Quasi-Succinct Indices

or

The Revenge of Elias and Fano

Sebastiano Vigna
Dipartimento di Informatica
Università degli Studi di Milano
Italia

Inverted Indices

- The backbone of search engines (and more)
- Main problem: store a sequence of increasing integers in little space so to be able to enumerate the list / pick the *k*-th integer / skip to the first integer larger than or equal to *b* quickly
- Maps to rank/selection/predecessor search
- For positions the problem is a bit more articulated (and complicated)

The Classical Solution

- Middle 80s/start of 90s (apparently depends on who you talk to)
- Turn the sequence $x_0, x_1, x_2, \ldots$ into *gaps* $x_0, x_1 - x_0, x_2 - x_1, \ldots$
- Hope that the numbers will be small and well (predictably) distributed
- Use some instantaneous code to store the gaps

Lot Of Research

- Zillions of different codes and kinds of codes
- Problem: sequential decoding easy, rank/selection/predecessor *very inefficient*
- Solution: various kind of *skip tables* that make it possible to “jump” in the middle of the gap sequence
- In retrospective, it looks a little bit contrived, doesn’t it?

Why Gaps?

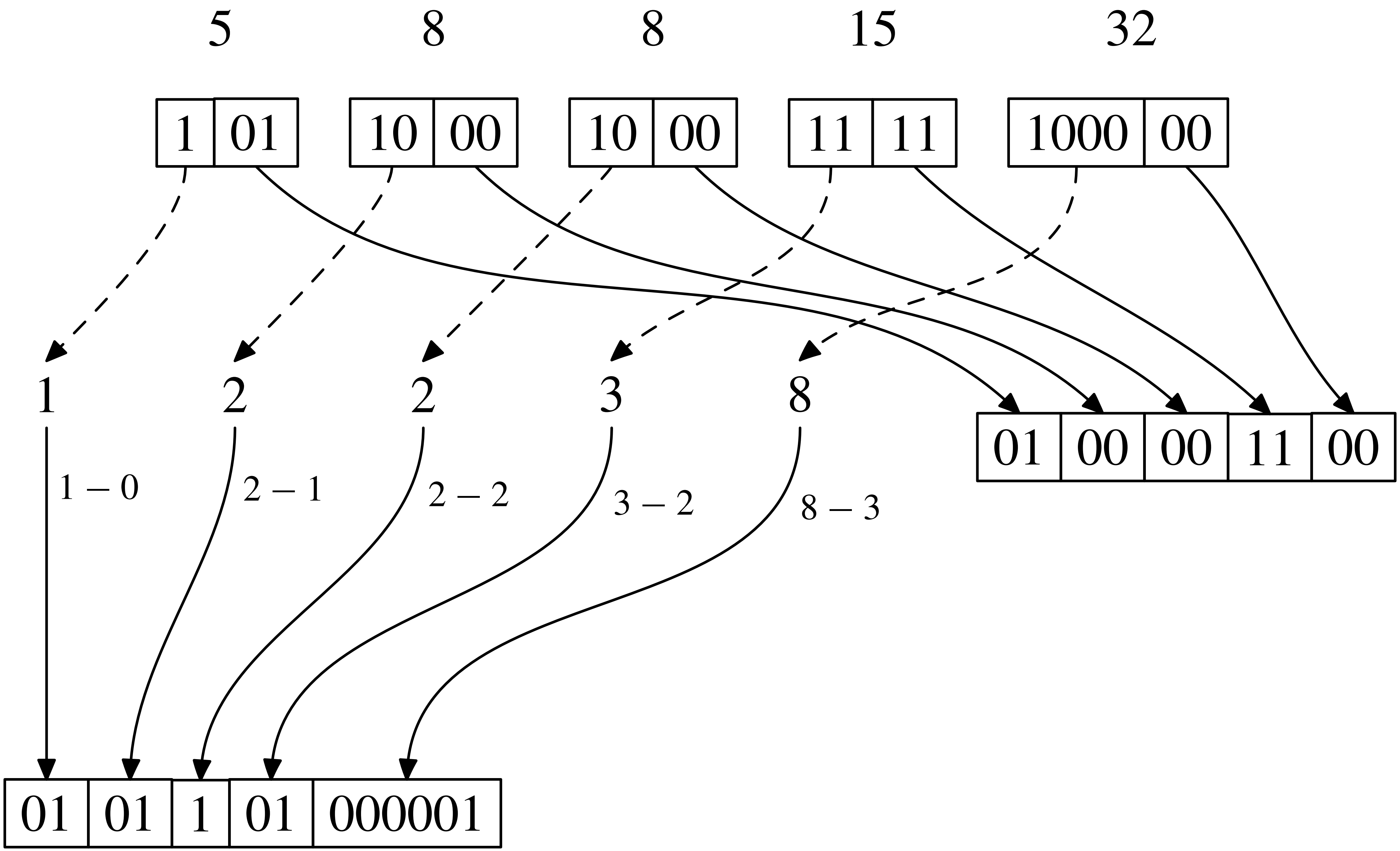
- Maybe we can approach the problem in a completely different way
- Maybe gaps were *not a good idea in the first place*
- Maybe there are nice, efficient ways of store sequences of integers that do not require gaps
- So, back (1975!) to the future (now)!

