Numerical Conversion to Mixed Base without Division – and Vice-Versa

Dick Grune
dick@dickgrune.com

Feb. 2, 2011

1 Introduction

As soon as there were computers they were used for numerical conversion, in base 10 from binary to decimal, but also in mixed base, for example from pennies to £sd. Numerical conversion is usually done by repeated division, but not all of the early computers had a divide instruction, so other means were required. Division can of course be simulated by repeated subtraction, but that is quite inefficient. The algorithm presented here does conversion, possibly to mixed base, in a few instructions per digit; I think I saw the technique first in code that came with the (British) Elliott 503 computer in the late 1960s. That computer had a divide instruction, so the technique itself must be older.

2 Algorithm

The algorithm computes the next digit d for an input value v by first repeatedly subtracting a suitable number, the **downstep** from v, until it is negative or zero, while at the same time raising d appropriately; and then in the same fashion adding a **correction** (usually a unit) to v until it is positive or zero, while lowering d. The downsteps and corrections are kept in a table, two entries for each digit. For example, the table for computing the tens in decimal conversion is

```
{30, 3},
{10, 1},
```

which means first subtracting thirties from v while raising d digit by threes, and then adding tens to v while lowering d by ones. Each of these actions keeps the value of the combination of v and d unchanged.

The initial value for d is obtained from a template; Figure 1 shows the code. The variable e is a pointer to the pertinent table entry; the computation of one digit moves it two entries forward. The algorithm converts a number to base 10 at a cost of 4.3 actions per digit.

This table-based approach is more efficient than just counting down (6.5 actions/digit), but equally importantly, allows the individual specification of the base for each digit.

```
set e to point to the top of the conversion table
while template is not exhausted:
    get next digit d from template (usually a '0')
    while v > 0:
        decrease v by e->value
        increase d by e->repr
    shift e to next table entry
    while v < 0:
        increase v by e->value
        de rease d by e->repr
    insert d into the string
    shift e to next table entry
```

Figure 1. Code for converting v to a string.

Template: `"L00.00s.00d."`

```
{10*10*20*12, 1},    /* skip the pounds sign */
{10*10*20*12, 1},
{ 3*10*20*12, 3},    /* ten pounds */
{ 1*10*20*12, 1},
{    3*20*12, 3},    /* one pound */
{    1*20*12, 1},
{    1*20*12, 1},    /* skip the dot */
{    1*20*12, 1},
{    1*10*12, 1},    /* ten shillings */
{    1*10*12, 1},
{       3*12, 3},    /* one shilling */
{       1*12, 1},
{       1*12, 1},    /* skip the s */
{       1*12, 1},
{       1*12, 1},    /* skip the dot */
{       1*12, 1},
{       1*10, 1},    /* ten pence */
{       1*10, 1},
{          3, 3},    /* one penny */
{          1, 1}
```

Figure 2. Table for conversion from pennies to £sd.

```
set e to point to the top of the conversion table
while template not exhausted:
    if e->repr:
        get next digit d from template
    while v > 0:
        decrease v by e->value
        increase d by e->repr
    if e->repr < 0:
        insert d into the string
    shift e to next table entry
    replace v by -v
```

Figure 3. Code for converting v to a string with one while loop less.

The table in Figure 2 converts sums up to £99.19s.11d. from pennies, starting from the template `L00.00s.00d.`. The non-numeric characters in the template are skipped by treating them as digits of base 1. The bases of the "big" digits, those for shillings and pence, are 20 and 12 resp., but their values must be printed in the decimal system; the table shows how this is handled. Although the downsteps and corrections, 24000, 7200, etc. could be entered directly in the table, it is more convenient and understandable to write them down as the product of their factors, to show how they are constructed.

The downstep can be chosen arbitrarily, as far as correctness is concerned. As long as the correction is the unit value of the digit under conversion, the algorithm will always give the correct answer, but some downstep values are more efficient than others. If the downstep is equal to the unit, division by simple subtraction results. The downstep in the tables was chosen to approximate the square root of the base. This seems intuitively a good choice; below we show that it indeed is.

The algorithm is not very robust: converting a negative value or one that does not fit the template results in (understandable) garbage. For example, converting -1 with the template `0000` yields `/999`. Here `/` represents the digit `-1`, and indeed $(-1) * 1000 + 9 * 100 + 9 * 10 + 9 = -1$.

In good 1960s style the two while loops in Figure 1 can be combined into one.[1] The trick is to keep v positive by inverting it for the second while statement, and correct the effect by inverting the digit correction in the corresponding entry in the table. The code for this is in Figure 3. It is to be used with a table containing entries like

```
{30, 3},
{10,-1},
```

and contains more trickery than I care to explain.

A working demo of the above algorithms in C can be found in the file `demo.c`.

