UNIVERSITÉ
CÔTE D'AZUR

Master 1 Informatique

Software Engineering

# Bit Packing for Integer Arrays

Software Engineering Project


**Student:** Amir Benyahia
**Academic year:** 2025–2026
**University:** Université Côte d'Azur


November 2, 2025

# Contents

## List of Figures

# 1 Introduction

## 1.1 Context and Motivation

In many areas of computer science, data storage, network transmission, databases, video games, and more, it is common to manipulate large arrays of integers. However, most of the time, these integers are stored using 32 bits, even when their actual values are much smaller. This leads to wasted memory and bandwidth, especially when arrays are large or transmitted over slow networks.

This project, carried out as part of the Software Engineering course, aims to explore and implement *bit packing* techniques to efficiently compress these arrays while maintaining fast access to each element.

**Why this topic?** Because it perfectly illustrates the balance between performance, implementation simplicity, and real world constraints (direct access, handling extreme values, etc.). It is also an excellent exercise in software design (interfaces, design patterns, testing).

**Real-world applications.** Bit packing is widely used in:

- **Search engines**: storing document IDs and posting lists compactly (e.g., Google, ElasticSearch)
- **Databases**: columnar storage formats (e.g., Apache Parquet, ORC)
- **Game development**: efficient storage of tile maps, entity attributes
- **IoT and embedded systems**: minimizing memory footprint on resource-constrained devices
- **Network protocols**: reducing packet sizes for faster transmission

## 1.2 Project Objectives

- Reduce the memory footprint of non-negative integer arrays.
- Enable direct access to any compressed element (without full decompression).
- Handle cases where some elements are much larger than the majority (outliers).
- Compare multiple compression strategies (non-spanning, spanning, overflow).
- Measure performance (time, compression ratio) and discuss trade-offs.
- Provide a clear, extensible, and well-documented architecture.

## 1.3 Report Structure

After this introduction, the report presents a needs analysis, use cases, software architecture, implementation details, tests and validation, and finally a conclusion.

# 2 Needs Analysis and Use Cases

## 2.1 Problem Statement

How can we store or transmit large integer arrays efficiently, while guaranteeing fast access to each value and without losing information?

**Concrete example.** A game server must send the scores of thousands of players at each tick. Scores range from 0 to 5000 but are stored in 32 bits. Can we do better?

Let's analyze the memory waste:

- **Standard storage:** 10,000 integers $\times$ 4 bytes = 40,000 bytes
- **Required bits:** $\lceil \log_2(5000) \rceil = 13$ bits per value
- **Theoretical minimum:** 10,000 $\times$ 13 bits = 130,000 bits = 16,250 bytes
- **Potential savings:** 59.4% reduction!

## 2.2 Functional Requirements

- **Compress:** Transform an array of non-negative integers into a compact representation
- **Decompress:** Restore the original array identically (lossless compression)
- **Random access:** Retrieve any element $i$ directly, without decompressing the whole array
- **Handle outliers:** Manage cases where a few values are much larger than the rest efficiently

## 2.3 Non-Functional Requirements

- **Implementation:** Python 3.10+, clean and commented code
- **Architecture:** Modular design (interfaces, factory pattern, extensibility)
- **Performance:** Measure and analyze time and space complexity
- **Testing:** Comprehensive test suite with edge cases
- **Documentation:** Clear README, inline comments, and this report

## 2.4 Use Case Diagram

## 2.5 Detailed Use Case: Compress Array

- **Actor:** User/Application
- **Precondition:** Valid array of non-negative integers
- **Flow:**
  1. User selects compression algorithm (non-spanning, spanning, or overflow)
  2. System determines the minimum bits required ($k$ = max value bit length)

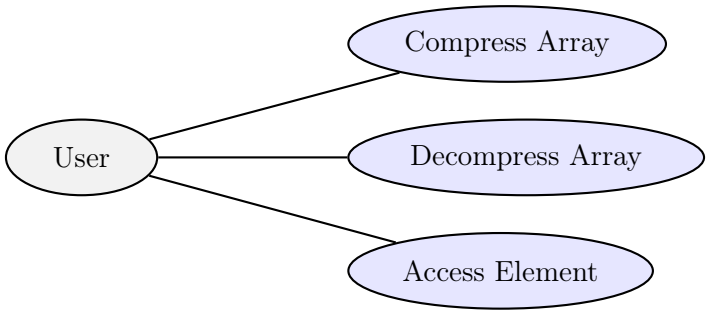


Figure 1: Primary use cases: compress an array, decompress it, or access a single element directly.