Elias–Fano Representation

- Elias developed in 1975 a quasi-succinct representation for monotone sequences (JACM); Fano discusses it in a report
- At that time, probably no more than a curiosity
- (My 2€¢: should be taught in the first year of any CS curriculum)
- Inspired several modern succinct data structures

High Bits/Low Bits

- Given n and u we have a monotone sequence $0 \leq x_0, x_1, x_2, \ldots, x_{n-1} \leq u$
- Store the lower $\ell = \log(u / n)$ bits explicitly
- Store the upper bits as a sequence of unary coded gaps (0^k1 represents k)
- We use at most $2 + \log(u / n)$ bits per element
- Close to the succinct bound: quasi-succinct!
- (Less than half a bit away, as Elias proves)



$5, 8, 8, 15, 32 \leq u = 36, \ell = 2$

Advantages

- Almost optimal space usage
- Distribution-free
- Reading sequentially requires very few logical operations (you might be surprised)
- Restrict the rank/selection problem to a nice ~2*n* bits array with half zeroes, half ones
- It's beautiful :-)
- So, what about rank/select?

Looking up (Selection)

- Suppose you want to get the *k*-th element quickly
- Just scan the upper bits, one word at a time, doing population counting (one clock)
- Cost of searching: 100ps/element (yes, that's picoseconds) per element on an i7 @ 3.4GHz
- When you get to the right word, complete sequentially and pick the lower bits

Searching (Successor)

- It's exactly the same: only, you count *zeroes*
- Zeroes tells you how much the *upper bits* are increasing, which is the important thing
- Just skip $b >> \ell$ *upper zeroes* and complete sequentially
- Due to the balance between ones and zeroes, on average always 100ps per element (this must be made more precise, see the paper)

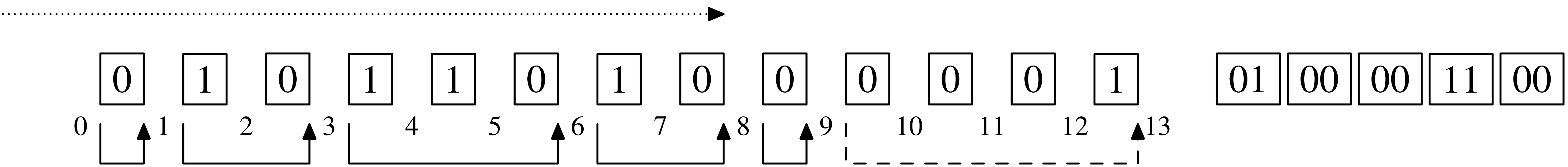
“Complete Sequentially”?

- Not really
- There are *broadword algorithms for selection* (I wrote the first one in 2005; improved later by Simon Gog)
- Fixed number of operations to skip *k* unary codes
- Final phase at ~500ps/element

```
int select_in_word( const uint64_t x, const int k ) {
   uint64_t byte_sums = x - ((x & 0xaaaaaaaaaaaaaaaaULL) >> 1);
   byte_sums = (byte_sums & 0x3333333333333333ULL) + ((byte_sums >> 2)
       & 0x3333333333333333ULL);
   byte_sums = (byte_sums + (byte_sums >> 4)) & 0x0f0f0f0f0f0f0f0fULL;
   byte_sums *= 0x0101010101010101ULL;
   const uint64_t k_step_8 = k * 0x0101010101010101ULL;
   const int place = ((((k_step_8 | 0x8080808080808080ULL) - byte_sums)
      & 0x8080808080808080ULL) >> 7) * 0x0101010101010101ULL >> 53 & ~0x7;
   return place + select_in_byte[x >> place & 0xFF |
      k - ((byte_sums << 8) >> place & 0xFF) << 8];
}
```

Not Fast enough?

