Sized Naturals as foundation for hierarchical labeling and extend hexadecimals for any bit string size

draft version 0.4.8 – *Peter Krauss*, *Thierry Jean, José Damico, Alan C. Alves, Everton Bortolini*, *Rui Pedro Julião*.

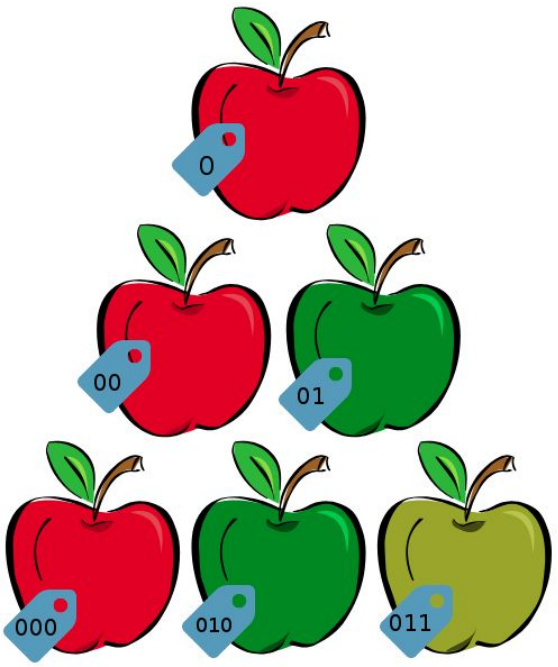
ABSTRACT

When the elements of a nested set are labeled with Natural numbers, the hierarchical structure is not preserved; but when the binary representation of the numeric labels is replaced by variable-length bit strings, and leading zeros are used (00 and 0 as distinct entities), it is possible to express labels with hierarchical syntax, preserving the original nested structure. We formally define this type of numerical labeling system, calling it Sized Naturals, and we show that it can be transformed into human-readable encodings (such as base4, base8 or base16), the base extensions base4h, base8h and base16h. We adopt as a formal reference-model the finite Cantor set — strictly, the hierarchical tree of the Cantor set, which is isomorphic to a Complete binary tree —, with hierarchy degree k, C_k, proposing a simple labeling process, of the elements of C_k mapped into elements of the Sized Naturals. The base extensions proposal use the “half digit” concept, a complementar syntax for ordinary base representation, where last digit can use a complementar alphabet to represent half of values of ordinary digits. We also offer algorithms for implementation of the (base extensions) conversion, and discuss possible optimizations.

key-words: human-readable encoding, hierarchical labeling, leading zeros, base conversion.

Introduction

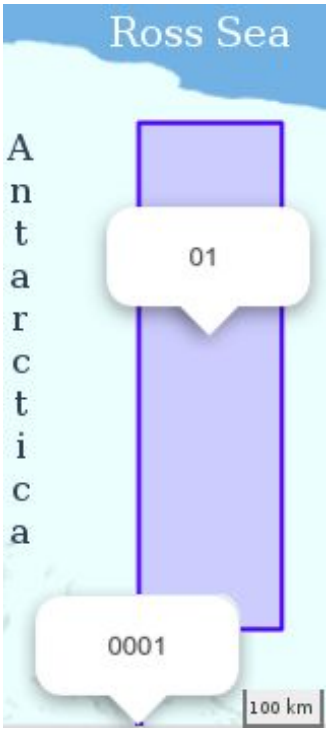
Sometimes we need Natural numbers (ℕ) for counting and labeling, but a kind of "number" where 0 is not equal to 00, in order to represent labels, hashes, hierarchical indexes or any that needs to differentiate 0 and 00, preserving other numeric interpretations, like order (e.g. 011>010) and the freedom to translate its positional notation to some other base (e.g. binary to hexadecimal).



In the set of apples beside, each apple was identified by a variable-length bit string, so the set *A* of the identifiers is
A = { `0`, `00`, `000`, `01`, `010`, `011` }.

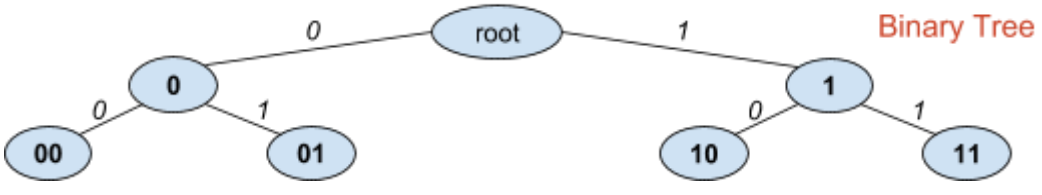
Each identifier of *A* can also be interpreted as a binary number, for example decimal value of binary `01` is 1, or, with base-subscript notation, $[01]_2=[1]_{10}$. Also $[010]_2=[2]_{10}$ and $[011]_2=[3]_{10}$, but the bit string is not a "pure number", there are some loss of information when adopting equivalence in $[0]_2 = [00]_2 = [000]_2 = [0]_{10}$. We need to rescue the number of bits (the apple labels has 1, 2 and 3 bits).

