DATA20021

University of Helsinki, Department of Computer Science

Information Retrieval

Lecture 3: Index Compression

Simon J. Puglisi

puglisi@cs.helsinki.fi

Spring 2020

This Week

16.1: Introduction to Indexing
- Boolean Retrieval model
- Inverted Indexes

21.1: Index Compression
- unary, gamma, variable-byte coding
- (Partitioned) Elias-Fano coding (used by Google, facebook)

23.1: Index Construction
- preprocessing documents prior to search (stemming, et c.)
- building the index efficiently

28.1: Web Crawling
- large scale web search engine architecture

30.1: Query Processing
- scoring and ranking search results
- Vector-Space model

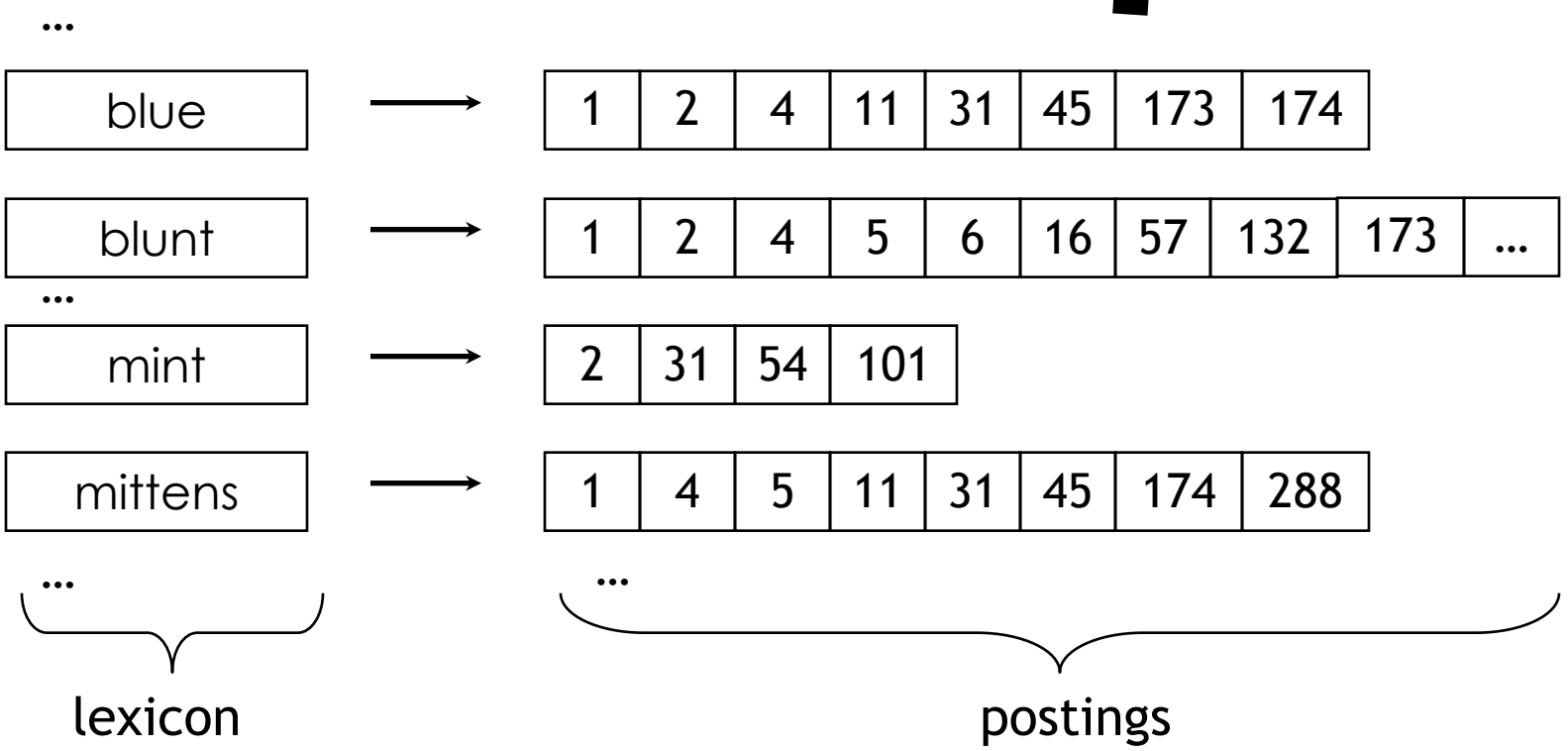
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast.
  - This is true of the decompression algorithms we will use.

Inverted Indexes

- Crawl the web, gather a collection of documents
- For each word t in the collection, store a list of all documents containing t:
- **Query:** blue mittens

Inverted Index



Boolean Retrieval with an Inverted Index

- **Query:** blue mittens

blue	→	1	2	4	11	31	45	173	174

1	4	11	31	45	174

Intersect lists

mittens	→	1	4	5	11	31	45	174	288

Why does Google use compression?

- We can compress both components of an inverted index
- Today: techniques for compressing the lists
  - Lists much bigger than lexicon (factor 10 at least)
  - (stuff for lexicon compression later in the course)
- Motivation: if we don't compress the lists we'll have to store them on disk because they're enormous
- Compression allows us to keep them (or most of them) in memory. If decompressing in memory is faster than reading from disk we get a saving. This is the case with int codes.
- Even if we still have to store some lists on disk, storing them compressed means we can read them into memory faster at intersection time

- A key idea for compressing inverted lists is to observe that elements of *L* are monotonically increasing, so we can transform the list by taking differences (gaps):

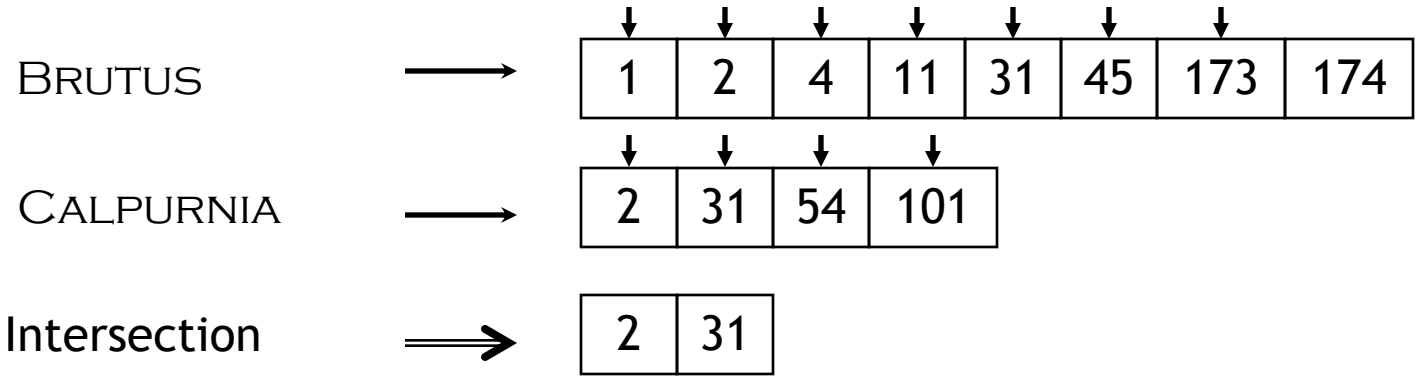
  L = 3,7,11,23,29,37,41,…

  D(L) = 3,4,4,12,6,8,4,…

- The advantage of doing this is to make the integers smaller, and suitable for integer codes
  - We will then assign smaller integers smaller codes

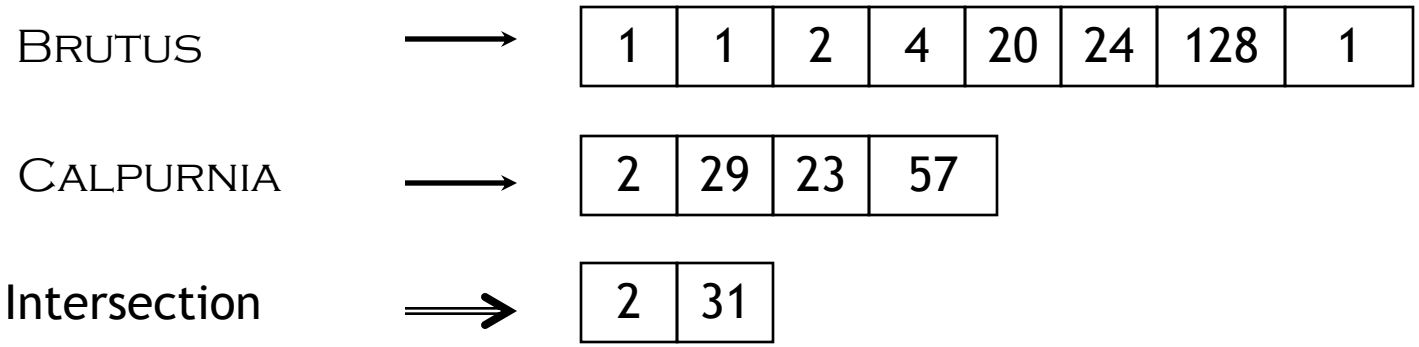
- We can easily "degap" the list during intersection
  - Adding items together as we scan the lists adds little overhead

Simple Conjunctive Query (two terms – no gaps)



- This is linear in the length of the postings lists: O(p+q).
- This only works if postings lists are sorted.