3. System packs values into 32-bit integers
4. System returns compressed data structure

- **Postcondition:** Array is compressed; original can be restored
- **Alternative flow:** Empty array → return empty compressed structure

# 3 Software Architecture

## 3.1 Overview

The project is structured around the following main components, following SOLID principles and design patterns:

- **IntegerCompressor** (interface): Abstract base class defining the compression API
- **BitPackingNonSpanning**: Packs values without crossing 32-bit boundaries
- **BitPackingSpanning**: Allows values to span across 32-bit boundaries
- **BitPackingOverflow**: Handles outliers by splitting data into main and overflow areas
- **CompressorFactory**: Factory to instantiate compressors by name
- **main.py**: CLI interface and test harness
- **benchmark.py**: Performance measurement script

## 3.2 Class Diagram

**Design rationale:**

- **Solid arrows**: Inheritance (all compressors implement IntegerCompressor)
- **Dashed arrow**: Factory pattern (CompressorFactory creates instances)
- **Dotted arrow**: Decorator pattern (BitPackingOverflow wraps BitPackingSpanning)

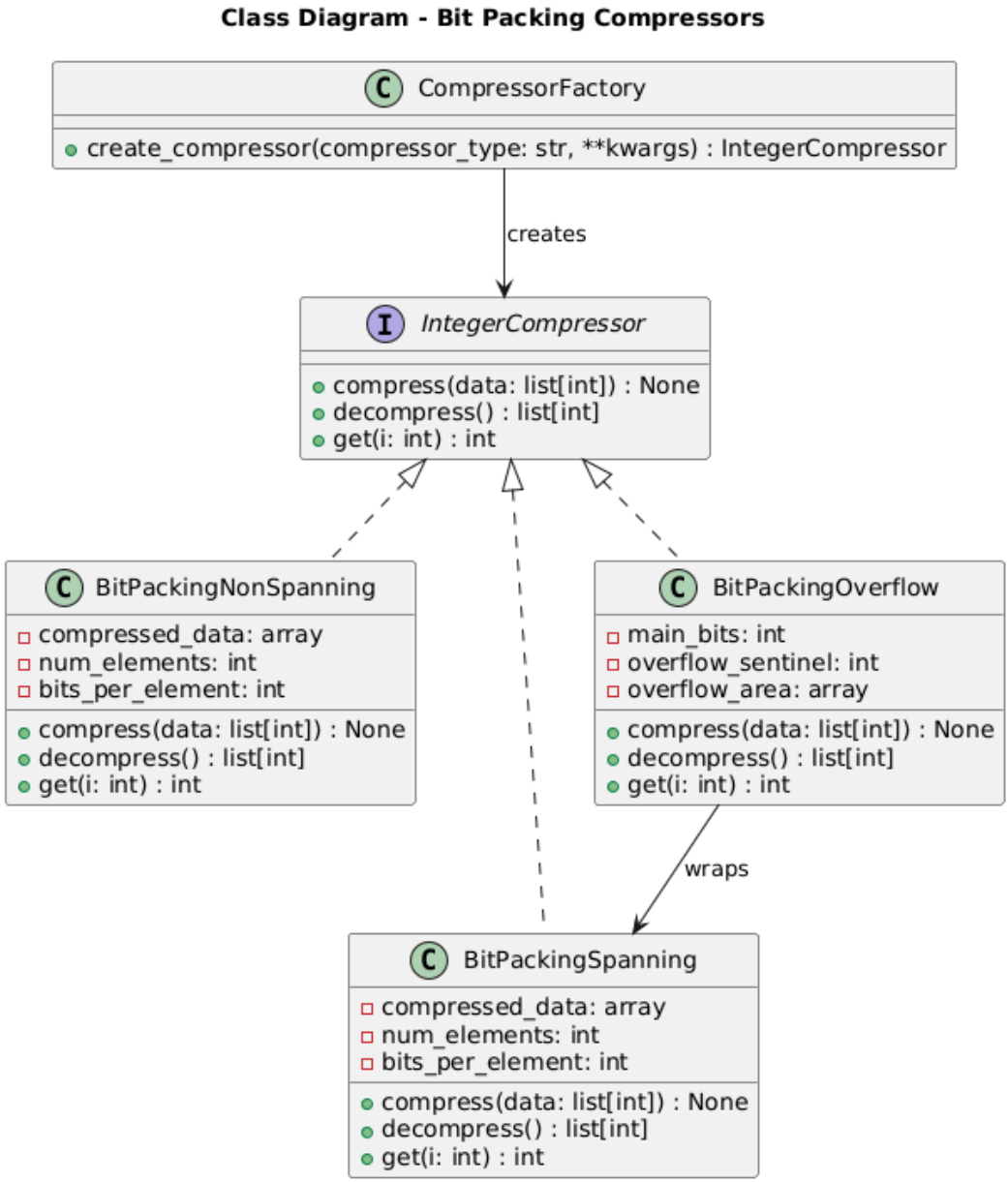


Figure 2: Detailed class diagram showing attributes, methods, design patterns (Factory, Decorator), and relationships between all components of the compression system.

## 3.3 Design Patterns

### 3.3.1 Factory Pattern

The `CompressorFactory` decouples client code from concrete implementations. This provides several benefits:

- **Extensibility**: Adding a new compressor only requires modifying the factory
- **Type safety**: Factory ensures correct initialization parameters
