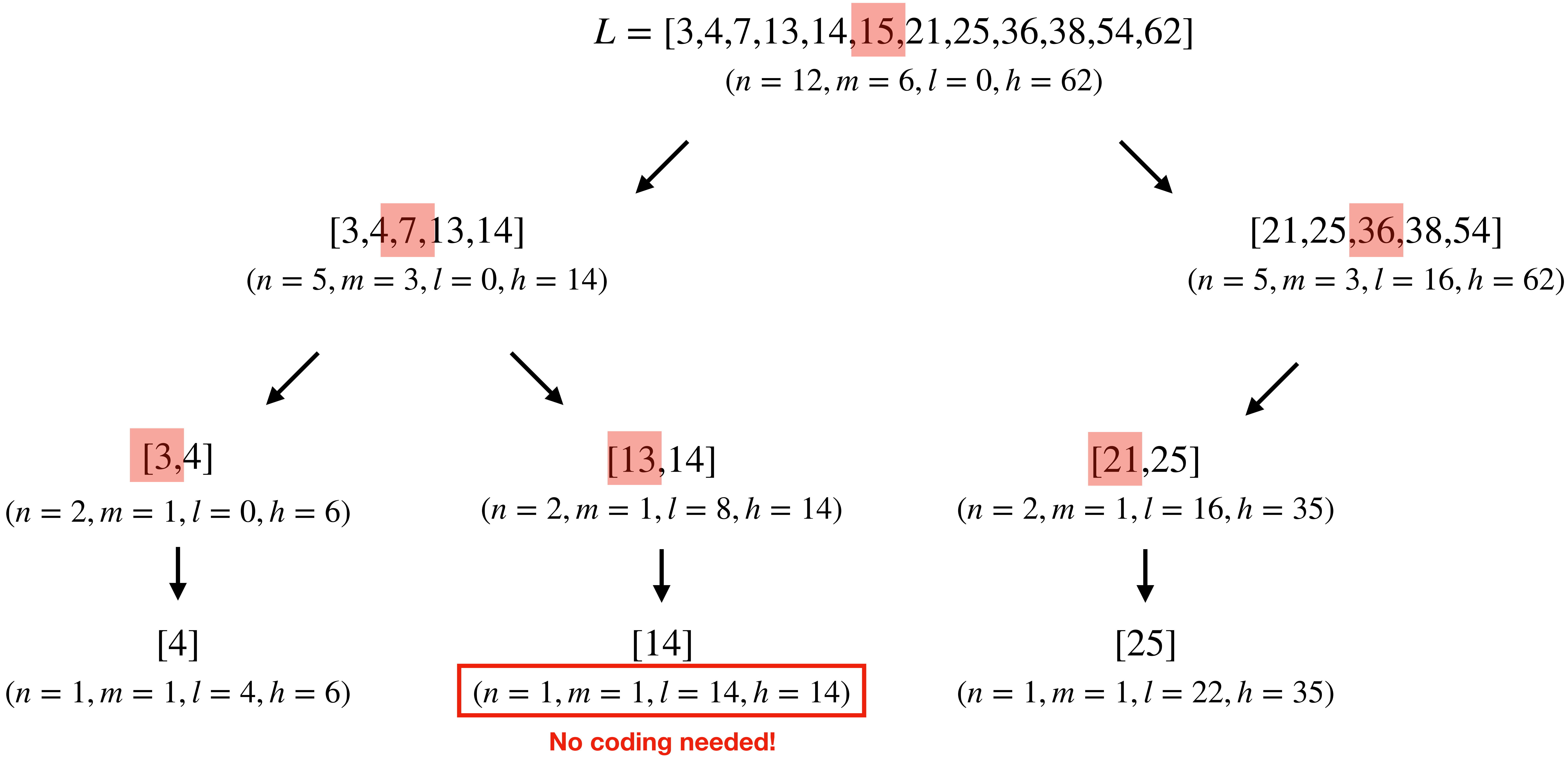
$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[21,25]$

$(n = 2, m = 1, l = 16, h = 35)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

