# Efficient Lexicographic Encoding of Numbers

Peter Seymour

Mar 28, 2020

## 1 Introduction

This paper explores ways of textually representing numbers such that their natural order is preserved as a lexicographic order of their representations. Once this is achieved numbers can be used within any text system alongside other text whilst only requiring a single order relation. New number types can be added and an order implied without requiring any modification to the system. This is particulary useful when building a data storage system that is extensible in the range of available types.

A textual representation of a number amounts to constructing a sequence of symbols. The most primitive construction is to represent the natural numbers. This construction can then be used to build up more complex representations such as signed integers, floating point numbers and tuples.

## 2 Natural numbers

Chose a base $b > 1$ then for any $n \in \mathbb{N}$ it can be expanded as $n = a_0 + a_1 b + \ldots + a_k b^k$ for some $k$. Now writing the coefficients in reverse order gives the usual representation. However, performing a lexicographic comparison requires the sequence of symbols to be right aligned so that coefficients are compared only where their degree is the same. For a left aligned representation a sequence of zeroes needs to be added but for the general case this will not be bounded.

For the rest of this paper it will be assumed that $b = 10$. The problem outlined above can be seen if we try to compare 11 with 7:

```
7
11
```

The lexicographic order does not agree with the numerical order. Analysing this more carefully leads to the conclusion that once a prefix has been used it must be used for all subsequent numbers or a proceeding prefix used. However, since this prefix is finite it must eventually reach a maximum and then be used for the rest of the natural numbers.

For instance:

```
0
1
10
11
```

At this stage all numbers not sharing the prefix could be removed along with the prefix and a countable set of shorter representations would remain. This would still represent all the natural numbers and leads to the conclusion that changing a prefix only affects a finite prefix of the natural numbers. In the most degenerate case this results in unary notation:

```
0
00
000
0000
```

A slightly less denegerate case using all the available symbols is:

```
0
1
⋮
9
90
91
⋮
99
990
991
⋮
```

Looking at the lengths of the representations gives for unary a length equal to $n+1$ and for the second alternative a length equal to $\lceil n/10 \rceil$ which is only a modest improvement. These representations though are useful for *small* numbers and are therefore useful as a component of more complicated representations.

The usual representation has a length of order $log_{10}n$ which is significantly better than the two alternatives listed above. The construction that now follows is a hybrid of both approaches.

## 2.1 A Recursive Representation

The usual reprentation works when the lengths of any two encodings are the same. Where the lengths differ the shorter one always preceeds the longer. The simple solution is to prepend the representations with their lengths. Of course the lengths must be represented in such as way that they can be compared lexicographically. Going back to the first example of 7 and 11 yields:

$$\begin{array}{l}\underline{\texttt{1}}\texttt{7}\\ \underline{\texttt{2}}\texttt{11}\end{array}$$

Which is now correctly ordered and will work for all numbers whose length is no greater than 9. For lengths greater than 9, the process is repeated. So for a length of 10 the representation is prepended with `10` and its length in turn prepended, namely `2`. This almost works but consider 123 and 1234567890:

$$\begin{array}{l}\underline{\texttt{3}}\texttt{123}\\ \texttt{2}\underline{\texttt{10}}\texttt{1234567890}\end{array}$$

The underlining highlights the problem that a sequence of 2 lengths is being compared with a sequence of 1. This is precisely the orignal problem of left-alignment causing numbers of different degrees to be compared. The length of this sequence needs to be prepended but given it is a much *smaller* number the unary representation can be used. Here a new symbol `+` [1] is used that follows all other digits to distinguish it from the regular digits that will follow. This symbol counts in unary the length of the sequence of numbers.

$$\begin{array}{l}\texttt{+7}\\ \texttt{++}\underline{\texttt{3}}\texttt{123}\\ \texttt{+++2}\underline{\texttt{10}}\texttt{1234567890}\end{array}$$

## 2.2 Zero

In many respects zero has length zero since setting $len_{10}(n) = \lceil log_{10}(n+1) \rceil$ is a natural measure of the length. With this in mind zero can be represented by a possibly empty sequence of `0`s, although a single `0` is preferred. This becomes more of an issue for the integers where not handling zero separately would cause it to have both a positive and negative version. When not using additional characters zero needs to have a single prefix sequence, see section 7.

