A gentle introduction to the: Succinct Data Structure Library

Algorithm Engineering A.Y. 2021/22
Università di Pisa

Giorgio Vinciguerra
Teaching assistant

Lab lectures

- **Not** part of the syllabus but useful for your future career
- Tools for working with big data: Compressed/succinct data structures
  - Complex algorithms become possible in memory, no clusters and no disks
  - The libraries we will see offer the same API of standard data structures

From theory to practice

- Integer coding via Elias-γ and -δ codes (§11.1 of the notes)
- Plain bitvectors with rank/select support (§15.1.1)
- Compressed bitvectors via Elias-Fano coding (§11.6 and §15.1.2)
- Pointerless programming (§15.1)

The Succinct Data Structure Library (SDSL)

https://github.com/simongog/sdsl-lite

(see also the fork https://github.com/xxsds/sdsl-lite)

- An **easy-to-use**, **highly-efficient**, **configurable**, and **extensible** library of succinct data structures for researchers and practitioners
- Implements highlights of 40 research articles
- Faithful to the original proposals while using modern instruction sets
- Written in C++11, API familiar to C++ STL users
- Wrappers in other languages (e.g. Python) available online

SDSL resources

Cheatsheet: http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf

API Docs: http://algo2.iti.kit.edu/gog/docs/html/index.html

Examples: https://github.com/simongog/sdsl-lite/examples

sdsl Cheat Sheet

Data structures

The library code is in the `sdsl` namespace. Either import the namespace in your program (`using namespace sdsl;`) or qualify all identifiers by a `sdsl::`-prefix.

Each section corresponds to a header file. The file is hyperlinked as part of the section heading.

We have two types of data structures in **sdsl**. *Self-contained* and *support* structures. A support object `s` can extend a self-contained object `o` (e.g. add functionality), but requires access to `o`. Support structures contain the substring `support` in their class names.

Integer Vectors (IV)

The core of the library is the class `int_vector<`w`>`. Parameter w corresponds to the fixed length of each element in bits. For $w = 8, 16, 32, 64, 1$ the length is fixed during compile time and the vectors correspond to `std::vector<uint`w`_t>` resp. `std::vector<bool>`. If $w = 0$ (default) the length can be set during runtime. *Constructor:* `int_vector<>(`n`,`x`,`ℓ`)`, with n equals size, x default integer value, ℓ width of integer (has no effect for $w > 0$).

Public methods: `operator[i]`, `size()`, `width()`, `data()`.

Manipulating `int_vector<`w`> v`

Method	*Description*
`v[`i`]=`x	Set entry `v[`i`]` to x.
`v.width(`ℓ`)`	Set width to ℓ, if $w = 0$.
`v.resize(`n`)`	Resize `v` to n elements.
Useful methods in namespace `sdsl::util`:	
`set_to_value(v,`k`)`	Set `v[`i`]=`k for each i.
`set_to_id(v)`	Set `v[`i`]=`i for each i.
`set_random_bits(v)`	Set elements to random bits.
`mod(v,`m`)`	Set `v[`i`]=v[`i`]` mod m for each i.
`bit_compress(v)`	Gets $x = \max_i$ `v[`i`]` and $\ell = \lceil \log(x-1) \rceil + 1$ and packs the entries in ℓ-bit integers.
`expand_width(v,`ℓ`)`	Expands the width of each integer to ℓ bits, if $\ell \geq$ `v.width()`.

Compressed Integer Vectors (CIV)

For a vector `v`, `enc_vector` stores the self-delimiting coded deltas (`v[`i+1`]`$-$`v[`i`]`). Fast random access is achieved by sampling values of `v` at rate `t_dens`. Available coder are `coder::elias_delta`, `coder::elias_gamma`, and `coder::fibonacci`.

Class `vlc_vector` stores each `v[i]` as self-delimiting codeword. Samples at rate `t_dens` are inserted for fast random access.

Class `dac_vector` stores for each value x the least (`t_b` $-$ 1) significant bits plus a bit which is set if $x \geq 2^{b-1}$. In the latter case, the process is repeated with $x' = x/2^{b-1}$.

Bitvectors (BV)

Representations for a bitvector of length n with m set bits.

Class	*Description*	*Space*
`bit_vector`	plain bitvector	$64\lceil n/64+1 \rceil$
`bit_vector_il`	interleaved bitvector	$\approx n(1+64/K)$
`rrr_vector`	H_0-compressed bitvector	$\approx \lceil \log \binom{n}{m} \rceil$
`sd_vector`	sparse bitvector	$\approx m \cdot (2 + \log \frac{n}{m})$
`hyb_vector`	hybrid bitvector	

`bit_vector` equals `int_vector<1>` and is therefore dynamic.
Public Methods: `operator[`i`]`, `size()`, `begin()`, `end()`
Public Types: `rank_1_type`, `select_1_type`, `select_0_type`[1].
Each bitvector can be constructed out of a `bit_vector` object.

Rank Supports (RS)

RSs add rank functionality to BV. Methods `rank(`i`)` and `operator(`i`)` return the number of set bits[2] in the prefix $[0..i)$ of the supported BV for $i \in [0, n]$.