Simple Conjunctive Query (two terms – gaps)



- As we scan each list during intersect work out the original value of each item by keeping track of the sum of the gaps

Gap Encoding

		inverted list								
the	docIDs	…		283042		283043		283044		283045	…
	gaps	…			1		1		1		…
computer	docIDs	…		283047		283154		283159		283202	…
	gaps	…			107		5		43		…
arachnid	docIDs	252000		500100							
	gaps	252000	248100								

Variable Length Encoding

What we want:

- For arachnid and other rare terms: use about 20 bits per gap.
- For the and other very frequent terms: use about 1 bit per gap.

In order to implement this, we need to devise some form of variable length encoding.

- Use few bits for small gaps, many bits for large gaps.

Integer representations…

Plain binary representations of integers

- $\text{binary}_k(n)$ = binary representation of an integer n using k-bits.

- $\text{binary}_4(13) = 1101$, $\text{binary}_7(13) = 0001101$

- Idea for encoding integers: just use minimal binary codes!
  - After all, that's small…

Just use binary representations…?

- 283154, 5, 43, 3
- 10001010010000010010, 101, 101011, 11
- 1000101001000010010101101011111
  - Uhhh…
- We have to be able to decompress as well!
  - Our coding scheme must be prefix free
  - We need a prefix code

Unary codes

- Our first prefix code for ints is the *unary* code

- Represents integer n as n ‘0’s with a final ‘1’.
  - So the unary code for integer n has length n+1.
  - (Note: we could reverse the roles of 0s and 1s – that is called a *negated* unary code)

- Unary code for 0 is 1
- Unary code for 3 is 0001
- Unary code for 40 is 0000000000000000000000000000000000000001

Elias codes…

Elias γ codes

- Our first non-trivial integer code is the γ (gamma) code, discovered by Peter Elias in 1975.
- Represent an integer n > 0 as a pair of selector and offset.
- Offset is the integer in binary, with the leading bit chopped off.
  - For example 13 → 1101 → 101
- Selector is the length of offset
  - For 13 (offset 101), this is 3.
  - Encode selector in unary code: 0001.
- Gamma code of 13 is the concatenation of selector & offset:
  - 0001, 101 = 0001101.

Elias γ codes

- Thinking back, the problem with our earlier “brain wave” of just using the minimal binary representations was that, later, we didn’t know the length of each code

  4, 43, 3 became 10110101111

- γ codes solve that problem by prefixing the minimal binary code with an unambiguous representation of the length of the minimal binary codes (a unary representation)

  000110000000101011001111

- But there is also a further optimization present in gamma codes…

Elias γ codes

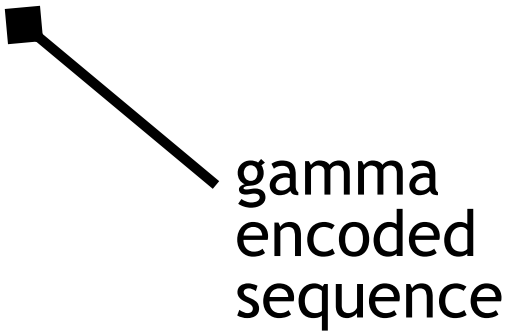
- …the most significant bit of the minimal binary code is *always* 1. So we don't need to store it.

  0001100000001101011001111

  0001100000000110101100111

- And so now we can reduce the selectors by one bit

  001010000010101101 1



Decoding a Gamma code

- The γ code for 13 is 0001101
  - How do we decode this?

- Read 1's until we hit a 0
  - 0001 = 3
  - We have the selector.
  - This tells us the number of bits in the offset.

- Read 3 more bits
  - We have the offset (101)
  - Put a 1 on the front of the offset and we have the original number
  - 1101 = 13

Decoding a Gamma code

- What are the numbers in this γ-encoded sequence?
  - 0001111000001101010100100

Length of a Gamma code

- The length of offset is $\lfloor \log_2 n \rfloor$ bits.
- The length of selector is $\lfloor \log_2 n \rfloor + 1$ bits,
- So the length of the entire code is $2\lfloor \log_2 n \rfloor + 1$ bits.
  - So *γ* codes are always of odd length.
- *γ* codes are just over twice the size of the minimal binary code

Variable byte (vbyte) codes…

Variable byte (VB) code

- Developed (for IR) at RMIT – Scholer et al., 2002 (?)
- Used by many commercial/research systems
- Blend of variable-length coding and sensitivity to memory alignment (they respect byte boundaries)
- Input: array of integers (gaps)
- Output: array of bytes

Variable byte (VB) code

- Dedicate 1 bit in each byte we output (high bit) to be a continuation bit c.
  - 00000000
- If the gap G fits within 7 bits, binary-encode it in the 7 available bits and set c = 0.
  - Eg. Gap of 29 = 11101 fits in 7 bits so we output: 00011101
  - Eg. Gap of 117 = 1110101 fits in 7 bits so we output: 01110101
- Else: set c = 1, encode lower-order 7 bits and then use additional bytes to encode the higher order bits using same algorithm.
  - Eg. Gap of 767 = 1011100101 > 7 bits so:
  - Put lower 7 bits (1100101) in first byte and set the c bit: 11100101
  - We now have the bits 101 to deal with, < 7 bits so output 00000101
- At the end, the continuation bit of the last byte is 0 (c = 0) and the other bytes is 1 (c = 1). So we know when we have decoded a gap!

Variable byte (VB) code

- Another example:
- 214577
- 11010000110001100001
- 0110001 → 10110001
- 0001100 → 10001100
- 1101 → 00001101

- 10110001
- 10001100
- 00001101
- 00011010000110001100001 = 214577

VB code examples

docIDs	824	829	215406
gaps	824	5	214577
in binary	1100111000	101	1101000110 00110001
VB codes	10111000 00000110	00000101	10110001 10001101 00001100

VB encoded list:

10111000000001100000010110110001100011010000110010111000000001100000010110110001100011010000110

VB code encoding algorithm

```
VBEncodeNumber(n)
1       bytes ← {}, i ← 0
2       while n >= 128 do
3               bytes[i] ← 128 + (n mod 128)
6               n ← n div 128
7               i ← i + 1
8       end while
9       bytes[i] ← n
0       return bytes
```

VB code decoding algorithm

```
VBDecodeNumber(bytestream)
1       n ← 0, i ← 0
2       d ← 1
3       while bytestream[i] ≥ 128 do
4               n ← n + d * (bytestream[i] - 128)
5               d ← d * 128
6               i ← i + 1
7       end while
8       n ← n + d * bytestream[i]
9       return n
```

Other variable length codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc
- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those
- Recent work on word-aligned codes that efficiently “pack” a variable number of gaps into one word
- See Ahn & Moffat 2005; Lemire et al. 2018

Compression of GOV2 collection

data structure	size
collection (text + xml, etc)	426.0 GB
collection (text)	23.0 GB
T/D incidence matrix	192.5 GB
documents	25 M
lexicon (terms)	35 M
postings, 32-bit per entry	22.5 GB
postings, 25-bits per entry	17.2 GB
postings, gamma	8.5 GB
postings, vbyte encoded	10 GB