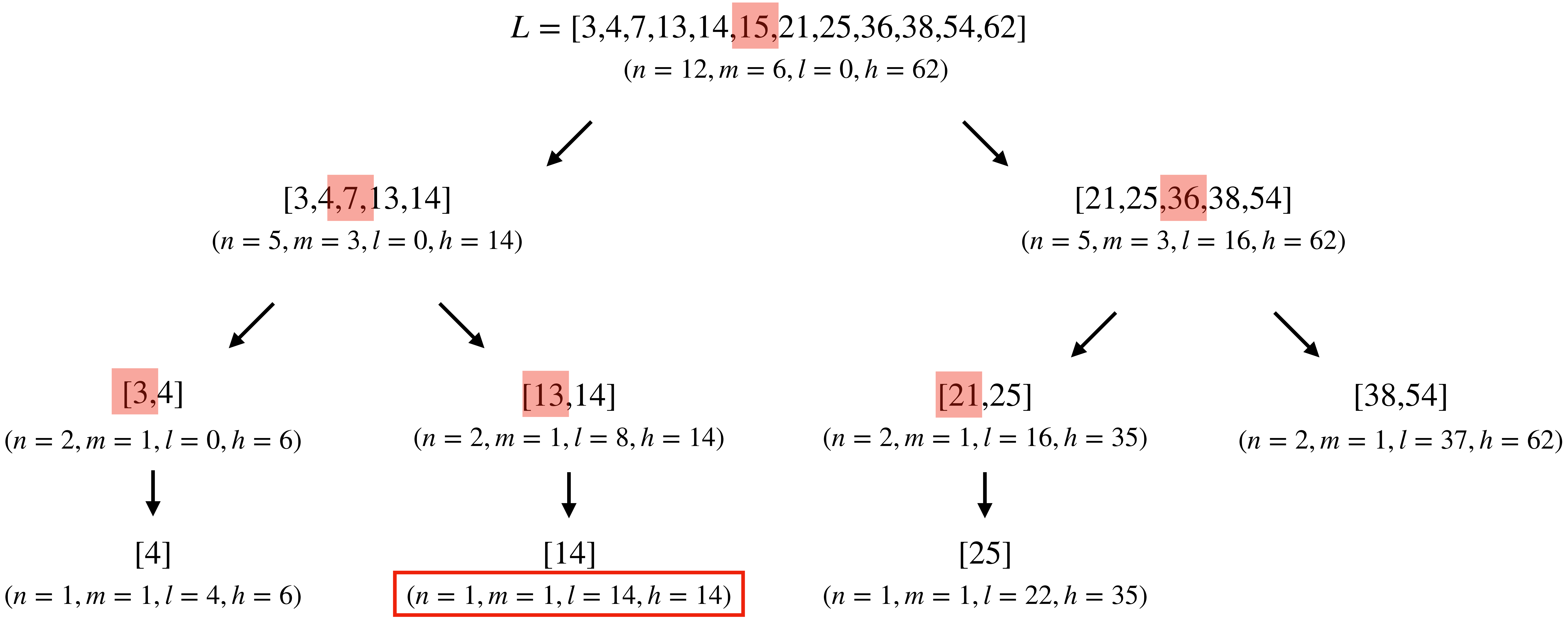
$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

$[25]$

$(n = 1, m = 1, l = 22, h = 35)$

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[21,25]$

$(n = 2, m = 1, l = 16, h = 35)$

$[38,54]$

$(n = 2, m = 1, l = 37, h = 62)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

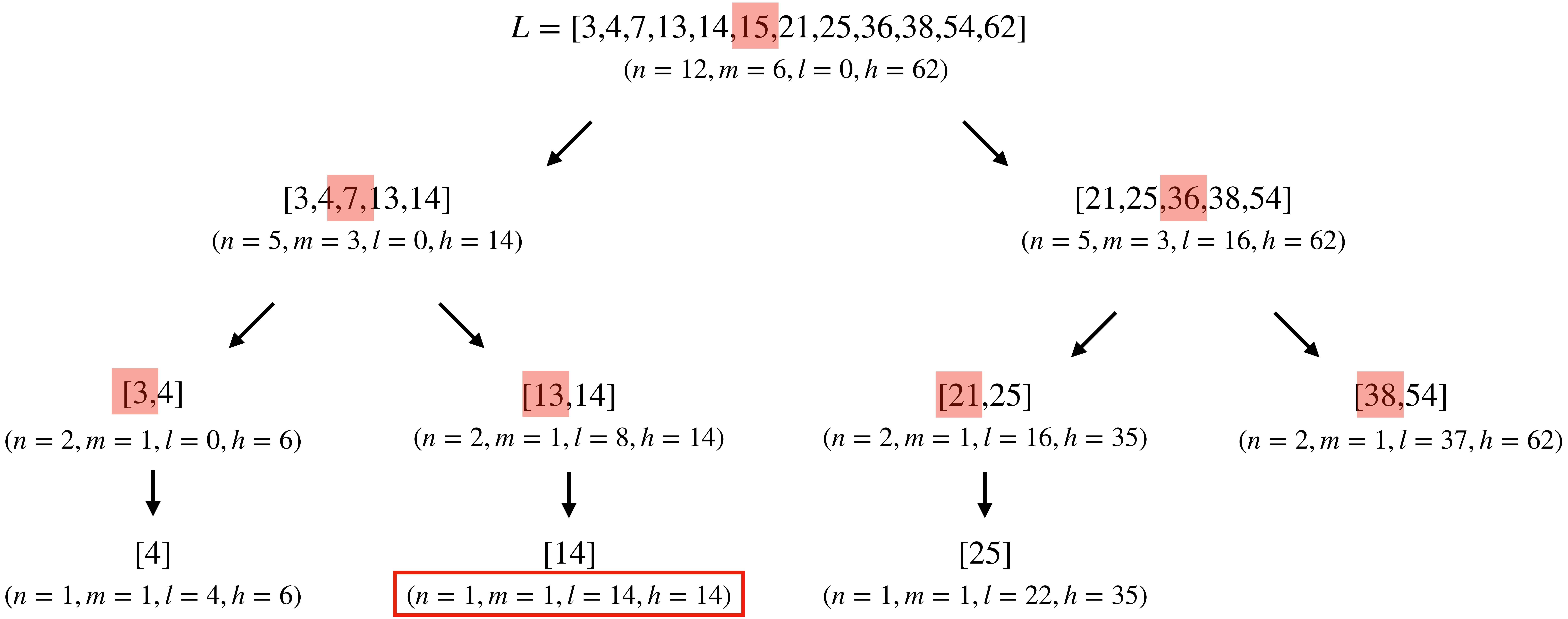
$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

