

◆ 1001 ÷ 11、余りはいくつ？（計算機での話） ◆

「 $(-7) \div 3$ 」の余りについては、数学の自然な立場から見れば

$$(-7) \div 3 = -3、余り 2$$

であるとの結論を得ました。そのついでに計算機、いわゆるコンピュータの場合について話しましょう。

コンピュータでは内部の計算を2進数するのが普通です。私たちは0~9までの10種類の数字を使って様々な数を表しています。これが10進数です。ですから2進数は2種類の数字、すなわち0と1だけを使って様々な数を表すものです。使う数字が少ないので10進数に比べると桁上がりが頻繁で、人間にとっては扱いづらいかもしれません。しかしコンピュータにとっては、たとえば電流が「流れた/流れない」の2値で判断するほうが確実に正確です。もちろん2進数であっても10進数であっても、計算の規則自体はまったく同じです。

それなのになぜ同じ話題を取り上げたのかって？ それはコンピュータには特殊な事情があるからです。特殊な事情のひとつに、誤差の問題がありますが、ここでは負の数に関する話題に絞ります*1。

まず話を簡単にするため、この章に登場するコンピュータは4ビットで演算を行うものとしします。つまり、4桁の2進数0000~1111だけが扱えるということです。このとき「 $111 \div 11$ 」は「 $7 \div 3$ 」を意味しますから、計算結果は「 $111 \div 11 = 10、余り 1$ 」となるでしょう。

ところでコンピュータが4ビットで演算をする場合は、0~15までの数しか扱えないことになります。ただこのことはコンピュータにとって致命的な欠陥にはなりません。というのは、たとえひとつの変数が4ビット分しか使えなくても、いくつかの変数を組み合わせれば8ビットにでも16ビットにでもできるからです。これは10進数が変数 a や b に0~9までしか使えなくても、 $10a + b$ のように組み合わせれば桁数を増やせることと似ています。

コンピュータが困るのは大きい桁数の扱いではなく、負の数の扱いなのです。

そのためにコンピュータ内部では最上位の桁を符号のために使うことがあります。つまり最上位が0なら正の数、1なら負の数という仕組みです。この約束により正の数は

2進数	0000	0001	0010	0011	0100	0101	0110	0111
10進数	0	1	2	3	4	5	6	7

の対応となり、負の数は

*1 $\frac{1}{5}$ は10進数では有限小数0.2ですが、2進数では0.00110011...と無限循環小数になります。

2進数	1000	1001	1010	1011	1100	1101	1110	1111
10進数	-8	-7	-6	-5	-4	-3	-2	-1

になるのです。負の数は慣れないとわかりづらいでしょうが、1000 から 1111 までは 1 ずつ増えていますし、「1111 + 1 = 0000」は「-1 + 1 = 0」を意味するわけですからこれでよいのです。

実はこのことがちょっとした混乱をもたらします。2進数では 0111 + 1 = 1000 ですから 0111 の次の数は 1000 ということになります。これを 10進数で言えば 7 の次の数は -8 となってしまうのです。ただ、このことは大きな問題にはなりません。なぜなら 4 ビットの 2進数の世界では、数は -8 ~ 7 までしかないのです。だからその世界の最大数の次が、その世界の最小数になるという循環を含んでもかまわないでしょう。

混乱は四則演算をするときに発生します。私たちが普通に 2進数を考えたとき、111 ÷ 11 は 10進数の 7 ÷ 3 にあたります。しかし最上位の 1 を負の符号で扱ってあげれば、111 ÷ 11 は 10進数の (-1) ÷ (-1) になってしまうかもしれません。もっとも 4 ビットのコンピュータであることが前提なら、これは 0111 ÷ 0011 ですから余計な心配は無用です。困るのは次のようなときです。

1111 ÷ 0011 が符号を持たない 2進数の計算であれば 15 ÷ 3 ですが、最上位の桁 1 が負の数を表す 1 であるとすれば (-1) ÷ 3 です。コンピュータにとって、最上位の 1 は、死活問題と呼んでいい程の重大事なのです。

そのためにコンピュータは、今扱っている数が「符号付き数」なのか「符号なし数」なのかを、常に把握しながら計算をしているのです。それだと、計算規則が複雑になりそうに思えますが、コンピュータはこれを楽に回避しています。

コンピュータは各ビットの 0 と 1 を反転させることはお手のものです。つまり 1011 なら各ビットを反転させて 0100 とします。そして反転させた数に 1 を加えると、「符号交換」された数ができあがるのです。いくつかの具体例をあげましょう。[] 内の数は 10進数を表しています。

1011	[-5]	→	0100 + 1 = 0101	[5]
0011	[3]	→	1100 + 1 = 1101	[-3]
0001	[1]	→	1110 + 1 = 1111	[-1]
1111	[-1]	→	0000 + 1 = 0001	[1]
0000	[0]	→	1111 + 1 = 0000	[0]

するとコンピュータ内部で負の数の計算をするとき、一度各ビットを反転させて計算し、再び反転させるというやり方が成立します。「1010 ÷ 0011」の計算を例に説明しましょう。これは「(-6) ÷ 3」の計算です。

$$1010 \div 0011 = (0101 + 1) \div 0011 \quad (1)$$

$$\begin{aligned}
 &= 0110 \div 0011 \\
 &= 0010 \\
 &= (1101 + 1) \\
 &= 1110
 \end{aligned} \tag{2}$$

(1) と (2) の () 内がビット反転による符号交換をしているところです。きっちり「 $(-6) \div 3 = -2$ 」となりました。式にすると長ったらしい気もしますが、コンピュータ内部では「(反転) \rightarrow (割り算) \rightarrow (反転)」の手順を機械的に踏んでいるだけです。

ところがこの機械的な作業が次のような割り算に影響します。「 $1001 \div 0011$ 」を例にとりましょう。これは「 $(-7) \div 3$ 」の計算です。

$$1001 \div 0011 = (0110 + 1) \div 0011 \tag{3}$$

$$= 0111 \div 0011$$

$$= 0010、余り 0001 \tag{4}$$

$$= (1101 + 1)、余り (1110 + 1) \tag{5}$$

$$= 1110、余り 1111 \tag{6}$$

() 内が符号交換になっています。(4) で答がでたものの、これを符号交換させなくては正しい答になりません。そこで(5)の符号交換を経て(6)になるわけですが、(6)は10進数では「 $(-7) \div 3 = -2、余り -1$ 」ということです。コンピュータにとって楽な計算をすることが、結果的に余りに負の数を出すことになっていたのです。このような仕組みのために、数学の自然な感覚と違う結果—しかし日常の感覚には近い—になるのは皮肉なものです*2。

*2 ソフトウェアの仕様によっては「 $(-7) \div 3 = -3、余り 2$ 」となっています。