Class	*Compatible BV*	*+Bits*	*Time*
`rank_support_v`	`bit_vector`	$0.25n$	$\mathcal{O}(1)$
`rank_support_v5`	`bit_vector`	$0.0625n$	$\mathcal{O}(1)$
`rank_support_scan`	`bit_vector`	64	$\mathcal{O}(n)$
`rank_support_il`	`bit_vector_il`	128	$\mathcal{O}(1)$
`rank_support_rrr`	`rrr_vector`	80	$\mathcal{O}(k)$
`rank_support_sd`	`sd_vector`	64	$\mathcal{O}(\log \frac{n}{m})$
`rank_support_hyb`	`hyb_vector`	64	-

Call `util::init_support(rs,bv)` to initialize rank structure `rs` to bitvector `bv`. Call `rs(`i`)` to get `rank(`i`)` $= \sum_{k=0}^{k<i}$ `bv[`k`]`

Select Supports (SLS)

SLSs add select functionality to BV. Let m be the number of set bits in BV. Methods `select(`i`)` and `operator(`i`)` return the position of the i-th set bit[3] in BV for $i \in [1..m]$.

Class	*Compatible BV*	*+Bits*	*Time*
`select_support_mcl`	`bit_vector`	$\leq 0.2n$	$\mathcal{O}(1)$
`select_support_scan`	`bit_vector`	64	$\mathcal{O}(n)$
`select_support_il`	`bit_vector_il`	64	$\mathcal{O}(\log n)$
`select_support_rrr`	`rrr_vector`	64	$\mathcal{O}(\log n)$
`select_support_sd`	`sd_vector`	64	$\mathcal{O}(1)$

Call `util::init_support(sls,bv)` to initialize `sls` to bitvector `bv`. Call `sls(`i`)` to get `select(`i`)` $= \min\{j \mid$ `rank(`$j+1$`)` $= i\}$.

Wavelet Trees (WT=BV+RS+SLS)

Wavelet trees represent sequences over byte or integer alphabets of size σ and consist of a tree of BVs. Rank and select on the sequences is reduced to rank and select on BVs, and the runtime is multiplied by a factor in $[H_0, \log \sigma]$.

Class	*Shape*	`lex_ordered`	*Default alphabet*	Travers-able
`wt_rlmn`	underlying WT dependent			×
`wt_gmr`	none	×	integer	×
`wt_ap`	none	×	integer	×
`wt_huff`	Huffman	×	byte	✓
`wm_int`	Balanced	×	integer	✓
`wt_blcd`	Balanced	✓	byte	✓
`wt_hutu`	Hu-Tucker	✓	byte	✓
`wt_int`	Balanced	✓	integer	✓

Public types: `value_type`, `size_type`, and `node_type` (if WT is traversable). In the following let c be a symbol, i,j,k, and q integers, v a node, and r a range.
Public methods: `size()`, `operator[`i`]`, `rank(`i`,`c`)`, `select(`i`,`c`)`, `inverse_select(`i`)`, `begin()`, `end()`.
Traversable WTs provide also: `root()`, `is_leaf(`v`)`, `empty(`v`)`, `size(`v`)`, `sym(`v`)`, `expand(`v`)`, `expand(`v`,`r`)`, `expand(`v`,std::vector<`r`>)`, `bit_vec(`v`)`, `seq(`v`)`.
`lex_ordered` WTs provide also: `lex_count(`i`,`j`,`c`)` and `lex_smaller_count(`i`,`c`)`. `wt_int` provides: `range_search_2d`.
`wt_algorithm.hpp` contains the following generic WT method (let wt be a WT object): `intersect(`wt`, vector<`r`>)`, `quantile_freq(`wt`,`i`,`j`,`q`)`, `interval_symbols(`wt`,`i`,`j`,`k`,...)`, `symbol_lte(`wt`,`c`)`, `symbol_gte(`wt`,`c`)`, `restricted_unique_range_values(`wt`,`x_i`,`x_j`,`y_i`,`y_j`)`.

Suffix Arrays (CSA=IV+WT)

Compressed suffix arrays use CIVs or WTs to represent the suffix arrays (SA), its inverse (ISA), BWT, Ψ, and LF. CSAs can be built over byte and integer alphabets.

Class	*Description*
`csa_bitcompressed`	Based on SA and ISA stored in a IV.
`csa_sada`	Based on Ψ stored in a CIV.
`csa_wt`	Based on the BWT stored in a WT.

Public methods: `operator[`i`]`, `size()`, `begin()`, `end()`.
Public members: `isa`, `bwt`, `lf`, `psi`, `text`, `L`, `F`, `C`, `char2comp`, `comp2char`, `sigma`.
Policy classes: alphabet strategy (e.g. `byte_alphabet`, `succinct_byte_alphabet`, `int_alphabet`) and SA sampling strategy (e.g. `sa_order_sa_sampling`, `text_order_sa_sampling`)

Longest Common Prefix (LCP) Arrays

Class	*Description*
`lcp_bitcompressed`	Values in a `int_vector<>`.
`lcp_dac`	Direct accessible codes used.
`lcp_byte`	Small values in a byte; 2 words per large.
`lcp_wt`	Small values in a WT; 1 word per large.
`lcp_vlc`	Values in a `vlc_vector`.
`lcp_support_sada`	Values stored permuted. CSA needed.
`lcp_support_tree`	Only depths of CST inner nodes stored.
`lcp_support_tree2`	+ large values are sampled using LF.

Public methods: `operator[`i`]`, `size()`, `begin()`, `end()`

Balanced Parentheses Supports (BPS)

We represent a sequence of parentheses as a `bit_vector`. An opening/closing parenthesis corresponds to `1`/`0`.

Class	*Description*
`bp_support_g`	Two-level pioneer structure.
`bp_support_gg`	Multi-level pioneer structure.
`bp_support_sada`	Min-max-tree over excess sequence.