$[25]$

$(n = 1, m = 1, l = 22, h = 35)$

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[21,25]$

$(n = 2, m = 1, l = 16, h = 35)$

$[38,54]$

$(n = 2, m = 1, l = 37, h = 62)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

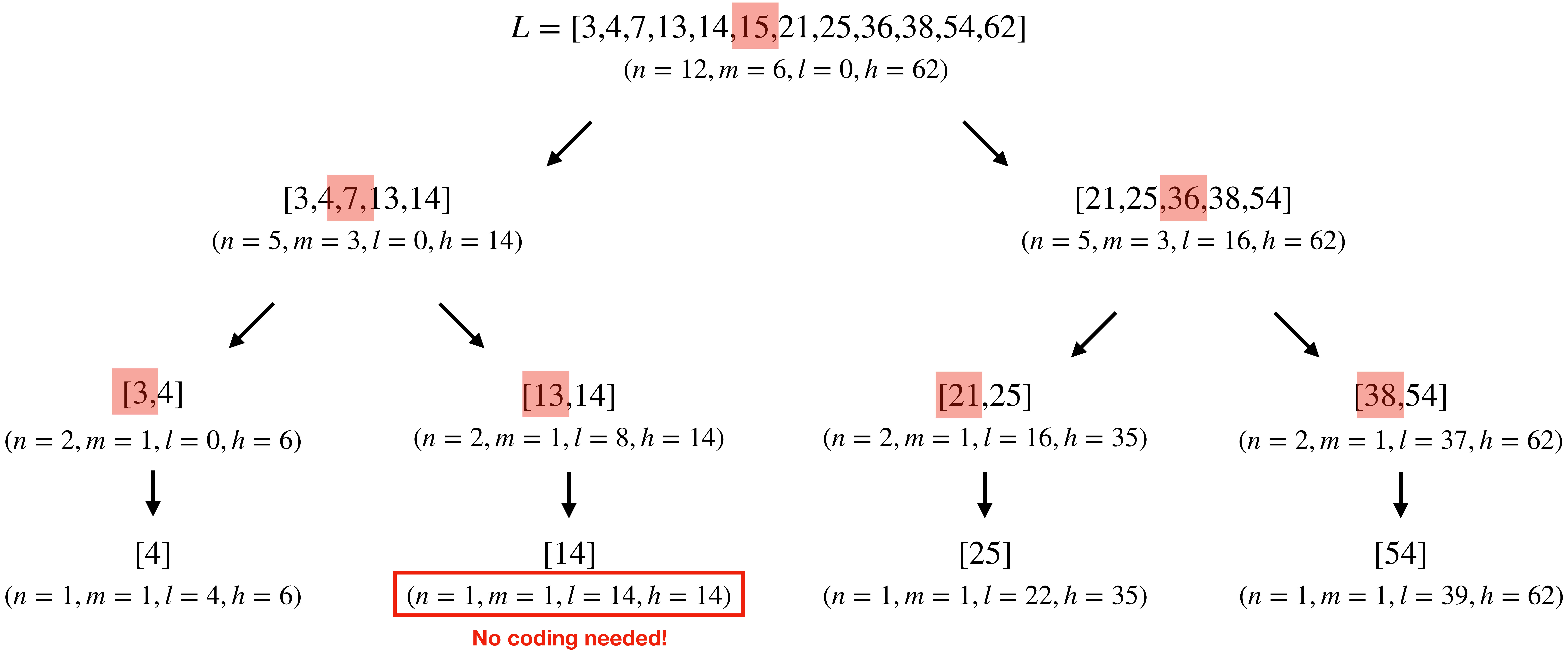
$(n = 1, m = 1, l = 14, h = 14)$

$[25]$

$(n = 1, m = 1, l = 22, h = 35)$

No coding needed!

Binary Interpolative Coding — Example



$L = [3,4,7,13,14,15,21,25,36,38,54,62]$

$(n = 12, m = 6, l = 0, h = 62)$

$[3,4,7,13,14]$

$(n = 5, m = 3, l = 0, h = 14)$

$[21,25,36,38,54]$

$(n = 5, m = 3, l = 16, h = 62)$

$[3,4]$

$(n = 2, m = 1, l = 0, h = 6)$

$[13,14]$

$(n = 2, m = 1, l = 8, h = 14)$

$[21,25]$

$(n = 2, m = 1, l = 16, h = 35)$

$[38,54]$

$(n = 2, m = 1, l = 37, h = 62)$

$[4]$

$(n = 1, m = 1, l = 4, h = 6)$

$[14]$

$\boxed{(n = 1, m = 1, l = 14, h = 14)}$

No coding needed!

