UNIVERSITÉ CÔTE D'AZUR

DIGITAL SYSTEMS FOR HUMANS
GRADUATE SCHOOL AND RESEARCH

Data Compression for Speeding Up Transmission

Software Engineering Project 2025

Yunus Emre Dogan

Université Côte d'Azur

Abstract

The transmission of large integer arrays over networks often leads to unnecessary bandwidth consumption due to fixed 32-bit storage per integer. This project proposes a set of bit-level compression strategies, collectively known as **Bit Packing**, to reduce data size while preserving random access to each element.

Three methods were implemented in Java: (1) a *two-consecutive* approach that allows integers to span across two 32-bit words, (2) a *non-consecutive* version that prevents boundary overlaps, and (3) an *overflow-based* method that separates rare large values into a secondary storage area.

All approaches are evaluated through compression and decompression benchmarks for arrays of increasing size, and their performance is compared in terms of speed and compression ratio. The objective is to determine the trade-off between compression efficiency and random access performance for each method.

1 Introduction

Efficient data transmission has become one of the fundamental challenges in modern computing and networking. When large arrays of integers are transmitted using standard 32-bit storage, a significant amount of bandwidth is wasted because many values require far fewer bits to represent. Reducing this redundancy can, therefore, lead to faster communication, lower storage requirements, and overall improved system performance.

This project addresses this issue through the concept of **Bit Packing** — a technique that stores each integer using only the minimal number of bits necessary to represent it. The key challenge is to achieve this compression without losing the ability to directly access any element of the compressed array. To study different trade-offs, three Java implementations were developed, each exploring a distinct bit-packing strategy with specific trade-offs in space efficiency and access speed. These approaches are described in detail in the next section.
Each method was designed, implemented, and analyzed in terms of compression efficiency and access time.

2 Approach and Design

The project explores several strategies for compressing integer arrays through the concept of **Bit Packing**. Although all approaches share the same core principle — storing integers compactly within 32-bit words — they differ in how they balance compression efficiency and access simplicity.
The general workflow for all approaches can be summarized as follows:

1. Compute the bit-length required to represent each integer in the input array.
2. Choose a packing strategy based on this information:
    - use the global maximum bit length (fixed-size packing),
    - restrict values to single 32-bit boundaries (non-consecutive packing),
    - or select a bit-length threshold and handle outliers separately (overflow model).
3. Pack the integers into 32-bit words while preserving direct access through the `get(i)` function and decompress the integers afterwards.

To analyze different trade-offs between compression ratio, access speed, and implementation complexity, three Java implementations were developed:

- **Method 1 – Two-Consecutive-Integer Packing**
- **Method 2 – Non-Consecutive Packing**
- **Method 3 – Overflow Packing**

Before describing each method in detail, it is important to note that all implementations share the same core functional structure. Each compression class implements three primary functions:

- **compress():** Encodes the input array into a compact bit-packed format.
- **get():** Retrieves the value at a specific index directly from the compressed data without full decompression.
- **decompress():** Reconstructs the original array by iteratively calling `get(i)` for each index.

While these functions exist in all three methods, each one differs in how bit boundaries, packing rules, and access operations are handled, as explained in the following subsections.

2.1 Method 1: Two-Consecutive-Integer Packing

This method is implemented in the TwoConsecutiveInt subclass derived from the base BitPacking class. The compression algorithm begins by determining the number of bits required to represent the largest value in the input array. Let this number be k. Each integer in the input is then represented using exactly k bits. The algorithm maintains a pointer that tracks the current bit position inside the output buffer.

If the next value to encode can fully fit within the remaining bits of the current 32-bit word, it is written directly using a left-shift operation. Otherwise, the portion that fits is stored in the current word, and the remaining bits are written into the next word, thus creating an overlap across two consecutive integers.