Public methods: `find_open(`i`)`, `find_close(`i`)`, `enclose(`i`)`, `double_enclose(`i`,`j`)`, `excess(`i`)`, `rr_enclose(`i`,`j`)`, `rank(`i`)`[4], `select(`i`)`.
Call `util::init_support(bps,bv)` to initialize a BPS `bps` to `bit_vector bv`.

Suffix Trees (CST=CSA+LCP+BPS)

A CST can be parametrized by any combination of CSA ,LCP, and BPS. The operation of each part can still be accessed through member varaibles. The additional operations are listed below. CSTs can be built for byte or integer alphabets.

Class	*Description*
`cst_sada`	Represents a node as position in BPS. Navigational operations are fast (they are directly translated in BPS operations on the DFS-BPS). Space: $4n+o(n)+\|CSA\|+\|LCP\|$ bits.
`cst_sct3`	Represents nodes as intervals. Fast construction, but slower navigational operations. Space: $3n+o(n)+\|CSA\|+\|LCP\|$

Public types: `node_type`. In the following let v and w be nodes and i, d, lb, rb integers.
Public methods: `size()`, `nodes()`, `root()`, `begin()`, `end()`, `begin_bottom_up()`, `end_bottom_up`, `size(`v`)`, `is_leaf(`v`)`, `degree(`v`)`, `depth(`v`)`, `node_depth(`v`)`, `edge(`v`,` d`)`, `lb(`v`)`, `rb(`v`)`, `id(`v`)`, `inv_id(`i`)`, `sn(`v`)`, `select_leaf(`i`)`, `node(`lb`,` rb`)`, `parent(`v`)`, `sibling(`v`)`, `lca(`v`,` w`)`, `select_child(`v`,` i`)`, `child(`v`,` c`)`, `children(`v`)`, `sl(`v`)`, `wl(`v`,` c`)`, `leftmost_leaf(`v`)`, `rightmost_leaf(`v`)`
Public members: `csa`, `lcp`.
The traversal example shows how to use the DFS-iterator.

Range Min/Max Query (RMQ)

A RMQ `rmq` can be used to determine the position of the minimum value[5] in an arbitrary subrange $[i, j]$ of an preprocessed vector `v`. Operator `operator(`i`,`j`)` returns $x = \min\{r \mid r \in [i,j] \wedge \text{v}[r] \leq \text{v}[k]\ \forall k \in [i,j]\}$

Class	*Space*	*Time*
`rmq_support_sparse_table`	$n \log^2 n$	$\mathcal{O}(1)$
`rmq_succint_sada`	$4n + o(n)$	$\mathcal{O}(1)$
`rmq_succint_sct`	$2n + o(n)$	$\mathcal{O}(1)$

Constructing data structures

Let `o` be a WT-, CSA-, or CST-object. Object `o` is built with `construct(o,file,num_bytes=0)` from a sequence stored in `file`. File is interpreted dependent on the value of `num_bytes`:

Value	*File interpreted as*
`num_bytes=0`	serialized `int_vector<>`.
`num_bytes=1`	byte sequence of length `util::file_size(file)`.
`num_bytes=2`	16-bit word sequence.
`num_bytes=4`	32-bit word sequence.
`num_bytes=8`	64-bit word sequence.
`num_bytes=d`	Parse decimal numbers.

Note: `construct` writes/reads data to/from disk during construction. Accessing disk for small instances is a considerable overhead. `construct_im(o,data,num_bytes=0)` will build `o` using only main memory. Have a look at this handy tool for an example.

Configuring construction

The locations and names of the intermediate files can be configured by a `cache_config` object. It is constructed by `cache_config(del,tmp_dir,id, map)` where `del` is a boolean variable which specifies if the intermediate files should be deleted after construction, `tmp_dir` is a path to the directory

The core of the library: `int_vector`

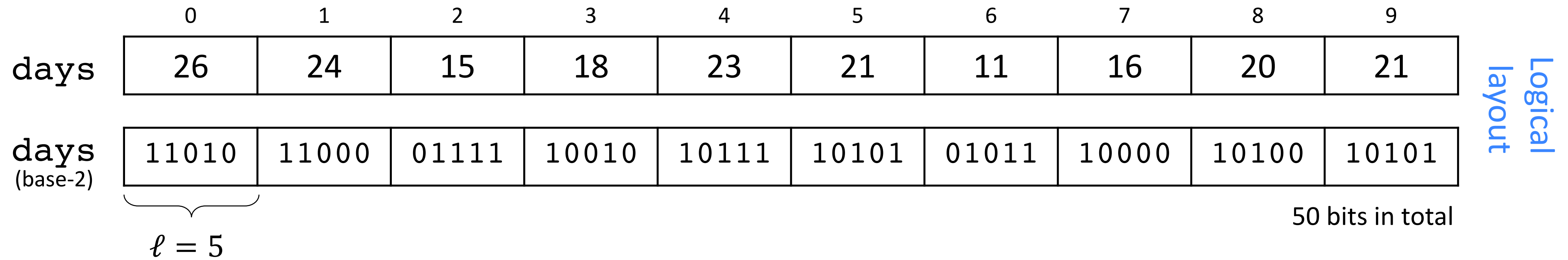
- Say you need to store the number of days per month worked by each employee of a company
- The `<cstdint>` header defines the (portable) types `int`n`_t` and `uint`n`_t` for $n = 8, 16, 32, 64$
- So you would normally declare something like