$[25]$

$(n = 1, m = 1, l = 22, h = 35)$

$[54]$

$(n = 1, m = 1, l = 39, h = 62)$

Directly-Addressable

Brisaboa *et al.*, 2013

- This is another representation that supports random Access.
- The idea is to choose a value of b and represent each integer x in L with chunks of $b+1$ bits: b bits encode a part of the codeword of x, the extra control bit indicates whether another chunk is necessary or not.
- So if $r = \lceil x/2^b \rceil$ is the number of chunks for x, then x is encoded with $b+r$ bits.
- All the n b-bit chunks are concatenated together in a codeword stream C_1, and all the control bits into a bit-vector B_1 of n bits whose i-th bit is 1 if the representation of $L[i]$ is followed by another chunk or `0` otherwise.
  Then repeat for all the integers that have 2 or more chunks.
- With Rank_1 queries on the bit-vectors B_i is possible to support random Access.

Example for $L = [2,7,12,5,13,142,61,129]$ and $b = 3$.

	2	7	12	5	13	142	61	129
C_1 =	010	111	100	101	101	110	011	001
B_1 =	0	0	1	0	1	1	1	1
C_2 =			001		001	001	110	000
B_2 =			0		0	1	0	1
C_3 =						010		010
B_3 =						0		0

Directly-Addressable

Brisaboa *et al.*, 2013

- This is another representation that supports random Access.
- The idea is to choose a value of b and represent each integer x in L with chunks of $b+1$ bits: b bits encode a part of the codeword of x, the extra control bit indicates whether another chunk is necessary or not.
- So if $r = \lceil x/2^b \rceil$ is the number of chunks for x, then x is encoded with $b + r$ bits.
- All the n b-bit chunks are concatenated together in a codeword stream C_1, and all the control bits into a bit-vector B_1 of n bits whose i-th bit is 1 if the representation of $L[i]$ is followed by another chunk or `0` otherwise.
  Then repeat for all the integers that have 2 or more chunks.
- With Rank_1 queries on the bit-vectors B_i is possible to support random Access.

$L[5] = ?$

Example for $L = [2,7,12,5,13,142,61,129]$
and $b = 3$.

	2	7	12	5	13	142	61	129
C_1 =	010	111	100	101	101	110	011	001
B_1 =	0	0	1	0	1	1	1	1
C_2 =			001		001	001	110	000
B_2 =			0		0	1	0	1
C_3 =						010		010
B_3 =						0		0

Directly-Addressable

Brisaboa *et al.*, 2013

- This is another representation that supports random Access.
- The idea is to choose a value of b and represent each integer x in L with chunks of $b+1$ bits: b bits encode a part of the codeword of x, the extra control bit indicates whether another chunk is necessary or not.
- So if $r = \lceil x/2^b \rceil$ is the number of chunks for x, then x is encoded with $b+r$ bits.
- All the n b-bit chunks are concatenated together in a codeword stream C_1, and all the control bits into a bit-vector B_1 of n bits whose i-th bit is 1 if the representation of $L[i]$ is followed by another chunk or `0` otherwise.
  Then repeat for all the integers that have 2 or more chunks.
- With Rank$_1$ queries on the bit-vectors B_i is possible to support random Access.

$L[5] = ?$

$C_1[5] = 101$

Example for $L = [2,7,12,5,13,142,61,129]$ and $b = 3$.

	2	7	12	5	13	142	61	129
C_1 =	010	111	100	101	101	110	011	001
B_1 =	0	0	1	0	1	1	1	1
C_2 =			001		001	001	110	000
B_2 =			0		0	1	0	1
C_3 =						010		010
B_3 =						0		0

Directly-Addressable

Brisaboa *et al.*, 2013

- This is another representation that supports random Access.
- The idea is to choose a value of b and represent each integer x in L with chunks of $b+1$ bits: b bits encode a part of the codeword of x, the extra control bit indicates whether another chunk is necessary or not.
- So if $r = \lceil x/2^b \rceil$ is the number of chunks for x, then x is encoded with $b+r$ bits.
- All the n b-bit chunks are concatenated together in a codeword stream C_1, and all the control bits into a bit-vector B_1 of n bits whose i-th bit is 1 if the representation of $L[i]$ is followed by another chunk or `0` otherwise.
  Then repeat for all the integers that have 2 or more chunks.
- With Rank$_1$ queries on the bit-vectors B_i is possible to support random Access.

$L[5] = ?$

$C_1[5] = 101$

Since $B_1[5] = 1$, then continue: Rank$_1(B_1{,}5) = 2$

Example for $L = [2{,}7{,}12{,}5{,}13{,}142{,}61{,}129]$ and $b = 3$.

