Nested Numbers

Ethan Holland Simmons

October 2, 2019

*Abstract*

*If given a single number, many operations can take place on that number, and each operation will yield a different value. A single number can hold any amount of values inside it, just the operations need to change. The number can also gain more values by adding operations to it.*

## 1.) Introduction:

All the values need to be represented by a single value. Each value needs to be able to be read from the single number. Possible operations can be addition and subtraction, but those alter the number.

Example:

Let the single number need to store 1 and 3. If a number is increased by a constant, and decreased by a constant, the number could be 2, with the constant being 1. To retrieve the values (1 and 3), the number (2) would just be increased or decreased by 1. If the value of 3 would change to 4, this no longer works, and another constant would need to be introduced.

More possible operations could be multiplication or division, but this has a similar issue. Two operations that don’t interfere with each other are integer division (or floor division), and modulos.

Example:

Let the number need to store the values 1 and 3 again. This can be done by multiplying the first by a constant, and adding the second. The values are retrieved by the floor division, and modulos operations. The constant can be 10. The first value is multiplied by the constant (1 \* 10) and the second is added (3), resulting is 13. The first value is retrieved by the floor division of the constant ( ⌊13 / 10⌋ = ⌊1.3⌋ = 1 ). The second value can be retrieved by modulos ( 13 mod 10 = 3).

There is a restriction: the second value has to be less than the constant, otherwise the modulus operation will return the value modulo the constant. The first value will also change (it will increase by the floored result of the second value, divided by the constant).

Example:

Let the number need to store 1 and 13. If following the same method, the single number is generated by 10\*1 + 13 = 23. The first value is then supposed to be 2 ( ⌊23 / 10⌋ = 2 ), and the second is supposed to be 3 ( 23 mod 10 = 3 ).

## 2.) Expanding

The only restriction is that the second value has to be less than the constant. There is no such restriction for the first value. This allows the first value to be a concentration of two other values, following the same method. This can be repeated on the first value of the first value, and expanded forever. Thus, a list of any numbers, a, where 0 a < k, they can be combined into a single value.

Example:

Let the number need to store the values 1, 2 and 3, with the constant 10. The first two (1 and 2) are combined (1\*10 + 2 = 12) and then 12 and 3 are combined ( 12 \* 10 + 3 = 123 ). Each value is retrieved by reversing this. First, ⌊123 / 10⌋ gives 12, and 123 mod 10 gives 3. The first value is then operated on again (⌊12/10⌋ = 1 and 12 mod 10 = 2). Therefore, all the values are retrieved.

## 3.) Computation

* Let a list of values, *a*, be denoted by a0, a1, a2, … an
* Let there be an integer *k* where 0 *k*
* Every value *a* satisfies 0 *a* < *k*
* Let the combined form be *c*
* *c* =
* To find any index of of *a* from *c*:
* To rewrite any index :

Example:

a = [ 1, 2, 3, 4 ]

k = 10

c = 4321 = (1\*100 + 2\*101 + 3\*102 + 4\*103 )

TEST:

a0 should be 1

a0 = (4321 mod 101) / (100) = (1) / (1) = 1 = 1

a3 should be 4

a3 = (4321 mod 104) / (103) = 4321 / 1000 = 4.321 = 4

## 4.) Uses

This concept can condense an array into a single integer. A python3 program can demonstrate this:

#PROGRAM START

lst=[10,20,30,4]

key=31

sum=0

for index in range(0,len(lst)): #creates the list

sum += lst[ index ] \* (key\*\*index)

print("sum:",sum)

def returnIndex(index):

return( int( (sum%(key\*\*(index+1) ) ) / (key\*\*index) ) )

for index in range(0,len(lst)): #retrieves values from the sum

print( returnIndex(index) )

#PROGRAM END

The uses for this can be for memory optimization. The array *a* holds 4 integer values. Each integer value resided in an allocated part of memory *x* bytes long. When condensed, it only is held in a location *x* bytes long, vs 4*x* bytes (as long as the end value is within the byte range).

This can also be used as a cryptographic scheme. Each character of the message can be represented by *a* and the encryption key can be *k*. The secret key will be the only way to find each *a* value. This has multiple strengths. First, it is very simple math, requiring a very basic knowledge of arithmetic. The most complex operation is the selection of the random key. Second, it is very fast. To encrypt the message, a random number (larger than the highest value that could be sent) is generated. Next, each character is stored into the number. This only requires a single addition, a single multiplication, and a single exponent. To retrieve the message, each character can be retrieved by 2 exponentials, 2 multiplications, a single addition, a single modulus operation, a single division, and a floor operation.

Example:

message = “zyx” = [ “z”,”y”,”x” ] = [ 122, 121, 120 ] (in ascii)

k = random number + 122 (to be larger than the max value) = 30 + 122 = 152

*c* = = 2790994

message = [ (2790994 mod 1521) / (1520) ,(2790994 mod 1522) / (1521) , (2790994 mod 1523) / (1522) ] = [ 122, 121 120 ] = [ “z”, “y”, “x” ] = message

The secret in this is the k value, and the only restriction is that it has to be larger than the largest value.

Lastly, this can be used for data compression. A classic array would have to allocate enough space to contain the maximum value, with wasted space. Java’s int type has 4 byte values, or -2,147,483,647 to +2,147,483,647. If working with smaller values, this creates a lot of wasted space. The following is a hypothetical situation in java.

Example:

Store this list in memory:

1,2,3,1776,4,5,6,7,8,9

This is 4 bytes \* 10 cells = 40 bytes = 320 bits

This with nested numbers would be:

1590913820463683291420337747885

constant: 1777

This is 101 bits, plus an 11 bit constant (but for simplicity this can be a full 4 byte integer). This is still only 133 bits, about 2.5 times smaller than the original list.

The constant can be some arbitrarily large number, or just one more than the maximum number (this is most optimised) to ensure each value works. The values could just be taken modulo *k* before operations to ensure there is no corruption of values.

## 4.) Conclusion

A single value can hold all the information to construct an ordered list of numbers.This is made possible by integer division and the modulus operator, and how they interact with different parts of a number. Any amount of values can be stored in a single number. This can help with data compression. This can also be used as a cryptographic scheme. In practice, the constant can be some well known, large number (excluding the cryptographic instances).