Document reordering,…

Digression: clustering in postings lists

- When document ids are clustered inside a postings list we get small gaps values that vbyte and gamma codes can compress well
- Document ids can become clustered inside postings lists in at least two ways…
- Firstly, it can be *natural*
  - e.g., When a term is trending (inside Twitter's inverted index)
- Secondly, we can try to *induce it* before building index: give documents containing similar words similar ids
  - Common heuristic: order the documents by URL
  - Within a domain (say helsinki.fi, ford.com) documents use a similar vocabulary

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

Document identifier mapping

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

Document identifier mapping

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

Reordered inverted lists

L1: 1 2 3 4 6 L2: 3 4 7 8 9 L3: 2 3 4 5

Document identifier mapping

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

Reordered inverted lists

L1: 1 2 3 4 6 L2: 3 4 7 8 9 L3: 2 3 4 5

New gap values

L1: 1 1 1 2 L2: 1 3 1 1 L3: 1 1 1

Document identifier mapping

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

larger gaps, less compressible

Reordered inverted lists

L1: 1 2 3 4 6 L2: 3 4 7 8 9 L3: 2 3 4 5

New gap values

L1: 1 1 1 2 L2: 1 3 1 1 L3: 1 1 1

Document identifier mapping

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

larger gaps, less compressible

Reordered inverted lists

L1: 1 2 3 4 6 L2: 3 4 7 8 9 L3: 2 3 4 5

New gap values

L1: 1 1 1 2 L2: 1 3 1 1 L3: 1 1 1

smaller gaps, better compressible

Document identifier mapping

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original inverted lists

L1: 1 3 6 8 9 L2: 2 4 5 6 9 L3: 3 6 7 9

Original gap values

L1: 2 3 2 1 L2: 2 1 1 3 L3: 3 1 2

larger gaps, less compressible

Reordered inverted lists

L1: 1 2 3 4 6 L2: 3 4 7 8 9

New gap values

L1: 1 1 1 2 L2: 1 3 1 1 L3: 1 1 1

smaller gaps, better compressible

~20% smaller lists on GOV2 collection

Compressing Graphs and Indexes with Recursive Graph Bisection

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Igor Kabiljo
Facebook
ikabiljo@fb.com

Brian Karrer
Facebook
briankarrer@fb.com

Giuseppe Ottaviano
Facebook
ott@fb.com

Sergey Pupyrev
Facebook
spupyrev@fb.com

Alon Shalita
Facebook
alon@fb.com

ABSTRACT

Graph reordering is a powerful technique to increase the locality of the representations of graphs, which can be helpful in several applications. We study how the technique can be used to improve compression of graphs and inverted indexes.

We extend the recent theoretical model of Chierichetti et al. (KDD 2009) for graph compression, and show how it can be employed for compression-friendly reordering of social networks and web graphs and for assigning document identifiers in inverted indexes. We design and implement a novel theoretically sound reordering algorithm that is based on recursive graph bisection.

Our experiments show a significant improvement of the compression rate of graph and indexes over existing heuristics. The new method is relatively simple and allows efficient

and results in a higher compression ratio. We stress that the success of applying a particular encoding algorithm strongly depends on the distribution of gaps in an adjacency list: a sequence of small and regular gaps is more compressible than a sequence of large and random ones.

This observation has motivated the approach of assigning identifiers in a way that optimizes compression. *Graph reordering* has been successfully applied for social networks [7, 12]. In that scenario, placing similar social actors nearby in the resulting order yields a significant compression improvement. Similarly, lexicographic locality is utilized for compressing the Web graph: when pages are ordered by URL, proximal pages have similar sets of neighbors, which results in an increased compression ratio of the graph, when compared with the compression obtained using the original

Elias-Fano encoding,…

Elias-Fano representation

Input is an array of n increasing non-negative integers (e.g., a postings list) and an upperbound on them, z:

$$0 \leq x_0 < x_1 < x_2 \ldots < x_{n-2} < x_{n-1} \leq z$$

We will represent the sequence in two bit arrays:

- the lower $l = \max\{0, \lfloor \log(z/n) \rfloor\}$ bits of each x_i are stored explicitly and contiguously in the lower-bits array
- the upper bits are stored in the upper-bits array as a sequence of unary-coded gaps

- E.g., list: 5, 8, 9, 15, 32 with upperbound 36
  - $l = \max\{0, \lfloor \log(36/5) \rfloor\}$ = 2 bits