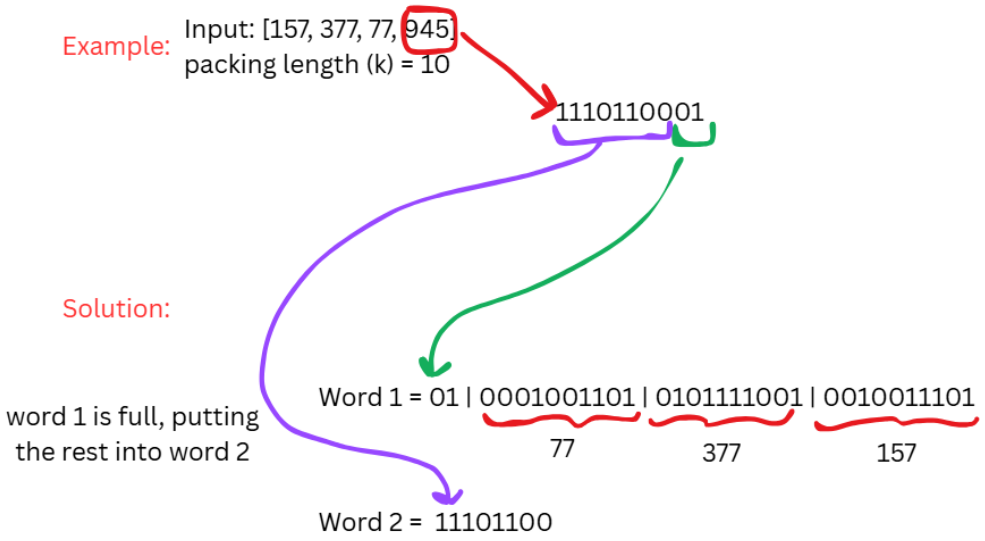

Figure 1: Illustration of the Two-Consecutive-Integer Packing process with packing length $k = 10$. Each integer is represented using 10 bits, and when the remaining bits in the 32-bit word are insufficient, the value is split across two consecutive words.

This design enables near-optimal compression density, since every available bit in the output buffer is utilized. However, the trade-off is that decoding or direct access (`get(i)`) becomes more complex, because an element may be split across two different 32-bit words.

2.2 Method 2: Non-Consecutive Integer Packing

This method is implemented in the NotTwoConsecutiveInt subclass derived from the base BitPacking class. This method aims to simplify access operations and avoid cross-word dependencies by ensuring that each compressed integer is stored entirely within a single 32-bit word.
After computing the bit width k required to represent the largest element of the input array, the algorithm determines how many integers can fit inside one 32-bit word:

$$n = \left\lfloor \frac{32}{k} \right\rfloor$$

Each word thus contains n compressed integers, with the remaining unused bits left as padding. Although this introduces some wasted space, it significantly improves random access performance and simplifies both compression and decompression logic.

When compressing, the algorithm packs up to n integers per word. The value of n ensures that no integer crosses the boundary between two words.

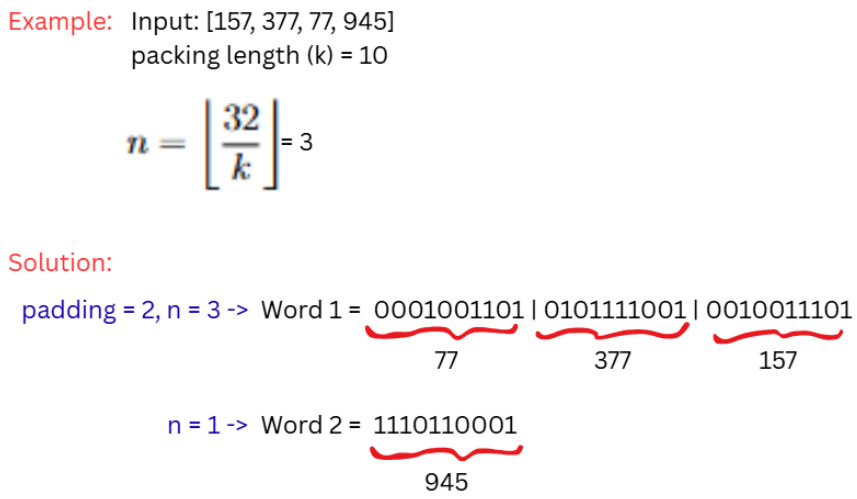

Figure 2: Example of Non-Consecutive Bit Packing: each integer fits entirely within one 32-bit word.

During decompression, each word can be processed independently, and the i-th value can be retrieved using only one 32-bit word access.

This design trades a small amount of space efficiency for a substantial increase in simplicity and speed of random access.

2.3 Method 3: Overflow-Based Compression

This method introduces an **overflow area** to efficiently handle large integers that would otherwise increase the global bit width. The algorithm begins by computing the bit length of each value and selecting a *threshold* — typically around the 75th percentile of all bit lengths. Values with bit lengths below or equal to this threshold are directly encoded in the main compressed region, while larger values are redirected to a separate overflow area.

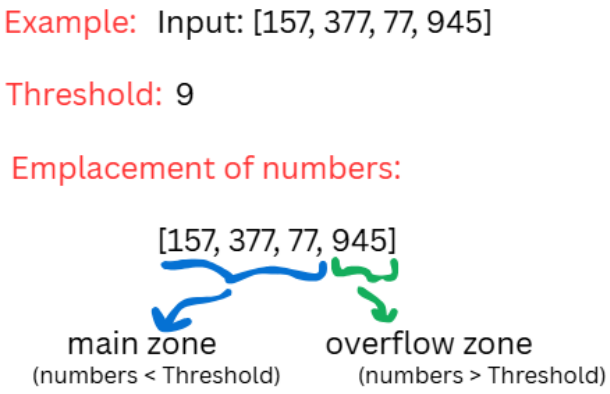

Figure 3: Illustration of overflow separation based on the threshold value (9 bits). Values under the threshold are stored in the main zone, while larger ones go into the overflow zone.