	2	7	12	5	13	142	61	129
C_1 =	010	111	100	101	101	110	011	001
B_1 =	0	0	1	0	1	1	1	1
C_2 =			001		001	001	110	000
B_2 =			0		0	1	0	1
C_3 =						010		010
B_3 =						0		0

Directly-Addressable

Brisaboa *et al.*, 2013

- This is another representation that supports random Access.
- The idea is to choose a value of b and represent each integer x in L with chunks of $b + 1$ bits: b bits encode a part of the codeword of x, the extra control bit indicates whether another chunk is necessary or not.
- So if $r = \lceil x/2^b \rceil$ is the number of chunks for x, then x is encoded with $b + r$ bits.
- All the n b-bit chunks are concatenated together in a codeword stream C_1, and all the control bits into a bit-vector B_1 of n bits whose i-th bit is 1 if the representation of $L[i]$ is followed by another chunk or `0` otherwise.
  Then repeat for all the integers that have 2 or more chunks.
- With Rank$_1$ queries on the bit-vectors B_i is possible to support random Access.

$L[5] = ?$

$C_1[5] = 101$

Since $B_1[5] = 1$, then continue: Rank$_1(B_1{,}5) = 2$

$C_2[2] = 001$

Example for $L = [2{,}7{,}12{,}5{,}13{,}142{,}61{,}129]$ and $b = 3$.

		2	7	12	5	13	142	61	129
C_1	=	010	111	100	101	101	110	011	001
B_1	=	0	0	1	0	1	1	1	1
C_2	=			001		001	001	110	000
B_2	=			0		0	1	0	1
C_3	=						010		010
B_3	=						0		0

Directly-Addressable

Brisaboa *et al.*, 2013

- This is another representation that supports random Access.
- The idea is to choose a value of b and represent each integer x in L with chunks of $b + 1$ bits: b bits encode a part of the codeword of x, the extra control bit indicates whether another chunk is necessary or not.
- So if $r = \lceil x/2^b \rceil$ is the number of chunks for x, then x is encoded with $b + r$ bits.
- All the n b-bit chunks are concatenated together in a codeword stream C_1, and all the control bits into a bit-vector B_1 of n bits whose i-th bit is 1 if the representation of $L[i]$ is followed by another chunk or `0` otherwise.
  Then repeat for all the integers that have 2 or more chunks.
- With Rank$_1$ queries on the bit-vectors B_i is possible to support random Access.

$L[5] = ?$

$C_1[5] = 101$

Since $B_1[5] = 1$, then continue: Rank$_1(B_1, 5) = 2$

$C_2[2] = 001$

Since $B_2[2] = 0$, then we have to stop and return 001.101

Example for $L = [2,7,12,5,13,142,61,129]$ and $b = 3$.

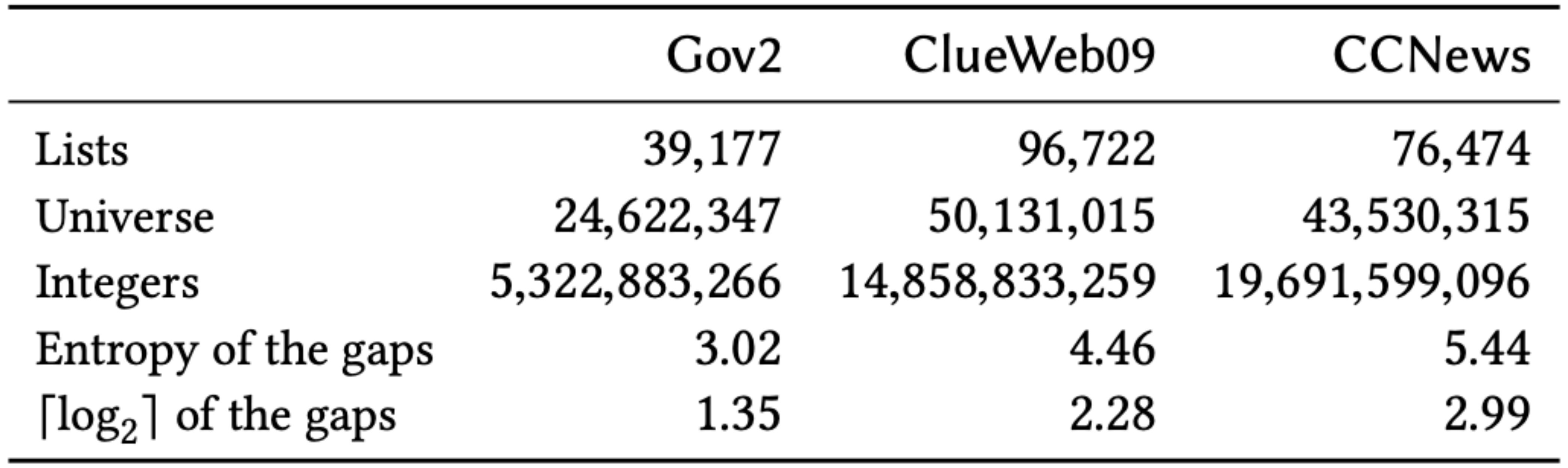
		2	7	12	5	13	142	61	129
C_1	=	010	111	100	101	101	110	011	001
B_1	=	0	0	1	0	1	1	1	1
C_2	=			001		001	001	110	000
B_2	=			0		0	1	0	1
C_3	=						010		010
B_3	=						0		0

Hybrid Approaches

- Representing the blocks of the list L with different compressors gives different space/time trade-offs.
- Usually, using the simple characteristic bit-vector of a block whenever it is dense, and another encoder E (like Elias-Fano or Variable-Byte) when it is sparse, gives better performance (high compression ratio and faster queries) than using E alone.

