Compression of Integers.

toomuchmath • 14 Apr 2026

Being a student of computer science myself, whenever I have to optimize code, generally we have to compromise on space complexity. For example, consider the problem where you have to return the indices (i, j) of two elements from an integer array nums such that nums[i] + nums[j] = target.

There are two ways to solve this. The brute-force approach dictates running two nested for loops to find the indices. Another approach is to store the elements and their respective indices into a map, and retrieve the indices from there.

In the brute-force method, the time complexity is $O(n^2)$ in the worst case, and the space complexity is $O(1)$ (ignoring the space required to store the answer). In the optimized case, we achieve a time complexity of $O(n)$, but the space complexity increases to $O(n)$ in the worst case. Here, n represents the number of elements in the array.

This simple tradeoff led me to wonder: could we compress the integers themselves? If we could somehow encode that entire $O(n)$ hash map—or the array itself—into a single integer that only takes $O(1)$ space under the Uniform Cost Model, we could theoretically break the classic time-space tradeoff. That is, suppose we are given a sequence of integers Q, and we want to find a single integer that can carry all the information from that entire sequence.

Note that in computer science, as an integer grows, it requires more bits to store it in memory. So, even if we compress these integers into one, the resulting number can be so large that it ultimately requires the same number of bits. Memory-wise, this might not be efficient in reality. However, here we are speaking strictly in mathematical terms, assuming that storing a single integer z —no matter if $z \to \infty$—still requires exactly one unit of storage. Generally, theoretical models that require unit storage even for arbitrarily large integers are known as the **Uniform Cost Model**.

Assumptions and Terminology

Before we actually go into the methods to encode the numbers, for consistency and to ensure we are all on the same page, it is important to lay out the formal definitions. Don’t worry—it will not be that difficult to understand.

The Domain: Let S be the set of all finite sequences (including the empty sequence) of non-negative integers. So, an element $Q \in S$ looks like $Q = (q_1, q_2, \ldots, q_n)$ where $q_i \in \mathbb{Z}_{\geq 0}$.

The Codomain: Let the target space be the set of non-negative integers $\mathbb{Z}_{\geq 0}$.

The Encoding Function: We are looking to define a function $\tau : S \to \mathbb{Z}_{\geq 0}$.

Requirement: The function τ must be injective. That is, for every unique Q, there exists an $n \in \mathbb{Z}_{\geq 0}$ with a one-to-one mapping such that $N = \tau(Q)$. Mathematically, if we have two sequences Q and Q', then

$$\tau(Q) = \tau(Q') \implies Q = Q'.$$

Also note that here, we refer to Q as a sequence rather than a set. In pure math, standard sets cannot contain duplicate elements and have no strict order. But in our case—much like an array in computer science—we absolutely care about duplicate elements, and the exact order of those elements matters. If two sequences contain the same numbers but in a different order or with different frequencies, they are entirely different sequences.

Even though I just stated the mathematical definitions, I want to give you a rough idea of these terms and symbols so that you can carry them forward without much effort. Here, Q represents any sequence formed by arbitrarily choosing non-negative integers. In computer science terms, it is basically an array. Now let's discuss τ a little bit. One thing to remember about τ is that for every unique input Q, there must be one and only one unique output N. Why? Because suppose two different sequences Q and Q' have the same output N—then how do we identify which input was given to the function? To avoid this ambiguity, the function τ must be injective.

Don't worry about the codomain thing. It is just there to define the function clearly (and to scare you), but just remember the output must be a non-negative integer. Treat it like a return type in functions. That's it.

Now to the question of surjection. That is, for every N in the codomain, there must be a sequence Q. In simpler terms, you gave me the number N, and then does there exist an array Q which on compression results in N? Now this is interesting. Because we are not binding our τ to be surjective. But it will be interesting to note that depending upon techniques, we will look at whether τ is surjective or not. And once it becomes surjective, it will be a bijection.