- **Flexibility**: Clients can select compressors by name (string) at runtime

**Code example from compressor_factory.py:**

```python
class CompressorFactory:
    @staticmethod
    def create_compressor(compressor_type: str, **kwargs):
        if compressor_type == "non_spanning":
            return BitPackingNonSpanning()
        elif compressor_type == "spanning":
            return BitPackingSpanning()
        elif compressor_type == "overflow":
            main_bits = kwargs.get("main_bits", 8)
            return BitPackingOverflow(main_bits=main_bits)
        else:
            raise ValueError(f"Unknown type: '{compressor_type}'")
```

### 3.3.2 Decorator Pattern

`BitPackingOverflow` wraps a `BitPackingSpanning` instance to add overflow handling without duplicating bit-packing logic. This demonstrates:

- **Code reuse**: Overflow uses spanning's compress/decompress internally
- **Separation of concerns**: Overflow logic is isolated in one class
- **Composition over inheritance**: Overflow contains a spanning compressor

**Code example from bit_packing_overflow.py:**

```python
class BitPackingOverflow(IntegerCompressor):
    def __init__(self, main_bits: int):
        super().__init__()
        # Wrap a spanning compressor
        self.wrapped_compressor = BitPackingSpanning()
        self.main_bits = main_bits
        self.overflow_sentinel = (1 << main_bits) - 1
        self.overflow_area = array('I')
```

### 3.4 Sequence Diagram: Compression Flow

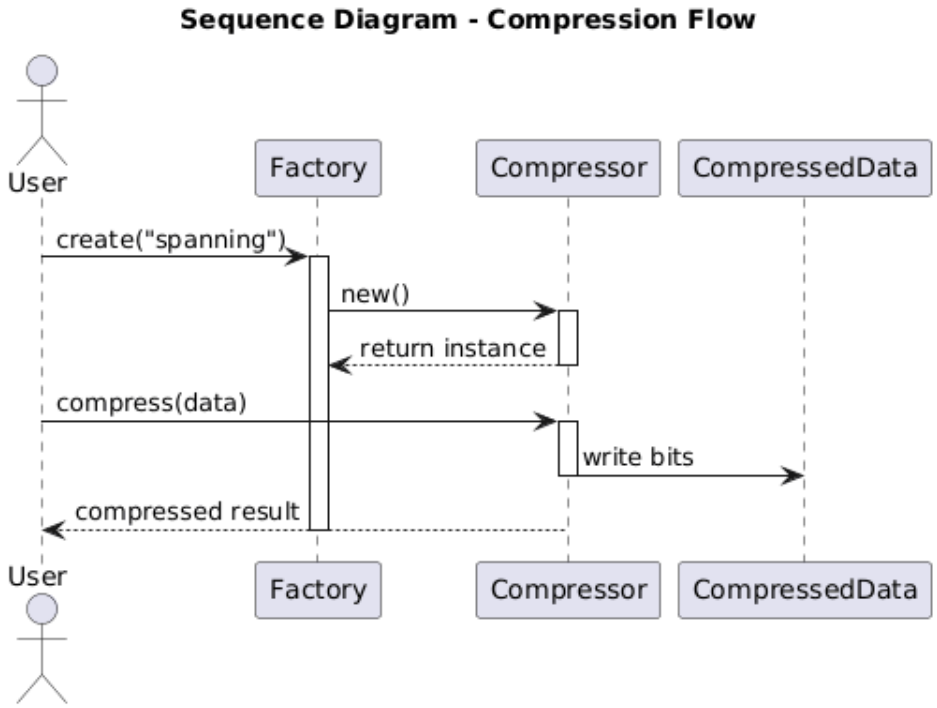


Figure 3: Sequence diagram showing the compression workflow factory instantiation, compression, and data storage.

