A Crash Course on Data Compression

3.1 Some List Compressors in C++

Giulio Ermanno Pibiri

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

@giulio_pibiri

@jermp

Overview

- C++ implementation of:
  Elias-Fano, Binary Interpolative Coding
- Compress/Uncompress some (long) integer lists
- Write-to/Load-from disk the compressed file

General Approach

For the wanted `codec` (compressor/decompressor), we would like to expose the following (minimal) interface:

```
struct codec {
    void encode(uint32_t const* input, uint32_t n);
    void decode(uint32_t* output) const;
    uint32_t size() const;
    void save(std::ofstream& out) const;
    void load(std::ifstream& in);
};
```

takes a pointer to a memory area (of, at least, 4n bytes), `input`, interprets the bytes pointed to by `input` as a list of n `uint32_t` integers, and encodes the list

decode the compressed list and writes the decoded integers as uncompressed `uint32_t` values to the memory pointed to by the `output` pointer

write the bytes to an output stream (e.g., a file)

read the bytes from an input stream (e.g., a file), and fill the memory of the `codec`

Assumption: the input list is **sorted**.

Our Plan

```
struct elias_fano {
    elias_fano() : m_l(0), m_universe(0) {}

    void encode(uint32_t const* input, uint32_t n);
    void decode(uint32_t* output) const;

    uint32_t size() const;
    uint32_t access(uint32_t i) const;

    void save(std::ofstream& out) const;
    void load(std::ifstream& in);

private:
    uint32_t m_l;
    uint32_t m_universe;
    bit_vector m_high_bits;
    bit_vector m_low_bits;
    darray m_high_bits_darray;
};
```

random access: it returns the i-th integer

provides the `select` query to efficiently implement `access`

```
struct interpolative {
    interpolative() : m_size(0), m_universe(0) {}

    void encode(uint32_t const* input, uint64_t n);
    void decode(uint32_t* output) const;

    uint32_t size() const;

    void save(std::ofstream& out) const;
    void load(std::ifstream& in);

private:
    uint32_t m_size;
    uint32_t m_universe;
    bit_vector m_bits;
};
```

elias_fano::encode

```
void encode(uint32_t const* input, uint64_t n) {
    if (n == 0) return;
    uint32_t u = input[n - 1];
    m_universe = u;

    assert(u >= n);
    m_l = std::ceil(std::log2(static_cast<double>(u) / n));
    uint64_t num_clusters = (u >> m_l) + 1;

    bit_vector_builder bvb_high_bits;
    bvb_high_bits.resize(n + num_clusters);

    bit_vector_builder bvb_low_bits;
    bvb_low_bits.resize(n * m_l);

    uint64_t low_mask = (uint64_t(1) << m_l) - 1;
    for (size_t i = 0; i != n; ++i, ++input) {
        uint32_t x = *input;
        bvb_low_bits.set_bits(i * m_l, x & low_mask, m_l);
        bvb_high_bits.set_bits((x >> m_l) + i, 1, 1);
    }

    bvb_high_bits.build(m_high_bits);
    bvb_low_bits.build(m_low_bits);
    m_high_bits_darray.build(m_high_bits);
}
```