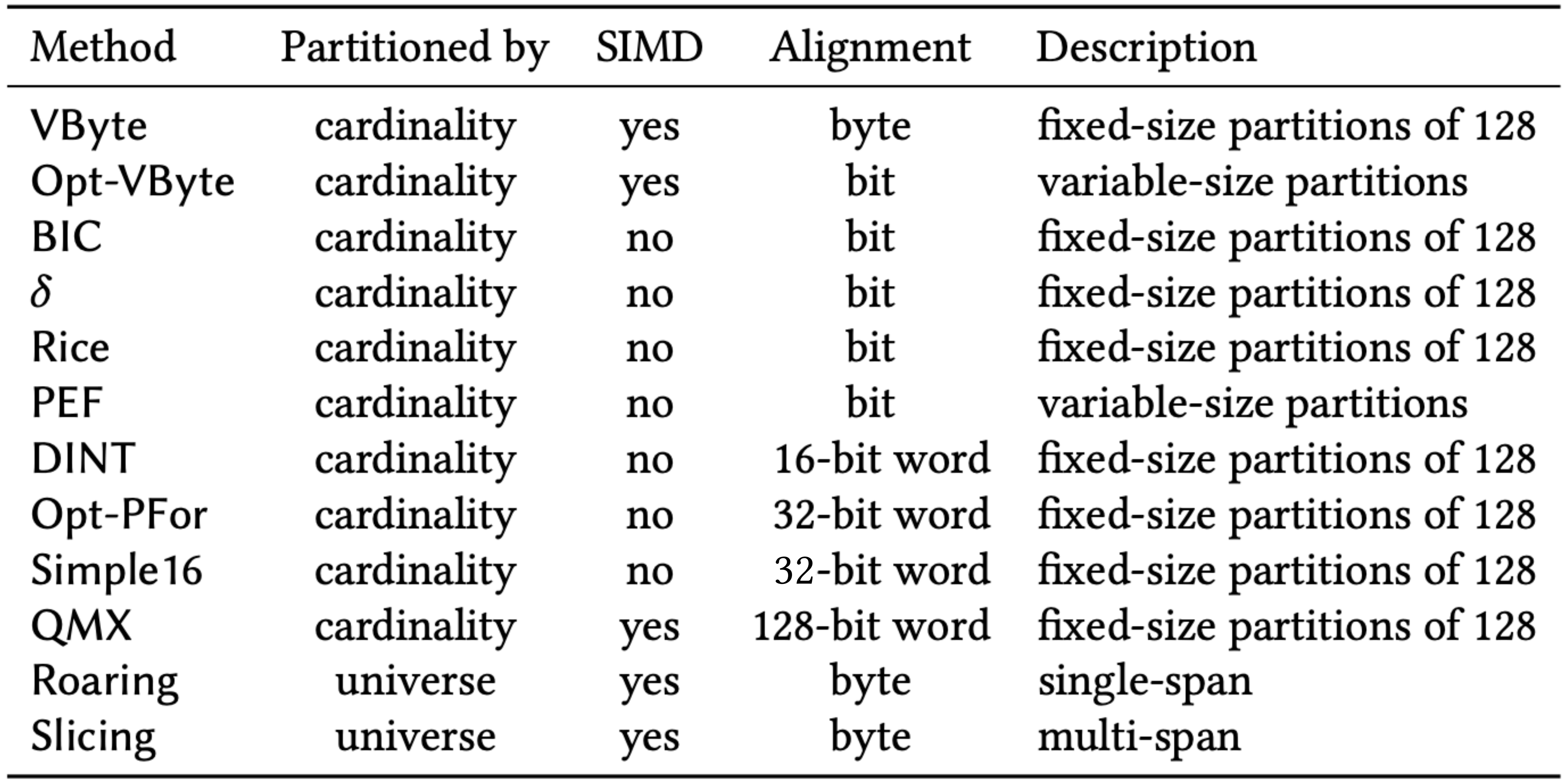
Performance — Datasets and Methods

Datasets



	Gov2	ClueWeb09	CCNews
Lists	39,177	96,722	76,474
Universe	24,622,347	50,131,015	43,530,315
Integers	5,322,883,266	14,858,833,259	19,691,599,096
Entropy of the gaps	3.02	4.46	5.44
$\lceil \log_2 \rceil$ of the gaps	1.35	2.28	2.99