Concatenation

Many mathematical techniques exist to achieve this encoding. Before going into highly abstract methods, a common and intuitive method we can instantly think of is the concatenation of all integers in the sequence.

Before defining this method mathematically, let's rigorously define what a "digit length" is using the floor function $\lfloor \cdot \rfloor$. Consider an integer $N \in \mathbb{Z}_{\geq 0}$. The digit length $d(N)$ is defined as the smallest integer $k \geq 1$ such that:

$$\left\lfloor \frac{N}{10^k} \right\rfloor = 0$$

Now, let's jump to the first method.

Let's first define pure, unpadded concatenation mathematically. We define a function $\mathcal{C}$ on a sequence $Q = (q_1, q_2, \ldots, q_n)$. The standard concatenation is:

$$\mathcal{C}(Q) = q_1 \cdot 10^{d(q_2)+d(q_3)+\cdots+d(q_n)} + q_2 \cdot 10^{d(q_3)+\cdots+d(q_n)} + \cdots + q_n$$

More compactly, using summation notation:

$$\mathcal{C}(Q) = \sum_{i=1}^{n} \left(q_i \cdot 10^{\sum_{j=i+1}^{n} d(q_j)} \right)$$

I know it looks dangerous! But the meaning is simply putting the numbers together side-by-side. Suppose $Q_1 = (11, 12, 13, 79, 65)$, then:

$$\mathcal{C}(Q_1) = 1112137965$$

Just that. It was me trying to be mathematical! You don't actually need to evaluate that horrifying equation by hand.

This pure concatenation is clearly effective when the digit length for every element in the sequence is exactly the same. However, a problem arises when we attempt to decode it. Suppose you are given $m = \mathcal{C}(Q)$. To retrieve Q, you need to divide m into perfect blocks of digits. But what must the block size be?

To solve this, we must encode the uniform block size—let's call it b—directly into our resulting integer. Furthermore, we must account for sequences with varying digit lengths by padding them.

Here is the complete logic for our injective function $\tau(Q)$:

First, find the maximum digit length in the sequence, $b = \max(d(q_i))$. Pad all numbers in the sequence with leading zeros so that every element has exactly b digits. Next, concatenate them. Finally, to encode the block size b, pick a digit that is strictly different from the last digit of your concatenated number, and append it exactly b times at the end.

Let's standardize the picking of this digit. Let's call it the **appending digit**, denoted by l, defined entirely by the very last integer in the sequence, q_n:

$$l = (q_n \bmod 10 + 1) \bmod 10$$

Because of this definition, l is guaranteed to be different from the last digit of the concatenated block.

Let's look at $Q_1 = (11, 12, 13, 79, 65)$. The block size is $b = 2$. The last digit of q_n is 5. So, $l = (5 + 1) \bmod 10 = 6$. We append 6 exactly 2 times:

$$\tau(Q_1) = 111213796566$$

How do we get the sequence back? Count the trailing identical digits until the digit changes. There are two 6s, so our block size is $b = 2$. Discard the 6s, and divide the remaining number into blocks of 2.

Now, let's look at a sequence with varying digits, which forces padding. Suppose $Q_2 = (7, 77, 777, 777)$.The maximum block size is $b = 3$. Padding the sequence gives us:

$$(007, 077, 777, 777)$$

Concatenating yields 007077777777. The last integer q_n ends in 7, so $l = 8$. We append it 3 times:

$$\tau(Q_2) = 007077777777888 = 7077777777888$$

Notice that mathematically, the integer loses its leading zeros! Because of this, we must **always start dividing into blocks from right to left** during the decoding phase. If we attempt to decode 7077777777888 (after removing the 8s) from the left, the missing zeros will cause catastrophic errors. By parsing in blocks of 3 from the right, the structure perfectly reconstructs itself, naturally restoring the implicit leading zeros.

But here is the twist. Try to encode $Q_a = (7, 0)$ and $Q_b = (0, 7, 0)$.