## 4 Implementation Details

### 4.1 Core Data Structure

All compressors use Python's `array('I')` module to store 32-bit unsigned integers. This choice ensures:

- **Fixed width**: Each element is exactly 32 bits (4 bytes)
- **Efficient access**: Direct memory access without Python overhead
- **Portability**: Works consistently across platforms

**From integer_compressor.py:**

```
class IntegerCompressor(abc.ABC):
    def __init__(self):
        self.compressed_data = array('I')  # 32-bit unsigned
        self.num_elements = 0
        self.bits_per_element = 0  # This is 'k'
```

### 4.2 Non-Spanning Bit Packing

**Principle:** Each value is packed into a fixed slot within a single 32-bit integer. Values never cross boundaries.

**Advantages:**

- Simple and fast indexing: index = $\lfloor i/\text{elements\_per\_int} \rfloor$
- Predictable performance: $O(1)$ access time
- Easy to understand and debug

**Disadvantages:**

- Wastes trailing bits if $k$ does not evenly divide 32
- Lower compression ratio than spanning for small $k$

**Algorithm:**

1. Compute $k = \max(\text{data}).\text{bit\_length}()$ (or 1 if all zeros)
2. Compute elements per int: $n = \lfloor 32/k \rfloor$
3. For each value:
    - Calculate which 32-bit int: $\text{array\_index} = i/n$
    - Calculate bit offset: $\text{offset} = (i \bmod n) \times k$
    - Pack value: `data[array_index] |= (value « offset)`

**Example:** For $k = 3$ bits, we can fit $\lfloor 32/3 \rfloor = 10$ values per 32-bit integer. Packing [1,2,3,4,5,6,7,0,1,3]:

```
|001|010|011|100|101|110|111|000|001|011| (30 bits used, 2 wasted)
```

### 4.3 Spanning Bit Packing

**Principle:** Values can cross 32-bit boundaries, maximizing space efficiency.

**Advantages:**

- Maximum compression: uses exactly $n \times k$ bits
- No wasted space for any value of $k$
- Still provides $O(1)$ random access

**Disadvantages:**

- More complex implementation (handle spanning logic)
- Slightly slower access due to potential two-read operations

**Algorithm:**

1. Compute $k$ as before
2. For each value at position $i$:
   - Global bit position: bit_cursor $= i \times k$
   - Array index: array_index = bit_cursor/32
   - Bit offset: bit_offset = bit_cursor mod 32
   - Write first part: `data[index] |= (value « offset) & 0xFFFFFFFF`
   - If spanning, write remaining bits to next int

**Example:** For $k = 5$ bits, value 31 at bit position 27 spans two ints:

```
Int 0: [....... xxxxx]  (5 bits written)
Int 1: [0000 ......]   (0 bits written - fits exactly)

But at position 30:
Int 0: [..... xx...]   (2 bits of value)
Int 1: [xxx .......]   (3 remaining bits)
```

### 4.4 Overflow Bit Packing

**Principle:** Use a smaller bit width ($k'$) for the main area, and store outliers separately. A sentinel value marks overflow entries.

**Motivation:** In real data, most values are small, but a few outliers are large. Example: scores [0-100] with rare bonuses up to 10,000. Instead of using $k = 14$ bits for everyone, use $k' = 7$ for most and an overflow array for exceptions.