Methods

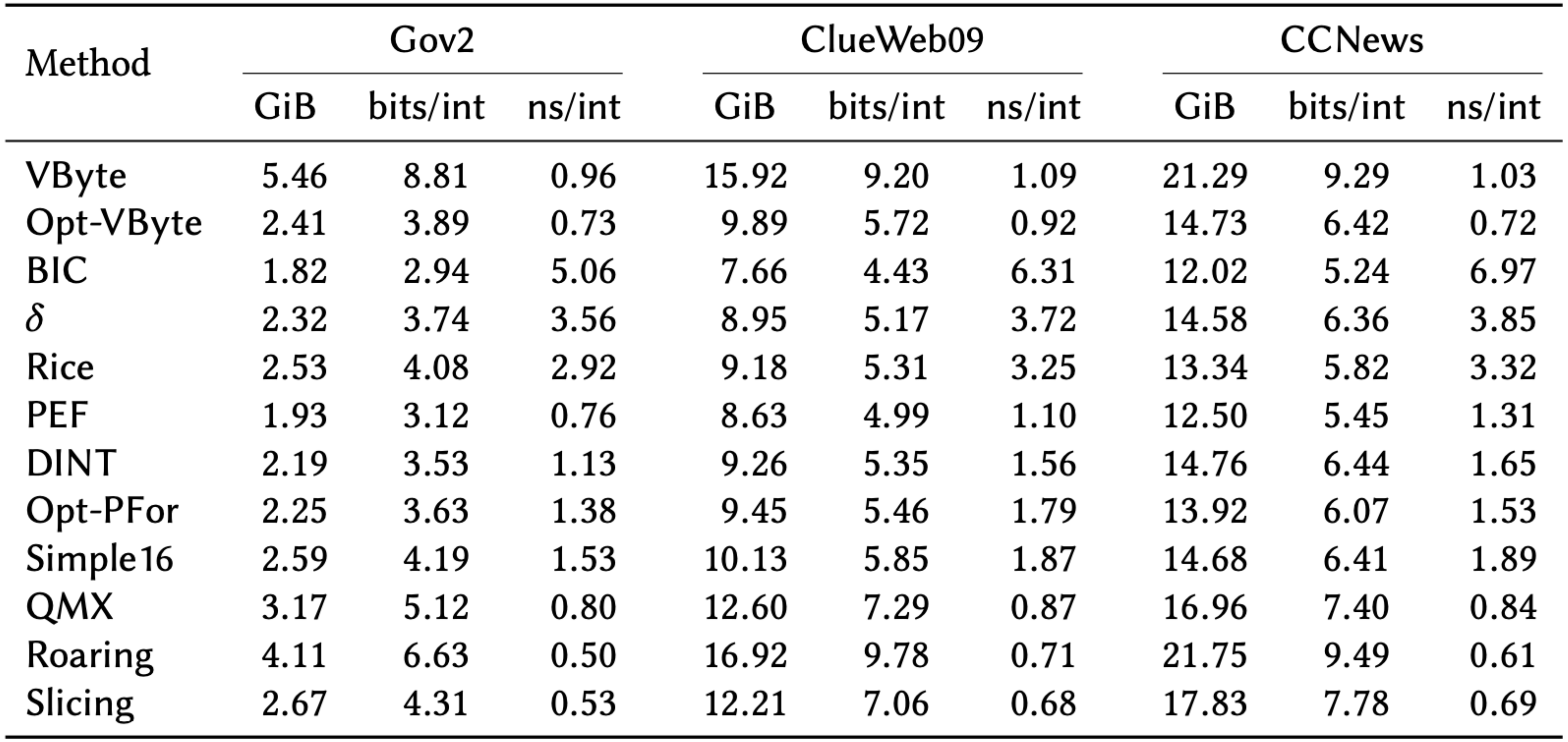


Method	Partitioned by	SIMD	Alignment	Description
VByte	cardinality	yes	byte	fixed-size partitions of 128
Opt-VByte	cardinality	yes	bit	variable-size partitions
BIC	cardinality	no	bit	fixed-size partitions of 128
δ	cardinality	no	bit	fixed-size partitions of 128
Rice	cardinality	no	bit	fixed-size partitions of 128
PEF	cardinality	no	bit	variable-size partitions
DINT	cardinality	no	16-bit word	fixed-size partitions of 128
Opt-PFor	cardinality	no	32-bit word	fixed-size partitions of 128
Simple16	cardinality	no	32-bit word	fixed-size partitions of 128
QMX	cardinality	yes	128-bit word	fixed-size partitions of 128
Roaring	universe	yes	byte	single-span
Slicing	universe	yes	byte	multi-span

Notes.

