The Elias–Fano coding method

Giulio Ermanno Pibiri
DAIS, Ca' Foscari University of Venice, Italy
giulioermanno.pibiri@unive.it

Abstract. These notes cover the Elias–Fano coding method, a way to represent a sorted integer sequence in almost-optimal space while allowing efficient search operations directly over the compressed representation.

Revised: April 25, 2026

1 Introduction

We consider the following problem.

Problem 1. (Sorted integer sequence coding) Given a sorted sequence A of $n > 0$ integers, $0 \leq x_0 < x_1 < \cdots < x_{n-1} < U$ for some universe size U, represent A in small space and support the following queries.

- Access(i) returns x_i for any $0 \leq i < n$.
- Successor(x) returns the smallest integer $y \in A$ that is $y \geq x$, or Nil if $x > x_{n-1}$.
- Predecessor(x) returns the largest integer $y \in A$ that is $y < x$, or Nil if $x \leq x_0$.

Let us first comment on the space strictly needed to represent A. There are $\binom{U}{n}$ different ways of selecting n distinct[1] integers from a universe of size $U \geq n$. The information-theoretic space lower bound for A is therefore $\lceil \log_2 \binom{U}{n} \rceil$ bits. In the following, we are interested in the case where U is much larger than n. Fact 1 gives a closed-form formula for $\log_2 \binom{U}{n}$ in such case (proof given in Section 5).

Fact 1 ($\log_2$ of binomial)**.** For any two integers U and n such that $n = o(\sqrt{U})$,

$$\log_2 \binom{U}{n} = n \log_2 \left(e \cdot \frac{U}{n} \right) - \frac{1}{2} \log_2(2\pi n) + o(1).$$

In these notes, to analyze the time complexity of an algorithm, we count the number of cache misses that it spends, as practical performance critically relates to them. We use the following simple model [12]: $C(Q) := \lceil Q/B \rceil$ is the number of cache misses that occur when the algorithm reads Q *consecutive* bits, using a cache with page (or *line*) size equal to B bits. For most architectures the value B is 512 (64 bytes). That is, B spans several memory words. In the following, we assume that $\lceil \log_2(U) \rceil \leq B$. (We omit ceiling operators when they are not essential.)

Given these preliminary remarks, the purpose of these notes is to prove the following result due to Peter Elias [2] and Robert Fano [3].

Theorem 1 (Elias–Fano)**.** There exists a representation of A that takes at most $n(L + 3) \leq n \log_2(\frac{U}{n}) + 2n$ bits and for $o(n)$ extra bits:

1. Access can be implemented with, at most, $1 + C_{\text{sel}}$ cache misses;
2. Successor/Predecessor can be implemented with, at most, $C_{\text{sel}} + C_{\text{scan}} + C(\Delta)$ cache misses;

[1] If we allow repetitions (i.e., two or more consecutive integers in A are equal), the number of ways is $\log_2 \binom{U+n-1}{n}$. Fact 1 applies anyway as $U + n - 1 \approx U$ when $U \gg n$.

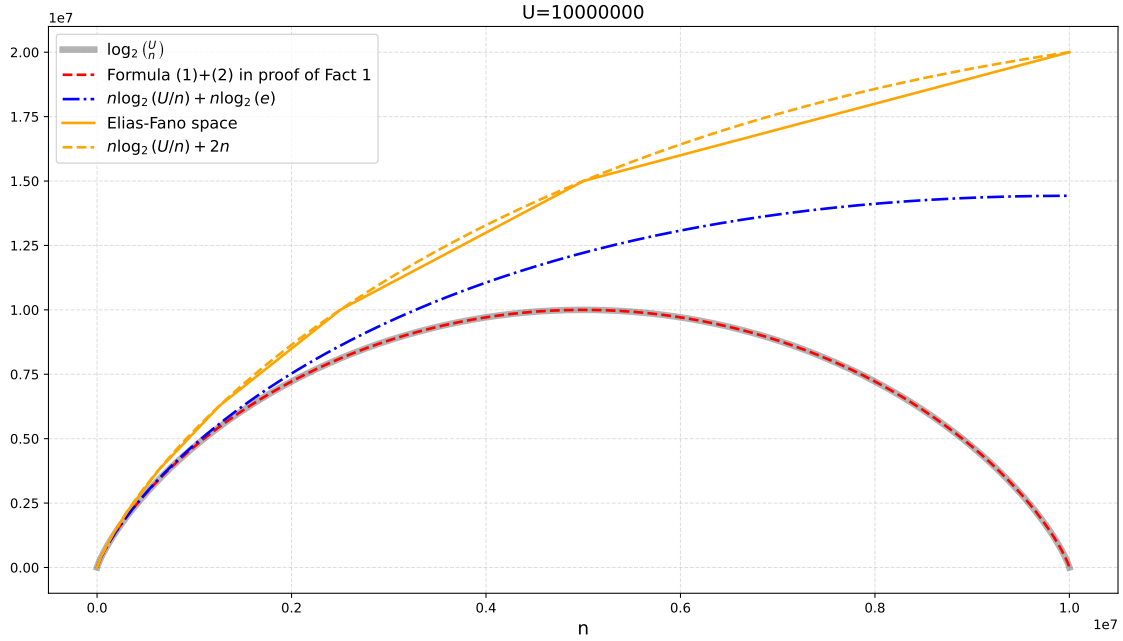

Figure 1: The exact value of $\log_2 \binom{U}{n}$ against the approximation $n \log_2(e \cdot U/n)$ for $U = 10^7$ and varying n. See Fact 1 and its proof in Appendix. We also report the space of Elias–Fano (see Fact 3).

where

- $C_{\text{sel}} = 2 + C(O(\log^4 n))$,
- $C_{\text{scan}} = C(U/n) + C(L \cdot (U/n + 1))$,
- and $\Delta := \max_{0 \le i < n-1} \{\lfloor x_{i+1}/2^L \rfloor - \lfloor x_i/2^L \rfloor\}$,
- and $L = \lfloor \log_2(U/n) \rfloor$.