Each packed element in the main array uses one additional **flag bit**: if the flag is 0, the bits represent the actual value; if it is 1, they represent an index that points to the true value in the overflow zone.

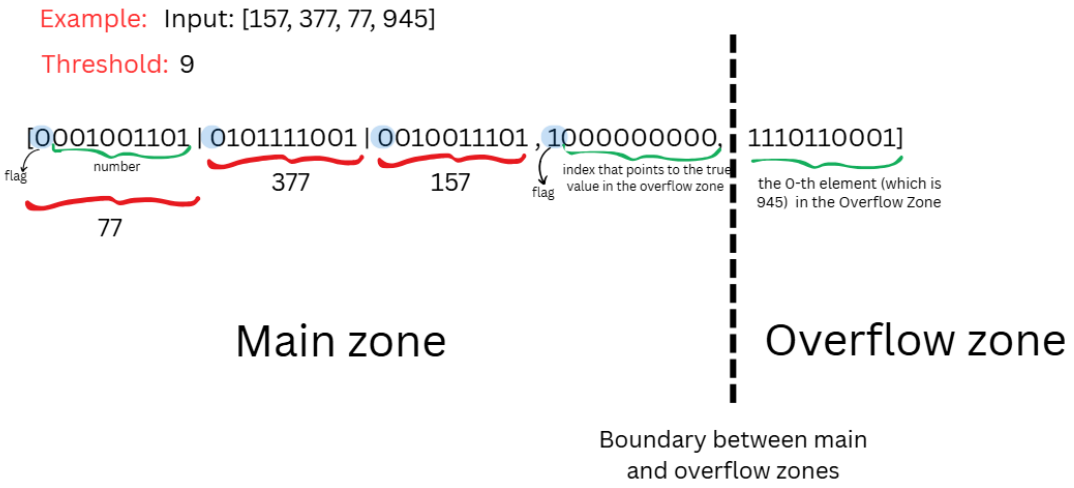

Figure 4: Visualization of the compressed representation: values under the threshold are stored in the Main Zone, while larger ones are redirected to the Overflow Zone. The flag bit distinguishes between the two.

During decompression or access, the algorithm reads this flag to determine whether to decode directly or retrieve the number from the overflow area.

2.3.1 Function Overview

The implementation of this method is divided into several functions, each responsible for a distinct step of the compression pipeline:

- `Packing_Length()` – computes the bit-length of each integer and determines a threshold value based on the 75th percentile of these lengths. This ensures that the majority of values fit within the main zone while leaving only a small fraction to be handled as overflows.
- `computePackedWordCount(n, w)` – estimates how many 32-bit words are required to store n packed values, each occupying w bits. This function guarantees accurate memory allocation for the compressed array.
- `compress()` – performs the main encoding. The total compressed array length is computed as:

  $$total\_len = packed\_int\_count + overflow\_count,$$

  ensuring enough space for both zones.
- `get(i)` – provides random access to the i-th element. It first checks the flag bit to decide whether to decode the value directly from the main zone or to retrieve it from the overflow area.

3 Implementation Details

The program was implemented in Java using an object-oriented structure. A base class (`BitPacking`) defines the core compression interface, while three subclasses implement different packing strategies. A simple factory method allows dynamic selection of the desired compression mode at runtime.

Common Utility Functions. All three compression methods share a few helper routines implemented in the base `BitPacking` class:

- `lengthOfBit(x)` – returns the number of bits required to represent an integer.
- `Packing_Length()` – determines the packing size (bit width) for the array based on the maximum value.
- `afficher()`: - Displays the compressed words for verification.

These functions provide a common foundation and are inherited by all derived classes to ensure consistent behavior across methods.

4 Benchmark and Performance Evaluation

4.1 Experimental Setup

To evaluate the efficiency of the three bit-packing implementations, a dedicated Java benchmarking program was created (`Benchmark.java`). This program measures both compression and decompression times for different array sizes, using high-resolution timing via `System.nanoTime()`.

Each experiment was executed for five repetitions, and the average execution time was recorded to minimize noise and account for potential fluctuations caused by the Java Virtual Machine (JVM) and garbage collection.

Arrays of varying sizes were tested to analyze how each method scales with input length:

$$\text{Sizes} = [10, 10^2, 10^3, 10^5, 10^6, 10^7, 10^8]$$

All input arrays were generated randomly with integer values ranging from 0 to 4096. All experiments were based on randomly generated integer arrays. In addition to the main test, two complementary benchmarks were conducted:

- **Value ranges:** [0, 64] and [0, 8000], in addition to the main [0, 4096] range.
- **Purpose:** To observe performance scaling across different bit-length intervals.
- **Metrics:** Only total execution time (compression + decompression) was plotted.
- **Note:** The overflow-based method was omitted due to its much higher compression time.