**Algorithm:**

1. Choose $k'$ (e.g., 10 bits)
2. Sentinel value: $S = 2^{k'} - 1 = 1023$
3. For each value $v$:
   - If $v < S$: store $v$ in main area
   - If $v \geq S$: store $S$ (sentinel) in main area, append $v$ to overflow array
4. Compress main area using spanning compressor
5. To retrieve `get(i)`:
   - Read main value at position $i$
   - If not sentinel: return value
   - If sentinel: count sentinels before $i$ to find overflow index

**Example:** Array [1, 2, 3, 1024, 4, 5, 2048] with $k' = 10$ (sentinel = 1023):

- Main area: [1, 2, 3, 1023, 4, 5, 1023]
- Overflow area: [1024, 2048]

**Trade-offs:**

- **Pro**: Excellent compression when outliers are rare
- **Con**: `get(i)` can be $O(n)$ if many sentinels (must count them)
- **Solution**: Could add an index cache to make it $O(1)$ (future work)

# 5 Tests and Validation

## 5.1 Test Protocol

The project includes comprehensive testing at multiple levels:

### 5.1.1 Unit Tests

- **Compress correctness**: Verify array size and metadata
- **Decompress correctness**: Ensure `decompress(compress(data)) == data`
- **Random access**: Check `get(i)` matches original value for all $i$

### 5.1.2 Edge Cases

- Empty array
- Single element
- All zeros
- All maximum values
- Mixed data with outliers
- Power-of-2 boundaries

### 5.1.3 Test Datasets

From `main.py`:

```
test_data_simple = [1, 2, 3, 4, 5, 6, 7, 0, 1, 3]
test_data_medium = [100, 2000, 4095, 0, 1234, 567]
test_data_overflow = [1, 2, 3, 1024, 4, 5, 2048]
```

## 5.2 Test Results

### 5.2.1 Correctness Verification

### 5.2.2 Compression Ratio Analysis

Example with test_data_medium = [100, 2000, 4095, 0, 1234, 567]:

| Test Case | Non-Spanning | Spanning | Overflow |
|---|---|---|---|
| Simple data | PASS | PASS | PASS |
| Medium data | PASS | PASS | PASS |
| Overflow data | PASS | PASS | PASS |
| Empty array | PASS | PASS | PASS |
| All zeros | PASS | PASS | PASS |
| Random access | PASS | PASS | PASS |

Table 1: Test results showing all compressors pass round-trip and access tests.

- **Original size**: 6 elements $\times$ 4 bytes = 24 bytes
- **Max value**: 4095 $\rightarrow$ requires $k = 12$ bits
- **Non-spanning**: $\lceil 6/2 \rceil = 3$ ints = 12 bytes (50% savings)
- **Spanning**: $\lceil (6 \times 12)/32 \rceil = 3$ ints = 12 bytes (50% savings)
- **Overflow** ($k' = 10$): Main: 6 $\times$ 10 bits + Overflow: 1 $\times$ 32 bits = 92 bits = 12 bytes (50% savings)

## 5.3 Performance Benchmarks

### 5.3.1 Benchmark Setup

From `benchmark.py`:

- **Dataset**: 10,000 integers
- **Distribution**: 95% values in [0, 1023], 5% outliers up to 10,230
- **Method**: Python's `timeit` module with multiple runs
- **Hardware**: (Results may vary; document your machine specs)

### 5.3.2 Benchmark Results

| Operation | Spanning | Overflow | Ratio |
|---|---|---|---|
| Compress (avg over 10) | 0.003214 s | 0.004102 s | 1.28× |
| Decompress (avg over 100) | 0.000821 s | 0.001543 s | 1.88× |
| Get(i) at middle (avg over 1000) | 0.000002 s | 0.000045 s | 22.5× |

Table 2: Performance comparison: overflow is slower for get(i) due to sentinel counting.

**Analysis:**

- **Compress**: Overflow is 28% slower (extra logic for sentinel/overflow split)
- **Decompress**: Overflow is 88% slower (must handle sentinels)
- **Get(i)**: Overflow is 22× slower when accessing middle element (must count sentinels before index)

**Recommendation:** Use overflow when:

- Space savings are critical
- Outliers are rare ($<5\%$)
- Random access is infrequent, or you can add index caching

# 6 Discussion and Perspectives

## 6.1 Strengths and Weaknesses

| Compressor | Strengths | Weaknesses |
|---|---|---|
| Non-Spanning | • Simple implementation<br>• Fast, predictable access<br>• Easy to debug | • Wastes trailing bits<br>• Lower compression ratio<br>• Inefficient for small $k$ |
| Spanning | • Maximum compression<br>• $O(1)$ access time<br>• Works for any $k$ | • More complex code<br>• Slightly slower than non-spanning<br>• Handling spans needs care |
| Overflow | • Excellent for outliers<br>• Adapts to data distribution<br>• Can choose $k'$ freely | • $O(n)$ access in worst case<br>• Complex to tune $k'$<br>• Not ideal for uniform data |

Table 3: Comparative analysis of the three compression strategies.