[1] This was the time that E.W. Dijkstra, later structured programming guru, said: "Any program can be done with one instruction less."

```
set e to point to the top of the conversion table
while template is not exhausted:
    get next digit z from template
    get next digit d from string
    while d > z:
        increase v by e->value
        de rease d by e->repr
    shift e to next table entry
    while d < z:
        decrease v by e->value
        increase d by e->repr
    shift e to next table entry
```

Figure 4. Code for obtaining the value of a string.

3 Converting *from* a string

With minimal modification the algorithm can be run in reverse, obtaining the numerical value of a string. The code to do this for one digit is shown in Figure 4. Note that the digit z plays the same role as the 0 in Figure 1.

This algorithm is even less robust than the previous one; converting a string that does not exactly match the template gives utter nonsense.

4 The optimal value for the downstep

Intuitively it seems a good idea to set the downstep to the the square root of the base, but showing it really is takes some work. Basically we want to express the number of actions required for converting a value n as a function $c_1(n, N, x)$ of the downstep x, where N is the base, and then sum this function over all values of n between 0 to $N - 1$. This gives the cost $C(N, x)$ of converting one "decade" of size N, corresponding to one digit. So

$$C(N, x) = \Sigma_{n=0}^{N-1} c_1(n, N, x).$$

The function $c_1(n, N, x)$ as a function of n is a slowly rising saw tooth, each next tooth representing one more downstep. This is handled most easily by splitting n into $ix + j$, with $0 \le i \le (N/x)$ and $0 \le j \le (x - 1)$. This allows us to replace $\Sigma_{n=0}^{N-1} c_1(n, N, x)$ by $\Sigma_{i=0}^{(N/x)-1} \Sigma_{j=0}^{x-1} c_2(i, j, N, x)$. Note that this is an approximation when N is not a multiple of x; we will return to this further on.

We now turn to $c_2(i, j, N, x)$. If n contains i times x, there will be $i + 1$ downstep actions. The result is $ix + j - (i + 1)x = j - x$, and $x - j$ correction steps will follow; so

$$c_2(i, j, N, x) = i + 1 + x - j$$

Note that even for $n = 0$ a downstep action will normally follow, since $n = 0$ only means that the present digit is zero, not that v is zero. Only when $v = 0$ there will be no downstep, but we ignore that for our present computation.

We now get

$$\Sigma_{j=0}^{x-1} c_2(i, j, N, x) \;=\; \Sigma_{j=0}^{x-1}(i + 1 + x - j) \;=\; \Sigma_{j=0}^{x-1}(i + 1 + x) - \Sigma_{j=0}^{x-1} j \;=$$

$$x(i+1+x) - \frac{1}{2}x(x-1) = xi + \frac{1}{2}x(x+3)$$

written this way because we next want to sum over i:

$$C(N,x) \;=\; \Sigma_{i=0}^{(N/x)-1}\big(xi + \frac{1}{2}x(x+3)\big) \;= x\frac{1}{2}\frac{N}{x}(\frac{N}{x}-1) + \frac{N}{x}\frac{1}{2}x(x+3) \;=$$

$$\frac{N}{2x}(N-x) + \frac{N}{2x}x(x+3) \;= \frac{N}{2x}(x^2+2x+N) \;= \frac{N}{2}(x+2+\frac{N}{x}) \;=$$

$$C(N,x) \;=\; \frac{N}{2}(x+\frac{N}{x}) + N$$

It is well known that this formula has a minimum for $x = \sqrt{N}$ [2], confirming our intuition.

Both in replacing $\Sigma_{i=0}^{(N/x)-1} i$ by $\frac{1}{2}\frac{N}{x}(\frac{N}{x}-1)$ and in determining the minimum of $(x+\frac{N}{x})$ by differentiation we have stepped outside the integers and treated x as a real number, which is then rounded or truncated. This is acceptable since the minimum is quite shallow. Using 4 as a downstep requires 4.5 actions/digit (4.3 for downstep 3), and for $N = 100$ we find:

x	actual cost	$C(100,x)$
7	1170	1164.29
8	1134	1125.00
9	1100	1105.56
10	1100	1100.00 (= 11.00 actions / digit)
11	1110	1104.55
12	1134	1116.67

so the position of the exact minimum is not very important. Also, doing conversions with bases much larger than 10 is inefficient: by decomposing the conversion base 100 into two conversions of base 10 each, by using the table

```
{30, 30},
{10, 10},
{ 3,  3},
{ 1,  1},
```

we can do a conversion for 8.6 actions per digit.

[2] $\frac{d}{dx}(x+\frac{N}{x}) = 1 - \frac{N}{x^2}$; set $\frac{d}{dx}$ to zero, and obtain $1 = \frac{N}{x^2} \;\rightarrow\; x = \sqrt{N}$