$\ell = \lceil \log_2(U/n) \rceil$

$U/2^\ell + 1$

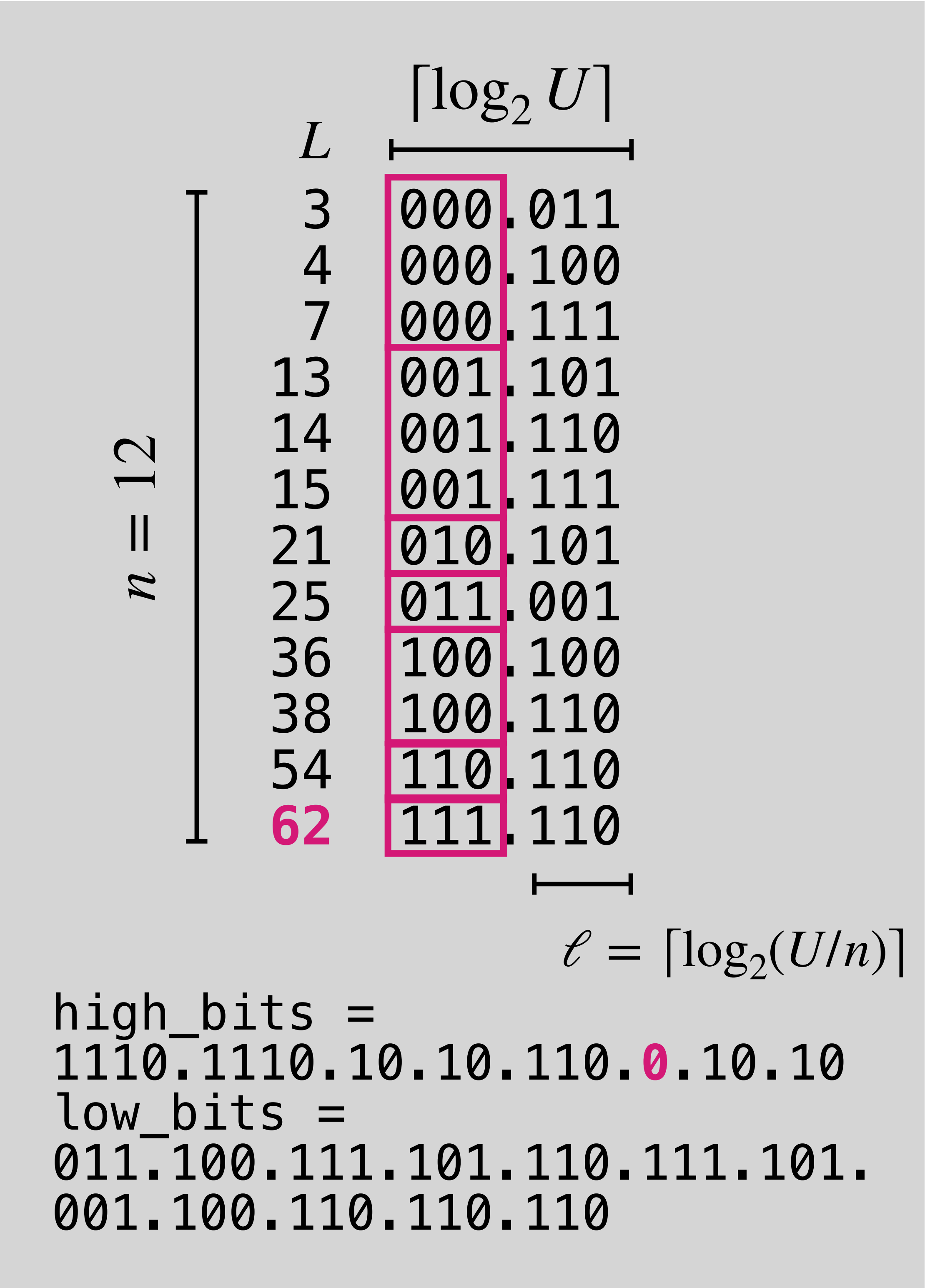
all zeros

isolate the lower bits by masking and write them into position

set a bit for every integer in the list

build the bit-vectors

Elias-Fano example



elias_fano::decode

```cpp
uint64_t access(uint64_t i) const {
    assert(i < size());
    uint64_t high = (m_high_bits_darray.select(m_high_bits, i) - i) << m_l;
    uint64_t low = m_low_bits.get_bits(i * m_l, m_l);
    return high | low;
}
```

Pseudo-code

Access(i):
$l = low\_bits[i]$
$h = \text{Select}_1(high\_bits, i) - i$
return $(h \ll \ell) \mid l$

one-to-one mapping between pseudo-code and C++

```cpp
void decode(uint32_t* output) const {
    uint32_t list_size = size();
    for (uint64_t i = 0; i != list_size; ++i) {
        uint64_t x = access(i);
        *output = x;
        ++output;
    }
}
```

decode each integer using `access` (**warning**: definitely not the best we can do for performance!)