2 Construction

Since the integers of A are sorted, the bits near the most significant tend to be similar for consecutive integers. The main idea of the Elias–Fano code is therefore to exploit this similarity by coding the "high" part of each integer most succinctly.

The binary $\lceil \log_2(U+1) \rceil$-bit representation of each integer of A is split into two parts: its $0 \le \ell \le \lceil \log_2(U+1) \rceil$ least significant bits and the remaining $h = \lceil \log_2(U+1) \rceil - \ell$ most significant bits. We call these parts the *low* and *high* parts respectively. All the n low parts are written explicitly in a vector A_{low} of ℓ-bit integers, whereas the high parts are coded using a bitvector A_{high} of length $n + \lfloor U/2^\ell \rfloor + 1$ bits, as follows. The elements of A can be viewed as logically clustered into $\lfloor U/2^\ell \rfloor + 1$ clusters, $A_0, \ldots, A_{\lfloor U/2^\ell \rfloor}$, such that A_j contains the consecutive elements of A that have their high bits equal to j. The bitvector A_{high} is then

$$1^{|A_0|}0.1^{|A_1|}0.1^{|A_2|}0. \ \ldots \ 1^{|A_{\lfloor U/2^\ell \rfloor}|}0$$

(where the '.' symbol is just to enhance the readability of the expression). That is, A_{high} writes the cardinalities of the clusters in unary code. Figure 2 illustrates an example. (Note that $|A_j|$ could be 0, i.e., no element in A has high part equal to j. In this case, the unary code is a single 0 bit. Also note that *runs of zeros* might be present in A_{high}. The length of the longest such run is Δ.) It follows that A_{high} has exactly n bits set, one per element of A, and the 0 bits delimit the $\lfloor U/2^\ell \rfloor + 1$ clusters.

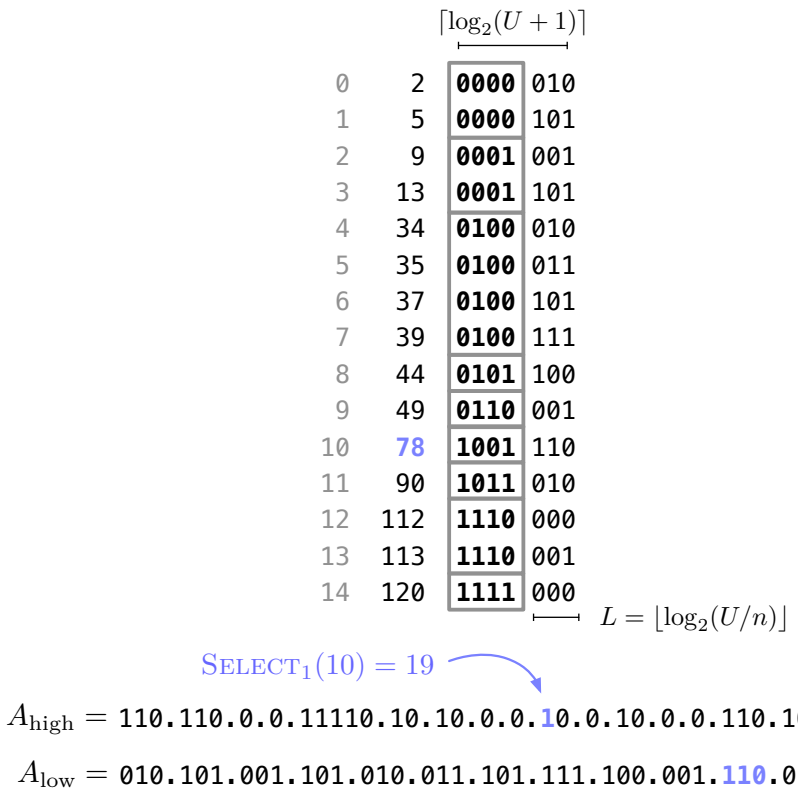

Figure 2: An example sequence $A = [2, 5, 9, 13, 34, 35, 37, 39, 44, 49, 78, 90, 112, 113, 120]$, with $n = 15$ and $U = 127$, encoded with Elias–Fano. The gray boxes logically cluster the high bits. In the example, we have $L = \lfloor \log_2(U/n) \rfloor = 3$, $|A_{\text{high}}| = n + \lfloor U/2^L \rfloor + 1 = 15 + 15 + 1 = 31$ bits (the largest run of zeros is $\Delta = 3$ in this example), and $|A_{\text{low}}| = nL = 15 \cdot 3 = 45$ bits. The encoding consumes a total of 76 bits, whereas $n \log_2(U/n) + 2n = 76.2269$ bits. The steps and information accessed upon $\textsc{Access}(i)$ for $i = 10$ are highlighted in blue font. First, a $\textsc{Select}_1(10)$ query is executed on A_{high}. This query returns the position of 10-th one in A_{high}, which is 19. There must then be $19 - 10 = 9$ zeros up to position 19. This number of zeros is the high part of x_{10}. The low part of x_{10}, the bits 110 (the value 6 in decimal), is simply retrieved from A_{low}. Lastly, the decoded integer is returned as $9 \cdot 2^3 + 6 = 78$.