A solution to avoid this loss is transforming each bit string into a pair of decimal numbers (*size,value*):
A' = { (1,0), (2,0), (3,0), (2,1), (3,2), (3,3) }.
Interpreting some samples: the element `0` of *A* was transformed into (1,0) of *A*', it is informing that it has size of 1 bit and numeric value zero, $[0]_{10}=[0]_2$; the element `011` was transformed into (3,3) informing size 3 and value $[3]_{10}=[011]_2$.



Neither the bit strings of *A*, nor equivalent pairs of *A*' has a mathematical special name, but there are many applications using it, so we baptized "**Sized Naturals**". Examples of real life encoding systems that exhibit such labeling characteristics:

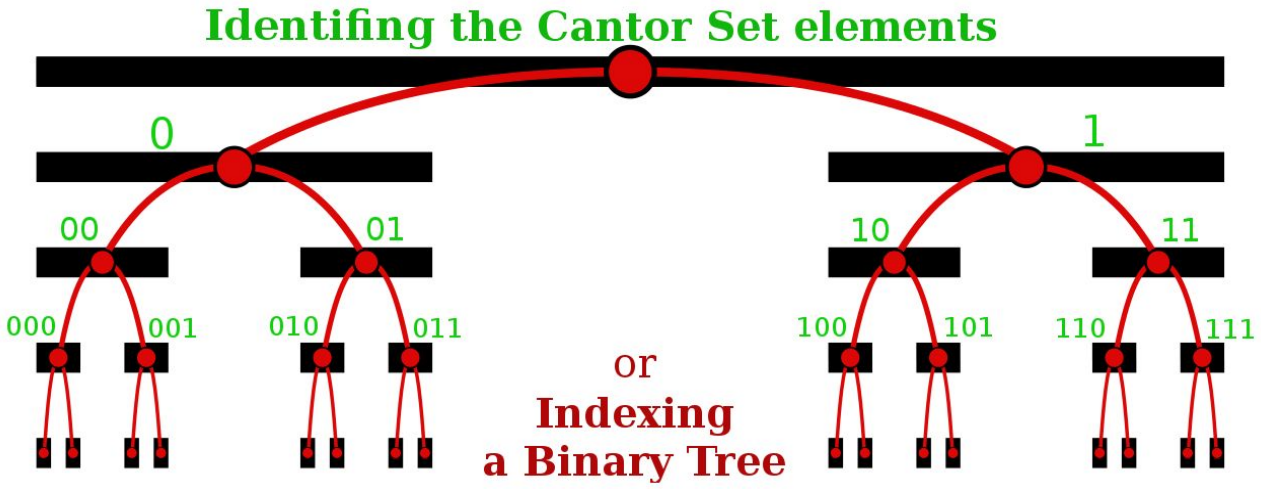
- Two checksums (e.g. CRC32 digests) with different lengths are distinct, we can't eliminate leading zeros.
  For instance `000fa339` is not equal to `fa339`. The CRC32 encoding system use base16 (hexadecimal) as standard representation, but internally CRC32 algorithm uses binary number representation.
- Two Geohashes with different lengths are distinct, e.g. `01` is a cell identifier of a geographic location little below Ross Sea with ~115000 km², and `0001` is a cell far below, with 3.5 km². Both, with same first-digit prefix, are contained into the bigger `0` cell. The Geohash encoding system uses base32 as standard representation, but internally the encode function can use bit string.
- Indexing a binary tree where left edges are labeled 0 and right labeled 1: a node is labeled by the concatenated string of branch labels from the root to the node. Example: to to reach node 01 from root, take the path of edges labeled 0 and 1.



The objective of this article is to define Sized Naturals and a hexadecimal representation for it.

Indexing hierarchical items with bit strings

In the Mathematics tradition the Cantor set is used as reference model to express concepts like self-similarity and subdivision rule. In Computation the equivalent reference model is the complete binary tree. We can use here any one, as reference for indexing and hierarchical labeling. Let's use the Set theory.



Each element of this "hierarchical tree of the Cantor set", a "Cantor bar" (illustrated in black), can be unically labeled by a bit string, therefore an identifier (ID), similar with the labelling of the set of apples. Each ID is also encoding its left (0) or right (1) positions, and its vertical position in the hierarchy of elements — each ID has a prefix that is the ID of its parent. Example: the parent of `011` is `01`.

Adopting the convention that **set X_k** is the set of identifiers of the Cantor bars, limited to k-bits.