The Mathematical Definition of $\tau(Q)$:

We can actually define this entire padding, concatenation, and appending process in one elegant mathematical formula. By treating each padded number as a “digit" in base-10^b, the implicit leading zeros are handled naturally by the powers of 10. The repeated appending digit l can be generated using the repdigit formula $\frac{10^b-1}{9}$.

For a non-empty sequence Q of length n:

$$\tau(Q) = 10^{b(n+1)} + \left(\sum_{i=1}^{n} q_i \cdot 10^{b(n-i+1)} \right) + l \cdot \frac{10^b - 1}{9} \tag{1}$$

(Note: If Q is the empty sequence, we can simply define $\tau(\emptyset) = 0$ to maintain injectivity).

Notice the addition of the $10^{b(n+1)}$ term at the beginning of our formula. This acts as a mathematical sentinel value (or anchor). To understand why this is strictly necessary for injectivity, we must remember that pure integers do not retain leading zeros.

Suppose we have two distinct sequences:

$$Q_a = (7, 0) \quad \text{and} \quad Q_b = (0, 7, 0).$$

Both have a maximum digit length of $b = 1$. If we were to encode them without our sentinel value, relying only on concatenation and the appending digit ($l = 1$ for both), the implicit leading zero in Q_b would vanish algebraically:

$$\tau_{\text{unsafe}}(Q_a) \to 701$$

$$\tau_{\text{unsafe}}(Q_b) \to 0701 \implies 701$$

We hit a collision:

$$\tau(Q_a) = \tau(Q_b),$$

utterly destroying our injectivity.

By adding $10^{b(n+1)}$, we are effectively placing a permanent 1 exactly one block-width to the left of our sequence. This 1 acts as a physical wall, trapping any leading zeros inside the integer’s magnitude.

$$\tau(Q_a) = 1701$$

$$\tau(Q_b) = 10701$$

During decoding, once we parse the integer right-to-left into blocks of size b, the final block will always be exactly 1. When we hit this sentinel block, we simply discard it and terminate the process, perfectly preserving all zeros that came before it. That’s it. This is our first basic, yet perfectly injective, compression

method. Let's talk about surjectivity little bit. Because not every integer ends in a valid sequence of identical digits that perfectly divides the remaining leading digits into blocks, this function τ is strictly injective, but clearly not surjective.

Now as being a student of mathematics, can we prove that $\tau(Q)$ is injective? The answer is yes. We can. So, let me state this as a theorem.

Theorem 1. *The τ defined as following is injective.*

$$\tau(Q) = 10^{b(n+1)} + \left(\sum_{i=1}^{n} q_i \cdot 10^{b(n-i+1)} \right) + l \cdot \frac{10^b - 1}{9}$$

Proof. By magic. □

Modulo-Quotient Decomposition

What do we do in this method? Well before that, we must just skim through the **Euclidean Division Lemma**. The lemma states that for every positive integer a and b, there exist unique integers q and r such that

$$a = bq + r$$

where $0 \leq r < b$. This lemma is a complicated way to say that there exist a unique quotient (q) and remainder (r) for every division.

What do we see that's interesting here? That a can uniquely be broken into q and r. So, we are doing ' "decomposition" using *remainders*. Hence the name of the method.

So how can we use this to encode our sequence Q into a single integer? Well, taking inspiration from **Golomb Coding**, just encode the quotient and remainder instead of the elements of the sequences. How exactly?