The space of the encoding is therefore $\text{EF}(\ell, n, U) = n\ell + n + \lfloor U/2^\ell \rfloor + 1$ bits. Solving

$$\frac{\partial}{\partial \ell} \text{EF}(\ell, n, U) = n - \frac{U \ln(2)}{2^\ell} = 0$$

gives us that the value of ℓ for which the space is minimized is $\ell^* = \log_2(\frac{U}{n}) + c$ with $c = \ln \ln(2) / \ln(2) \approx -0.5287$. Let $L = \lfloor \log_2(\frac{U}{n}) \rfloor$ and $F \in [0, 1)$ the fractional part of $\log_2(\frac{U}{n})$. By definition of floor function, we have $L \leq \log_2(\frac{U}{n}) < L + 1$. Summing c to every term in the previous inequality, we have that $L + c \leq \ell^* < L + c + 1$, i.e., $\ell^* \in [L - 0.5287, L + 0.4713)$. However, ℓ must be an integer in practice and we choose $\ell = L$. We claim that this choice is good. In fact, only when $F < -c - 1/2 \approx 0.0287$ we have that the integer closest to ℓ^* is $L - 1$ because $\ell^* = L + F + c < L - 1/2$ in this case. Vice versa, when $F \geq -c - 1/2$, then the integer closest to ℓ^* is L. In other words, the choice $\ell = L$ is optimal for $\approx 97\%$ of the cases.

Fact 2. $n \leq \lfloor \frac{U}{2^L} \rfloor < 2n$.

Proof. By definition of the floor function, we have $L \leq \log_2(\frac{U}{n}) < L + 1 \iff 2^L \leq \frac{U}{n} < 2 \cdot 2^L \iff n \cdot 2^L \leq U < 2n \cdot 2^L \iff n \leq \frac{U}{2^L} < 2n$. Now, since $\lfloor \frac{U}{2^L} \rfloor$ is the largest

Algorithm 1 The procedure ENCODE takes as input a sorted sequence A, its size n, and an upper bound U to the largest element in A, and encodes it with the Elias–Fano method. The operation $A_{\text{high}}.\text{SET}(j)$ sets the j-th bit of A_{high}. The operations at Line 7 and 8 can be implemented with bitwise operators of actual programming languages: $x \bmod 2^L$ as x `&` $(2^L - 1)$ (bitwise AND with the pre-computed constant $2^L - 1$) and $\lfloor x/2^L \rfloor$ as x `>>` L (bitwise right-shift by L bits).

```
1:  function ENCODE(A, n, U)
2:      L = ⌊log2(U/n)⌋
3:      let A_low be an array of n L-bit integers
4:      let A_high be a bitvector of n + ⌊U/2^L⌋ + 1 bits, initially all zeros
5:      for i = 0, ..., n − 1 do
6:          low(x_i) = x_i mod 2^L
7:          high(x_i) = ⌊x_i/2^L⌋
8:          A_low[i] = low(x_i)
9:          A_high.SET(high(x_i) + i)
10:     return (A_low, A_high)
```

integer that is less than or equal to $\frac{U}{2^L}$ and n is at most $\frac{U}{2^L}$, we conclude that n cannot be larger than the largest such integer. Hence $n \leq \lfloor \frac{U}{2^L} \rfloor$. The other inequality simply follows because $\lfloor \frac{U}{2^L} \rfloor \leq \frac{U}{2^L} < 2n$. □

The space of A_{high} is at most $3n$ bits for the previous fact[2]. In conclusion, the space is at most $n(L+3)$ bits. It is interesting to compare the latter bound with the information-theoretic minimum from Fact 1. But before doing it, we note the following.

Fact 3. $nL + n + \lfloor \frac{U}{2^L} \rfloor + 1 \leq n \log_2(\frac{U}{n}) + 2n$.

Proof. Since $L = \log_2(\frac{U}{n}) - F$, the space bound can be written as $n \log_2(\frac{U}{n}) - n(F-1) + \lfloor n2^F \rfloor + 1 \approx n \log_2(\frac{U}{n}) + n(2^F - F + 1)$. Now, consider the function $f(x) = 2^x - x + 1$, $0 \leq x \leq 1$. The function is convex and has a global minimum in $x = -c$. Since $f(0) = 1$ and $f(1) = 1$, we conclude that $2^F - F + 1 \leq 2$ for $F \in [0, 1)$ and the claim follows. □

Now, comparing the bound $n \log_2(\frac{U}{n}) + 2n$ with $n \log_2(\frac{U}{n}) + n \log_2(e)$ (i.e., Fact 1, discarding lower order terms that are negligible for large n), it is immediate to see that Elias–Fano spends at most $2 - \log_2(e) \approx 0.5573$ bits per element more than the minimum when n is large! (See also Figure 1.)

Algorithm 1 shows how the Elias–Fano encoding for A is obtained, i.e., how the arrays A_{low} and A_{high} are computed. The algorithm runs in $\Theta(n)$ time as each operation in the loop takes $O(1)$ time. It executes at most $C(nL) + C(3n)$ cache misses because the arrays A_{low} and A_{high} are accessed sequentially.

3 Queries

3.1 Access

The query ACCESS(i) decodes x_i from the encoded representation of A, for any $0 \leq i < n$. We have $x_i = \text{high}(x_i) \cdot 2^L + \text{low}(x_i)$. The two parts of x_i must be re-linked together upon ACCESS(i). The low bits can be simply retrieved as $\text{low}(x_i) = A_{\text{low}}[i]$, spending one cache miss. The high bits are computed by *searching* A_{high}. To do so efficiently, we introduce two more queries, defined over a bitvector. Assume the bitvector has u bits.

[2]The closer the fraction part of $\log_2(\frac{U}{n})$ is to 0 (resp. 1), the closer $\lfloor \frac{U}{2^L} \rfloor$ is to n (resp. $2n$).