5 8 9 15 32

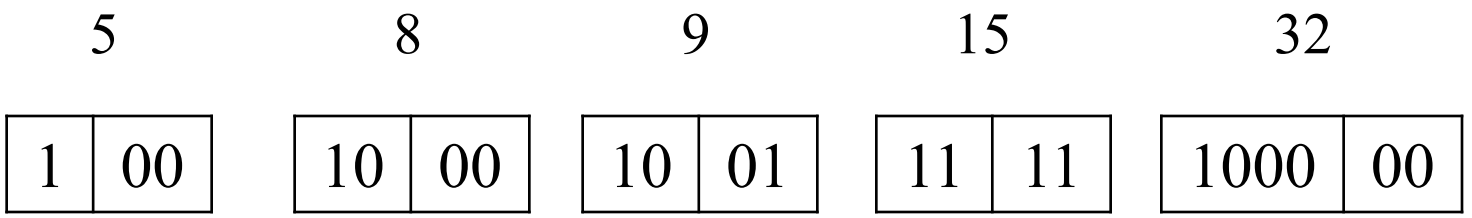
$l = 2, z = 36$

5 8 9 15 32

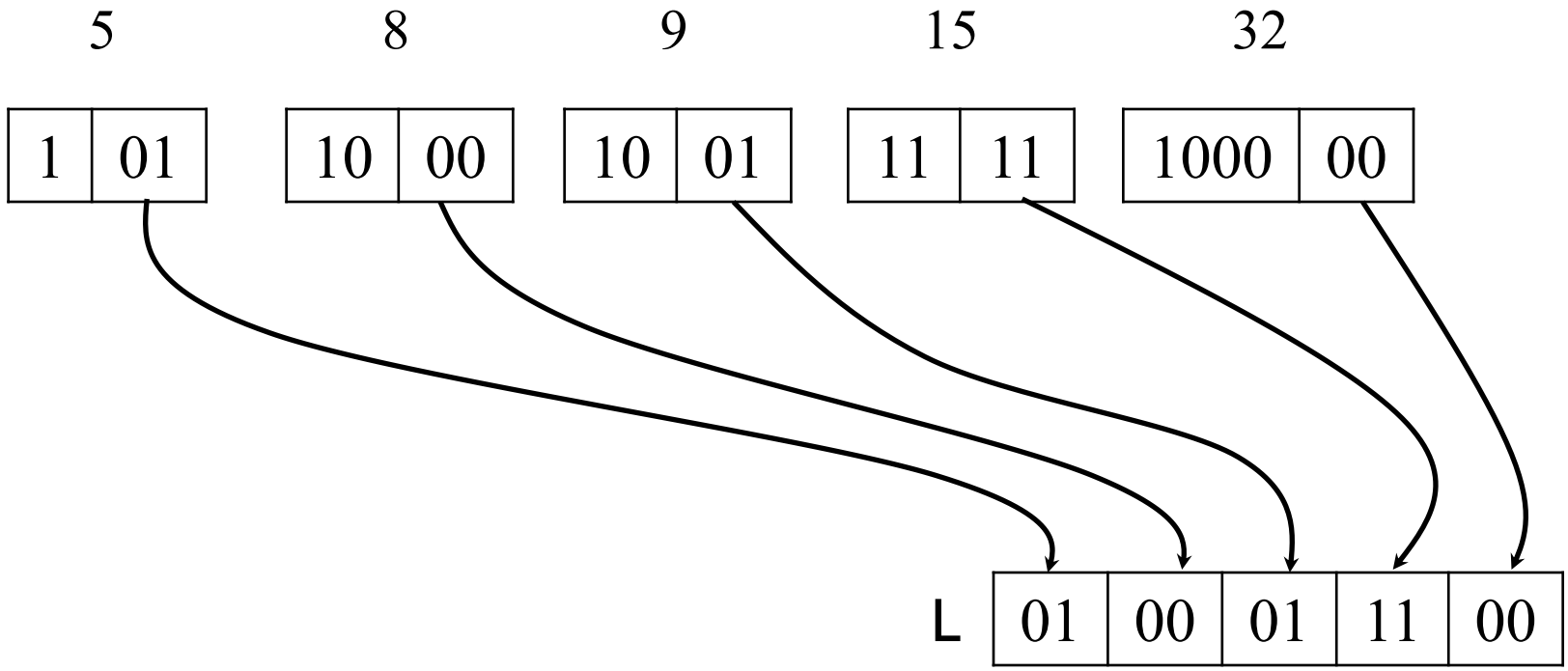
$l = 2, z = 36$

5	8	9	15	32
1 00	10 00	10 01	11 11	1000 00

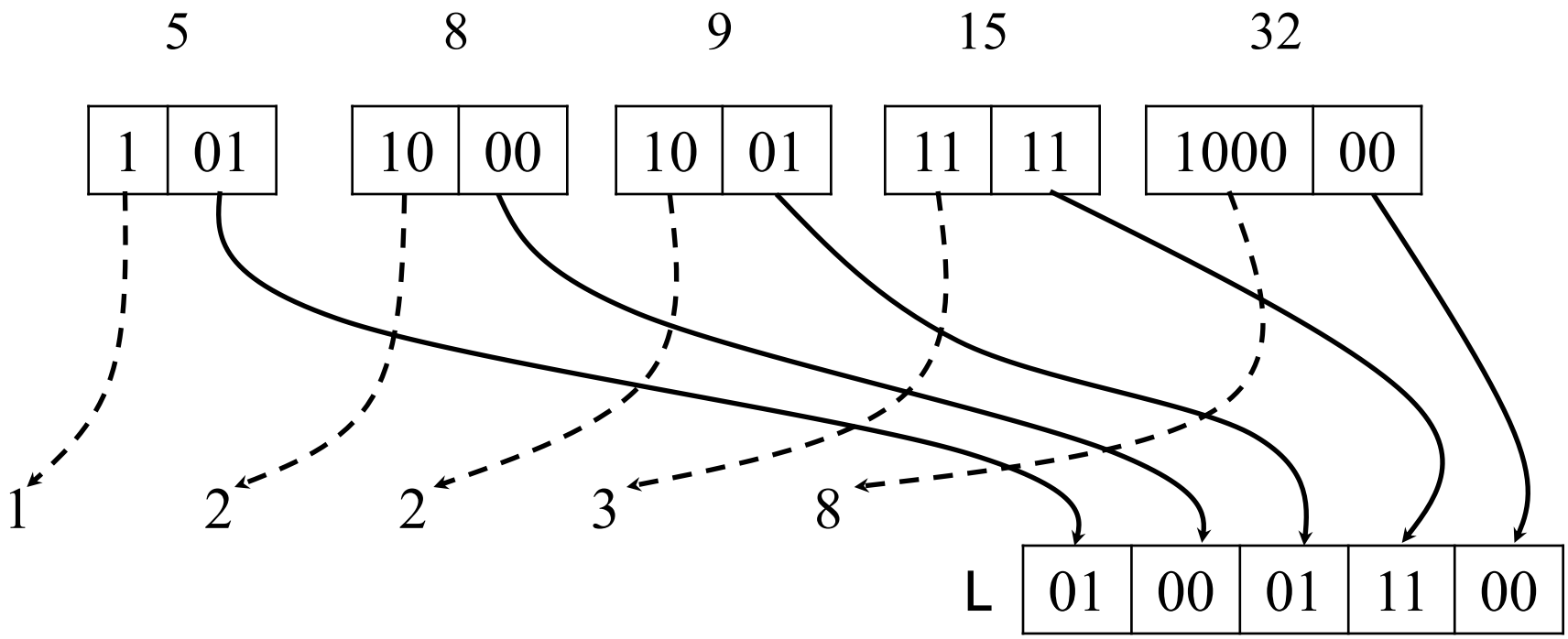
$l = 2, z = 36$



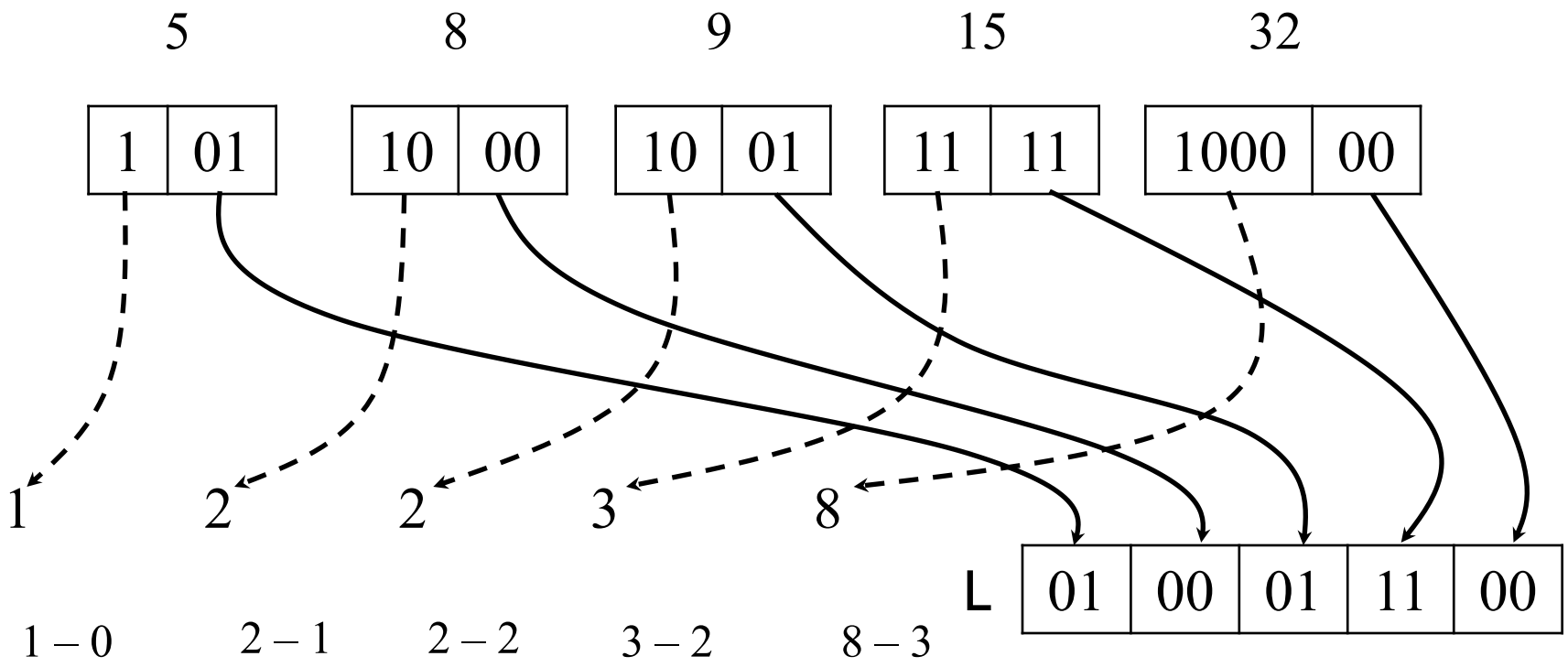
$l = 2, z = 36$

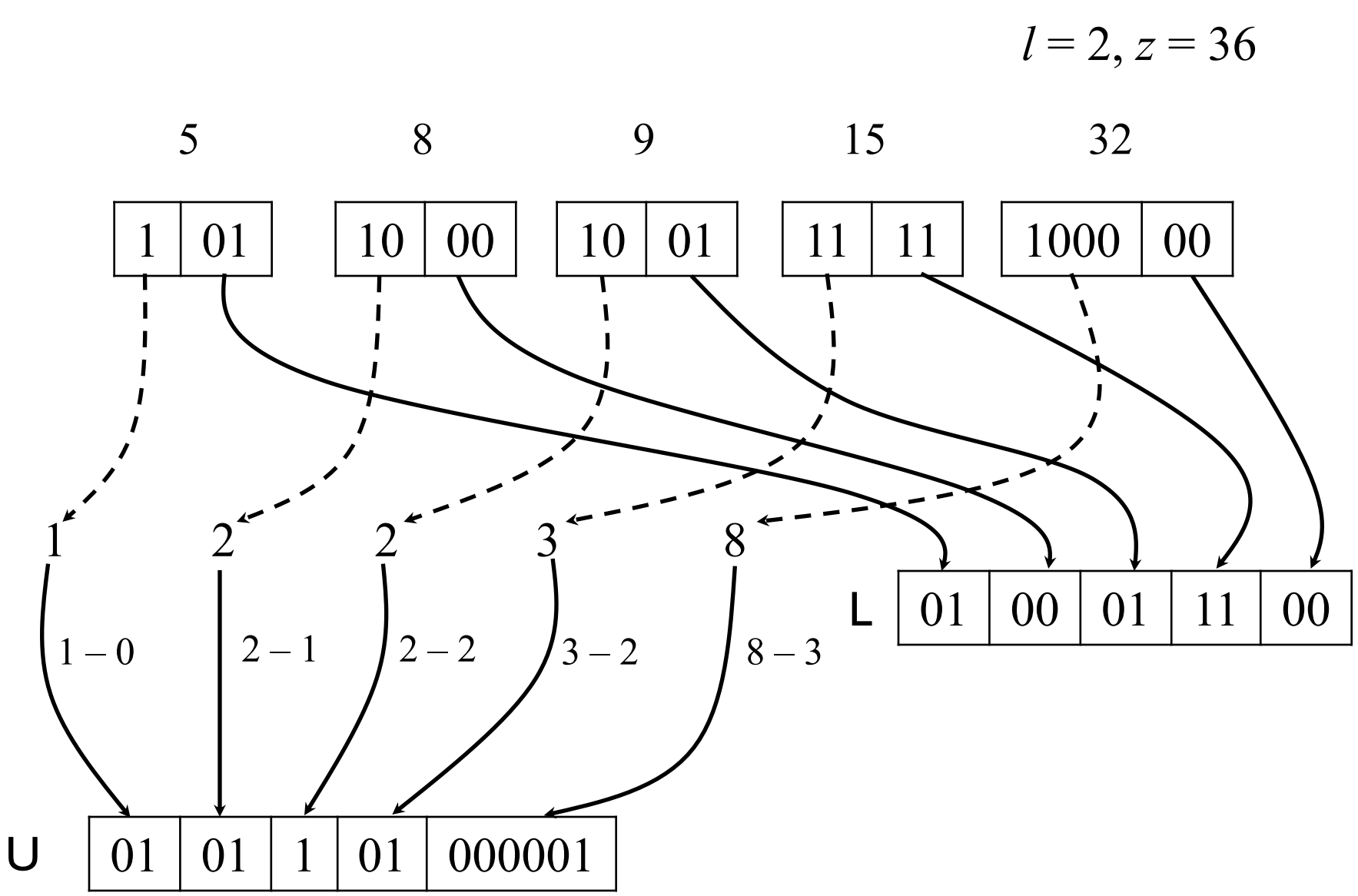