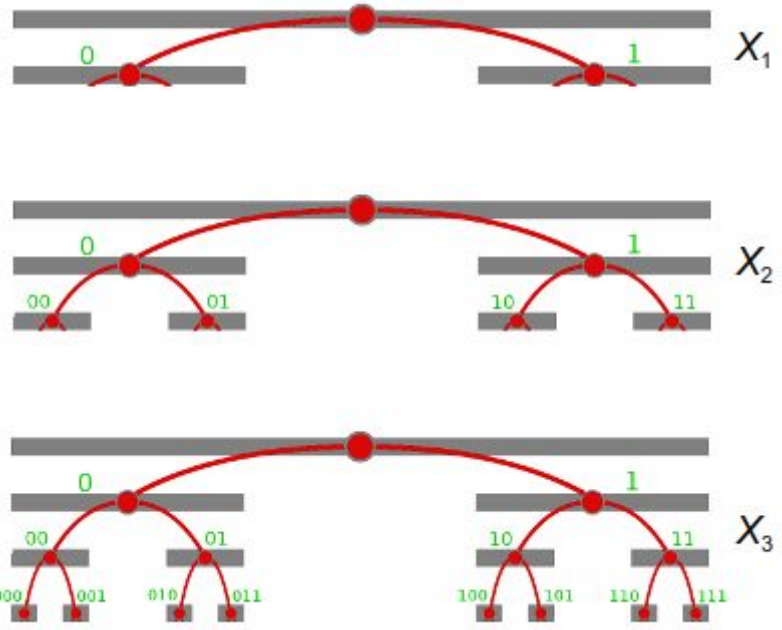
X_1 = {0, 1};

X_2 = {0, 00, 01, 1, 10, 11};

X_3 = {0, 00, 000, 001, 01, 010, 011, 1, ..., 111};

X_k = ...;

X_8 = {0, 00, 000, 000, ..., 11111110, 11111111}.



In usual applications the set X_k is the domain of a set of identifiers (or indexes), and k is finite. Example: the set X_3 is the domain of the illustrated set A of apples, $A \subset X_3$.

There are an intuitive recursive construction rule (illustrated) for each new X_k after X_1

$X_k = P_k \cup X_{k-1}$

where P_k is the set of all of 2^k numbers expressed as fixed-length (k) bit strings. Example:

$X_2 = P_2 \cup X_1 = \{00, 01, 10, 11\} \cup \{0, 1\}$.

The number of elements, $|X_k|$, after $|X_1|=2$, is the recurrence $|X_k| = |X_{k-1}| + 2^k$. Examples, for k ranging from 2 to 8:

$$|X_2| = 2+4=6; \quad |X_3| = 6+8=14; \quad |X_4| = 14+16=30; \quad ...; \quad |X_8| = 254+256=510.$$

By induction we see that $|X_k| = 2^{k+1} - 2$. As suggested by this power k+1, is possible to map any Sized Natural

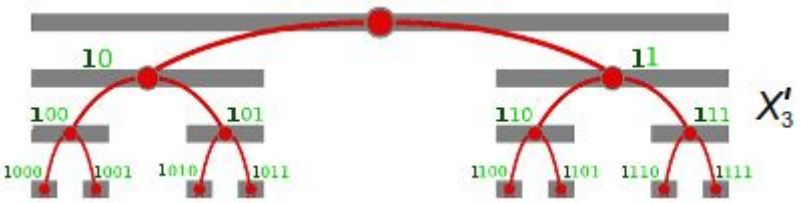
(l,n) to a number $n+2^{l+1}$, because it preserves n and the leading zeros information.

The **hierarchy of the elements of P_k** is visible in the illustration above: the last line, that is the subset P_k of new members, has elements x formed by the concatenation (operator $\oplus$) of prefix p and suffix s, $x=p\oplus s$, where p is the parent identifier and s is the left or right label. Example: the parent `01` of `010`=`01`$\oplus$`0` and `011`=`01`$\oplus$`1`.

The hidden bit implementation strategy

It's possible to "protect leading zeros" and represent a Sized Natural by a Natural number.
Any Sized Natural (l,n) can be mapped to a number $n+2^l$, without loss of the leading zeros.

The illustration beside show the Sized Naturals of k=3, transformed in a set X' of Naturals, where the first bit is the value 2^l added to n of each original (l,n) element of X_3.



This strategy can be used to simplify the internal representation in algorithms and, in some circumstances, as a mathematical alternative to the bit string or ordered pair representations. To "externalize" the internal representation, we must to remove the first bit, so we can say that the first bit is "hidden" from the outside of the algorithm.

Problem on non-binary representations

It is not possible, with usual non-binary positional representations like hexadecimal, to represent all Sized Naturals. Even the most simple, base4, as illustrated by question marks below.

Table 1

(size,value)	BitString	Base4	Base16
(1,0)	0	**?**	**?**
(2,0)	00	0	**?**
(3,0)	000	**?**	**?**
(4,0)	0000	00	0
(5,0)	00000	**?**	**?**
(6,0)	000000	000	**?**
(7,0)	0000000	**?**	**?**
(8,0)	00000000	0000	00
(8,1)	00000001	0001	01
(7,1)	0000001	**?**	**?**
(8,2)	00000010	0002	02
(8,3)	00000011	0003	03
(6,1)	000001	001	**?**
(7,2)	0000010	**?**	**?**
(8,4)	00000100	0010	04
(8,5)	00000101	0011	05
...	...	...	...

The table above shows that the pairs (l,n) of the first column, titled (*size*,*value*), can always be represented by a bit string of the second column, and vice-versa, but not always in base4 or base16.