For each method, three metrics were collected:

- **Compression Time (ns)** – the time required to encode the input array into the compressed form.
- **Decompression Time (ns)** – the time to reconstruct the original array from the compressed data.
- **Total Time (ns)** – the sum of compression and decompression times, representing end-to-end performance.

4.2 Results and Analysis

4.2.1 Numbers between [0, 4096]

The following tables summarize the average total compression time, decompression time, and the average total execution time (compression + decompression) for each method across different array sizes.

Table 1: Average compression time (μs)

Array Size	**Two-Consecutive**	**Non-Consecutive**	**Overflow**
10	5.14	4.47	190.03
10^2	5.41	5.20	58.42
10^3	45.08	42.20	178.07
10^5	1224.92	899.96	4453.68
10^6	2503.17	3565.33	11342.03
10^7	22592.37	23244.46	119322.98
10^8	173041.79	219952.53	990811.15

Table 2: Average decompression time (μs)

Array Size	**Two-Consecutive**	**Non-Consecutive**	**Overflow**
10	1.48	1.40	1.29
10^2	10.31	8.16	13.50
10^3	26.33	27.06	39.13
10^5	902.02	825.14	1576.84
10^6	3750.20	5674.57	5613.66
10^7	36192.02	58239.37	62329.13
10^8	344468.42	554882.97	548788.46

Table 3: Average total time (μs) for each method

Array Size	**Two-Consecutive**	**Non-Consecutive**	**Overflow**
10	6.63	5.87	191.33
10^2	15.73	13.36	71.93
10^3	71.41	69.26	217.20
10^5	2126.95	1725.10	6030.53
10^6	6253.38	9239.91	16955.70
10^7	58784.41	81483.84	181652.12
10^8	517510.22	774835.51	1539599.62

4.2.2 Random Input Performance: Range [0, 8000]

The following figures illustrate how the execution time evolves across different array sizes for randomly generated numbers in the range [0, 8000]. The results are split into three scale intervals to highlight how performance trends change as the data size increases.

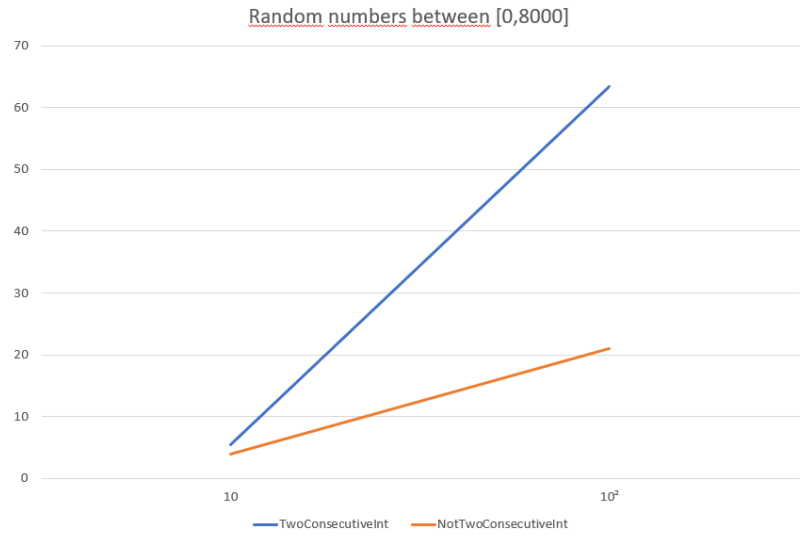

Figure 5: Performance comparison for small arrays (10 to 10^2 elements). Method 1 (Two-Consecutive) shows slightly higher total time than Method 2 (Non-Consecutive).

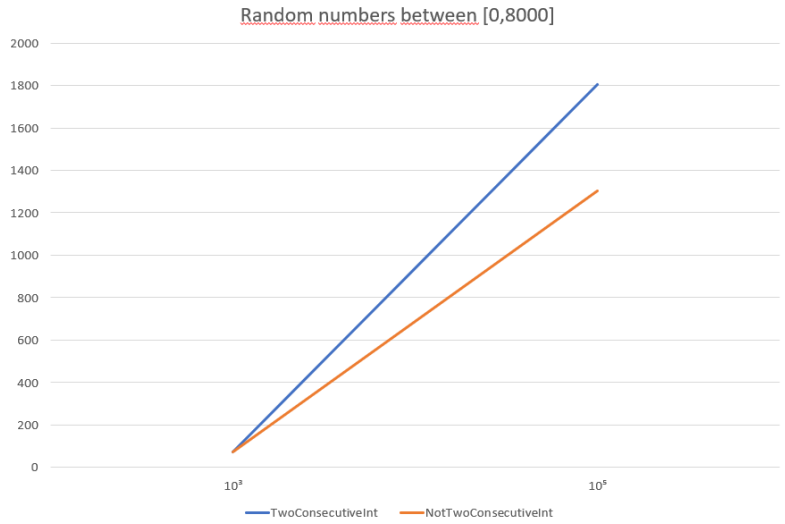

Figure 6: Medium-scale arrays (10^3 to 10^5). Method 1 continues to take longer, though the gap between the two methods remains consistent.