$l = 2, z = 36$



$l = 2, z = 36$

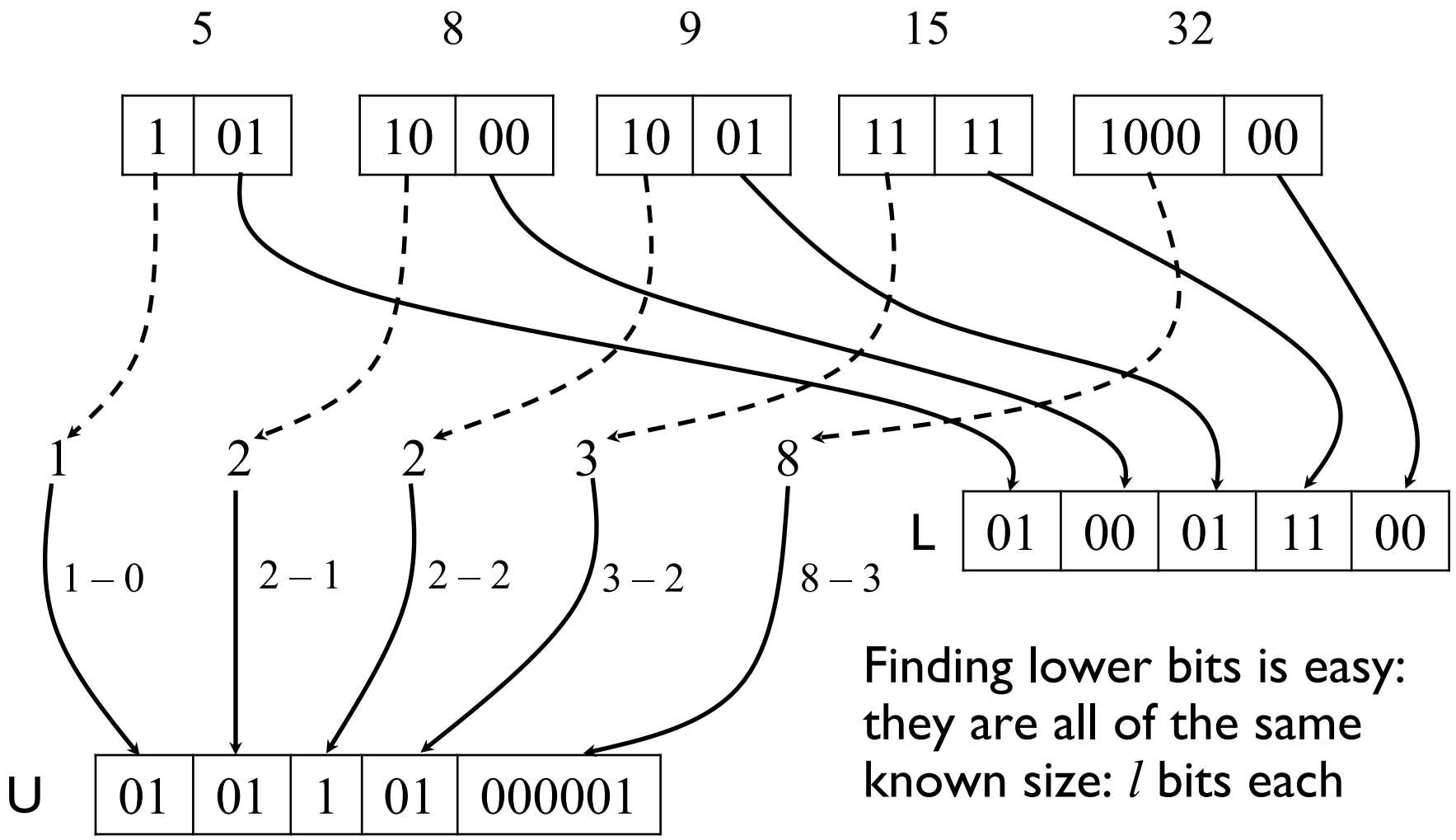

5
8
9
15
32
1
01
10
00
10
01
11
11
1000
00
1
2
2
3
8
L
01
00
01
11
00
1 – 0
2 – 1
2 – 2
3 – 2
8 – 3


$l = 2, z = 36$
5
8
9
15
32
1 01
10 00
10 01
11 11
1000 00
1
2
2
3
8
1 – 0
2 – 1
2 – 2
3 – 2
8 – 3
L
01 00 01 11 00
U
01 01 1 01 000001