Algorithm 2 ACCESS(i) decodes the i-th integer from the Elias–Fano encoding of A.

```
1: function Access(i)
2:     h = A_high.Select_1(i) − i
3:     l = A_low[i]
4:     x_i = h · 2^L + l
5:     return x_i
```

- The query $\text{RANK}_b(i)$ returns the number of bits equal to b up to position $0 \leq i < u$ (excluded), for $b \in \{0, 1\}$. Clearly, $\text{RANK}_b(i) = i - \text{RANK}_{\neg b}(i)$.
- Symmetrically, the query $\text{SELECT}_b(i)$ returns the position of the i-th bit b, for $b \in \{0, 1\}$. If the bitvector has z ones, then $\text{SELECT}_1(i)$ is valid only for $0 \leq i < z$, and $\text{SELECT}_0(i)$ is valid only for $0 \leq i < u - z$. In any other case, the query returns NIL.

Note that, by definition, the following two equivalences hold.

- $\text{RANK}_b(\text{SELECT}_b(i)) = i$;
- $\text{RANK}_b(\text{SELECT}_{\neg b}(i)) = \text{SELECT}_{\neg b}(i) - i$.

Let us now see how these queries permit to compute $\text{high}(x_i)$ efficiently from A_{high}. The value of $\text{high}(x_i)$ corresponds to the identifier of the cluster to which x_i belongs. Since A_{high} encodes the cardinalities of the clusters in unary code and following the cluster identifier order, i.e., for cluster $j = 0, \ldots, \lfloor U/2^L \rfloor$, it follows that $\text{high}(x_i)$ is the number of zeros up to position $\text{SELECT}_1(i)$ on A_{high}. This quantity is $\text{RANK}_0(\text{SELECT}_1(i))$ which, in turn, is $\text{SELECT}_1(i) - i$ for the previous equivalences. Algorithm 2 illustrates this procedure and Figure 2 shows the steps for $\text{ACCESS}(10) = 78$. Note that ACCESS does not need RANK but only SELECT_1.

A trivial implementation of $\text{SELECT}_1(i)$ would just scan A_{high}, counting the number of seen ones and stopping upon the i-th. This is illustrated in the loop of Algorithm 3, assuming Lines 2–4 are discarded and letting $s = i$ in Line 5. The queries $\text{POPCOUNT}(W)$ and $\text{SELECTINWORD}(W, i)$ return, respectively, the number of ones in the word W and the position of the i-th one in W. As queries can be implemented in constant time using intrinsics[3], it follows that SELECT_1 takes $O(n/w)$ in the worst case for word size w. We explain how to accelerate the query in Section 3.2.

3.2 Select

The idea to accelerate the simple linear-time algorithm for SELECT is to use some extra, little, space on top of the bitvector. In the following, we consider a bitvector V of u bits. Clark [1] was the first to show that SELECT_1 can be supported in $O(1)$ time with $o(u)$ extra bits but his data structure requires a non-trivial space usage in practice and many distinct memory accesses [4]. In the following, we describe the solution by Okanohara and Sadakane [5] – the *DArray* index – which is inspired by Clark's solution and we use in practice. (We assume SELECT_1 queries throughout the presentation although one can obviously flip the ones into zeros to support SELECT_0 as well.)

Theorem 2 (DArray)**.** Consider a bitvector V of u bits with $z > 0$ ones. There exists an index that takes $o(z)$ bits and supports SELECT_1 queries in at most $2 + C(O(\log^4 u))$ cache misses.

[3]Precisely, `__builtin_popcountll` for POPCOUNT and `_pdep_u64` (parallel bits deposit) for SELECTINWORD. See references [7, 8] for further details.

Algorithm 3 $\textsc{Select}_1(i)$ returns the position of the i-th one from the bitvector V. (We assume words of size 64 bits.) The proper block to scan is identified in constant time with the help of the arrays I, S, and D which constitute the DArray index.

```
1:  function Select_1(i)
2:      p = I[⌊i/L_1⌋]
3:      if p < 0 then
4:          return S[−p − 1 + (i mod L_1)]
5:      s = p + D[⌊i/L_3⌋]
6:      r = i mod L_3
7:      if r = 0 then return s
8:      j = ⌊s/64⌋
9:      W = V[j] & (-1 << (s mod 64))          ▷ Clear bits in word W before s.
10:     loop
11:         c = PopCount(W)
12:         if r < c then break                ▷ The wanted position lies in W.
13:         r = r − c
14:         j = j + 1
15:         W = V[j]
16:     return j · 64 + SelectInWord(W, r)
```

The extra bits used for $\textsc{Select}_1$ as well as the number of cache misses claimed for $\textsc{Access}$ (Point 1. of Theorem 1) follow by using Theorem 2 for A_{high} ($u \leq 3n$ and $z = n$).

We now explain the result in Theorem 2. Let L_1, L_2, and L_3 be integer quantities to be fixed later. The z positions of the ones are split into groups of size L_1 (except for, possibly, the last group). Each group defines a block of bits in V. A block is called *sparse* if its length is larger than L_2 bits, *dense* otherwise. Sparse blocks are represented verbatim, i.e., the positions of the L_1 ones are coded using $\log_2(u)$-bit integers. A dense block, instead, is sparsified: we keep one 1-bit position every L_3 such positions. The positions are coded relatively to the beginning of each block, hence taking $\log_2(L_2)$ bits per position. The data structure therefore stores three arrays, I, S, D. The *inventory* array $I[0..\lceil z/L_1 \rceil)$ is such that $I[i] := \textsc{Select}_1(i \cdot L_1)$ if block i is dense; otherwise, $I[i] = -p-1$ where p is the start position in S of the 1-bit positions of block i. The space for I is therefore $z/L_1 \cdot \log_2(u)$ bits. The array S holds the positions of the L_1 ones in sparse blocks. As we have at most z/L_2 sparse blocks, its space is $z/L_2 \cdot L_1 \cdot \log_2(u)$ bits at most. Lastly, the array $D[0..\lceil z/L_3 \rceil)$ is such that $D[i]$ is the position of the $(i \cdot L_3)$-th one relative to the start position of the comprising block if the $(i \cdot L_3)$-th one belongs to a dense block; otherwise $D[i] = -1$. The space for D is $z/L_3 \cdot \log_2(L_2)$ bits.