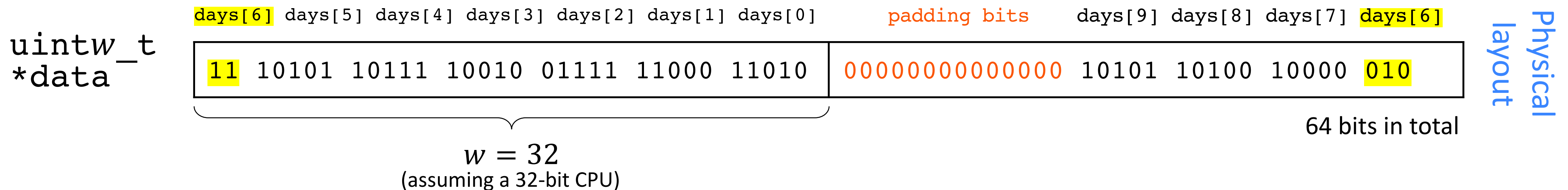
```
std::vector<uint8_t> days(n_employees);
```

- What is the space? 1 byte per employee
- But only 5 bits are needed (31 = 11111_2)
- We are wasting 8/5-1 = 60% of extra space per employee!

The core of the library: `int_vector`

- `int_vector<`ℓ`>` is a container of integers of fixed bit-width ℓ





$= 1^\ell$

```
j = i*ℓ
days[i] = (data[j/w] >> (j % w)) & ((1 << ℓ) - 1)
```

Exercise: what is the formula when `days[i]` spans two words?

Solution to the exercise

```
j = i * ℓ

offset = j % w

if (offset + ℓ <= w) // the integer spans one word
  return (data[j/w] >> (j % w)) & ((1 << ℓ ) - 1)
else                 // the integer spans two words
  return (data[j/w] >> (j % w)) |
         (data[j/w+1] & ((1 << ((offset+ℓ) % w)) - 1)) << (w-offset)
```

Example 1: `int_vector<`ℓ`>`, bit-width ℓ set at compile-time

```
#include <iostream>
#include <sdsl/vectors.hpp>

int main() {
    sdsl::int_vector<5> v = {26, 24, 15, 18, 23, 21, 11, 16, 20, 21};
    std::cout << v << std::endl;
    v[3] = 10;
    v[1] = 194; // == 0b11000010
    std::cout << v << std::endl;
    std::cout << v.bit_size() << std::endl;
    std::cout << sdsl::size_in_bytes(v) << std::endl;
    return 0;
}
```

Output

```
26 24 15 18 23 21 11 16 20 21
26 2 15 10 23 21 11 16 20 21
50
16
```

16 ← 8 byte for the compressed v, 8 bytes for `v.size()`

Example 2: `int_vector<>`, bit-width set at run-time

```
#include <iostream>
#include <sdsl/vectors.hpp>

int main() {
    sdsl::int_vector<> v = {26, 24, 15, 18, 23, 21, 11, 16, 20, 21};
    std::cout << sdsl::size_in_bytes(v) << std::endl;
    sdsl::util::bit_compress(v);
    std::cout << v << std::endl;
    std::cout << (int) v.width() << std::endl;
    std::cout << sdsl::size_in_bytes(v) << std::endl;
    return 0;
}
```

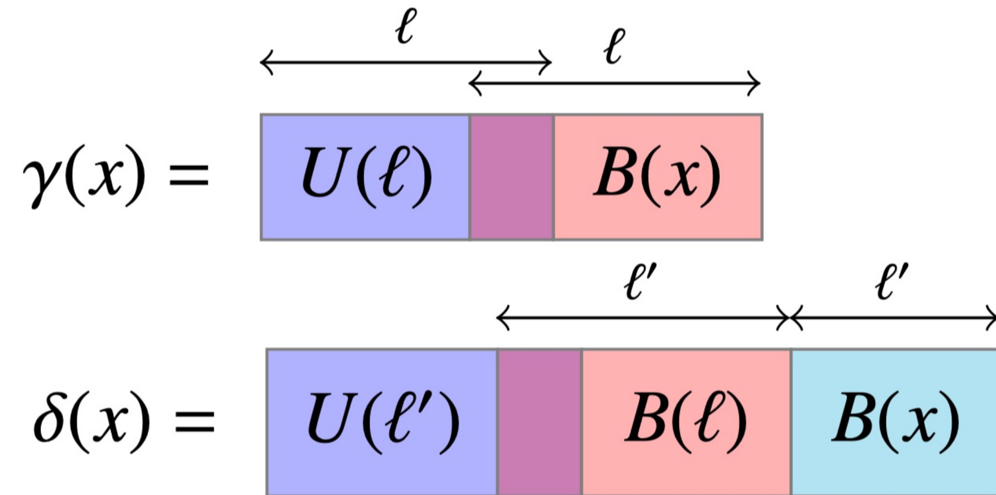
Output

```
89
26 24 15 18 23 21 11 16 20 21
5
17
```

89 ← 80 bytes for the 10 ints, 8 bytes for `v.size()`, 1 byte for the bit-width

17 ← 8 byte for the compressed v, 8 bytes for `v.size()`, 1 byte for the bit-width