write the decoded integer and advances the output pointer

interpolative::encode

```
void encode(uint32_t const* input, uint64_t n) {
    if (n == 0) return;
    bit_vector_builder builder;
    m_size = n;
    m_universe = input[n - 1];
    encode(builder, input, m_size - 1, 0, m_universe);
    builder.build(m_bits);
}
```

bit-vector builder to write the codewords

initial lower and upper bounds

build the bit-vector

```
void encode(bit_vector_builder& builder, uint32_t const* input,
            uint32_t n, uint32_t lo, uint32_t hi) {
    if (n == 0) return;

    assert(lo <= hi);
    assert(hi - lo >= n - 1);

    if (hi - lo + 1 == n) return;

    uint32_t m = n / 2;
    uint32_t x = input[m];
    write_binary(builder, x - lo - m, hi - lo - n + 1);

    encode(builder, input, m, lo, x - 1);
    encode(builder, input + m + 1, n - m - 1, x + 1, hi);
}
```

stop recursion

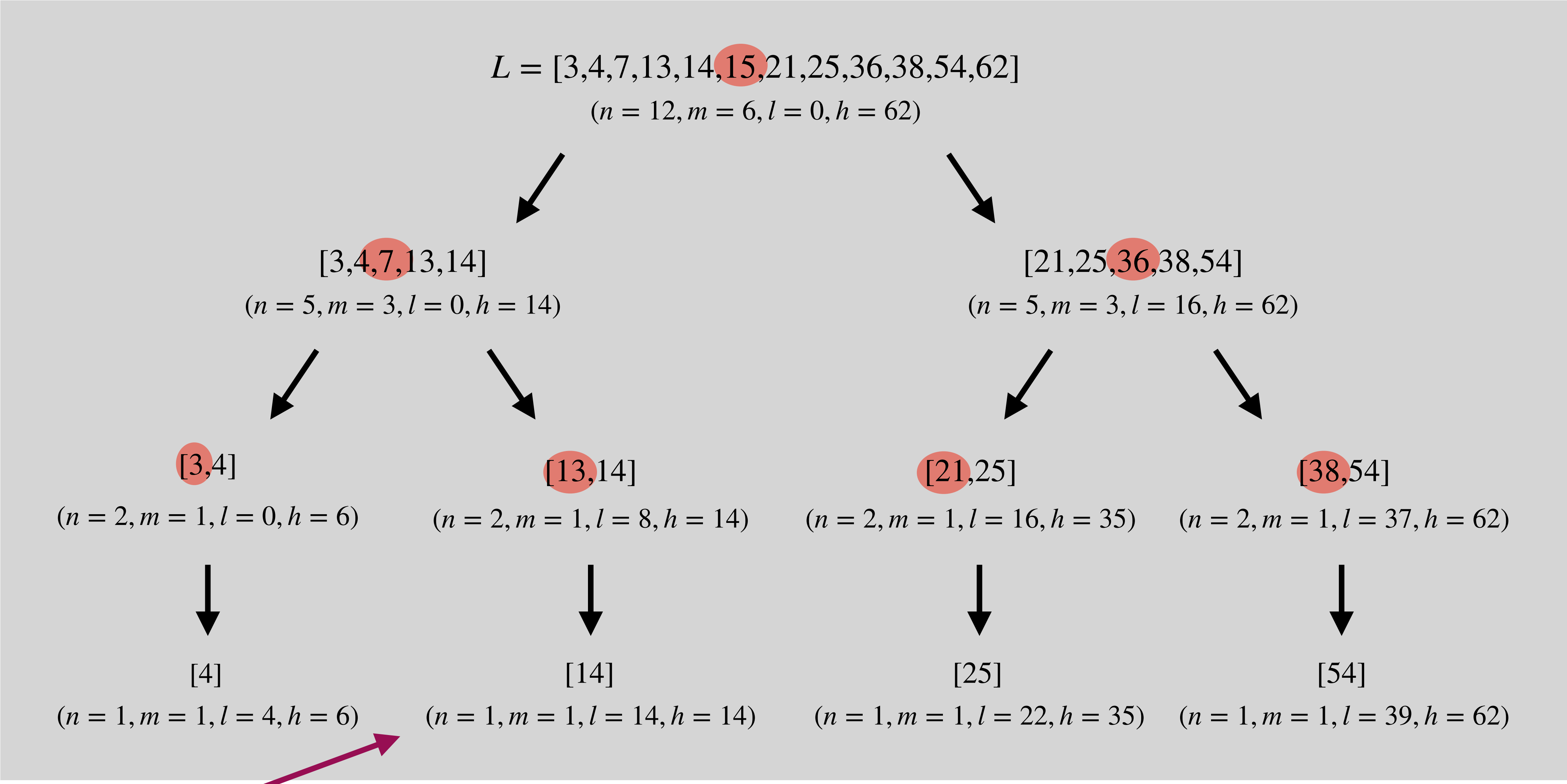
invariants

run of consecutive ints: stop recursion

encode the middle element

recursive calls

Interpolative example



write the integer $x \leq r$ using $b = \lceil \log_2(r+1) \rceil$ bits

```
void write_binary(bit_vector_builder& builder, uint32_t x, uint32_t r) {
    assert(r > 0);
    assert(x <= r);
    uint32_t b = msb(r) + 1;
    builder.append_bits(x, b);
}
```

interpolative::decode

```
void decode(uint32_t* output) const {
    bit_vector_iterator it(m_bits);
    output[m_size - 1] = m_universe;
    decode(it, output, m_size - 1, 0, m_universe);
}
```

run of consecutive ints: decode implicitly via a simple for loop

```
void decode(bit_vector_iterator& it, uint32_t* output,
            uint32_t n, uint32_t lo, uint32_t hi) const {
    if (n == 0) return;

    assert(lo <= hi);
    assert(hi - lo >= n - 1);

    if (hi - lo + 1 == n) {
        for (uint32_t i = 0; i != n; ++i) output[i] = lo++;
        return;
    }

    uint32_t m = n / 2;
    uint32_t x = read_binary(it, hi - lo - n + 1) + lo + m;
    output[m] = x;

    decode(it, output, m, lo, x - 1);
    decode(it, output + m + 1, n - m - 1, x + 1, hi);
}
```