- Choose a prime p immediately greater than the mean of sequence Q.
- Get the quotients. Let me mathematically write it. Suppose D be the sequence of quotients.
- $$D = \left(\left\lfloor \frac{q_1}{p} \right\rfloor, \left\lfloor \frac{q_2}{p} \right\rfloor, \ldots, \left\lfloor \frac{q_n}{p} \right\rfloor \right)$$

  Get the remainder sequence R as follows.
- $$R = (q_1 \bmod p, q_2 \bmod p, \ldots, q_n \bmod p)$$

  Use $\tau_{\text{concat}}(Q)$ to encode (D, R, p). That's it. Basically do the following.

$$\tau_{\text{concat}}\left(\tau_{\text{concat}}(D), \tau_{\text{concat}}(R), p\right)$$

So, let me mathematically write this new τ function.

$$\tau_{\text{MQD}}(Q) = \tau_{\text{concat}}\left((\tau_{\text{concat}}(D), \tau_{\text{concat}}(R), p)\right) \tag{2}$$

In this method, it is not necessary to use prime. It can be anything or any number, even just the floor of the mean. But I am choosing it to be prime because I feel like it and I am very moody.

Let's try to encode Q_2, and Q_1 will be encoded by angels.

So, $Q_2 = (7, 77, 777, 777)$. So, the value would be $\mu_{Q_2} = 409.5$. The prime would be $p = 419$. Now D would be $D = (0, 0, 1, 1)$ and $R = (7, 77, 358, 358)$.

Okay, now $\tau_{\text{concat}}(D) = 100112$ and $\tau_{\text{concat}}(R) = 1007077358358999$, and we already have prime p. Because our new sequence is a 3-tuple and $\tau_{\text{concat}}(R)$ has 16 digits, our maximum block size is now $b = 16$. We pad everything to 16 digits, calculate our sentinel value, and concatenate to get $\tau_{\text{MQD}}(Q_2)$:

$$1000000000010011210070773583589990000000000000419000000000000000$$

The method for decomposition is the same. Count appending digits and remove them. Divide into blocks, remove the sentinel value, and get D and R. And do the same for D and R and use p to get the sequence Q_2.

Also, τ_{MQD} is injective but not necessarily surjective, as it is just τ_{concat} in some different form. '

Polynomial Interpolation over a Finite Field

Until now, our methods have been more "algorithmic" than mathematical—we were essentially writing equations for algorithms. We haven't fully embraced mathematical rigor, but the time has come to do so.

As the name suggests, we are discussing *polynomial interpolation*. What exactly is that? Fundamentally, it is like drawing a graph. In interpolation, we are given a set of points and must find an equation that satisfies them. In other words, we seek the unique graph passing through those points.

Let us take our sequence Q. Suppose we want to find a polynomial $f(x)$ that satisfies the following points:

$$(1, q_1), (2, q_2), \ldots, (n, q_n)$$

Mathematical law dictates that for any n points, there exists exactly one unique polynomial $f(x)$ of degree at most $n-1$ that passes through all of them. This is known as **Lagrange Interpolation**. The "wild idea" here is this: instead of compressing the sequence Q directly, we construct this polynomial and simply extract its coefficients!

$$f(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{n-1} a_i x^i$$

Let θ be the sequence of our coefficients:

$$\theta = (a_0, a_1, a_2, \ldots, a_{n-1})$$

We now only need to encode θ. This can be done using any preferred method, such as concatenation or modulo-quotient decomposition. As a tip, try to use the method that results in the fewest digits (e.g., use concatenation when there are no outliers).

However, if you have studied **Lagrange Interpolation** before, you know that the coefficients usually end up being rational numbers. But our domain is the set S—the set of all finite sequences of *non-negative integers*. We need our coefficients to be non-negative integers as well. This is exactly where the **Finite Field** (specifically a Galois Field, denoted as $GF(p)$ or $\mathbb{Z}_p$) enters the picture.

In this method, we utilize the “moody" prime p, ensuring that $p > n$ and $p > \max(q_i)$. Why? Because by performing our interpolation entirely within $GF(p)$, fractions become fundamentally impossible. Every coefficient a_i is mathematically guaranteed to be a “clean" integer such that $0 \leq a_i < p$.