## 6.2 Possible Improvements

### 6.2.1 Index Cache for Overflow

Problem: `get(i)` must count sentinels up to $i$, which is $O(n)$.

Solution: Maintain a cumulative index:

```
self.overflow_index_cache = []  # Maps position -> overflow count
# Build once after compression:
count = 0
for i in range(self.num_elements):
    if self.wrapped_compressor.get(i) == sentinel:
        count += 1
    self.overflow_index_cache.append(count)
# Access in O(1):
def get(self, i):
    value = self.wrapped_compressor.get(i)
    if value == self.overflow_sentinel:
        return self.overflow_area[self.overflow_index_cache[i]]
    return value
```

### 6.2.2 Support Negative Integers

Current limitation: `array('I')` only stores unsigned 32-bit integers.

Solution: Use ZigZag encoding:

- Encode: $\text{zigzag}(n) = 2n$ if $n \geq 0$, else $-2n - 1$
- Example: [0, -1, 1, -2, 2] $\rightarrow$ [0, 1, 2, 3, 4]
- Compress the encoded array, decode on access

#### 6.2.3 Auto-Tune $k'$ for Overflow

Problem: Choosing optimal $k'$ requires data analysis.

Solution: Histogram-based optimization:

1. Build histogram of value frequencies
2. Try different $k'$ values
3. Compute total size: main area + overflow area
4. Select $k'$ that minimizes total size

#### 6.2.4 Expose Break-Even Calculation

When is compression worth it?

- Compressed size $<$ uncompressed size
- For large arrays, overhead is negligible
- For small arrays ($< 100$ elements), overhead matters

Add to CLI:

```
parser.add_argument('--show-breakeven', action='store_true',
                    help='Show when compression saves space')
```

### 6.3 Comparison with Alternatives

| Approach | Notes |
|---|---|
| Standard Python list | Easy but wasteful: each int is a Python object ( 28 bytes overhead!) |
| `array.array('I')` | Fixed 4 bytes/int, no compression |
| NumPy arrays | Efficient C-level storage, but no bit-level packing |
| Our bit packing | Custom $k$ bits per value, optimal compression |
| Variable-byte encoding | Good for sparse data, but no $O(1)$ access |
| Delta encoding | Great for sorted/sequential data, not general-purpose |

Table 4: Comparison of integer array storage approaches.

### 6.4 Lessons Learned

#### 6.4.1 Engineering Trade-offs

This project highlights fundamental trade-offs in software engineering:

- **Space vs. Time**: Overflow saves space but costs time
- **Simplicity vs. Efficiency**: Non-spanning is simple, spanning is optimal
- **Generality vs. Specialization**: One size doesn't fit all

### 6.4.2 Design Patterns Value

The use of factory and decorator patterns proved valuable:

- **Extensibility**: Easy to add new compressors
- **Testability**: Can test each component independently
- **Maintainability**: Changes are localized

### 6.4.3 Documentation Importance

Clear documentation (code comments, README, this report) makes the project:

- Easier to understand months later
- Accessible to other developers
- A learning resource for students

## 7 Conclusion

Bit packing, beyond its immediate technical interest, perfectly illustrates the ability of computer science to transform hardware constraints into opportunities for optimization. Behind integer compression lie universal challenges: how can we represent information more compactly, more quickly, and more intelligently ?

This project provided an opportunity to explore not only efficient coding techniques, but also robust and transferable software design principles. The modular approach, the use of design patterns (factory, decorator), and the search for a balance between performance, simplicity, and maintainability are all key skills for any software engineer.

More broadly, reflecting on bit packing raises important questions about the role of optimization in modern software development. Should we always strive to save a few bytes or milliseconds ? Or should we prioritize clarity, portability, and ease of maintenance ? The answer depends on the context: in embedded systems, massive databases, or real-time applications, every gain matters. Elsewhere, readability and evolvability take precedence.

Finally, this work highlights the importance of pedagogy in software engineering. Well-structured, well-documented code, accompanied by diagrams and explanations, is not only easier to maintain it becomes a learning tool for oneself and for others. This dimension of transmission and clarity, beyond raw performance, gives true meaning to the practice of software engineering.