UNIVERSITÉ
CÔTE D'AZUR

Master 1 Informatique

Software Engineering

Bit Packing for Integer Arrays

Software Engineering Project

Student:	Amir Benyahia
Instructor:	Jean-Charles Régin
Academic year:	2025–2026
University:	Université Côte d'Azur

November 2, 2025

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Objectives . . . 1

2 Software Design and Architecture **1**
- 2.1 Environment and Constraints . . . 1
- 2.2 Main Components . . . 1
- 2.3 Design Patterns . . . 1

3 Implementation Details **2**
- 3.1 Non-Spanning Bit Packing . . . 2
- 3.2 Spanning Bit Packing . . . 2
- 3.3 Overflow Bit Packing . . . 2

4 Benchmark and Performance Analysis **2**
- 4.1 Protocol . . . 2
- 4.2 Results (example run) . . . 2
- 4.3 Discussion . . . 3
- 4.4 When is compression worthwhile? . . . 3

5 Limits and Bonus (Negative Integers) **3**

6 Conclusion **3**

1 Introduction

1.1 Context

Transmitting integer arrays efficiently is a common need. When values are small, using full 32-bit integers wastes bandwidth. This report studies and implements *Bit Packing* to compress arrays while still supporting direct access to the i-th value after compression.

1.2 Objectives

We aim to:

- compute the minimum number of bits k needed to represent the maximum of the input,
- store each integer using exactly k bits,
- support direct access via `get(i)` without full decompression,
- implement two versions (non-spanning and spanning), plus an overflow variant,
- measure the performance of `compress`, `decompress` and `get`.

This report presents the design, implementation and performance evaluation of the system.

2 Software Design and Architecture

2.1 Environment and Constraints

Implementation is in Python 3. To emulate fixed 32-bit blocks, we use `array('I')` (unsigned 32-bit C integers). Because Python integers are arbitrary precision, we explicitly mask writes with `0xFFFFFFFF` where needed to avoid unintended overflow.

2.2 Main Components

The codebase is organized as follows:

- **Interface**: `IntegerCompressor` with `compress`, `decompress`, `get`,
- **Strategies**: `BitPackingNonSpanning`, `BitPackingSpanning`, `BitPackingOverflow`,
- **Factory**: `CompressorFactory` (selects the compressor by name),
- **CLI/Demo**: `main.py`,
- **Benchmark**: `benchmark.py` (uses `timeit`).

2.3 Design Patterns

We use two patterns:

- **Factory**: decouples client code from concrete implementations. Adding a new compressor only touches the factory.
- **Decorator**: `BitPackingOverflow` wraps a `BitPackingSpanning` compressor to add overflow handling without duplicating low-level bit packing logic.

3 Implementation Details

3.1 Non-Spanning Bit Packing

We compute $k =$ `max(data).bit_length()` (or 1 for an all-zero array). We then pack $\lfloor 32/k \rfloor$ elements per 32-bit slot. This wastes trailing bits but yields simple $O(1)$ indexing: an integer is at

$$\text{array\_index} = \left\lfloor \frac{i}{\text{elements\_per\_int}} \right\rfloor, \quad \text{bit\_offset} = (i \bmod \text{elements\_per\_int}) \times k.$$

3.2 Spanning Bit Packing

We maintain a global `bit_cursor` and write each k-bit value possibly across two consecutive 32-bit integers. Before OR-ing into the `array('I')`, we mask with `0xFFFFFFFF` to keep only the lower 32 bits. The `get(i)` function reconstructs a k-bit value by reading one or two 32-bit cells and masking.

3.3 Overflow Bit Packing

We introduce a smaller width k' for the *main* area and reserve the maximum value $(1 \ll k') - 1$ as a sentinel. Values $\geq$ sentinel go to an *overflow area* stored separately. During `get(i)`, if the main value is the sentinel, we count how many sentinels appear before i to index the overflow array.

Note on complexity. The naive `get(i)` for overflow is $O(n)$ in the worst case (counting prior sentinels). A simple improvement is to precompute a prefix count or a list of overflow positions during `compress` to obtain $O(1)$ average-time access. This is left as future work.

Main bits default. In the CLI (`main.py`), `-main-bits` defaults to 3. In the factory, if no argument is given, the default is 8. For benchmarking and examples, we explicitly set $k' = 10$ to avoid ambiguity.