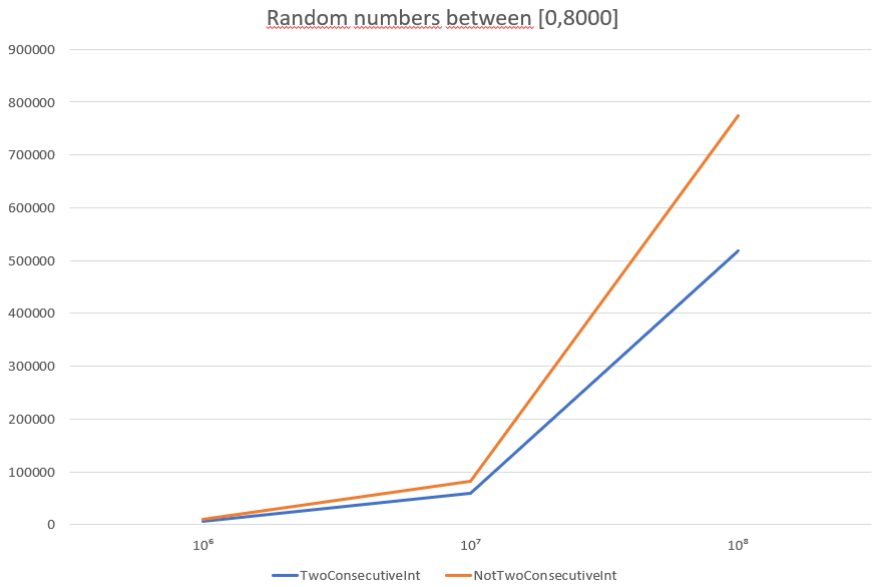

Figure 7: Large-scale arrays (10^6 to 10^8). At this scale, Method 1 (Two-Consecutive) overtakes Method 2, showing better scalability and lower total time as array size increases.

Observation: For numbers in the range [0, 8000], both methods show a clear linear growth in execution time as the array size increases. For small and medium datasets (10–10^5 elements), the *Non-Consecutive* method performs faster due to its simpler structure and lack of cross-word operations. However, for larger datasets (10^6–10^8 elements), the *Two-Consecutive* approach becomes more efficient, as its tighter bit packing results in better memory locality and reduced data movement. Overall, this method scales more effectively with growing array size.

4.2.3 Random Input Performance: Range [0, 64]

The following figures show the execution time across different array sizes for randomly generated integers within the range [0, 64]. This experiment evaluates how the three compression methods perform when handling small integer values that require few bits for representation.