Integer coders



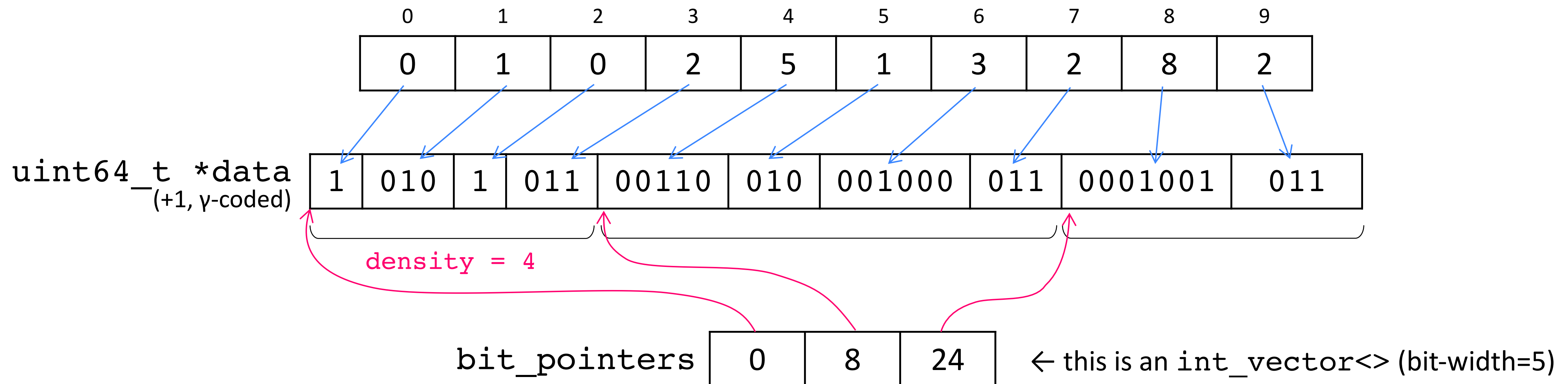
$\gamma(9) =$ 0001001

$\delta(14) =$ 00010011110

- `sdsl::coder::elias_gamma`
- `sdsl::coder::elias_delta`

Compressed integer vectors: `vlc_vector`

- Stores Elias-γ and -δ codes contiguously
- Zeros are permitted (internally, it uses $x + 1$ instead of x)
- How to implement `vlc_vector::operator[]`?



`density` is a *space-time trade-off parameter*: decrease it for faster random access but higher space usage

Example 3: `vlc_vector<coder,density>`

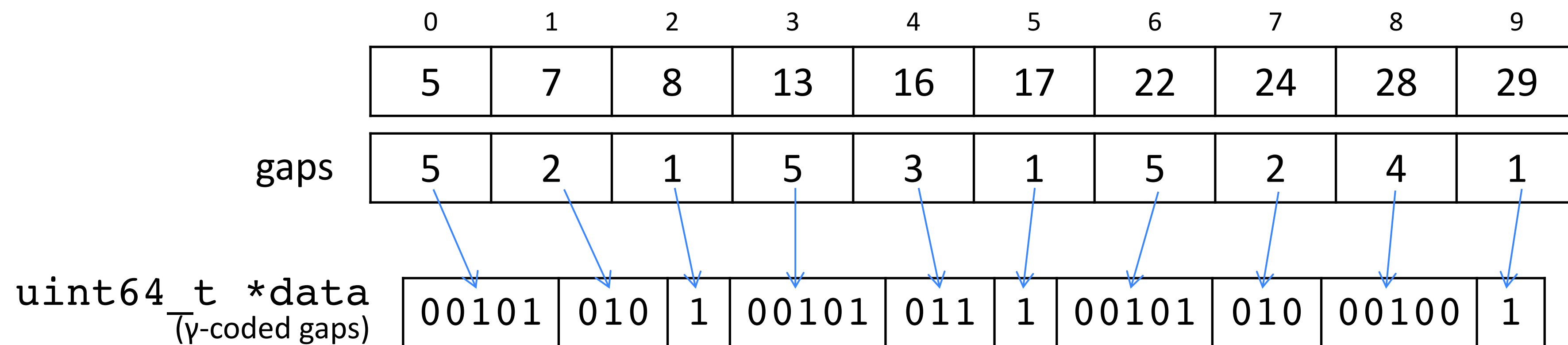
```
#include <iostream>
#include <sdsl/vectors.hpp>

int main() {
    sdsl::int_vector<> v(10 * (1 << 20));
    v[100] = 1ULL << 63;
    sdsl::util::bit_compress(v);
    std::cout << size_in_mega_bytes(v) << std::endl;
    sdsl::vlc_vector<sdsl::coder::elias_delta, 128> vlc(v);
    std::cout << size_in_mega_bytes(vlc) << std::endl;
    return 0;
}
```