Size of the Elias-Fano representation

- We have n increasing integers drawn from a universe of size z
- L contains n items, each of $l = \lfloor \log(z/n) \rfloor$ bits:
  - $n \lfloor \log(z/n) \rfloor$ bits
- U contains at most n 1 bits
- The number of 0s is at most $z/2^{\lfloor \log(z/n) \rfloor} \leq 2n$ bits
- Overall $\leq 2n + n\lfloor \log(z/n) \rfloor$ bits

access(i): getting the ith integer in the sequence

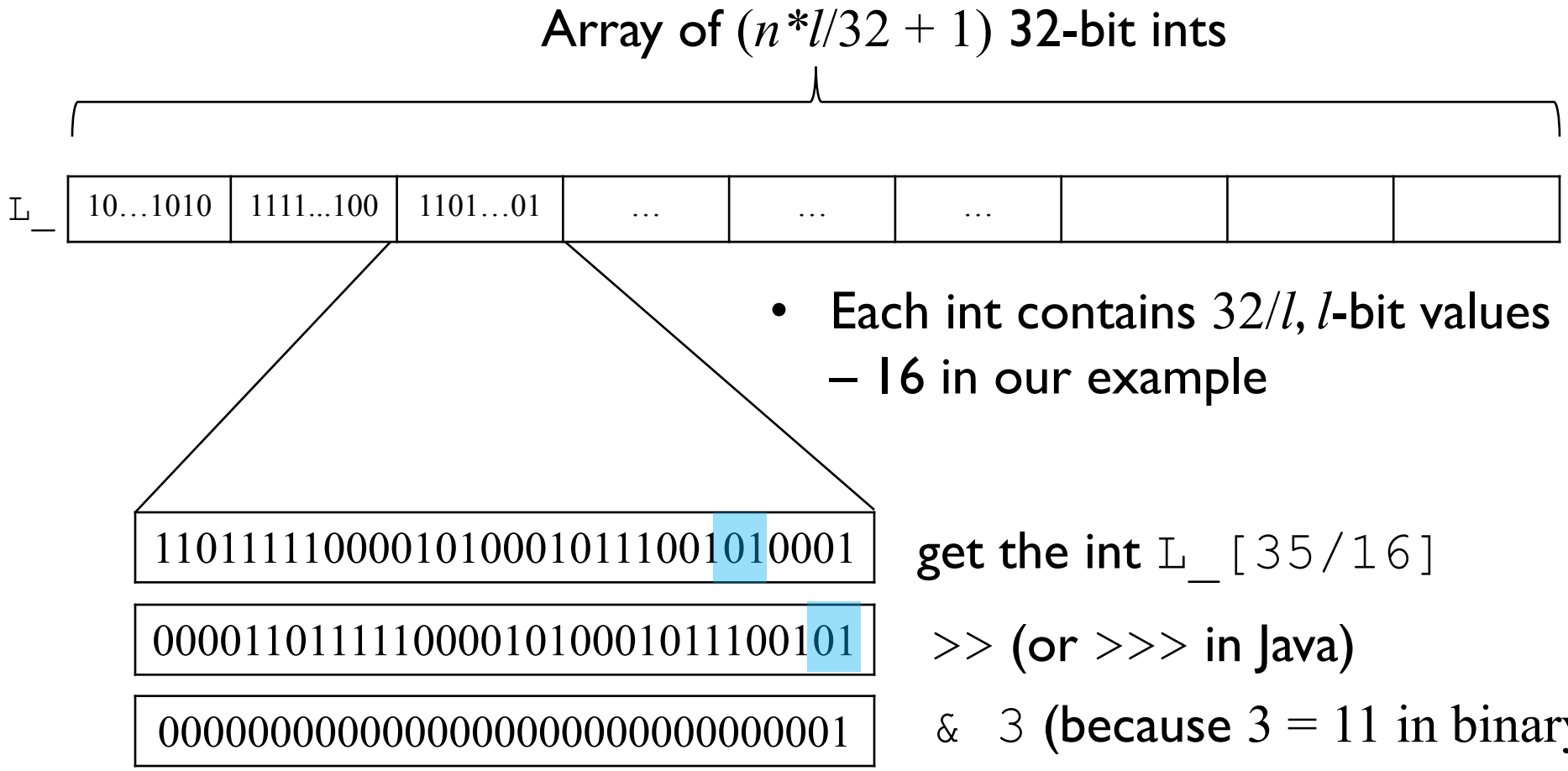


Finding lower bits is easy: they are all of the same known size: l bits each

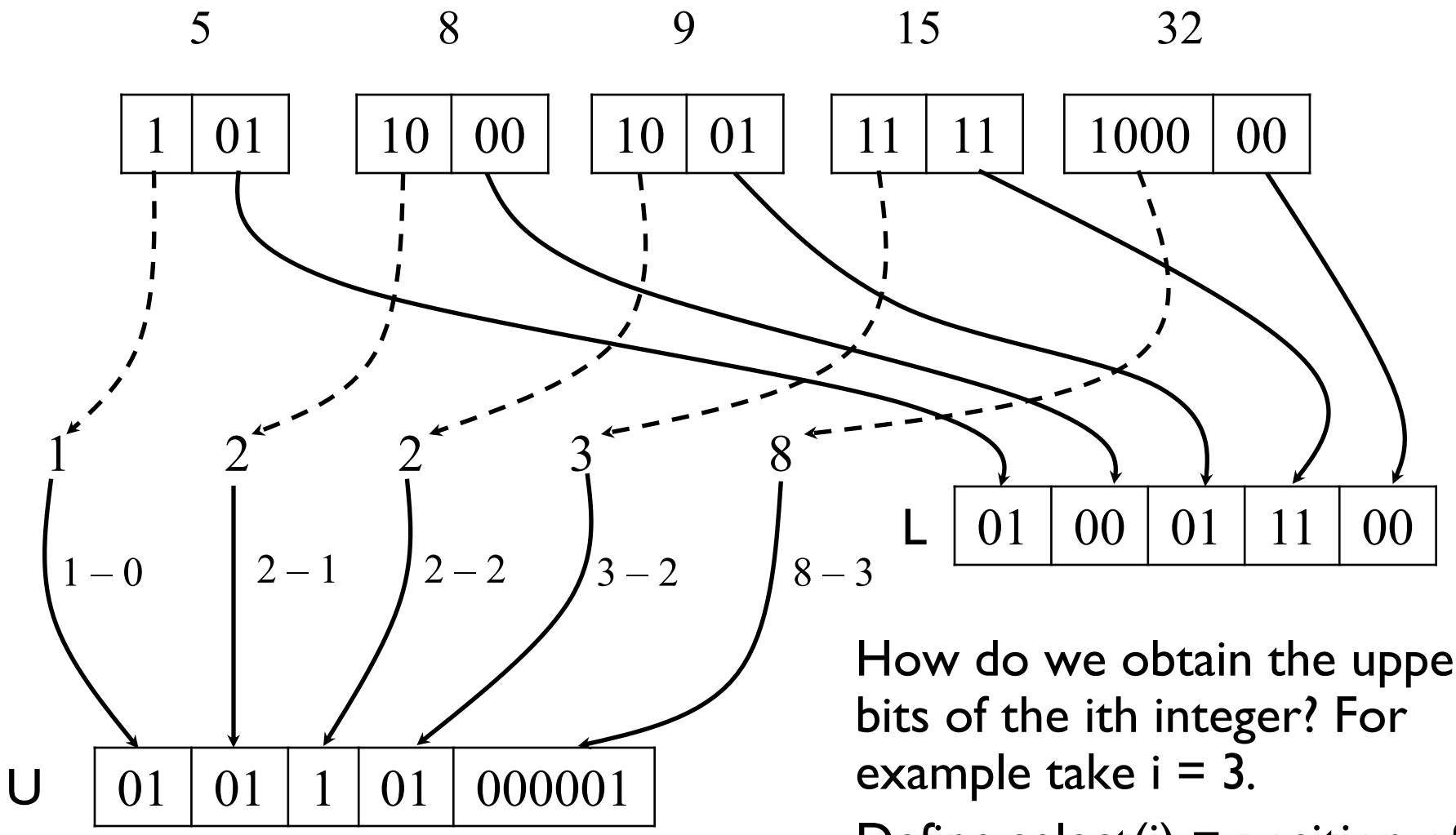
Lower bits for i^{th} value are L[$li..li+l$-1]

Implementing L…

- Let's think for a minute about how we might actually implement L in (close to) nl bits
  - (assuming l is a factor of 32, and therefore a power of 2)