Refer to Algorithm 3. A $\textsc{Select}_1(i)$ query, $0 \leq i < z$, first checks $p = I[\lfloor i/L_1 \rfloor]$: if $p < 0$, then the block is sparse and the query is answered as $S[-p - 1 + (i \bmod L_1)]$; otherwise the position $D[\lfloor i/L_3 \rfloor]$ is retrieved and a sequential scan of at most L_2 bits is executed starting from position $p + D[\lfloor i/L_3 \rfloor]$ in V, looking for the $(i \bmod L_3)$-th one after that position. It follows that the final scan of a block takes $O(\frac{L_2}{\log u})$ time assuming words of $w = \Theta(\log u)$ bits. The number of cache misses per query is: 2, if i belongs to a sparse block; $2 + C(L_2)$, if i belongs to a dense block. Figure 3 shows an example of the data structure and some queries.

Choosing $L_1 = O(\log^2 u)$, $L_2 = O(\log^4 u)$, and $L_3 = O(\log u)$, all the three arrays I, S, and D take $o(z)$ bits and the number of cache misses per query is at most $2+C(O(\log^4 u))$.

In practice, L_1, L_2, and L_3 are suitable constants, chosen to achieve a good balance between space and query efficiency. For example, we might use $L_1 = 2^{10}$, $L_2 = 2^{16}$

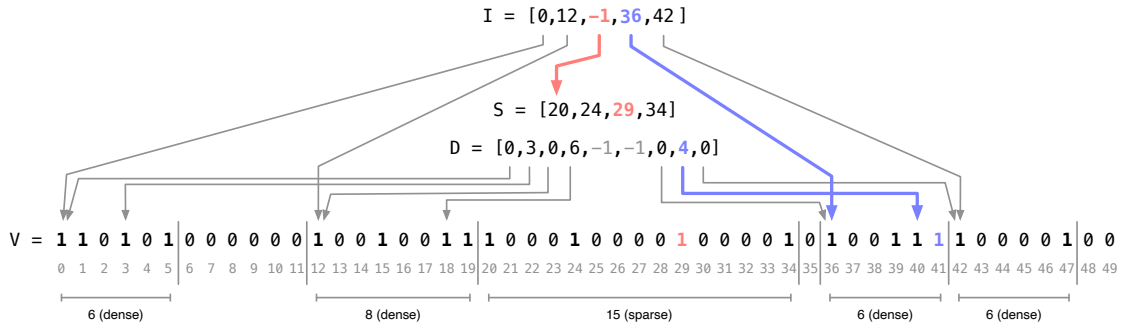

Figure 3: An example of the DArray index for a bitvector V with $u = 50$ bits and $z = 18$ ones, using $L_1 = 4$, $L_2 = 8$, and $L_3 = 2$. The example highlights the accessed information to answer two queries, $\textsc{Select}_1(15)$ (in blue) and $\textsc{Select}_1(10)$ (in red). For $i = 15$, the entry $\lfloor i/L_1 \rfloor = \lfloor 15/4 \rfloor = 3$ of the inventory is first accessed. Since $I[3] \geq 0$, the 15-th one belongs to a dense block. Hence, also the entry $\lfloor i/L_3 \rfloor = \lfloor 15/2 \rfloor = 7$ of the array D is accessed. Lastly, a final scan of the bitvector starting from position $I[3] + D[7] = 40$, and looking for the $(i \bmod L_3) = (15 \bmod 2) = 1$-th bit after that position, is executed to finally determine position 41. For $i = 10$ instead, the inventory entry $\lfloor 10/4 \rfloor = 2$ is accessed. Since $I[2] < 0$, then the 10-th one belongs to a sparse block, hence the query is answered retrieving the position $S[-I[2] - 1 + (10 \bmod 4)] = S[2] = 29$.

(each element of D can be thus coded using 16-bit words), and $L_3 = 2^5$. For this choice, two more considerations follow when the DArray is used over the bitvector A_{high} of the Elias–Fano encoding of A:

- It is unlikely to have sparse blocks in A_{high} as the average density of the ones is *at least* $1/3$. The influence of sparse blocks in the space is almost negligible and the $o(n)$ term in the space bound of Theorem 1 is therefore close to $\frac{n}{L_3} \cdot 16 + \frac{n}{L_1} \cdot 64 = \frac{9}{16}n \approx 0.5625n$.
- The cost $C(L_2)$ is utterly pessimistic. For example, if the z ones were uniformly spread in B, we would have an average distance between two consecutive ones of *at most* 3 and the number of cache misses involved in the scan of a block would be $C(3 \cdot L_3) \ll C(L_2)$ for our choices $L_3 = 2^5$ and $L_2 = 2^{16}$. Furthermore, the cost $C(3 \cdot L_3)$ is 1 for any machine having cache line B larger than $3 \cdot 32 = 96$ bits; the typical cache line size is 512 bits (see Section 1). Therefore, in practice the number of cache misses is closer to 3 than $2 + C(O(\log^4 u))$ for non-pathological cases.