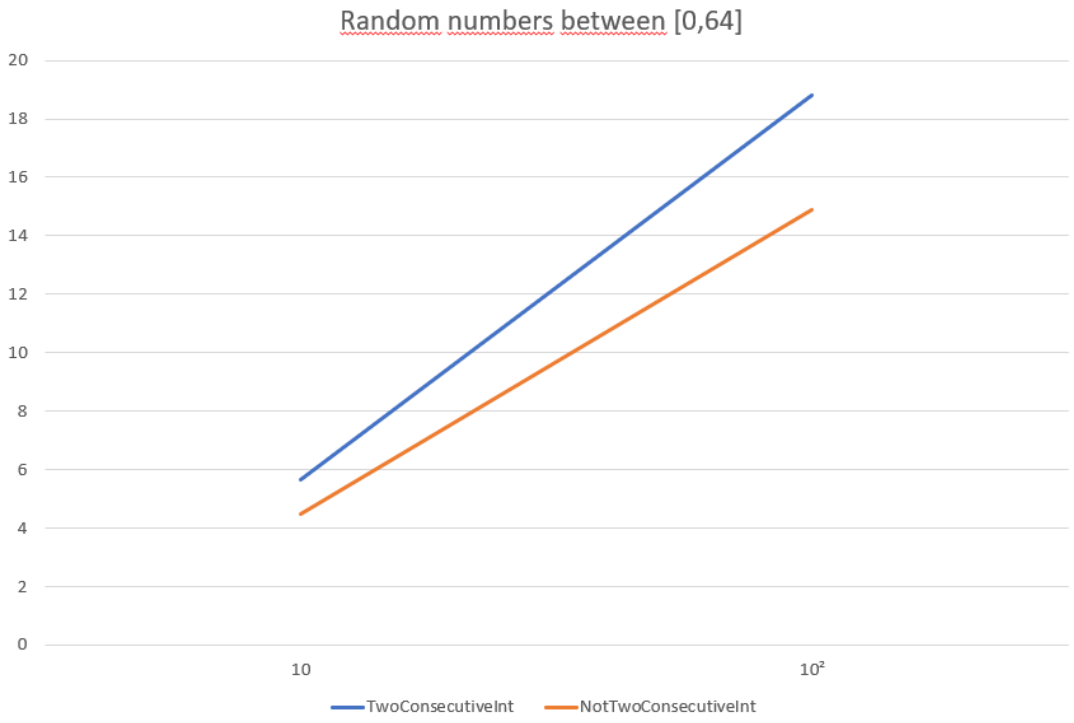

Figure 8: Execution times for small arrays (10–10^2 elements).

Figure 9: Execution times for medium arrays (10^3–10^5 elements).

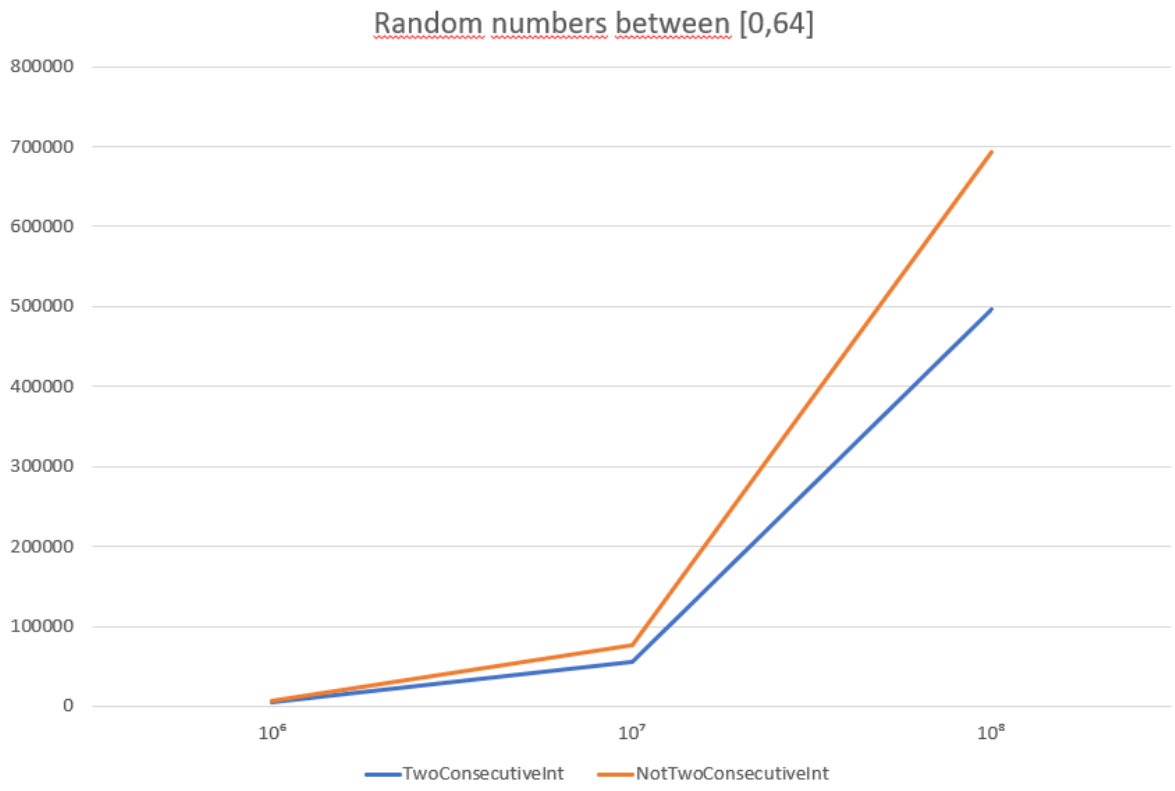

Figure 10: Execution times for large arrays (10^6–10^8 elements).

Observation: A similar performance trend was observed for numbers in the range [0, 64]. The *Non-Consecutive* method remains slightly faster for small arrays, while the *Two-Consecutive* version overtakes it for larger datasets, showing the same scalability advantage described in the previous section.