A bit string x can be converted to a base b only when the number of bits *x_l* of the the bit string is a multiple of $dpd=ceil(log_2(b))$, i. e. the number of bits per digit of the base b. Expressing in terms of remainder (modulo operation), it is only possible when ***x_l* % dpd = 0**.
Examples: in Table-1 the column *base4* haven't question marks when size is 2, 4, 6 or any other even size. It its because when b=4 we need $\log_2(4)$=2 bits/digit. Column *base16* haven't question marks when size is 4, 8 or any

multiple of 4, because when *b*=16 we need $\log_2(16)=4$ bits/digit.

Testing a naive solution

Showing how the most simple strategy not works. A common solution to preserve leading zeros is to use an external symbol instead of zero. For example the RFC 4648 suggests,

> *(...) the use of padding ("=") in base-encoded data is not required or used.* (section 3.2)

Let's use the letter "Z" as external symbol. Using the illustrated set *A* of apples to exemplify:
A = { 0, 00, 000, 01, 010, 011 } = { 0, Z0, ZZ0, 01, Z10, Z11 }
Can we preserve this external symbol when we translate the set to base4?
For example "ZZ", "ZZZZ10" and "ZZ11" have exact translations to base4: "Z", "ZZ2" and "Z3".
But what about "0", "1", "ZZ0" or "ZZ1"?
To translate base2⇒base4 components we need to add more external symbols, suppose A, B, C, D and E:
0⇒A; Z0⇒C; ZZ0⇒ZA; 01⇒1; Z10⇒DA; Z11⇒DE. This translation results in
A' = { A, C, ZA, 1, DA, DE }.

Conclusion: extending base4 alphabet with symbols "Z", "A", ..., "E" does not seem useful. When translating leading zeros from base2, we lost hierarchy visualization. The demand of a lot of new external symbols is also a problem, human-readability seeks something simple, that seems base4 in the most of representation.

We will show in the next sections that there are a reasonable solution extending base4 alphabet,
but not translating the leading zeros, the solution only adds a new last digit when need it.

Formal definition of Sized Naturals

The fundamentals of Set Theory are described in Halmos (1960), the conventions about Natural number set (e.g. adopting 0 as element of $\mathbb{N}$) reinforced in ISO 80000-2:2009. The classic reference-models for *bit string* concept are the “strip of tape of bits” of Turing (1937) and the binary message of Shannon (1948).
“The set of the **Sized Naturals**”, is a particular denomination of “the set of all bit strings”, and the following is its formal definition, oriented to the introduced applications of labeling. The denomination Sized Naturals is also indifferent about the of element's representation, as bit string or as ordinated pair of numbers.

We can transform all elements of $\mathbb{N}$ into bit strings, by the element’s binary representation (base2), forming the set *L* of labels of $\mathbb{N}$. Still, *L* is a subset of Sized Naturals, because bit strings can use leading zeros, representing distinct elements.

The bit-length of a bit string is its number of bits. By other hand, the standard way to express the "size" of a element of $\mathbb{N}$ is by expressing the number of digits in its base2 representation, which is known as the bit-length of the number. To avoid confusion we adopt the "minimum Bit-Length" (minBL) function:

$$minBL(n) = \begin{cases} \lceil log_2(n) + 1 \rceil & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

All Sized Naturals **with bit-length *k*** can be expressed as a set $\boldsymbol{P_k}$

$$P_k = \{\forall x = (k, n) \mid n \in \mathbb{N} \ \wedge \ minBL(n) \leq k\}$$

The aimed finite set $\boldsymbol{X_k}$ **of all Sized Naturals with maximum bit-length *k*** can be defined by recurrence:

$$X_k = \begin{cases} P_k \cup X_{k-1} & \text{if } k > 1 \\ P_1 & \text{if } k = 1. \end{cases}$$

Table-1 shows an example, a sample of X_8 set. Because of finite size of the real-world fractal structures, the set X_k of labels must be mapped into finite Cantor set, as described by Merlo et al. (2003). In this context, the elements of a set P_k can be used as ordered labels of the Cantor bars of level *k*, and, using analog construction rules, all the elements of X_k will be mapped to a full hierarchical structure of level *k*, named by Merlo et al. as “hierarchical tree of the Cantor set”.

Any element of X_k can be expressed as bit string by the function *toBitString*(*l*,*n*), that is the binary representation of *n* with padding zeros to *l*. They are semantically equivalent: the bit string and the ordered pair representations.

The **hierarchy** expressed by the recurrence is embedded in both element representation options:

- as bit string *x* of length *l*, have a prefix of length *l*-1 that is the bit string representation of its parent.
- as ordered pair, *x*=(*l*,*n*), the parent is the pair (*l*-1,*m*) where ***m* = *floor*(*n* / 2)**.

When we not want to to make explicit the hierarchy, we can use a simplified definition of X_k

$$X_k = \{\forall x = (l, n) \mid l, n \in \mathbb{N} \ \wedge \ minBL(n) \leq l \leq k\}$$

The functions *minBL*(*n*) and *toBitString*(*l,n*) are also expressed in Javascript at appendix.