3.3 Iteration

A useful operation usually defined on a data type representing a collection of items is to iterate through all items. In our case, we would like to iterate through all the integers of A in order from the i-th, for some $0 \leq i < n$. A simple loop calling $\textsc{Access}$ at each iteration would clearly suffice, but would also call $\textsc{Select}_1$ at each iteration. This is expensive. However, since the elements are decoded *in order* we can cache the position of $\textsc{Select}_1(i)$ on A_{high} and use that position to compute $\textsc{Select}_1(i+1)$ without an explicit call to $\textsc{Select}_1$ for decoding the $(i+1)$-th element.

Algorithm 4 implements an iterator interface for a sequence coded with Elias–Fano. The method $\textsc{Iterator}(i)$ instantiates an iterator object that holds a *state* that is updated at every forward (resp. backward) movement with the method $\textsc{Next}$ (resp. $\textsc{Prev}$). The value that is currently pointed to by the iterator is decoded with the method $\textsc{Value}$. The state is made up of the position of the currently decoded element, *pos*, and two iterator

Algorithm 4 The iterator interface for a sequence coded with Elias–Fano. Function ITERATOR returns an iterator object that holds the state $(pos, it_{\text{low}}, it_{\text{high}})$.

```
1:  function ITERATOR(i, hint = NIL)
2:      pos = i
3:      j = hint
4:      if j = NIL then
5:          j = A_high.SELECT_1(i)
6:      it_high = A_high.ITERATOR(j)        ▷ Skip to next 1 after position j if A_high[j] = 0.
7:      it_low = A_low.ITERATOR(i)

8:  function VALUE()
9:      h = it_high.VALUE() − pos
10:     l = it_low.VALUE()
11:     return h · 2^L + l

12: function NEXT()
13:     if pos < n then
14:         pos = pos + 1
15:         it_high.NEXT()                  ▷ Skip to next 1 after current position.
16:         it_low.NEXT()

17: function PREV()
18:     if pos > 0 then
19:         pos = pos − 1
20:         it_high.PREV()                  ▷ Skip to previous 1 before current position.
21:         it_low.PREV()
```

objects, it_{high} and it_{low}, respectively iterating through the bits of A_{high} and the sequence of low parts A_{low}.

The semantics of it_{low} is immediate as it iterates through an array of L-bit integers and, at each call of NEXT or PREV, it updates the state of the iterator accordingly. The semantics of it_{high}, instead, is as follows. The method A_{high}.ITERATOR(j) skips to the next 1 after position j if $A_{\text{high}}[j] = 0$; the method it_{high}.NEXT()/PREV() skips to the next/previous 1 after/before the current position; and the method it_{high}.VALUE() returns the position of the pointed-to 1 bit in A_{high}.

Algorithm 4 does not issue any SELECT$_1$ query apart from, possibly, one during the instantiation of the iterator object if no "hint" is given to the ITERATOR method. The method VALUE runs in constant time and does not cause any cache miss because $it_{\text{low/high}}$.VALUE() access memory that has been already fetched by either the method ITERATOR or NEXT/PREV. Instantiating an iterator over A_{low} (Line 7) also takes constant time and at most one cache miss; it_{low}.NEXT()/PREV() takes constant time. Instantiating an iterator over A_{high} (Line 6) can take up to $O(\frac{\Delta}{w})$ time, instead, to skip to the next 1 bit and $C(\Delta)$ cache misses. The same holds for the methods NEXT and PREV of it_{high}.

Iterating through all elements of A costs at most $C(nL) + C(3n)$ cache misses as A_{low} and A_{high} are accessed separately.

Algorithm 5 SUCCESSOR(x) returns the smallest value $y \in A$ that is $y \geq x$. We assume that the largest element of A, MAX $= x_{n-1}$, is stored redundantly and returned directly rather than retrieved with ACCESS($n-1$).

```
1:  function SUCCESSOR(x)
2:      if x > MAX then return NIL
3:      high(x) = ⌊x/2^L⌋
4:      i = 0
5:      p = NIL
6:      if high(x) > 0 then
7:          p = A_high.SELECT_0(high(x) − 1)
8:          i = p − high(x) + 1
9:      it = ITERATOR(i, p)      ▷ Position p is used as a "hint" to aid SELECT_1 on A_high.
10:     y = it.VALUE()
11:     while y < x do
12:         it.NEXT()
13:         y = it.VALUE()
14:     return y
```

3.4 Successor and Predecessor

Both SUCCESSOR and PREDECESSOR can be implemented with binary search by executing an ACCESS query at each step. This algorithm thus involves $O((1 + C_{\text{sel}}) \log n)$ cache misses, where $C_{\text{sel}} = 2 + C(O(\log^4 n))$ is the number of cache misses spent by a SELECT$_b$ query on A_{high} using the DArray index from Theorem 2. In the following, we explain how to avoid the binary search and thus reduce the number of cache misses combining SELECT$_0$ queries on A_{high} with iteration.

Successor. Let us consider SUCCESSOR(x) first. From x, we compute $\text{high}(x) = \lfloor x/2^L \rfloor$. If $\text{high}(x) > 0$, then $i = \text{SELECT}_0(\text{high}(x) - 1) - \text{high}(x) + 1$ is the index of the first element of A whose high bits are greater than or equal to high(x) (if $\text{high}(x) = 0$, we let $i = 0$). Symmetrically, $j = \text{SELECT}_0(\text{high}(x)) - \text{high}(x)$ gives us the index of the first element of A whose high bits are strictly larger than high(x). Since a cluster contains at most $2^L \leq U/n$ elements, we have that $j - i \leq U/n$ elements, and the successor could be determined by binary searching the range $A[i..j]$ for a total of $2C_{\text{sel}} + O(\log(U/n)(1+C_{\text{sel}}))$ cache misses.