read $b = \lceil \log_2(r+1) \rceil$ bits and interpret them as the integer x

```
uint32_t read_binary(bit_vector_iterator& it, uint32_t r) {
    assert(r > 0);
    uint32_t b = msb(r) + 1;
    uint32_t x = it.take(b);
    assert(x <= r);
    return x;
}
```

How to compile and run the code

From a terminal window, move into this folder and type the following commands.

To compile in a "debug" environment, define (`-D`) the `DEBUG` flag to enable all asserts:

```
g++ -std=c++11 -DDEBUG compress.cpp -o compress
g++ -std=c++11 -DDEBUG -mbmi2 -msse4.2 decompress.cpp -o decompress
g++ -std=c++11 -DDEBUG -mbmi2 -msse4.2 check.cpp -o check
```

NOTE. Note the two extra compiler flags `-mbmi2` and `-msse4.2` that are needed, respectively, for the two hardware instructions `pdep` (parallel bit deposit) and `popcnt` (population count). Both these two special instructions are used in the implementation of the `select` query used by Elias-Fano' access algorithm.

To compile for maximum speed, disable all asserts (`-DNDEBUG`) and also use the optimization flags `-O3` and `-march=native` :

```
g++ -std=c++11 -DNDEBUG -O3 -march=native compress.cpp -o compress
g++ -std=c++11 -DNDEBUG -O3 -mbmi2 -msse4.2 -march=native decompress.cpp -o decompress
g++ -std=c++11 -DNDEBUG -O3 -mbmi2 -msse4.2 -march=native check.cpp -o check
```

Now, first unzip the file `lists.txt.gz` in the folder `2_integer_codes/code` which contains 10 sorted integer lists:

```
gunzip ../../2_integer_codes/code/lists.txt.gz
```

Then use the program `./compress` to actually compress the lists. The program expects the following arguments:

```
Usage: ./compress [type] [input_lists_filename] [output_filename]
```

where `type` is one of the following: `ef` or `bic` for, respectively, Elias-Fano or Binary Interpolative Coding.

The script `run_all.sh` shows all the examples. To run it, use:

```
bash run_all.sh
```

Micro benchmark

On a desktop Mac book pro (16-inch, 2019) with a 2.6 GHz 6-Core Intel Core i7 processor, I got the following results (compiling with optimization flags `-O3` and `-march=native`).

Code	bits/int	ns/int
ef	6.81	5.40
bic	2.61	6.60