Output

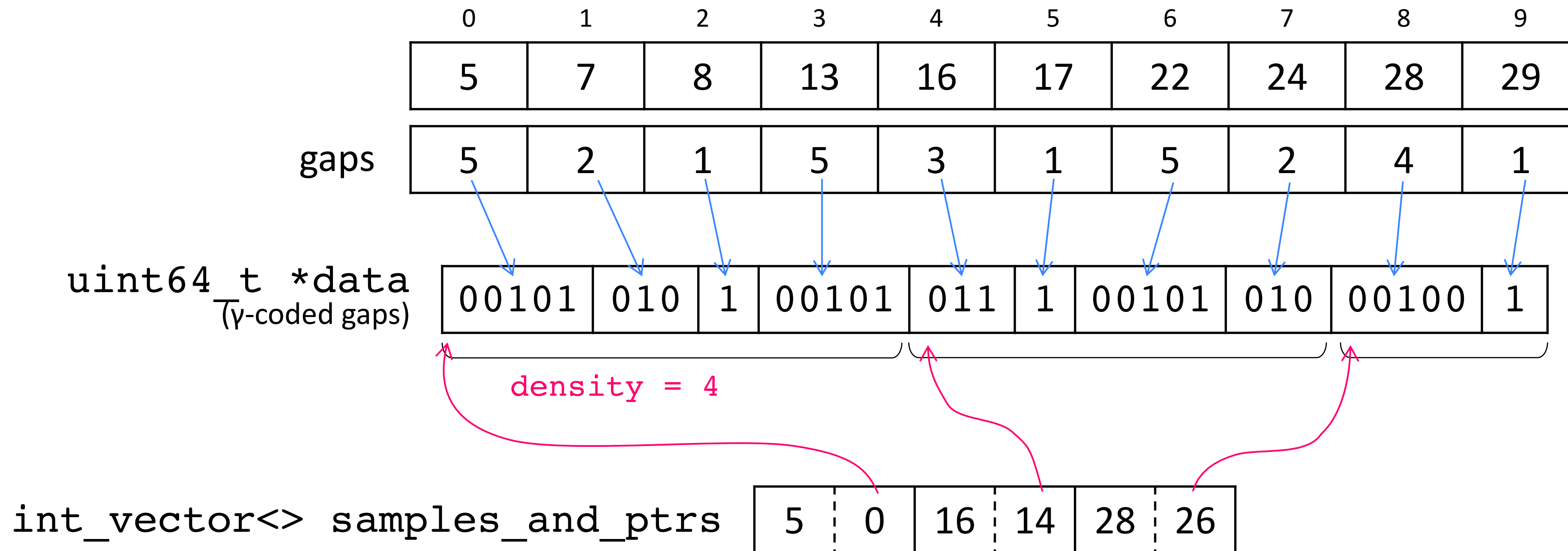
```
80
1.48442
```

Compressed integer vectors: `enc_vector`

- Say you have increasing integers $x_1, x_2, x_3 \ \dots, x_n$
  A postings list (search engines), an adjacency list (graphs)
- γ/δ-code the gaps $x_1, x_2 - x_1, x_3 - x_2, \dots, x_n - x_{n-1}$
- How to implement `vlc_vector::operator[]`?