However in practice, it is often better to answer the query by scanning A from the i-th element (and without computing the value $j = \text{SELECT}_0(\text{high}(x)) - \text{high}(x)$). When scanning from the i-th element, the following cases can happen. Let $p = \text{SELECT}_0(\text{high}(x) - 1)$ if $\text{high}(x) > 0$ or NIL otherwise (with a little abuse of notation, we assume that NIL$+1 = 0$ in the following discussion).

1. The bit in position $p+1$ of A_{high} is 0: then cluster high(x) is empty and the successor of x is x_i (minimum element in the next non-empty cluster). The low bits of x_i are retrieved with 1 cache miss, whereas its high bits are computed by scanning A_{high} from position $p + 1$ until the next bit set. Since the longest run of zeros in A_{high} has length Δ, $C(\Delta)$ cache misses are issued during the scan.
2. The bit in position $p + 1$ of A_{high} is 1, so the cluster high(x) is not empty. The elements in the cluster all have the same high bits (equal to high(x)). Now, two cases can happen:

   (a) The successor is not larger than the largest element in the cluster, so it belongs to the cluster. Scanning up to U/n elements therefore costs $C(U/n) + C(L \cdot U/n)$

cache misses.

(b) The successor is larger than the largest element in the cluster, so it is the minimum in the next non-empty cluster. The cost is at most $C(U/n)+C(\Delta)+C(L\cdot(U/n+1))$.

Call $C_{\text{scan}} = C(U/n)+C(L\cdot(U/n+1))$. The cost of SUCCESSOR is at most $C_{\text{sel}}+C_{\text{scan}}+C(\Delta)$ cache misses. Algorithm 5 shows the pseudocode for SUCCESSOR.

We discuss some examples. Refer to Figure 2 and consider $x = 57$; $\text{high}(x) = 7$. Then $p = \text{SELECT}_0(\text{high}(x)-1) = p = \text{SELECT}_0(6) = 16$ and $i = p-\text{high}(x)+1 = 16-7+1 = 10$ is the index of the first element whose high bits are ≥ 7. In this case, the bit in position $p+1$ is 0, hence cluster 7 is empty and $\text{SUCCESSOR}(57) = x_{10}$. To recover the high bits of x_{10}, we scan A_{high} from position $p+1$ until a 1 bit is found (or A_{high} is exhausted) and counting the number of zeros in between. In this case, the next 1 bit is found in position 19 and the number of zeros between position $p+1$ (included) and position 19 is 2. Thus the high bits of x_{10} are the integer $7+2=9$.

Now consider $x = 37$; $\text{high}(x) = 4$. Then $p = \text{SELECT}_0(\text{high}(x)-1) = \text{SELECT}_0(3) = 7$ and $i = p-\text{high}(x)+1 = 7-4+1 = 4$ is the index of the first element whose high bits are ≥ 4. The bit in position $p+1 = 8$ is 1, hence cluster 4 is not empty and $\text{SUCCESSOR}(37) = 37$ is determined by scanning the elements of cluster 4.

Predecessor. The algorithm for PREDECESSOR is similar. If the bit in position $p+1$ of A_{high} is 1 and x is larger than the first (smallest) element in cluster $\text{high}(x)$, then the PREDECESSOR belongs to the cluster and the cost is at most $C(U/n)+C(\Delta)+C(L\cdot(U/n+1))$[4]. Otherwise, the PREDECESSOR is element x_{i-1} and to determine its high bits we pay a cost of $C(\Delta)$ in the worst case. The worst-case cost is therefore identical to that of SUCCESSOR. Algorithm 6 shows the pseudocode for PREDECESSOR.

Refer again to Figure 2 and consider $x = 33$; $\text{high}(x) = 4$. Then $p = \text{SELECT}_0(\text{high}(x)-1) = \text{SELECT}_0(3) = 7$ and $i = p-\text{high}(x)+1 = 7-4+1 = 4$ is the index of the first element whose high bits are ≥ 4. The bit in position $p+1 = 8$ is 1, hence cluster 4 is not empty, but x is not larger than the first element from the cluster which is 34. Hence $\text{PREDECESSOR}(33) = x_{i-1} = x_3 = 13$. To determine the high bits of x_3 we scan A_{high} backwards from position p and count the number of zeros until the first bit 1. In this case, we count two zeros, hence the high bits of x_3 are the integer $3-2=1$.

For $x = 37$, instead, we proceed in the same way as for $\text{SUCCESSOR}(37)$ and since x is larger than the first element in cluster 4, 34, $\text{PREDECESSOR}(37)$ is determined by scanning the elements cluster 4.

In summary, the idea behind these algorithms is to replace a SELECT_1 query to recover the high bits of $\text{SUCCESSOR}(x)/\text{PREDECESSOR}(x)$ with a scan of at most Δ bits. Albeit $\Delta \approx 2n$ in pathological cases, the scan is beneficial whenever $C(\Delta) < 2 + C_{\text{sel}}$, which is likely to be the case in practice when Δ is small.

Using again Theorem 2 on the zeros of A_{high} ($u \leq 3n$ and $z = 2n$) to implement SELECT_0, the extra bits and number of cache misses claimed in Point 2. of Theorem 1 follow.

4 Further readings and implementations

The Elias–Fano coding method has found its main use in compressing inverted indexes for search engines. In this context, the method has been re-discovered (e.g., in 1998 by

[4]We could avoid the cost $C(\Delta)$ noting that the predecessor must lie within the cluster: if all the elements of the cluster have been examined (the runs on 1s in A_{high} is exhausted), the predecessor is the last element in the cluster and there is no need to compute the high bits (nor the low bits) of the first element in the next cluster. The cost would therefore be $C_{\text{sel}} + \max\{C_{\text{scan}}, C(\Delta)+1\}$.