- Fix a quantum q (I use 256)
- Store in a table the position of each q-th zero, or q-th one
- Go there in constant time and search from there
- On average, again constant time because of the balance between zeroes and ones
- Extreme locality: one memory access per skip



- 5, 8, 8, 15, 32 ≤ u = 36, ℓ = 2
- We to skip to 22, so we skip 22 >> ℓ = 5 zeroes
- We getting to position 9, so we are in the middle of the unary code associated with the element of index 9 - 5 = 4
- A unary-code read (the dashed arrow) returns 3
- We now know that the upper bits of the current element (of index 4) are 3 + 5 = 8
- Since the block of lower bits of index 4 is zero, we return 32
- If we have skip pointers with q=4, we can start from the dotted arrow

Enough of Fun with Bits

- We want to store an inverted index
- There are document *pointers*, *counts* and *positions*
- For pointers we obviously use a quasi-succinct list with skips
- Counts? Positions?
- Important: we can store *strictly* monotone sequences quasi-succinctly by storing $x_i - i$!

Using Duality Perversely

- Instead of storing counts $c_0, c_1, c_2, \ldots$, we store their *prefix sums* (a.k.a. *cumulative function*) $c_0, c_0 + c_1, c_0 + c_1 + c_2, \ldots$

- Instead of storing positions, we store the *prefix sums of their gaps*

- Positions $p^i_0, p^i_1, p^i_2, \ldots, p^i_{c_i - 1}$ are first turned into $p^i_0 + 1, p^i_1 - p^i_0, p^i_2 - p^i_1, \ldots, p^i_{c_i - 1} - p^i_{c_i - 2}$

- All such sequences are concatenated and stored as a prefix sum

- Key observation: *the counts cumulative function is the indexing function for positions*

Fast & Compact

- Decoding speed faster than other approaches (but not for counts/positions!)
- Compression *definitely* better than other approaches, even for the smallest lists, except for very slow stuff like Golomb
- Locality of access *definitely* better than other approaches
- Note that reading sequentially and skipping mix well together

It Works

- In April 2012 I visited Facebook
- They were working on their new feature—GraphSearch
- Problem: how do you find *very quickly* which of your friends like to cook?
- Hard intersection problem—small vs. big

Interaction

- I handed to Mike Curtiss (formerly at Google) my preprint
- I had the gut feeling that the Elias–Fano representation was exactly what they were looking for: fast intersection of small and big lists
- Note that Shuai Ding (PForDelta) works with Mike

Ten Months Later

- On January 27 I got an email from Mike
- "We wanted to let you know that we have open-sourced a C ++ implementation of your index representation. We are currently using this in production, because it is faster than any other approaches."
- "Any other" includes Google's GroupVarint and variants of PForDelta
- I guess they benchmarked the thing thoroughly...

Fantastic Feedback

- My code was completely rewritten by a competent engineer using stuff I've never seen for unaligned access
- Some more speed improvements
- Very clever idea: read directly cumulative unary codes by bit cancellation
- That stuff found its way back into MG4J (our Java search engine)

What Now?

- Let's improve this, e.g., *better implementations*
- There's decades of engineering and optimization on gaps, very little on this, yet it is faster and compresses better!
- Beautiful code by Philip Pronin (Facebook) on GitHub:

```
int64_t get_next_upper_bits() {
   while( word == 0 ) word = upper_bits[ ++curr ];
   const int64_t upper_bits = curr * 64 +
          __builtin_ctzll( word ) - index++;
   word &= word - 1;
   return upper_bits;
}
```

Benchmarks

- Benchmarks in the WSDM 2012 paper are obsolete (code is now much better thanks to Philip)
- New benchmarks soon using Haswell, which has a single-instruction x &= $x-1$ (thanks to Giuseppe Ottaviano for making me notice this)
- Difficult comparison with the literature, as the authors of the main paper about compression of positions refuse to give their code (the URL in the paper is fake)

When Does It Shine?

- Heavy skipping: "Romeo and Juliet"
- Proximity-based search, like...
- find phrases
- find documents containing a bag of words *within k positions*
- In general, when skipping is more important than pure enumeration
- Particularly efficient when coupled with *optimally lazy* proximity algorithms [Boldi & Vigna 2006]

When Does It Crawl?

- Enumeration oriented tasks (no skipping)
- TF-IDF-like scoring (e.g., BM25) on Boolean disjunctions (but you can have a separate fast count index)
- Everywhere appearing phrases: “home page”

Try It!

- On MG4J: http://mg4j.di.unimi.it/
- On WebGraph: http://webgraph.di.unimi.it/
- Facebook: https://github.com/facebook/folly/
- Lucene codec?
- Questions?