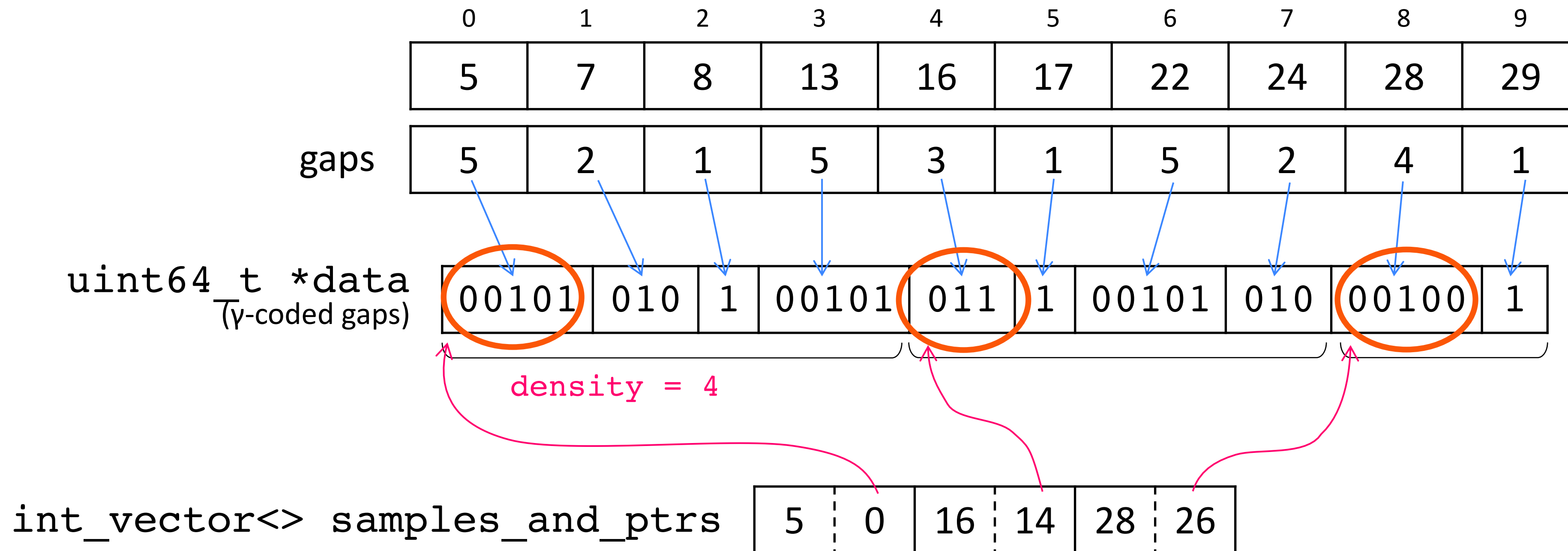


Compressed integer vectors: `enc_vector`

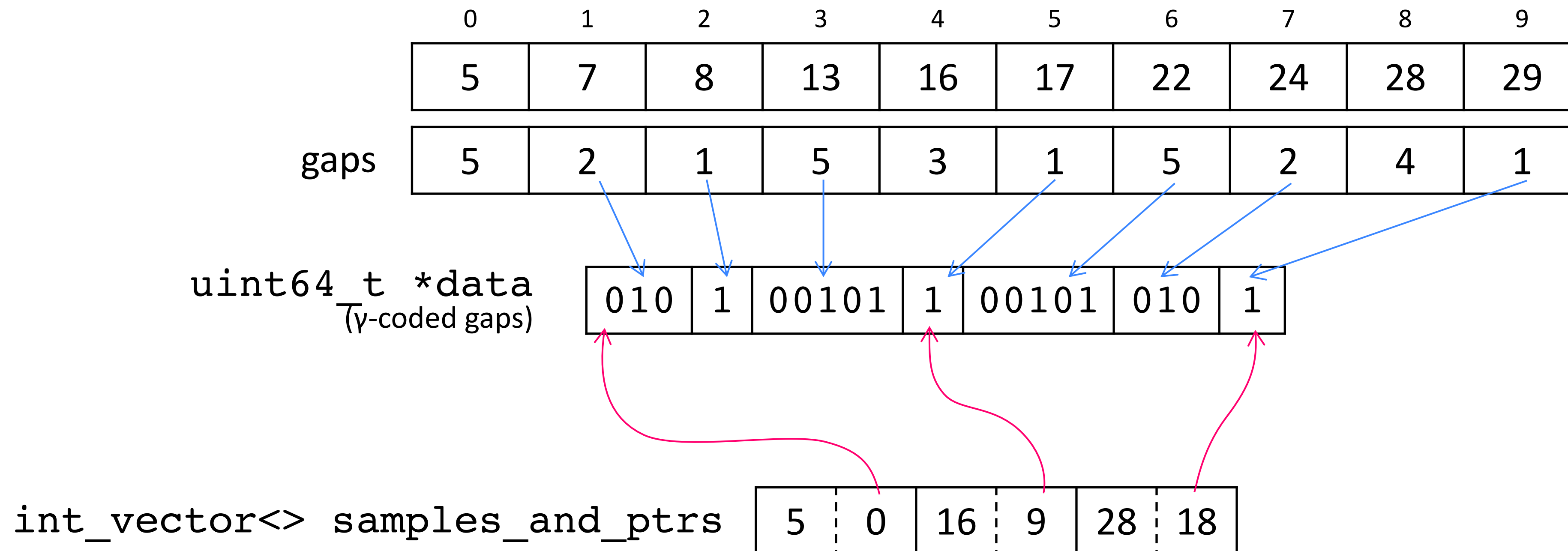


Compressed integer vectors: **enc_vector**

- There is some redundancy here…



Compressed integer vectors: `enc_vector`



`density` is a *space-time trade-off parameter*: decrease it for faster random access but higher space usage

Example 4: `enc_vector<coder,density>`

```
#include <random>
#include <iostream>
#include <sdsl/vectors.hpp>

sdsl::int_vector<> random_data(size_t n, uint64_t u) { // Algorithm 3.4 of the notes
    if (n > u)
        throw std::invalid_argument("n > u");
    std::mt19937 rnd(std::random_device{}());
    sdsl::int_vector<> out(n);
    for (size_t j = 1, s = 0; j <= u && s < n; ++j)
        if (rnd() % (u - j + 1) <  n - s)
            out[s++] = j;
    return out;
}

...
```

Example 4: `enc_vector<coder,density>`

```
#include <random>
#include <iostream>
#include <sdsl/vectors.hpp>

sdsl::int_vector<> random_data(size_t n, uint64_t u) {...}

int main() {
    auto data = random_data(10 * (1 << 20), 1 << 24);
    std::cout << sdsl::size_in_mega_bytes(data) << std::endl;

    sdsl::vlc_vector<sdsl::coder::elias_delta, 128> vlc(data);
    std::cout << sdsl::size_in_mega_bytes(vlc) << std::endl;

    sdsl::enc_vector<sdsl::coder::elias_delta, 128> enc_delta(data);
    std::cout << sdsl::size_in_mega_bytes(enc_delta) << std::endl;


    return 0;
}
```

What if we use `sdsl::coder::elias_gamma`?

Output

```
80
39.0282
3.19339
```

Example 4(bis): enc_vector<coder,density>

```
#include <random>
#include <iostream>
#include <sdsl/vectors.hpp>

sdsl::int_vector<> random_data(size_t n, uint64_t u) {...}

int main() {
    auto data = random_data(10 * (1 << 20), 1 << 24);
    std::cout << sdsl::size_in_mega_bytes(data) << std::endl;

    sdsl::vlc_vector<sdsl::coder::elias_delta, 128> vlc(data);
    std::cout << sdsl::size_in_mega_bytes(vlc) << std::endl;

    sdsl::enc_vector<sdsl::coder::elias_delta, 128> enc_delta(data);
    std::cout << sdsl::size_in_mega_bytes(enc_delta) << std::endl;

    sdsl::enc_vector<sdsl::coder::elias_gamma, 128> enc_gamma(data);
    std::cout << sdsl::size_in_mega_bytes(enc_gamma) << std::endl;
    return 0;
}
```

Output

```
80
39.0282
3.19339
2.79236
```

Plain bitvectors (`bit_vector`)

- A specialised version of `int_vector<1>`
- Mutable
  - `b[i] = 1`
  - `b.flip()`
- Bitwise operations between bitvectors `b1 |= b2` (also `&=`, `^=`)
- Auxiliary data structures extend a (static) bitvector functionality

Class	+Bits	Time
rank_support_v	$0.25n$	$O(1)$
rank_support_scan	64	$O(n)$
select_support_mcl	$\leq 0.2n$	$O(1)$
select_support_scan	64	$O(n)$

Example 5: `bit_vector`

```
#include <iostream>
#include <sdsl/bit_vectors.hpp>

int main() {
    sdsl::bit_vector b1 = {1, 1, 0, 1, 0, 0, 1};
    sdsl::bit_vector b2 = {0, 0, 1, 1, 0, 1, 0};
    b1 |= b2;
    b1.flip();
    std::cout << b1 << std::endl;

    sdsl::bit_vector b(80 * (1 << 20), 0);
    for (size_t i = 0; i < b.size(); i += 100)
        b[i] = 1;
    std::cout << sdsl::size_in_mega_bytes(b) << std::endl;
}
```