4 Benchmark and Performance Analysis

4.1 Protocol

The script `benchmark.py` uses `timeit` with:

- dataset size: $N = 10{,}000$ integers,
- $k' = 10$ for overflow,
- roughly 5% overflow values (randomly selected),
- functions measured: `compress` (10 runs), `decompress` (100 runs), `get(i)` (1000 runs) with $i = N/2$.

4.2 Results (example run)

Times depend on hardware. The table below is illustrative; please refer to the script output for your machine.

Table 1: Example benchmark on a 10,000-integer dataset (average over runs).

Function	**Compressor**	**Average Time (s)**
`compress()`	`spanning`	0.0827
`compress()`	`overflow`	0.0828
`decompress()`	`spanning`	0.0053
`decompress()`	`overflow`	0.0054
`get(i)`	`spanning`	$< 10^{-6}$
`get(i)`	`overflow`	$< 10^{-6}$

4.3 Discussion

- **Compression**: spanning and overflow are close; splitting into main/overflow adds little cost.
- **Decompression**: similar as well; per-element overhead is small.
- **Access**: `get(i)` is effectively instantaneous at this scale; however, overflow can degrade if many sentinels occur and we do not precompute indices (future optimization).

4.4 When is compression worthwhile?

Let S_{orig} be the uncompressed size (bytes), S_{comp} the compressed size, B the network throughput (bytes/s), and T_{comp} the compression time. A simple model for total time with compression is:

$$T_{\text{with}} = T_{\text{comp}} + \frac{S_{\text{comp}}}{B}, \quad T_{\text{without}} = \frac{S_{\text{orig}}}{B}.$$

Compression is beneficial if $T_{\text{with}} < T_{\text{without}}$, i.e.

$$T_{\text{comp}} < \frac{S_{\text{orig}} - S_{\text{comp}}}{B}.$$

If we include client-side `decompress()` in the model, just replace T_{comp} by $T_{\text{comp}} + T_{\text{decomp}}$.

5 Limits and Bonus (Negative Integers)

Current implementation assumes non-negative 32-bit integers. Handling negatives can be added:

- **Sign bit**: store $|x|$ with one extra bit for the sign,
- **ZigZag**: map $\mathbb{Z} \to \mathbb{N}$ (e.g., $0 \mapsto 0$, $-1 \mapsto 1$, $1 \mapsto 2$...) then compress as unsigned.

6 Conclusion

We implemented and compared three bit-packing strategies fulfilling the project brief: *non-spanning*, *spanning* and *overflow*. The architecture (Factory + Decorator) keeps the code modular and easy to extend. In practice:

- **Non-spanning** is the simplest to reason about. It provides very cheap indexing (one division and one shift) but wastes the remaining bits in each 32-bit word, so the compression ratio is weaker when k does not divide 32.
- **Spanning** is a strong default: it achieves near-optimal packing for a fixed k while keeping `get(i)` in $O(1)$ (read one or two words, then mask/shift). The implementation is slightly more involved but pays off in space.
- **Overflow** is attractive on heavy-tailed data: most values are small and a few are very large. With a good choice of k' (we used $k' = 10$ in the benchmark), the main area stays compact and only outliers spill to the overflow area. The current `get(i)` is correct but can degrade toward $O(n)$ if sentinels are frequent; caching prefix counts or positions fixes this in future work.

Our measurements with `timeit` show that the CPU cost of (de)compression is small compared to typical transmission times when the compression ratio is significant. The threshold model $\left(T_{\text{comp}} < \frac{S_{\text{orig}} - S_{\text{comp}}}{B}\right.$, optionally adding $\left.T_{\text{decomp}}\right)$ gives a clear decision rule: faster networks or poor compression ratios reduce the benefit, while slower links or better ratios amplify it.

Practical guidance. Use **spanning** by default. Prefer **non-spanning** if implementation simplicity or strict alignment constraints matter more than space. Switch to **overflow** when a small k' shrinks the main area and only a small fraction of values (e.g., $\leq 5\% - 10\%$) overflow; pick k' from a quick histogram to keep the sentinel rate low. By design, values $\geq (1 \ll k') - 1$ are treated as overflow (sentinel semantics is inclusive), which should be documented for users.