Truncated binary encoding

From Wikipedia, the free encyclopedia

Truncated binary encoding is an entropy encoding typically used for uniform probability distributions with a finite alphabet. It is parameterized by an alphabet with total size of number n. It is a slightly more general form of binary encoding when n is not a power of two.

If n is a power of two then the coded value for $0 \le x < n$ is the simple binary code for x of length $\log_2(n)$. Otherwise let $k = \text{floor}(\log_2(n))$ such that $2^k \le n < 2^{k+1}$ and let $u = 2^{k+1} - n$.

Truncated binary encoding assigns the first u symbols codewords of length k and then assigns the remaining $n - u$ symbols the **last** $n - u$ codewords of length $k + 1$. Because all the codewords of length $k + 1$ consist of an unassigned codeword of length k with a "0" or "1" appended, the resulting code is a prefix code.

Contents [hide]

Example with $n = 5$ [edit]

For example, for the alphabet {0, 1, 2, 3, 4}, $n = 5$ and $2^2 \le n < 2^3$, hence $k = 2$ and $u = 2^3 - 5 = 3$. Truncated binary encoding assigns the first u symbols the codewords 00, 01, and 10, all of length 2, then assigns the last $n - u$ symbols the codewords 110 and 111, the last two codewords of length 3.

For example, if n is 5, plain binary encoding and truncated binary encoding allocates the following codewords. Digits shown ~~struck~~ are not transmitted in truncated binary.

Truncated binary	Encoding			Standard binary
0	~~0~~	0	0	0
1	~~0~~	0	1	1
2	~~0~~	1	0	2
UNUSED	~~0~~	~~1~~	~~1~~	3
UNUSED	~~1~~	~~0~~	~~0~~	4
UNUSED	~~1~~	~~0~~	~~1~~	5/UNUSED
3	1	1	0	6/UNUSED
4	1	1	1	7/UNUSED

It takes 3 bits to encode n using straightforward binary encoding, hence $2^3 - n = 8 - 5 = 3$ are unused.

In numerical terms, to send a value x where $0 \le x < n$, and where there are $2^k \le n < 2^{k+1}$ symbols, there are $u = 2^{k+1} - n$ unused entries when the alphabet size is rounded up to the nearest power of two. The process to encode the number x in truncated binary is: If x is less than u, encode it in k binary bits. If x is greater than or equal to u, encode the value $x + u$ in $k + 1$ binary bits.

Example with $n = 10$ [edit]

Another example, encoding an alphabet of size 10 (between 0 and 9) requires 4 bits, but there are $2^4 - 10 = 6$ unused codes, so input values less than 6 have the first bit discarded, while input values greater than or equal to 6 are offset by 6 to the end of the binary space. (Unused patterns are not shown in this table.)

Input value	Offset	Offset value	Standard Binary	Truncated Binary
0	0	0	~~0~~000	000
1	0	1	~~0~~001	001
2	0	2	~~0~~010	010
3	0	3	~~0~~011	011
4	0	4	~~0~~100	100
5	0	5	~~0~~101	101
6	6	12	0110	1100
7	6	13	0111	1101
8	6	14	1000	1110
9	6	15	1001	1111

To decode, read the first *k* bits. If they encode a value less than *u*, decoding is complete. Otherwise, read an additional bit and subtract *u* from the result.

Example with $n = 7$ [edit]

Here is a more extreme case: with $n = 7$ the next power of 2 is 8 so $k = 2$ and $u = 2^3 - 7 = 1$:

Input value	Offset	Offset value	Standard Binary	Truncated Binary
0	0	0	~~0~~00	00
1	1	2	010	010
2	1	3	011	011
3	1	4	100	100
4	1	5	101	101
5	1	6	110	110
6	1	7	111	111

This last example demonstrates that a leading zero bit does not always indicate a short code; if $u < 2^k$, some long codes will begin with a zero bit.

Simple algorithm [edit]

Generate the truncated binary encoding for a value *x*, 0 <= *x* < *n*, where *n* > 0 is the size of the alphabet containing *x*. *n* need not be a power of two.

```
string TruncatedBinary (int x, int n)
{
        // Set k = floor(log2(n)), i.e., k such that 2^k <= n < 2^(k+1).
        int k = 0, t = n;
        while (t > 1) { k++;  t >>= 1; }

        // Set u to the number of unused codewords = 2^(k+1) - n.
        int u = (1 << k+1) - n;

        if (x < u)  return Binary(x, k);
        else  return Binary(x+u, k+1));
}
```

The routine *Binary* is expository; usually just the rightmost *len* bits of the variable *x* are desired. Here we simply output the binary code for *x* using *len* bits, padding with high-order 0's if necessary.

```
string Binary (int x, int len)
{
```

```
        string s = "";
        while (x != 0) {
                if (even(x))  s = '0' + s;
                else  s = '1' + s;
                x >>= 1;
        }
        while (s.Length < len)  s = '0' + s;
        return s;
}
```

See also [edit]

- Benford's law
- Golomb coding

Categories: Lossless compression algorithms