The basis for this interpolation modulo p is as follows: for each point j from 1 to n, the basis polynomial $\mathcal{L}_j(x)$ is defined as:

$$\mathcal{L}_j(x) \equiv \prod_{\substack{1 \leq m \leq n \\ m \neq j}} (x-m)(j-m)^{-1} \pmod{p}$$

Our final polynomial is the sum of these bases multiplied by our sequence values:

$$f(x) \equiv \sum_{j=1}^{n} q_j \cdot \mathcal{L}_j(x) \pmod{p} \tag{3}$$

When operating in $GF(p)$, every calculation is performed modulo p. Suppose your sequence is $Q' = (1, 8, 3)$ and you choose $p = 7$. Because you are in $GF(p)$, the 8 becomes $8 \bmod 7 = 1$. Your sequence transforms into $(1, 1, 3)$ before the polynomial is even built, resulting in a permanent loss of original data and threatening injectivity. Therefore, the prime p must be a “ceiling" higher than your largest value to ensure the sequence Q' remains unique.

Let's use this method to encode $Q_1 = (11, 12, 13, 79, 65)$. First, we find θ_{Q_1}. We choose $p = 83$ (the prime immediately greater than $\max(n, \max(q_i))$). For $\mathcal{L}_1$:

$$\begin{aligned}\mathcal{L}_1 &\equiv (x-2)(x-3)(x-4)(x-5) \cdot ((1-2)(1-3)(1-4)(1-5))^{-1} \pmod{83} \\ &\equiv 45(x-2)(x-3)(x-4)(x-5) \pmod{83}\end{aligned}$$

After calculating the remaining $\mathcal{L}_j$ terms, we form the polynomial $f(x)$:

$$\begin{aligned}f(x) \equiv\ & 11 \cdot 45(x-2)(x-3)(x-4)(x-5) \\ &+ 12 \cdot 69(x-1)(x-3)(x-4)(x-5) \\ &+ 13 \cdot 21(x-1)(x-2)(x-4)(x-5) \\ &+ 79 \cdot 69(x-1)(x-2)(x-3)(x-5) \\ &+ 65 \cdot 45(x-1)(x-2)(x-3)(x-4) \pmod{83} \\ &\equiv 12x^4 + 43x^3 + 23x^2 + 32x + 67 \pmod{83}\end{aligned}$$

Now we simply encode $\theta_{Q_1} = (67, 32, 23, 43, 12)$. (Note: coefficients are listed from a_0 to a_{n-1}).

Is this method injective? Lagrange interpolation is strictly bijective, but there is one potential problem. Suppose we have $Q_c = (5, 5, 5)$. The polynomial is simply $f(x) = 5$, giving $\theta_{Q_c} = (5)$. Now consider $Q_d = (5, 5, 5, 5, 5)$. Its polynomial is also $f(x) = 5$. If we only look at the coefficients, we hit a collision! How do we know if $f(x) = 5$ should be evaluated 3 times or 5 times?

We resolve this by refining our definition:

Coefficient sequence (θ): The coefficient sequence of the polynomial $f(x)$ must have a length strictly equal to the length of Q.

Under this rule, $\theta_{Q_c} = (5, 0, 0)$ and $\theta_{Q_d} = (5, 0, 0, 0, 0)$. The collision is avoided, and the method remains strictly injective.

Decoding is laughably simple: evaluate the polynomial at the known indices:

$$q_i \equiv f(i) \pmod{p} \quad \text{for } i \in \{1, 2, \ldots, n\}$$

Since we also need the "moody" prime p to retrieve the sequence, the final encoding formula is:

$$\tau_{GF(p)}(Q) = \tau_{\text{concat}}((\theta_Q, p)) \tag{4}$$

Don't tell me that domain of above equation is wrong! We append the moody prime p to the end of our coefficient sequence θ_Q, and pass the entire flat sequence into our τ_{concat} function.

Theorem 2. *The function $\tau_{GF(p)}$ is injective.*

Proof. Do I need to prove it again? □