4.3 Transmission Time and Latency Threshold

To quantify when compression becomes beneficial, we consider the total transmission time:

$$T_{\text{total}} = T_{\text{compress}} + T_{\text{send}} + T_{\text{decompress}}$$

where T_{send} depends linearly on the data size and the network latency t. If S_u and S_c denote the uncompressed and compressed sizes (in bits), then:

$$T_{\text{send}}^{(\text{uncompressed})} = S_u \cdot t, \qquad T_{\text{send}}^{(\text{compressed})} = S_c \cdot t$$

Compression is worthwhile when:

$$T_{\text{compress}} + T_{\text{decompress}} + S_c \cdot t < S_u \cdot t$$

which leads to the threshold condition:

$$t > \frac{T_{\text{compress}} + T_{\text{decompress}}}{S_u - S_c}$$

This expression defines the minimum network latency t above which using compression reduces the total transmission time.

Example. For an array of 10^6 integers:

- Uncompressed size: $S_u = 32 \times 10^6 = 32\,\text{Mbits}$
- Compressed (Two-Consecutive) size: $S_c = 12\,\text{Mbits}$
- Compression time: $T_{\text{compress}} = 2.5\,\text{ms}$
- Decompression time: $T_{\text{decompress}} = 3.7\,\text{ms}$

Substituting into the formula:

$$t > \frac{2.5 + 3.7}{32 - 12} = \frac{6.2}{20} = 0.31\,\text{ms}$$

Thus, for network latencies above approximately **0.3 milliseconds**, bit-packing compression becomes advantageous for arrays of this size.
In general, for small datasets, the compression overhead dominates and the threshold t is higher, while for large datasets the overhead becomes negligible, making compression beneficial even at lower latencies.

Experimental Summary. The table below summarizes the measured latency thresholds (t^*) for three dataset sizes ($n = 10^3$, 10^6, and 10^8) using random values in the range [0, 4096]. These results confirm how compression efficiency improves as the dataset size increases.

Method	$n = 10^3$	$n = 10^6$	$n = 10^8$
Two-Consecutive	8.07	**0.7410**	**0.2927**
Non-Consecutive	8.87	0.9237	0.3932
Overflow-Based	63.09	2.5840	1.1154

5 Challenges and Debugging Process

During the development of the three bit-packing methods, several implementation challenges and design corrections were encountered. This section summarizes the main issues faced and the reasoning behind the final design choices.

5.1 Bit Ordering and Masking Issues

Initially, bits were packed from the most significant side (left to right). However, this caused difficulties during masking and bit extraction since bitwise operations in Java are typically right-aligned. To simplify the logic and prevent incorrect masking results, the bit insertion direction was reversed — bits are now packed starting from the least significant position of each 32-bit word. This adjustment resolved several decoding inconsistencies observed during early tests.

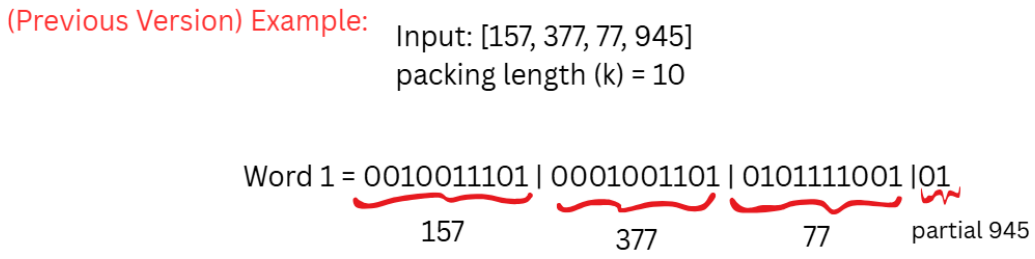

Figure 11: Incorrect bit ordering when packing from left to right (previous attempt).

5.2 Incorrect Word Count Calculation in Overflow Method

In the first version of the overflow-based compression, the number of packed 32-bit words was calculated as:

$$\texttt{packed\_int\_count} = \lceil \frac{n \times w}{32} \rceil$$

The first formula for computing the packed word count occasionally gave incorrect results, causing slight mismatches in memory allocation. The exact cause was not fully determined, though it was likely due to rounding or alignment effects. To fix this, a more reliable dynamic counting method — **computePackedWordCount()** — was implemented, which consistently produced correct results in all cases.

5.3 Threshold Selection Logic

At first, several threshold selection strategies were tested, including using the mean or median of bit lengths. However, these caused unstable results — especially when arrays contained a few large outliers. A fixed percentile-based approach (75%) was ultimately chosen, as it adaptively balances normal and overflow values.

However, this percentile-based approach is not always efficient. When the input data has a narrow or nearly uniform range, the percentile value tends to stabilize and produce the same threshold across multiple tests. In such cases, the overflow mechanism becomes redundant and does not provide meaningful compression benefits.

5.4 Overflow Zone Placement

Another early design decision concerned where to place overflow values in the compressed array. An initial implementation appended them in reverse order (from the end of the array backwards), but this complicated access operations and flag-index mapping.

