Radix-less Symbolic Efficiency

By Sven Nilsen, 2024

In this paper I explain why ternary computers have higher maximal symbolic efficiency theoretically than binary computers. To do this, I use a function for computing the permutative maximal sub-group product. This product corresponds to the maximal capacity of counting for any finite number of discrete symbols. I use this insight to construct an efficient radix-less system.

A computer is a machine that manipulates symbols. The architecture of computers can vary, but the usual approach is to have a place for memory and some central unit for processing information. Traditionally, computers have been designed to operate with binary states, `0` and `1`. However, some experimental designs have been based on ternary computers. These ternary designs have often been more efficient than binary, within reasonable ways to compare designs that eliminate bias.

The problem is: How can we compare binary versus ternary or other radixes in general in a way that does not depend on eliminating particular biases that hold for some classes of designs?

With other words, the idea is to develop a method to determine maximal symbolic efficiency.

Consider starting with N number of symbols. You can group them in any way, for example, pairing the symbols two by two such that the whole collection forms a binary number system. If you have 10 symbols, then using a single sub-group only allows representing 10 states. However, if you pair the 10 symbols into 5 sub-groups, then the number of states that can be represented is `2^5 = 32`. This means binary systems in general have higher symbolic efficiency than unary systems.

For any N number of symbols, the highest capacity one can construct from any grouping of the symbols into sub-groups is given by the number sequence https://oeis.org/A000792.

What is not easily inferred from this number sequence, is that the numbers for index 2 or greater is a product of the primes 2 and 3, where there is a maximum two 2s. With other words, most sub-groups for large values of N have size 3.

This shows that ternary systems have higher symbolic efficiency than binary systems on average, when all restrictions on designs are removed. There might be a constraint on the physical universe that ultimately results in binary systems being more efficient, but by default one should rationally assume that ternary systems are more efficient unless there is some evidence to suggest otherwise.

Ternary systems store more information than binary systems using fewer parts for computation, without necessarily knowing what these parts are materialistically. If you have some ternary system A which is more efficient than a binary system B, then there might be some way to optimize B into a new design B2 that is more efficient than A. However, if B can not be optimized, then the most optimal system is A, or some better design than A based on ternary computation.

Consider the following states in a 2-bit binary system:

00 01 10 11

The position of `0` and `1` in this system is used to distinguish between symbols. The symbols are:

0 in the first position, 1 in the first position, 0 in the second position, 1 in the second position

The definition of a symbol here, is the combination of position and value that makes it unique.

In this sense, the efficiency of symbolic reasoning is not measured in bits or trits, but by using the number of total distinct symbols which efficiency is the product of the sub-groups of symbols.

For binary systems, N bits give the following M distinct symbols:

$$M = 2 * N$$

For ternary systems, N trits give the following M distinct symbols:

$$M = 3 * N$$

However, using distinct symbols leads to a new problem: If ternary systems are more efficient than binary systems, then there is not always a one-to-one correspondence between the amount of states.

The actual problem here is the convention that numerical notation does not encode the radix. In some sense, the radix expresses the geometry in which the number is a position. Simply encoding a one-dimensional radix does not produce a practical notation, since one tends to favor a fixed radix instead of bridging the semantic gap between various needs for symbols and their required efficiency. In one sense, the key is to be able to control the symbolic efficiency by demand.

The solution is to create a practical numerical notation using multiple common bases of 2 and 3:

00	01			**Base 2**			630	631	632	**Base 3x3x3**
							640	641	642	
10	11	12		**Base 3**			650	651	652	
							730	731	732	
20	21	22	23	**Base 4**			740	741	742	
							750	751	752	
30	31	32		**Base 3x3**			830	831	832	
40	41	42					840	841	842	
50	51	52					850	851	852	
420	421	430	431	520	521	530	531			**Base 2x2x2**

In this notation, every number informs the reader about the base. When used with separators, one does not need to specify the radix. Any number in the sequence A000792, which are maximal symbolic efficient for any finite number of symbols, can be reached using Cartesian products of the radix-less numerical notation. At the same time, there are only 0-8 digits being used, which makes it compatible as a subset of standard decimal notation.
The number 9 is never used in the radix-less numerical notation, which makes it possible to use it as a separator. For example, when needing a number system with Base 6, one can use Base 2 and Base 3 separated by 9:

00910	(0 in Base 2, 0 in Base 3)
01912	(1 in Base 2, 2 in Base 3)

This produces the cyclic sequence:

00910, 00911, 00912, 01910, 01911, 01912, 00910, …

When increasing by one, an algorithm can output the next number while preserving the base.

The notation above fixes the placement of symbols for random memory access in some text file. This solves a major problem with standard decimal notation that requires sequential reading to solve for ambiguities with numbers starting with `0`s. By fixing the number of symbols used, the operating system can start reading the file at the precise location where the number is. The problem is usually not to keep around enough memory to store data, but to access or change the data quickly.

Without making such advancements in numerical notation, one would have to build computers from scratch using ternary gates to take advantage of better symbolic efficiency. This is a very costly investment for the world today that is using binary systems everywhere.

Notice that in Base 2, the prefix `0` is redundant and can be removed:

```
00      =>      0
01      =>      1
```

However, this requires reading the length of the number for Base 3, which starts with `1`. Binary data is common, so it is often better to just store blobs of binary data when needed. The benefit of not having to read the whole number to access the whole data file and have a well defined semantics from a fixed position in the file, outweights the benefit of slightly better compression. The point is that in the human readable version of the data, a human user can interpret the number efficiently.

There is another trick that one might use:

```
42094209420             (0 in Base 2x2x2, 0 in Base 2x2x2, 0 in Base 2x2x2)
```

Since 9 works as a separator, which is used for readability when there is no other character like space ` ` available, one can take advantage that the radix-less numerical notation is never ambiguous by simply removing `9` everywhere:

```
420420420               (0 in Base 2x2x2, 0 in Base 2x2x2, 0 in Base 2x2x2)
```

It can be difficult for a human reader to interpret such numbers, but a computer has no problems.

As an exercise, one might try interpret this number:

```
52121
```

This might seem ambiguous because it looks like `52` is Base 3x3 followed by `12` in Base 3. However, the remaining `1` does not correspond to any number. The correct answer is `521921`. Reading a number without any separator in radix-less numerical notation can be difficult for humans, but once the placement is sorted out, a computer can access each symbol directly. When a computer changes the number by random access, it can modify it while preserving the format.