All above definitions also applies to the term “Sized Integers”, used in some implementations. As in usual generalizations from $\mathbb{N}$ to $\mathbb{Z}$, can be accomplished with some control and carrying the minus sign.

Hidden bit representation

The hidden bit strategy is consistent. There are a bijective relation $f: X_k \rightarrow H_k \subset \mathbb{N}$ and its inverse f^{-1} that maps any Sized Natural into a Natural number,

$$f(l,n) = n+2^l \qquad\qquad f^{-1}(m) = [minBL(m)-1\ ,\ m-2^{minBL(m)}].$$

In programming languages we can promote an entity to first-class citizen when it supports all the operations generally available to other primitive entities. That is the case, the elements of H_k are primitive, is possible to use it as primary key in a database, or in optimized sort algorithms, as `m_compare_lexOrder(a,b)` at appendix.

Positional representation of Sized Naturals

Natural numbers can be expressed with positional notation, using the rule of "remove leading zeros". The rule is used in any base (radix) representation. The Sized Natural's representation is like “Natural numbers *without the rule of remove leading zeros*”. The solution to accomplish forbidden conversions is to extend base4 and base16 alphabets, adding a last digit when forbidden, with these new symbols (in bolds below).

Table 2

(size,value)	BitString	Base4h	Base16h	(size,value)	BitString	Base4h	Base16h
(1,0)	0	**G**	**G**	(1,1)	1	**H**	**H**
(2,0)	00	0	**I**	(2,2)	10	2	**K**
(3,0)	000	0**G**	**M**	(3,4)	100	2**G**	**Q**
(4,0)	0000	00	0	(4,8)	1000	20	8
(4,1)	0001	01	1	(4,9)	1001	21	9
(3,1)	001	0**H**	**N**	(3,5)	101	2**H**	**R**
(4,2)	0010	02	2	(4,10)	1010	22	a
(4,3)	0011	03	3	(4,11)	1011	23	b
(2,1)	01	1	**J**	(2,3)	11	3	**L**
(3,2)	010	1**G**	**O**	(3,6)	110	3**G**	**S**
(4,4)	0100	10	4	(4,12)	1100	30	c
(4,5)	0101	11	5	(4,13)	1101	31	d
(3,3)	011	1**H**	**P**	(3,7)	111	3**H**	**T**
(4,6)	0110	12	6	(4,14)	1110	32	e
(4,7)	0111	13	7	(4,15)	1111	33	f
...	...	...	...	...	...	...	...
(8,29)	...	0131	1d	(10,117)	...	01311	1d**J**
(9,58)	...	0131**G**	1d**G**	(9,59)	...	0131**H**	1d**H**
(10,116)	...	01310	1d**I**	(10,118)	...	01312	1d**K**

This table shows the adopted solution, that not affects prefix hierarchy. It will be explained in the next subsections.

Base2

The base2 representation is the simplest and the **canonic** one. The Sized Natural bit string representation is the ordinary base2 augmented with the "use leading zeros" rule (`00` and `0` are distinct elements).

Base4h

How to convert base2 one-digit numbers `0` and `1` to base4?
The translations $[00]_2=[0]_4$ and $[01]_2=[1]_4$ make sense for bit strings, but, what about $[0]_2=[?]_4$ and $[1]_2=[?]_4$, the one bit translations?

The solution is to use a fake digit that represent these values. To avoid confusion with hexadecimal letters we can start with `G` to represent `0` and `H` to represent `1`. It will be named **half digit values** because its numeric values uses a half of bits of the ordinary's base4 digit values. The "**base4** with extended alphabet for **half** digits" was shortened to "base**4h**". Table-2 is illustrating base4h representation of all elements of the X_4 set.

> NOTE: to use base-subscript notation without ambiguity in base2 (where $[00]_2=[0]_2$), for example in $[0]_2=[G]_{4h}$, we can add "h" to say that it is a bit string of Sized Naturals, where $[00]_{2h} \neq [0]_{2h}$.

Base4h numbers are strings with usual base4 pattern and the *halfDigit* as optional suffix. This syntax rule, to recognize arbitrary base4h codes, can be expressed by a regular expression:

```
/^([0123]*)([GH]?)$/
```

The inverse, to translate from bit string with *b* bits, when *b* is even we can use ordinary base4 conversion, and when *b* is odd concatenate the *halfDigit*. Splitting (e.g. with Javascript) the value as prefix and suffix parts,

```
let part = bitString.match(/^((?:[01]{2,2})*)([01]*)$/)
```

the prefix (part[0]) will be translated to usual base4 number, and the suffix (part[1]), when exists (a remaining last bit), translated to halfDigit by this JSON map: `{"0":"G","1":"H"}`.

Example: to convert `001010010` into base4h, split into parts, part[0]=`00101001` of 2-bits blocks from begin, that will result in "0221", and part[1]=`0`, of remaining bit, resulting in "G". Concatenating part results, "0221G".