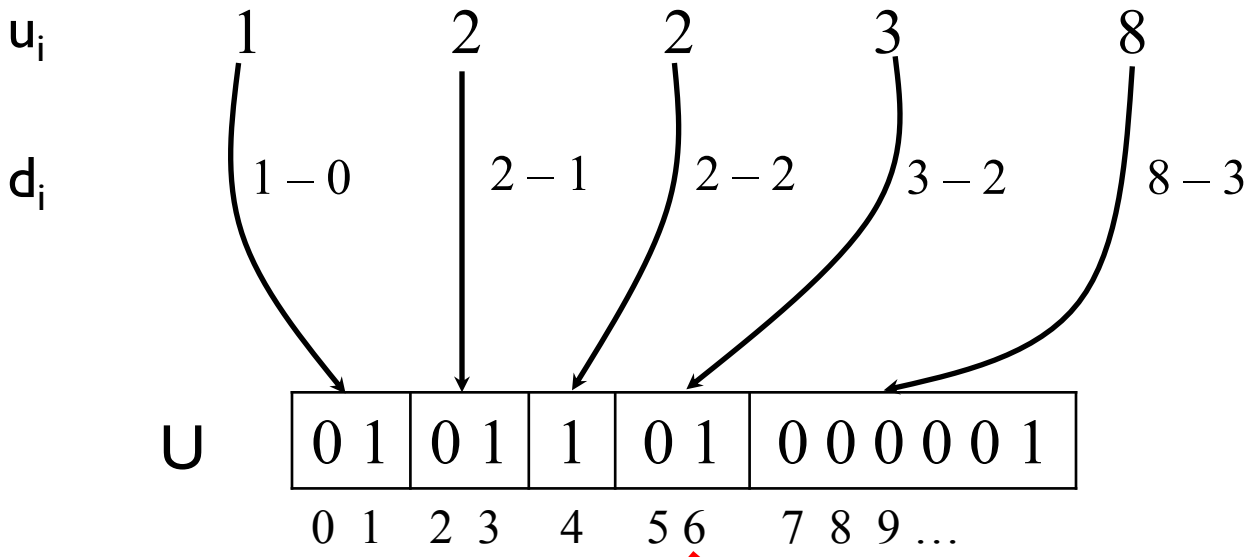
access(i): getting the ith integer in the sequence



How do we obtain the upper bits of the ith integer? For example take i = 3.

Define select(i) = position of ith 1 in a bitvector

Finding upper bits of i^{th} value

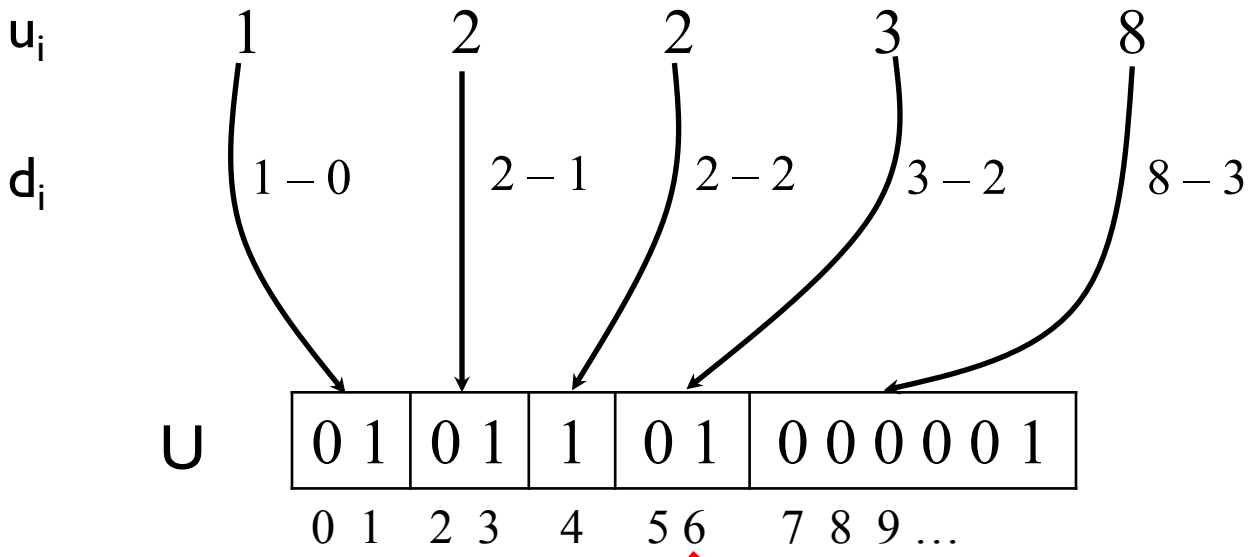


$select_B(i)$ = position of i^{th} '1' in a bitvector B (counting from 0)

$select_U(3) = 6$

$select_U(i)$ gives us the end of difference d_i

Finding upper bits of i^{th} value



$select_B(i)$ = position of i^{th} '1' in a bitvector B (counting from 0)

$select_U(3) = 6$

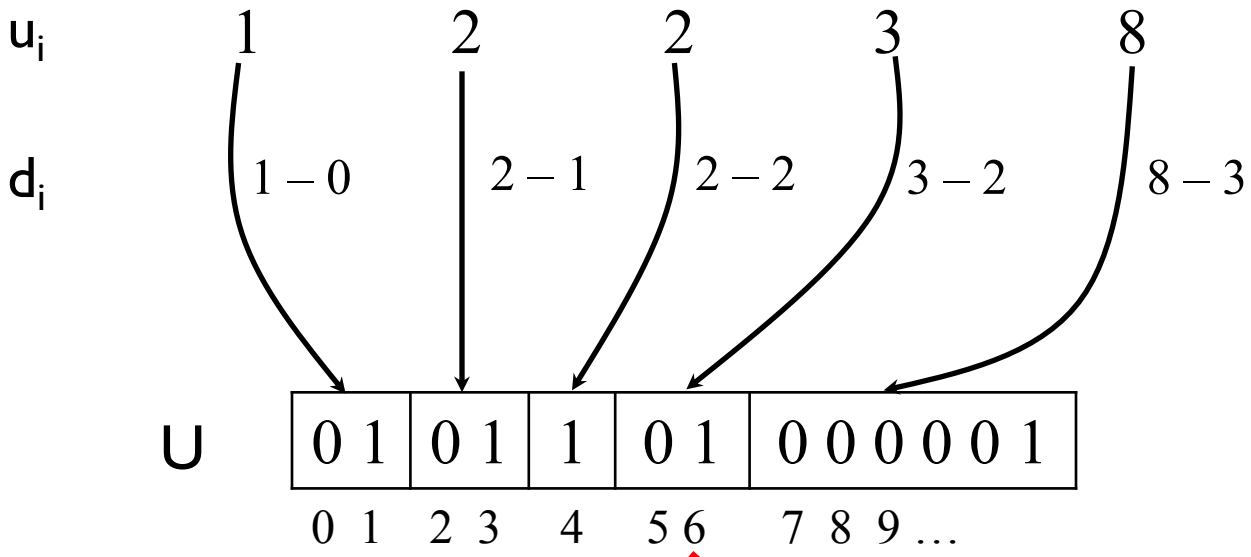
$select_U(i)$ gives us the end of difference d_i

Claim: value of upper bits u_i for ith value is: $select_U(i) - i$

$u_3 = select_U(3) - 3 = 6 - 3 = 3$

$u_4 = select_U(4) - 4 = 12 - 4 = 8$

Finding upper bits of i^{th} value



$select_B(i)$ = position of i^{th} '1' in a bitvector B (counting from 0)