Algorithm 6 PREDECESSOR(x) returns the largest value $y \in A$ that is $y < x$. We assume that the largest and smallest elements of A, MAX $= x_{n-1}$ and MIN $= x_0$ respectively, are stored redundantly and not retrieved with ACCESS($n-1$) and ACCESS(0).

```
1:  function PREDECESSOR(x)
2:      if x ≤ MIN then return NIL
3:      if x > MAX then return MAX
4:      high(x) = ⌊x/2^L⌋
5:      i = 0
6:      p = NIL
7:      if high(x) > 0 then
8:          p = A_high.SELECT_0(high(x) − 1)
9:          i = p − high(x) + 1
10:     it = ITERATOR(i, p)
11:     y = it.VALUE()
12:     if x ≤ y then
13:         it.PREV()
14:         y = it.VALUE()
15:         return y
16:     prev = y
17:     while y < x do
18:         prev = y
19:         it.NEXT()
20:         y = it.VALUE()
21:     return prev
```

Vo and Moffat [13] who described it as a "modified Rice code"), adapted (see the work by Vigna [11]), and extended (e.g., the *partitioned* Elias–Fano technique by Ottaviano and Venturini [6]). From a theoretical point of view, it is also possible to make the encoding dynamic [9], hence allowing insertion and removal of integers, while maintaining the asymptotic space usage and runtime for the operations.

The survey paper by Pibiri and Venturini [10] also contains a description of the method, and experimental comparisons against other methods.

Implementations. Some open-source implementations of Elias–Fano are available.

- C++: https://github.com/jermp/bits.
- C++: https://github.com/vigna/sux.
- Rust: https://github.com/vigna/sux-rs.

References

[1] David Clark. *Compact pat trees.* PhD thesis, University of Waterloo, 1997.

[2] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[3] Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

[4] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[5] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, pages 60–70, 2007.

[6] Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 273–282, 2014.

[7] Prashant Pandey, Michael A Bender, and Rob Johnson. A fast x86 implementation of select. *arXiv preprint arXiv:1706.00990*, 2017.

[8] Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Information Systems*, 99:101756, 2021.

[9] Giulio Ermanno Pibiri and Rossano Venturini. Dynamic elias-fano representation. In *28th Annual symposium on combinatorial pattern matching (CPM 2017)*, pages 30–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[10] Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing surveys*, 53(6):125:1–125:36, 2021.

[11] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 83–92, 2013.

[12] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys*, 33(2):209–271, 2001.

[13] Anh Ngoc Vo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 290–297, 1998.

5 Appendix

Fact 4 (Stirling's approximation)**.** For any large positive integer x, $\log_2(x!) = x\log_2(\frac{x}{e}) + \frac{1}{2}(\log_2 x + \log_2(2\pi)) + o(1)$.

Fact 5. For $x \in [-1, 1)$, $\log_2(1-x) = -\log_2(e) \cdot \sum_{k=1}^{\infty} \frac{x^k}{k} = -\log_2(e) \cdot (x + \frac{x^2}{2} + \frac{x^3}{3} + \cdots)$.

Fact 1 ($\log_2$ of binomial)**.** For any two integers U and n such that $n = o(\sqrt{U})$,

$$\log_2 \binom{U}{n} = n \log_2 \left(e \cdot \frac{U}{n}\right) - \frac{1}{2}\log_2(2\pi n) + o(1).$$

Proof. By applying Fact 4 to $\log_2 \binom{U}{n} = \log_2(\frac{U!}{(n!(U-n)!)})$ and simplifying, we have

$$\begin{aligned}
\log_2 \binom{U}{n} &= U \cdot \log_2 \left(\frac{U}{U-n}\right) - n \cdot \log_2 \left(\frac{n}{U-n}\right) \\
&\quad + \frac{1}{2}\Big(\log_2 \left(\frac{U}{n}\right) - \log_2(U-n) - \log_2(2\pi)\Big) + o(1) = \\
&\quad - U \cdot \log_2 \left(1 - \frac{n}{U}\right) + n \log_2 \left(\frac{U}{n}\Big(1 - \frac{n}{U}\Big)\right) \\
&\quad - \frac{1}{2}\log_2 \left(\frac{n(U-n)}{U}\right) - \frac{\log_2(2\pi)}{2} + o(1) = \\
&\quad - U \cdot \log_2 \left(1 - \frac{n}{U}\right) + n \log_2 \left(\frac{U}{n}\right) + n \log_2 \left(1 - \frac{n}{U}\right) \qquad (1) \\
&\quad - \frac{1}{2}\log_2(n) - \frac{1}{2}\log_2 \left(1 - \frac{n}{U}\right) - \frac{\log_2(2\pi)}{2} + o(1). \qquad (2)
\end{aligned}$$

Since $n = o(\sqrt{U})$, $\log_2(1 - \frac{n}{U}) = -\log_2(e) \cdot \frac{n}{U} - o(1)$ for Fact 5. The claim follows by plugging the latter equation into the previous expression. □

Fact 6. For any two integers U and n such that $1 \le n \le U$, $\binom{U}{n} < (e \cdot \frac{U}{n})^n$.

Proof. First observe that $\binom{U}{n} = \frac{U!}{n!(U-n)!} = \frac{U \cdot (U-1) \cdot \cdots \cdot (U-n+1)}{n!} < \frac{U^n}{n!}$. By truncating the Taylor expansion $e^n = \sum_{i=0}^{\infty} \frac{n^i}{i!}$ to the n-th term, it follows that $e^n > \frac{n^n}{n!} \iff n! > (\frac{n}{e})^n$. The claim follows by plugging the latter inequality into the former. □

Fact 6 tells us that the term $-\frac{1}{2}\log_2(2\pi n) + o(1)$ in Fact 1 is *always* negative.