Base16h

This encoding extension for base16 was inspired in the base4h encode. It uses the same “half digit” concept: a complementar syntax for ordinary base representation where last digit can use an alternative alphabet to represent half of values of ordinary digits.

We can use base16 (hexadecimal representation) for any integer, but when controlling the bit-length can use only base16-compatible lengths: 4 bits, 8 bits, 12 bits, ... multiples of 4.
So, how to transform into base16 bit strings as `0`, `1`, `00`, `01`, `10`, ... ?

The solution is to extend a hexadecimal representation, in a similar way to the previous one used for base4h: the last digit as a fake-digit that can represent all these incompatible values — so using the *halphDigit* values `G` and `H` for 1-bit values, and including more values for 2 bits (4 values) and 3 bits (8 values). The total is 2+4+8=14 values, they can be represented by the letters `G` to `T`.

The name of this new representation is **base16h**, because it is the ordinary base16 "plus an optional halfDigit", by “h” shortening “half”. Its string pattern is:

```
/^([0-9a-f]*)([G-T])?$/
```

The inverse, to translate from bit string with *b* bits, there are *b*%4 last bits to be translated to a *halfDigit*. Splitting (e.g. with Javascript) the value as prefix and suffix parts,

```
let part = bitString.match(/^((?:[01]{4,4})*)([01]*)$/)
```

the prefix (*part[0]*) will be translated to usual hexadecimal number, and the suffix (*part[1]*), when exists (with 1, 2 or 3 last bits), translated by this "last bits to halfDigit" JSON map:

```
{ "0":"G","1":"H",
  "00":"I","01":"J","10":"K","11":"L",
  "000":"M","001":"N","010":"O","011":"P","100":"Q","101":"R","110":"S","111":"T"
}
```

Example: to convert `0010100101` into base16h, split into part[0]=`00101001` of 4-bits blocks from begin, and part[1]=`01`, of remaining bits. Convert part[0] into ordinary hexadecimal (`00101001` is “29”), and part[1] by the JSON table above (`01` is “J”), so it results in “29J”.

Base8h

This encoding is less usual, but use the same pattern and implementation tham base16h.

String-detection pattern: `/^([0-9a-f]*)([G-T])?$/`
Split into parts: `bitString.match(/^((?:[01]{3,3})*)([01]*)$/)`
JSON map for halfDigit: `{ "0":"G","1":"H", "00":"I","01":"J","10":"K","11":"L" }`

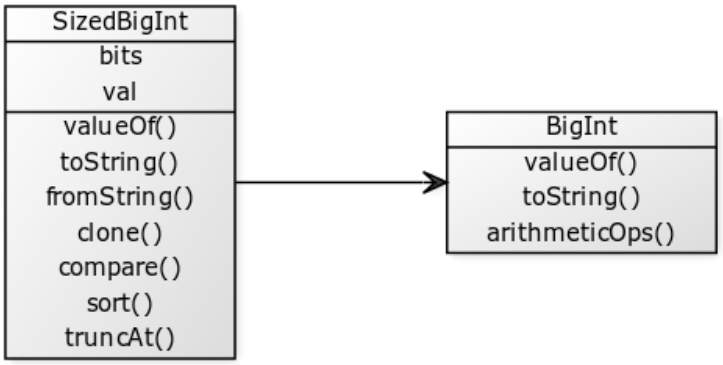
Example: to convert `0010100101` into base8h, split into part[0]=`001010010` of 3-bits blocks from begin, and part[1]=`1`, of remaining bits. Convert part[0] into ordinary octal (`001010010` is “122”), and part[1] by the JSON table above (`1` is “H”), so it results in “122H”.

Algorithms

All algorithms and conventions fixed in this article was tested using Javascript for Web context constraints and performance evaluations. The Sized Natural concept was adapted to the Javascript primitive type BigInt. In a future release we will develop C++ code to run as WebAssembly Javascript object.

SizedBigInts

Sized BigInt's are arbitrary-precision integers with defined number of bits, was implemented with license CC0 as Javascript class, that uses primitive datatype BigInt in internal representation. It can be accessed at https://github.com/ppKrauss/SizedBigInt.



The algorithm described below was tested by implementations using the SizedBigInt class. The class also was extend to represent discrete global grid cell identifiers, like Geohash, and its encoding options.

To store internally a Sized Natural of our formal definition, there are many ways. The choice is a matter of context (database or interface), simplicity, performance. Main options for internal representation:

- *s* bit string. Supposing some "finesh mark" (S2 example) to control the string length. Two ways:
  - bits: real buffer of bits.
  - characters: using the ASCII characters "0" and "1", in a text string.
- (*l*,*s*) length and bit string. When *s* have no "finish mark", the information is stored in a small integer.
- (*l*,*n*) length and number. Typically some native "big integer" for *n*. (bits,val) in our implementation.
- *m* only a number, using the hidden-bit strategy. The same big integer *n* of (*l*,*n*) pair, but using more one bit to protect leading zeros.

When the application use a limited (maximum) length, any of these strategies of internal representation can use this limit to optimize performance. The 32 and 64 bits are usual limits.