$select_U(3) = 6$

$select_U(i)$ gives us the end of difference d_i

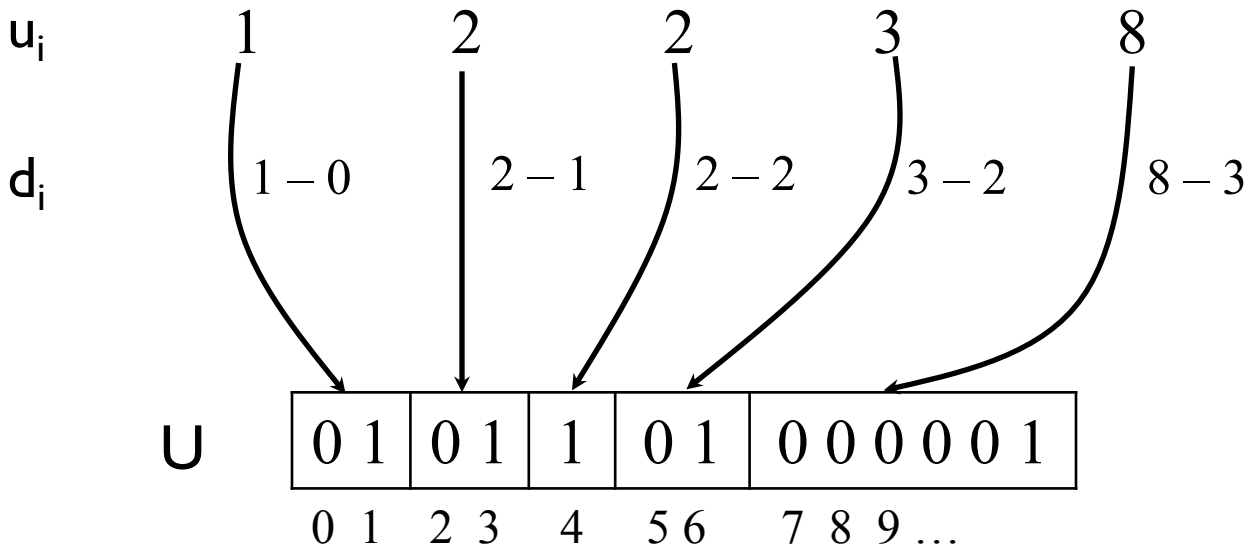
Claim: value of upper bits u_i for ith value is: $select_U(i) - i$

$u_3 = select_U(3) - 3 = 6 - 3 = 3$

$u_4 = select_U(4) - 4 = 12 - 4 = 8$

Why?

Finding upper bits of i^{th} value



We have stored (in U) <u>differences</u> between the upper-bits' values

u_i = sum of differences $d_0 + d_1 + \ldots d_i$.

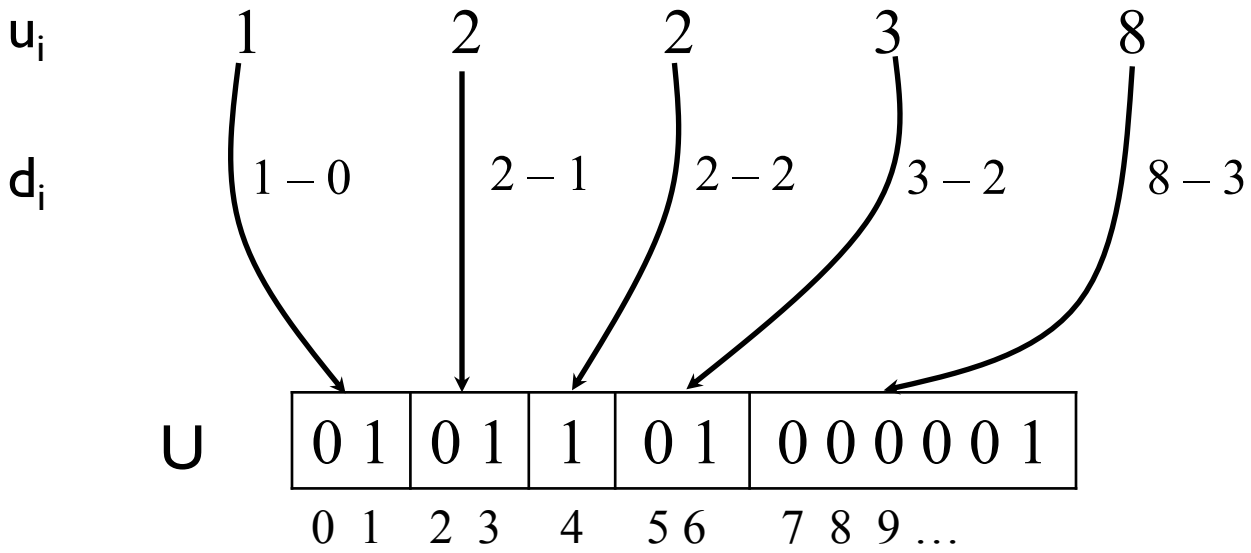
Differences encoded in unary, i.e. d_i: d_i 0s and one 1

sum of differences up to i = num of 0s up to select(i)

sum of 0s up to select(i) = select(i) – (num of 1s up to select(i))

= i, by definition

access(i): summary



- So provided we have select(i), we can access the upper bits
  - The lower bits are easy, as we've seen
- Because we want to access the original integers in left-to-right order during intersection, we ask select(1), then select(2), et c.
- We answer this sequence of queries by scanning U, always looking for the next 1 bit (the answer to our next select(i))
  - Determining the position of the next 1 in a bit array can be done very fast in practice with word-level tricks

nextGEq(i): finding the next item $\geq i$

- Having the ability to do access(i) on the Elias-Fano representation allows us to implement list intersection
- However, Elias-Fano allows us to easily implement a more powerful operation that can allow us to skip more items in the list during intersection…
- Specifically, nextGEq(i) returns the first (i.e. smallest) item in the list that is $\geq i$ – this is the next possible element that can be part of the resulting intersection
- This will be on the next Exercise Sheet (with some hints ;-)

- Elias-Fano combines strong theoretical guarantees and excellent practical performance
- It is used inside Google's search engine
- It's also used inside Facebook's graph search facility (and probably also inside their text search engine)
- There are also good public implementations:
  - https://github.com/ot/ds2i
  - https://github.com/vigna/sux/tree/master/sux

Partitioned Elias-Fano encoding,…

- Elias-Fano indexes can sometimes be significantly bigger than those obtained via other compression

- This inefficiency is caused by E-F's inability to exploit clustering in the sequence (where we would have small gaps)

- In short: Ottaviano and Venturini solve this problem by partitioning lists into chunks and storing separate E-F data structures for each chunk

Compression of GOV2 Collection

data structure	size
collection (text + xml, etc)	426.0 GB
collection (text)	40.0 GB
T/D incidence matrix	192.5 GB
documents	25 M
lexicon (terms)	35 M
postings, 32-bit per entry	22.5 GB
postings, 25-bits per entry	17.2 GB
postings, gamma	8.5 GB
postings, vbyte encoded	10 GB
postings, Elias-Fano encoded	7.42 GB
postings, Paritioned E-F	4.65 GB

Times for AND queries

- Rough numbers estimated from a few papers
  - gamma 3.0
  - vbyte 2.3
  - Elias-Fano 1.1
  - Partitioned E-F 1.2

- Partitioned E-F 10% than E-F for OR queries

Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient. (Works for other kinds of queries too).
- Only 1-2% of the total size of the collection.
- Only 11-12% of the total size of the text in the collection.
- However, we've ignored positional and frequency information.
- Compression can be applied to these components too.

Next Lecture

16.1: Introduction to Indexing
- Boolean Retrieval model
- Inverted Indexes

21.1: Index Compression
- variable-byte coding
- (Partitioned) Elias-Fano coding (used by Google, facebook)

23.1: Index Construction
- preprocessing documents prior to search (stemming, et c.)
- building the index efficiently

28.1: Web Crawling
- large scale web search engine architecture

30.1: Query Processing
- scoring and ranking search results
- Vector-Space model

Further reading

Anh and Moffat (2005) introduce an number of word-aligned (as opposed to byte-aligned) binary codes for list compression:

Inverted Index Compression Using Word-Aligned Binary Codes.
Information Retrieval 8(1): 151-166 (2005)

Lemire et al. (2018) describe some fast implementations of vbyte:

Stream VByte: Faster byte-oriented integer compression.
Information Processing Letters 130: 1-6 (2018)

Vigna (2013) shows first successful application of Elias-Fano codes to the inverted index

Quasi-succinct indices.
Proceedings of the Conference on Web Search and Data Mining 2013: 83-92

Venturini and Ottaviano improve Vigna's scheme via partitioning

Partitioned Elias-Fano indexes. Proceedings of SIGIR 2014: 273-282

Dhulipala et al. (2016) doc id reordering with recursive graph bisection

Compressing Graphs and Indexes with Recursive Graph Bisection.
Proceedings of KDD 2016: 1535-1544