Output

```
0000100
10
```

Recap

- **Integer vectors**
  - int_vector<>, bit-width set at run-time, starts with 64 bits by default
  - int_vector<ℓ>, bit-width ℓ fixed at compile-time
  - Implementation of random access
- **Compressed integer vectors**
  - vlc_vector<coder,density>, vector of γ/δ-coded integers
    - Implementation of random access via bit pointers
  - enc_vector<coder,density>, vector of γ/δ-coded gaps between ints
    - Implementation of random access via bit pointers and samples
- **Plain bitvectors**
  - bit_vector = int_vector<1>
  - Bitwise AND, OR, XOR
  - Set individual bit, flip all bits

Example 6: `bit_vector` with rank/select

```
#include <iostream>
#include <sdsl/bit_vectors.hpp>

int main() {
    sdsl::bit_vector b = {0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1};
    sdsl::rank_support_v<1> b_rank(&b);
    for (size_t i = 1; i <= b.size(); ++i)
        std::cout << b_rank(i) << " ";
    std::cout << std::endl;

    sdsl::select_support_mcl<1> b_select(&b);
    size_t ones = b_rank(b.size());
    for (size_t i = 1; i <= ones; ++i)
        std::cout << b_select(i) << " ";
}
```

counts #1s in b[0, i)

Output

```
b = 0 1 0 1 1 1 0 0 0 1 1
    0 1 1 2 3 4 4 4 4 5 6
    1 3 4 5 9 10
```

Elias-Fano compressed bitvectors (**sd_vector**)

- Construct from a bitvector

```
sdsl::bit_vector b = {0,1,0,1,1,1,0,0,0,1,1};
sdsl::sd_vector<> ef1(b);
```

- Construct from a vector of increasing integers

```
sdsl::int_vector<> v = {1, 3, 4, 5, 9, 10};
sdsl::sd_vector<> ef2(v.begin(), v.end());
```

- `ef1 == ef2`

Example 7: Elias-Fano

```cpp
#include <iostream>
#include <sdsl/bit_vectors.hpp>

sdsl::int_vector<> random_data(size_t n, uint64_t u) {...}

int main() {
    auto data = random_data(10 * (1 << 20), 1 << 25);
    sdsl::sd_vector<> ef(data.begin(), data.end());
    std::cout << sdsl::size_in_mega_bytes(data) << std::endl;
    std::cout << sdsl::size_in_mega_bytes(ef) << std::endl;
    return 0;
}
```

Output

```
80
5.25673
```

Example 8: Elias-Fano internals

```
#include <iostream>
#include <sdsl/bit_vectors.hpp>

int main() {
    sdsl::int_vector<> v = {1, 4, 7, 18, 24, 26, 30, 31};
    sdsl::sd_vector<> ef(v.begin(), v.end());
    uint64_t u = v[v.size() - 1] + 1;
    uint64_t bpi = std::ceil(std::log2(u));
    std::cout << "Data = " << v << std::endl
              << "Bitvector = " << ef << std::endl
              << "u = " << u << std::endl
              << "Bits per integer = " << bpi << std::endl
              << "Bits in the high part = " << bpi - ef.wl << std::endl
              << "Bits in the low part = " << (int) ef.wl << std::endl
              << "L = " << ef.low << " (in decimal)" << std::endl
              << "H = " << ef.high << std::endl;
    return 0;
}
```

<u>Technical note</u>: wrt the lecture notes, SDSL uses a different splitting point for the low/high parts (a difference of ±1); but in this example the encodings match

Output

```
Data = 1 4 7 18 24 26 30 31
Bitvector = 01001001000000000001000001010001
u = 32
Bits per integer = 5
Bits in the high part = 3
Bits in the low part = 2
L = 1 0 3 2 0 2 2 3 (in decimal)
H = 1011000100110110000000000
```

	high	low
1 =	000	01
4 =	001	00
7 =	001	11
18 =	100	10
24 =	110	00
26 =	110	10
30 =	111	10
31 =	111	11

$h = \lceil \log n \rceil = 3$

$b = \lceil \log u \rceil = 5$

$\ell = b - h = 2$

$L =$ 0100111000101011

bucket	0	1	2	3	4	5	6	7
$H =$	10	110	0	0	10	0	110	110

Example of §11.6

Example 9: Elias-Fano access and query

```cpp
#include <iostream>
#include <sdsl/bit_vectors.hpp>

int main() {
    sdsl::int_vector<> v = {1, 4, 7, 18, 24, 26, 30, 31};
    sdsl::sd_vector<> ef(v.begin(), v.end());
    std::cout << ef << std::endl;

    // Use as a bitvector with rank/select/operator[]
    sdsl::rank_support_sd<> ef_rank(&ef);
    sdsl::select_support_sd<> ef_select(&ef);
    std::cout << ef_rank(5) << std::endl;
    std::cout << ef_select(4) << std::endl;
    std::cout << ef[24] << std::endl; // access to the bit in pos 24

    // Use as an integer vector with access/nextGEQ given select/rank
    auto access = [&] (size_t i) { return ef_select(i + 1); };
    auto nextGEQ = [&] (uint64_t x) { return ef_select(ef_rank(x) + 1); };
    std::cout << access(5) << std::endl;
    std::cout << nextGEQ(4) << std::endl;
    std::cout << nextGEQ(27) << std::endl;
    return 0;
}
```

Output

```
01001001000000000001000001010011
2
18
1
26
4
30
```