Comparing codes

There no constraint in our Sized Natural definition about order, any one is valid. We adopted the *bit string lexicographic order* as canonic (see the line sequence at tables 1 and 2), that corresponds, in the Complete Binary Tree models, to the pre-order traversal. See functions `_compare_lexOrder(a,b)` at appendix. It is canonic because in a listing it groups same-prefix itens. The most simple, `pair_compare_lexOrder(a,b)` uses `pair_toBitString(l,n)` for basic string comparison. For best performance the ideal internal representation is *m* (hidden bit), that is direct numeric comparison — can be tested with `m_compare_lexOrder(a,b)`.

The library offers also the level-order. The `m_compare_levelOrder(a,b)` function it is a integer comparison. The `pair_compare_levelOrder(a,b)` function is implemented by *l* and *n* numerical comparisons: when a.*l*=b.*l* it compares a.*n* and b.*n*.

Trucating

The most usual is to truncate as prefix, see `_truncate(x,bits)` function at appendix. Sometimes is useful to split a Sized Natural into both, prefix and suffix new Sized Natural.

Base conversion

Mathematical libraries, like the native Javascript BigInt, have good performance in ordinary base conversion (see *BigInt.toString*(*radix*) method). The most frequently used ones are 4, 8,16, 32 and 64. The alphabet can be controlled by some standardization, see example in the appendix. The base4h and base16h, as shown above, have additional performance cost to split the bit string into prefix (that use fast native conversion of ordinary base) and halfDigit, that is a fast key-value conversion.

Terminology

Base: the web standards, as RFC 4648, use the term "base", but Javascript (ECMA-262) adopted the term "radix" in `parseInt(string,radix)`. The preferred term is *base*.

Base alphabet: is the "encoding alphabet", a set of UTF-8 symbols used as digit values of a specific base.

Base label: each pair (base,alphabet) need a short label. In the SizedBigInt class some labels was defined: base2, base4, base4h, base8, base16, base16h, base32, base32ghs, base32hex, base32pt, base32rfc, base64, base64url, base64rfc. See also the ID column of the *catalog-base.csv* file at class's git repository.

Default alphabet: is the alphabet adopted as standard for a specific base, associated with the label "baseX", for example "base4" is a synonymous for "base4js", the ECMA-262 standard for it. See *base label***.**

Padding: SizedBigInt's are numbers where padding zeros make difference (0 is not equal to 00). The RFC 4648 is not for numbers, adopted the convention of pad "=" characters at the end of encoded data.

Set, **element**, **number**, **Natural number** and **Integer** are terms of the Set theory (Halmos 1960), the formal mathematical foundation used here. In implementation context the **Javascript** semantic for integer, class, number, etc. is preferred.

Size and **length**: the term "size" was used in the title of this project, but the usual term for "size of the string" is length and, for binary numbers, bit-length, the preferred term.

Sized BigInt: an adaptation of the term "Sized Integer" for programming languages, like Javascript, where the term BigInt is used to express integer numbers.

Sized Integer: the formal definition expressed in this article is about Natural numbers (positive Integers), but it is easy to generalize, was only a simplification to avoid signal analysis and to reuse implementations.

Unsigned Sized Integer: synonym for Sized Natural. In Computation the term "unsigned integer" is preferred, instead "Natural number".

References

ECMA-262 v9.0 (2018), "ECMAScript® 2018 Language Specification".

ISO 80000-2:2009, "Quantities and units—Part 2: Mathematical signs and symbols to be used in the natural sciences and technology", Chapters 5 and 6.

J. Ferreirós (2007), "Labyrinth of Thought, A History of Set Theory and Its Role in Modern Mathematics", Second revised edition. ISBN 978-3-7643-8349-7.

P. Halmos (1960), "Naive Set Theory". ISBN 978-1-61427-131-4.

RFC 4648 (2006), "The Base16, Base32, and Base64 Data Encodings".

C. E. Shannon (1948), “A Mathematical Theory of Communication”. urn:doi:10.1002/j.1538-7305.1948.tb01338.x.

L. A. Steen, J. A. Seebach (1995), “Counterexamples in Topology”. ISBN 978-0-486-68735-3.

A. M. Turing (1937), “On Computable Numbers, with an Application to the Entscheidungsproblem”. urn:doi:10.1112/plms/s2-42.1.230.

D. R. Merlo, J. A. R. Martín-Romo, T. Alieva, M. Calvo (2003), “Fresnel Diffraction by Deterministic Fractal Gratings: An Experimental Study”, urn:doi:10.1134/1.1595227 (open)

Appendix - supplementary material

Basic Sized Naturals functions

Using only as functional specification and didactic reference, not optimized implementations neither offering pre-conditions or exceptions. All in Javascript, input *l* can be a *Number(l)* and *n* must be a *BigInt(n)*. BigInt literals (as *2n*) for some compilers must replaced by a constant definition, e.g. *const TWO = BigInt(2)*.

