UNIVERSITÉ CÔTE D'AZUR

SOFTWARE ENGINEERING
RAPPORT

Bit Packing Project Report

Student :
Valentin VINCENT

Teacher :
Jean-Charles RÉGIN

2 novembre 2025

Table des matières

1 Introduction

This project was a short project that involved the creation of a number compressor that compresses numbers by packing their binary versions together.
Bit packing is the process of storing multiple smaller numbers within a single larger binary container by allocating only the necessary number of bits for each value. Instead of reserving a full byte (8 bits) for every number, bit packing makes it possible to use as few bits as required.

This approach is particularly useful when dealing with large datasets or memory-constrained environments, as it can significantly reduce storage requirements and improve data transfer efficiency. Bit packing is commonly used in :

- **Data compression algorithms** to minimize file size.
- **Graphics and image formats** (e.g., textures, bitmaps) where pixel values or color components are packed tightly.
- **Communication protocols** to optimize bandwidth by minimizing the number of bits sent.
- **Embedded systems** where memory resources are extremely limited.

For Example :
We have an array of numbers [1,2,3,4], we will compress them by creating a "big" binary number that takes them all, and we should have something resembling this : 11011100 (1, 10, 11, 100).

We had the constraint to create 2 compression functions that use different methodologies.

With the compression, we had to make decompression functions and a get method that allow the user to know what is the number at a chosen position x.

2 Methodologies

The project was developed in Python, using rustic binary conversion functions instead of the integrated ones.

I used classes and one external function :

- A **Number** class that contains the number value and a method to convert it into binary. There is no method to convert it back to base 10 because during testing, I encountered errors.
- A **BitPacking_v1** class that contains the first compression methodology.
- A **BitPacking_v2** class that contains the second compression methodology.

2.1 First Class : BitPacking_v1

This class contains the first compression method and takes a **Number**'s list as a parameter.

2.1.1 Compression

The compression method is pretty simple, i move through my list of **Number** and convert them into binaries and then i put them together in a big binary number.

We still have a maximum size for our big number so i set a *max_ size* limit and when we reach the limit, we create a new binary number and add the rest of the numbers in the next number.

If there is missing space (for example, we have 30 spots filled instead of 32), the missing spots will be filled with zeros.

If a number is larger than the limit of a big number, it will be ignored with a message.

The trick to not use the binary manipulation, because I'm not used to this, I used the more rustic methods, my binary numbers were strings so it was simpler to manipulate.

I also saved the binary numbers (as int) in a second list for the decompression and get functions.

2.1.2 Decompression

For the decompression, I cheated a little bit and went through the list of binary numbers and converted back into integers and returned it.

2.1.3 Get

The get function is the same as the decompression function but instead of going through the whole list, I simply go the the index given in parameter in the binary numbers list and return the converted correpsponding number.

2.2 Second Class : BitPAcking_ v2

This class contains the second compression method and also takes a **Number**'s list as a parameter. It was a little more complex because the method was blurry for a long time until my classmate **zorena** gave me some inspiration on discord.

2.2.1 Compression

The compression here is more complex, and I'm not sure if what I produced is correct or fully follows the requirements.
I still go through my array of numbers, but this time, I don't pack them directly ; I will do this at the end.

I start by converting them into their binary versions and place them virtually using the [0-0] method, where the first number is the index of the big number and the second is the binary number position within it.

I create an instance for each binary number position and pack each instance into an array, so I end up with something like : `[[0-0], [0-1], [1-0], ...]`.
The placement depends on the *max_ size* defined.
When I get a number, I check if its size exceeds the max size. If it does, I create a new array and add it to the collection, but I still iterate through both the main and new arrays to ensure maximum space utilization.

Once this array is created, I generate an array of lists, each sub-list corresponding to the maximum number on the left of the [x-x] instance.
After this, I insert the binaries into the lists based on their instance positions in the first array. It was simpler for me to handle it this way.

Then to have the "Big" number, I use a similar method than the first method, manipulating strings to have the more lisible output.

2.2.2 Decompression

The decompression method is simple, I just go through my list of `[[x-x]` and convert the binary number into its corresponding interger version thanks to the indexes.

2.2.3 Get

The get it the same as the first method but this time, we take the [x-x] then get the number on the binary list and convert it.

2.3 Benchmarks

2.3.1 Setup

To truly understand the differences between the two methods, it was necessary to measure their performance to provide an estimate of their efficiency.
To achieve this, I implemented a benchmark to compare both methods using the following pipeline :

1. Each function (`compress`, `decompress`, `get`) is executed 100 consecutive times.
2. The total execution time of these 100 runs is measured, and then divided by 100 to obtain the average execution time per call.