## 2.3 Uniqueness

The mapping from the natural numbers to symbol sequences is neither onto nor one-to-one. The second condition allows for multiple representations of a number since `+1` can be inserted after the last `+` in any number.

$$\begin{array}{r}\texttt{++}\underline{\texttt{2}}\texttt{99}\\ \texttt{+++1}\underline{\texttt{2}}\texttt{99}\\ \texttt{++++}\underline{\texttt{1}}\texttt{1}\underline{\texttt{2}}\texttt{99}\\ \texttt{+++++1}\underline{\texttt{1}}\texttt{1}\underline{\texttt{2}}\texttt{99}\end{array}$$

Using multiple representations will disturb the ordering so a canonical representation is preferred. This can be described by a simple method of constructing the shortest sequence. A short fragment of pseudo-code best explains this where `len` is defined as `ceil(log10(n+1))`.

[1] ASCII does not preserve the order `-01234567890+` as required but other characters can be used

```
elen( n )
    if n > 0 then print '+'
    if len(n) > 1 then elen( len(n) )
    print n
```

The final length for a number $n > 1$ is $k + \Sigma_1^k len_{10}^i(n)$ where $k \geq 1$ is the smallest number such that $len_{10}^k(n) = 1$. Here $len_{10}(n) = \lceil log_{10}(n+1) \rceil$. This is $O(log_{10} n)$ and hence very efficient. A 1000 digit number would take only 1008 symbols.

$$\texttt{+++4}\underline{1000}\ldots$$

## 3 Integers

The integers can be constructed by placing two copies of the positive naturals end to end and reversing the order of the first. Since every positive natural begins with `+` a second copy can be denoted by using `-` in its place [2]. Here `-` preceeds all digits and hence `+` also. This successfully orders all negative numbers before positive ones but the relative order of negative numbers will be incorrect. To reverse this order it suffices to invert each digit $d$ by replacing it with $9 - d$.

For instance

[2]Alternatively `--` can be encoded as `009`, `---` can be encoded as `0009` etc

| | |
|---|---|
| -1234567891 | $---7\underline{89}8765432108$ |
| -1234567890 | $---7\underline{89}8765432109$ |
| -1234567889 | $---7\underline{89}8765432110$ |
| ⋮ | |
| -11 | $--\underline{7}88$ |
| -10 | $--\underline{7}89$ |
| -9 | `-0` |
| ⋮ | |
| -2 | `-7` |
| -1 | `-8` |
| 0 | `0` |
| +1 | `+1` |
| +2 | `+2` |
| ⋮ | |
| +9 | `+9` |
| +10 | $++\underline{2}10$ |
| +11 | $++\underline{2}11$ |
| ⋮ | |
| +1234567889 | $+++2\underline{10}1234567889$ |
| +1234567890 | $+++2\underline{10}1234567890$ |
| +1234567891 | $+++2\underline{10}1234567891$ |

## 4 Small Decimals

For a decimal $r \in (0,1)$ its usual expansion after the point (leading zeroes included but trailing zeroes removed) is left-aligned and naturally ordered. Extending this to small negative decimals requires modification firstly, by adding a sign to negative numbers for instance `-`. For consistency a `+` will be added to positive numbers. Secondly, the digits need to be inverted as for negative integers. Finally, a terminator is required to distinguish longer but lower numbers such as $-0.12$ and $-0.123$ which would be `-87` and `-876` respectively. This can be accomplished by a symbol that proceeds all digits such as `+`. Again for consistency `-` will be added at the end of the postive decimals. Zero can appear as its own digit without a sign enabling the representation of $r \in (-1,1)$.

For example

| | |
|---|---|
| -0.9995 | `-0004+` |
| -0.999 | `-000+` |
| -0.0123 | `-9876+` |
| -0.00123 | `-99876+` |
| -0.0001233 | `-9998766+` |
| -0.000123 | `-999876+` |
| 0 | `0` |
| +0.000123 | `+000123-` |
| +0.0001233 | `+0001233-` |
| +0.00123 | `+00123-` |
| +0.0123 | `+0123-` |
| +0.999 | `+999-` |
| +0.9995 | `+9995-` |

## 5 Large Decimals

Extending small decimals to include those larger than 1 is achieved by adding the integer component separately. Each number is then a pair consisting of its (signed) integer component and a small decimal. Where the integer component is zero but there are digits after the point it must take a sign namely `+0` or `-0`. The integer will be length prefixed so no symbol is required to indicate the point.

This is best shown by example.

| | |
|---|---|
| -100.5 | --$\underline{6}$8994+ |
| -10.5 | --$\underline{7}$894+ |
| -3.145 | `-3854+` |
| -3.14 | `-385+` |
| -1.01 | `-198+` |
| -1 | `-1+` |
| -0.0001233 | `-09998766+` |
| -0.000123 | `-0999876+` |
| 0 | `0` |
| +0.000123 | `+0000123-` |
| +0.0001233 | `+00001233-` |
| +1 | `+1-` |
| +1.01 | `+101-` |
| +3.14 | `+314-` |
| +3.145 | `+3145-` |
| +10.5 | ++$\underline{2}$105- |
| +100.5 | ++$\underline{3}$1005- |

## 6 Floating Point Numbers

A floating point number $f$ takes a mantissa $m \in [\frac{1}{10}, 1)$ and an integer exponent $e$ such that $f = 10^e \times \pm m$. This accounts for all decimals with a finite expansion.