Utility functions:

```
function minBL(n){
 // calculates bigInt_log2(n)+1, the minimum bit length of BigInt n.
 for(var count=0; n>1n; count++)  n = n/2n
 return count+1
}
function pair_minBL(x) {  return minBL(x[1])  }
function m_minBL(m) { return minBL(m)-1 }
```

Functions using pairs (*l*,*n*) as internal representation:

```
// javascript Array convention x[l,n]. So x[0] is l and x[1] is n.

function pair_toBitString(x) {
  // transforms Sized BigInt of (l,n) representation into a bit string representation.
  return x[1].toString(2).padStart(x[0],'0')  // l is x[0] and n is x[1]
}

function pair_fromBitString(s) {
  let l = s.length
  let n = BigInt("0b"+s)
  return [l,n]
}

function pair_truncate(x,bits) {
  return pair_fromBitString( pair_toBitString(x).slice(0,bits) );
```

```
}


function pair_compare_lexOrder(a,b) {
  // compare two SizedBigInt arrays (e.g. a[0]=l and a[1]=n)
  let str_a = pair_toBitString(a)
  let str_b = pair_toBitString(b)
  return (str_a>str_b)? 1: ( (str_a==str_b)? 0: -1 )
}

function pair_compare_levelOrder(a,b) {
  let bitsDiff = a[0] - b[0]   // compare bitLengths (l)
  if (bitsDiff) return bitsDiff;
  else { // when equal lengths, compare BigInts (n)
    let valDiff = a[1] - b[1]
    return valDiff? ((valDiff>0n)? 1: -1): 0
  }
}

function pair_lexOrder_next(x,maxBits=null,cycle=false) {
       // the successor of x in a context of lexicographical order, returning string
       let t = x[0]  // x[0] is the size, x[1] the value
       if (!t) return null;
       if (!maxBits) maxBits=t; else if (t>maxBits) return null;
       let s = pair_toBitString(x)
       if (t<maxBits) return s+'0';
       t--
       if (s[t]=='0') return s.slice(0,t)+'1';
       else return (s==''.padEnd(maxBits,'1'))? (cycle?'0':null): s.slice(1);
}
```

Functions using *m* of the hidden bit strategy as internal representation:

```
function m_toBitString(m) {
  // transforms Sized BigInt of m representation into a bit string representation.
  return (m===null)? '': m.toString(2).slice(1)
}

function m_fromBitString(s) {
  return s? BigInt("0b1"+s): null
}

function m_truncate(m,bits) { // can be optimezed
  return m_fromBitString( m_toBitString(m).slice(0,bits) )
}



function m_compare_lexOrder(a,b) {
  // compare two SizedBigInt of m representation
```

```
  let dif;
  let bdif = m_minBL(a) - m_minBL(b)
  if (bdif) {
       dif = (bdif>0)
         ? a/BigInt(2**bdif) - b      // normalize a
         : a - b/BigInt(2**(-bdif));  // normalize b
       if (!dif) dif = bdif;    // 0 before 00, 101 before 0101 etc.
  } else
      dif = a - b
  return dif? ((dif>0n)? 1: -1): 0
}
function m_compare_levelOrder(a,b) {
  let dif = a - b
  return dif? ((dif>0n)? 1: -1): 0
}
```

Encoding alphabets and conventions

Complete list of standard alphabets for base conversion. ID syntax: base+alphabetLabel.

base	alphabet label	id	bits /digit	alphabet (after space halfDigits)	Reference standard
2	js*	base2js	1	01	ECMA-262
4	js*	base4js	2	0123	ECMA-262
4	h	base4h	2	0123 GH	ECMA + HalfDigit for SizedBigInt
8	js*	base8js	3	01234567	ECMA-262
8	h	base8h	3	01234567 GH IJKL	ECMA + HalfDigit for SizedBigInt
10	iso*	base10iso	3.32	0123456789	ISO 80000-2:2009
16	js*	base16js	4	0123456789abcdef	ECMA-262 and RFC 4648/sec8
16	h	base16h	4	0123456789abcdef GH IJKL MNOPQRST	ECMA + HalfDigit for SizedBigInt
20	js*	base20js	4.32	0123456789abcdefghij	ECMA-262
20	olc	base20olc	4.32	23456789CFGHJMPQRVWX	OLC, Open Location Code
32	hex*	base32hex	5	0123456789abcdefghijklmnopqrstuv	ECMA-262 and RFC 4648/sec7
32	ghs	base32ghs	5	0123456789bcdefghjkmnpqrstuvwxyz	Geohash
32	nvu	base32nvu	5	0123456789BCDFGHJKLMNPQRSTUVWXYZ	No-Volgals except U (near non-silabic)
32	rfc	base32rfc	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ234567	RFC 4648/sec6
36	js*	base36js	5.17	0123456789abcdefghijklmnopqrstuvwxyz	ECMA-262
58	btc*	base58btc	5.86	123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz	Bitcoin convention
64	url*	base64url	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_	RFC 4648/sec5
64	rfc	base64rfc	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/	RFC 4648/sec4

(*) default base. For example base32 is interpreted by default as base32hex.