To ensure a fair comparison, I defined several test scenarios to observe how the algorithms behave under different conditions :
— A **small list** with **small numbers** (random values between 1 and 50).
— A **large list** with **small numbers** (random values between 1 and 50).
— A **small list** with **large numbers** (random values between 1 and 1000).
— A **large list** with **large numbers** (random values between 1 and 1000).
— A **list with very large numbers**, to observe the impact on `BitPacking_v2`, which may ignore them (random values between 1 and 10000).
I also varied the parameter `max_length` (with values 8, 16, and 32 bits), since it directly affects how the numbers are packed.

3 Results

The following tables summarize the performance results. Execution times represent the average duration for a single operation, expressed in microseconds (µs).

Results for `max_length` = 8

Dataset	Version	Compress (µs)	Decompress (µs)	Get (µs)
1. Small list, small numbers	v1	155.68	59.32	0.64
	v2	386.80	67.05	0.63
2. Large list, small numbers	v1	1699.38	585.63	0.51
	v2	12242.94	677.75	0.75
3. Small list, large numbers	v1	158.39	24.93	0.93
	v2	222.12	27.77	1.01
4. Large list, large numbers	v1	1579.40	245.92	0.49
	v2	3146.73	264.85	0.55
5. Medium list, very large numbers	v1	1041.59	6.38	0.48
	v2	1015.05	7.22	0.55

TABLE 1 – Performance results for `max_length = 8`.

We can see here that the second method is overall slower than the first one except on the compress method for a medium list of very large numbers.

Results for `max_length` = 16

Dataset	Version	Compress (µs)	Decompress (µs)	Get (µs)
1. Small list, small numbers	v1	123.32	59.65	0.58
	v2	313.40	67.29	0.62
2. Large list, small numbers	v1	1337.74	575.83	0.69
	v2	6671.27	682.45	0.75
3. Small list, large numbers	v1	264.23	110.14	0.94
	v2	583.96	118.62	0.84
4. Large list, large numbers	v1	3410.05	1090.96	1.33
	v2	14207.94	1198.37	1.28
5. Medium list, very large numbers	v1	1737.29	865.66	1.72
	v2	6778.12	913.97	1.79

TABLE 2 – Performance results for `max_length = 16`.

We see in the results that the first compress, decompress and get methods tend to be faster with a larger `max_length` but only on lists of small numbers. The seconds are getting slower.

Results for `max_length` = 32

Dataset	Version	Compress (µs)	Decompress (µs)	Get (µs)
1. Small list, small numbers	v1	114.58	59.19	0.58
	v2	273.83	68.43	0.62
2. Large list, small numbers	v1	1207.88	583.03	0.70
	v2	4822.89	677.72	0.73
3. Small list, large numbers	v1	208.35	110.67	0.82
	v2	504.58	118.44	0.98
4. Large list, large numbers	v1	2208.61	1077.03	0.49
	v2	8446.33	1194.88	0.52
5. Medium list, very large numbers	v1	1647.02	868.79	1.70
	v2	4947.08	934.70	1.73

TABLE 3 – Performance results for `max_length = 32`.

In this benchmark, we observe that the first methods are even faster on lists of small numbers and even faster on lists of big numbers compared to the second benchmark, but still slower than the first. The observations are the same for the second method.

3.0.1 Overall Observations

Overall, the first version consistently outperforms the second across most datasets and bit lengths.

Increasing `max_length` tends to improve performance for smaller values but has limited impact on larger ones.

These results suggest that, while the second method may offer structural or implementation advantages, it comes with a notable performance trade-off.

We also have to know that it is faster with a smaller `max_length` because when a number's binary size is larger than the parameter, it is ignored, so it could be a reason why the smaller `max_length` returns better results

4 Conclusion

This project was a complex but interesting exploration of data compression techniques using *Bit Packing*.

The main goal was to develop two different methods and compare their efficiency, which was achieved through a strong benchmarking.

The analysis of the results is clear : the `BitPacking_v1` method, simpler and more straightforward, proved to be significantly more efficient than the more complex logic of `BitPacking_v2` in almost all cases.

However, the results obtained with a maximum binary length max_length may be biased because binary numbers whose size exceed the limit are ignored.

Even though my implementations do not fully meet the original specifications—mainly because they use string operations instead of real bit manipulations—this project was an interesting exercise.

To go further, the code could be rewritten to use actual bitwise operations for more authentic and efficient compression. It could also be improved by adding support for negative numbers, by reserving an additional bit for sign representation in each number.

In conclusion, this was a short but highly instructive project, providing valuable insights into performance challenges, algorithm design, and technical trade-offs.