- Opt-VByte: hybrid approach between unary codes and VByte
- BIC: Binary Interpolative Coding
- PEF: Partitioned Elias-Fano
- QMX: Simple with 128-bit words
- Roaring: Elias-Fano partitioned by universe with 2 levels
- Slicing: Elias-Fano partitioned by universe with 3 levels

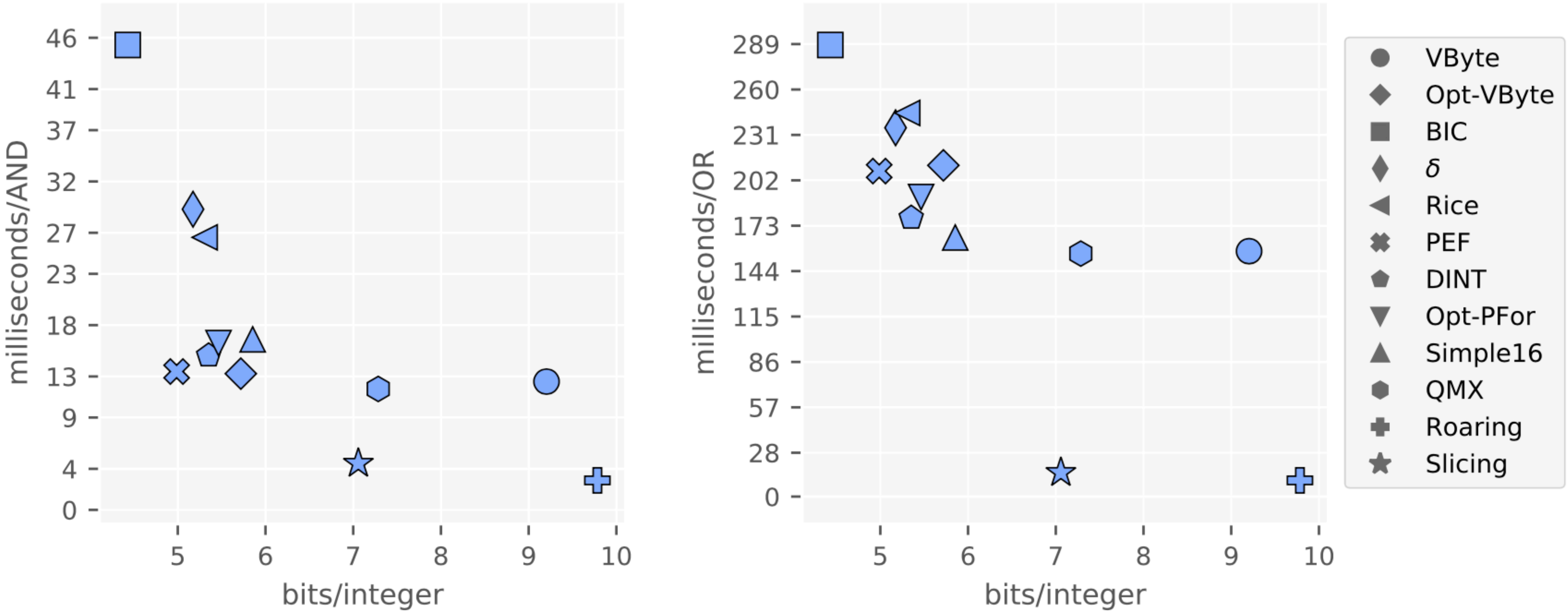
Performance — Space and Decoding



Method	Gov2			ClueWeb09			CCNews		
	GiB	bits/int	ns/int	GiB	bits/int	ns/int	GiB	bits/int	ns/int
VByte	5.46	8.81	0.96	15.92	9.20	1.09	21.29	9.29	1.03
Opt-VByte	2.41	3.89	0.73	9.89	5.72	0.92	14.73	6.42	0.72
BIC	1.82	2.94	5.06	7.66	4.43	6.31	12.02	5.24	6.97
δ	2.32	3.74	3.56	8.95	5.17	3.72	14.58	6.36	3.85
Rice	2.53	4.08	2.92	9.18	5.31	3.25	13.34	5.82	3.32
PEF	1.93	3.12	0.76	8.63	4.99	1.10	12.50	5.45	1.31
DINT	2.19	3.53	1.13	9.26	5.35	1.56	14.76	6.44	1.65
Opt-PFor	2.25	3.63	1.38	9.45	5.46	1.79	13.92	6.07	1.53
Simple16	2.59	4.19	1.53	10.13	5.85	1.87	14.68	6.41	1.89
QMX	3.17	5.12	0.80	12.60	7.29	0.87	16.96	7.40	0.84
Roaring	4.11	6.63	0.50	16.92	9.78	0.71	21.75	9.49	0.61
Slicing	2.67	4.31	0.53	12.21	7.06	0.68	17.83	7.78	0.69

- BIC and PEF are best for space; VByte and Roaring are worst for space.
- BIC is worst for decoding; VByte and Roaring are best for decoding.
- PEF gains its speed thanks to the efficient decoding of dense bit-vectors.
- All the other methods offer trade-offs between these two extremes.

Performance — Space vs. AND/OR Queries (on ClueWeb09)



- Partitioning by universe allows much faster queries.
- However, the gap in performance between methods partitioned by universe and cardinality diminishes when intersection involves more lists.
- For methods partitioned by cardinality: the performance of intersections (AND) is related to that of Succ; the performance of union (OR) is related to that of decoding.

Further Readings

- Section 3 (except 3.8) and 6 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

- Sections 4.2 and 4.3 (about Rank & Select) of:
  Gonzalo Navarro. 2016. *Compact Data Structures*. Cambridge University Press, ISBN 978-1-107-15238-0.