The mantissa will always have a non-zero digit after the point and for each number $f$ will have a unique exponent. This is the normalised representation and ensures comparison can be determined by the exponent where it differs otherwise by the mantissa.

The floating point number is then a triple $(+, e, m)$ or $(-, -e, m)$ consisiting of a sign followed by the exponent and mantissa. The exponent is represented as an integer using the recursive method and is negated for negative numbers to ensure ordering. The mantissa is represented as a postive small decimal but its sign must appear at the front of the entire representation. As with large decimals there is no need to include a delimiter since the exponent is length prefixed. Zero is represented as `0` since a sign symbol will be used for all other numbers. The resulting representations look like the large decimals with a sign added except the integer is used multiplicatively to scale the number rather than as an additive offset. In effect it is a run length encoded version of a large decimal and acts as a compression technique.

These example include a space after the sign for readability.

| | |
|---:|:---|
| $-0.1 \times 10^{11}$ | - --$\underline{7}$888+ |
| $-0.1 \times 10^{10}$ | - --$\underline{7}$898+ |
| -1.4 | - -885+ |
| -1.3 | - -886+ |
| -1 | - -88+ |
| -0.123 | - 0876+ |
| -0.0123 | - +1876+ |
| -0.001233 | - +28766+ |
| -0.00123 | - +2876+ |
| 0 | 0 |
| +0.00123 | + -7123- |
| +0.001233 | + -71233- |
| +0.0123 | + -8123- |
| +0.123 | + 0123- |
| +1 | + +11- |
| +1.3 | + +113- |
| +1.4 | + +114- |
| $+0.1 \times 10^{10}$ | + ++$\underline{2}$101- |
| $+0.1 \times 10^{11}$ | + ++$\underline{2}$111- |

## 7 Compact Prefixes

Adding two extra characters is not always desirable. In this case prefixes can be encoded without them at the expense of slightly longer small numbers since the prefix cannot be omitted. The prefix blocks are of a length upto the character base less two. Working base 16 and showing in hex:

| | |
|---|---|
| -17592186044417 | `63effffffffffe` |
| -17592186044416 | `63efffffffffff` |
| -17592186044415 | `6400000000000` |
| ⋮ | |
| -17 | `6dee` |
| -16 | `6def` |
| -15 | `70` |
| ⋮ | |
| -2 | `7d` |
| -1 | `7e` |
| 0 | `8` |
| +1 | `81` |
| +2 | `82` |
| ⋮ | |
| +15 | `8f` |
| +16 | `9210` |
| +17 | `9211` |
| ⋮ | |
| +17592186044415 | 9*b*`fffffffffff` |
| +17592186044416 | 9*c*`100000000000` |
| +17592186044417 | 9*c*`100000000001` |

This is in effect the same as using dedicated characters but the maximum and minimum characters take the place of `+` and `-` with a trailing value to enumerate a particular block. For consistency these prefixes use the upper half of the character range for positive numbers and the lower half for negative numbers.

```
06
07
1
2
⋮
ffc
ffd
ffe
fff8
fff9
⋮
ffffffd
ffffffe
fffffff8
fffffff9
```

For negative decimals that need a trailing character there is no combination of base characters that will work. However, the decimal component can be encoded over a lower base freeing up trailing characters. For example having an integer component base 16 and then the decimal component base 10 expressed as 10 hex digits (e.g. `3-c`) with `f` and `0` for trailing characters.

## 8 Conclusion

An efficient encoding of various number groups has been demonstrated that has the same length order, namely $log(n)$, as the usual written representation. This has been achieved by only adding two symbols to the alphabet or using a slightly more complex scheme over any symbol set. The constructions shown have been built from the natural numbers using length prefixing or the natural left-alignment of decimal expansions. This is a general technique that can be used to build complex numbers, tuples, etc. These numbers can be mixed with textual fragments to form hybrid strings. A simple example would be unbounded file naming where `+` is any legal character proceeding `9`.

| | |
|---:|:---|
| 1. | `photo+1.jpg` |
| 2. | `photo+2.jpg` |
| 3. | `photo+3.jpg` |
| ⋮ | |
| 10. | `photo++210.jpg` |
| 11. | `photo++211.jpg` |
| ⋮ | |
| 1234567890. | `photo+++2101234567890.jpg` |

## References

[1] "Efficient Lexicographic Encoding of Numbers", P. Seymour, 2008.