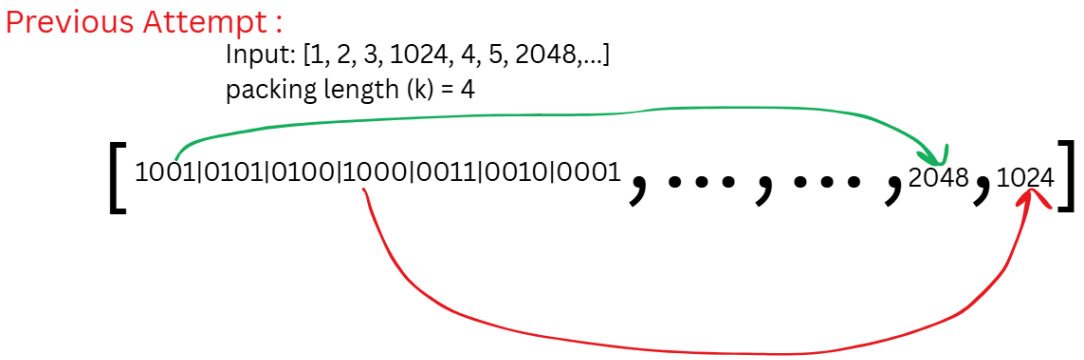

Figure 12: Previous overflow placement approach, where overflow values (e.g., 1024, 2048) were appended at the end of the packed sequence.

5.5 AI-Assisted Corrections and Improvements

During the final development phase, several subtle issues were analyzed and resolved with the help of an AI assistant. The tool was used strictly for debugging and reasoning support, while all design decisions and testing were carried out manually. The following cases summarize the most relevant AI-assisted improvements:

(1) Bit Length Calculation

The original version of the `lengthOfBit()` function used a logarithmic expression:

```
int lengthOfBit(int num){
    return (int)(Math.floor((Math.log(num) / Math.log(2)) + 1));
}
```

Although correct in theory, it was slow and sometimes produced rounding errors for powers of two. After AI-based review, the implementation was replaced with a faster bitwise method, resulting in improved accuracy and performance.

(2) Array Allocation in TwoConsecutiveInt

The compressed array length was initially computed as:

```
compressed_arr = new int[(int)Math.ceil(arr.length * size_packet)/32 + 2];
```

This floating-point computation sometimes underestimated the required memory size. After discussion with the AI assistant, it was replaced by a more robust integer-based calculation:

```
int bits = arr.length * size_packet;
compressed_arr = new int[(bits + 31) / 32];
```

This approach avoids rounding errors and guarantees sufficient memory for any bit length.

(3) Logical Shift Error in get()

A decoding bug caused negative numbers to appear (e.g., `-31` instead of `3041`). The AI assistant identified that the issue was due to the use of the arithmetic right shift operator (`>>`), which propagates the sign bit. Replacing it with the logical right shift operator (`>>>`) solved the issue:

```
value = (compressed_arr[wordIndex] >>> offset) & mask;
```

(4) Word Count Computation in Overflow Method

Originally, the number of packed 32-bit words was computed as:

```
packed_int_count = (int) Math.ceil(arr.length * w / 32.0);
int total_len = packed_int_count + overflow_count;
```

This method produced small alignment errors due to rounding. With the AI assistant's help, the formula was replaced with a precise iterative function that ensures correct word counting for all combinations of array sizes and bit widths:

```
int packed_int_count = computePackedWordCount(arr.length, w);
int total_len = packed_int_count + overflow_count;

int computePackedWordCount(int n, int w) {
    int words = 1;
    int pos = 0;
    for (int i = 0; i < n; i++) {
        if (pos + w > 32) {
            words++;
            pos = 0;
        }
        pos += w;
    }
    return words;
}
```

(5) Latency Threshold Validation

The theoretical expression for the transmission latency threshold was reviewed with AI assistance to confirm its dimensional consistency and practical interpretation. All numerical testing and code integration were conducted manually.

(6) Summary

These corrections were developed and validated manually by me. AI assistant used only as a support tool to analyze specific bugs and suggest possible improvements. I reviewed, understood, and integrated each suggested modification myself after testing and verifying the correctness. The AI assistance helped speed up debugging and improve mathematical precision, but the full design, coding, and integration work were performed by me.

6 Conclusion

This project explored three bit-level compression strategies for integer arrays, each representing a distinct balance between compression density and computational efficiency. The results show that while the Non-Consecutive method performs faster for smaller and moderately sized arrays, the Two-Consecutive approach achieves better scalability as the input size increases. The Overflow model offers adaptability for heterogeneous datasets but introduces additional complexity and overhead.

Overall, the study highlights that no single packing strategy is universally optimal — the best choice depends on data distribution and access requirements.