A-005

整数符号化の復号速度に関する研究
Study on decoding speed of integer coding

喜田 拓也†
Takuya Kida

1. はじめに

データ圧縮の技法のひとつに、整数の符号化[1]がある。整数の符号化とは、整数の並びを一意復号可能な符号化を用いてビット列に変換することである。整数の範囲が決まっている場合、単純にはその範囲を表現可能なビット数で固定長符号化すればよい。たとえば、0 から 255 までの整数を符号化するには、値を 2 進法で表現して 8 ビットの固定長符号化を行えばよい。実際、主要なプログラミング言語で整数を取り扱う際には、CPU のレジスタ長に合わせて 32 ビットや 64 ビットの固定長符号化が用いられる。

整数の並びに偏りがある場合、可変長符号化のほうが望ましい。出現頻度の高い値により短い符号語を割り当てることで、平均符号長をより小さく抑えることができる。ハフマン符号や算術符号などに代表されるエントロピー符号化[2]を整数の符号化に用いることも可能ではある。しかしこれらはアルファベットサイズが比較的小さい文字列データに適した符号化手法であり、範囲の広い（アルファベットサイズが大きい）整数の符号化には向いていない。

整数の並びに対する可変長符号化においては、通常、小さい値ほど出現頻度が高いものと仮定する。なぜならば、整数の符号化が行われるのは、前段で差分符号や MTF 符号（Move-to-Front 符号[1]）がしばしば行われるからである。それ以外にも、様々な現象（単語の出現頻度など）に Zipf の法則[2]が当てはまることが知られている。

小さい値により短い符号語を割り当てる整数の符号化としては、ゴロム符号や一連のイライアス符号が有名である[1]。それらの他に、Fraenkel と Klein らが提案したフィボナッチ符号[3]や可変バイト符号[4]、またそれらの変種がいくつか提案されている。これらの可変長符号は、符号系列から一意に符号語を切り出せるようにするため、すべて語頭符号になっている。語頭符号とは、その符号の任意の符号語は他の符号語の語頭（接頭辞）になっていないという条件（語頭条件）を満たす符号である。

本稿では、上述した既存の整数の可変長符号化方法について、実験的な比較評価を行う。特に、符号化後の圧縮データの利活用という観点から、その復号速度について重点的に議論する。

2. 準備

本節では、ガンマ符号とデルタ符号、フィボナッチ符号、可変バイト符号について概説する。ガンマ符号とデルタ符号はともに Peter Elias によって提案された整数の符号化である。イライアス符号と呼ばれる一連の符号化方法としては、この２つのほかにオメガ符号もあるが、今回は割愛する。

以降では、符号化の対象とする値 X は自然数（$X \geq 1$）と仮定する。0 を符号化したい場合には、値を+1 ずらして符号化すればよい。また、負の値を含めた範囲を符号化したい場合は、符号語の境界に 1 ビット追加すればよい。

2.1 ガンマ符号

まずガンマ符号の符号化処理について説明する。対象の値 n の 2 進法表現を X とする。まず X の桁数を求め、これを K とする。語頭に K 個の 0 を出力した後、X を出力した結果がガンマ符号となる。たとえば、6 を符号化する場合、X は 110 で、$K = 3$ である。よって、$K - 1 = 3 - 1 = 2$ 個の 0 を出力した後、110 を出力するので、6 のガンマ符号は 00 110 となる（ここでは桁数表現に用いるビットの区切りを見やすくするために、符号語の途中にスペースを挿入している。以降も同様に適宜スペースを挿入する）。

桁数 K の長さは $\lfloor \lg n \rfloor + 1$ である。よって、$\lfloor \lg n \rfloor$ 個分の 0 を出力した後に $\lfloor \lg n \rfloor + 1$ ビットを出力するため、ガンマ符号の符号長は $2\lfloor \lg n \rfloor + 1$ ビットである。

復号処理は次のようになる。まずビット列の先頭から 1 が来るまで 0 の個数を数え、その個数を K とする。最初の 1 を含めた $K + 1$ ビット分の系列を元の値 n の 2 進法表現とする。

2.2 デルタ符号

デルタ符号の符号化処理について説明する。対象の値n の 2 進法表現を X とする。まず X の桁数 K を求め、これをガンマ符号で出力する。その後、X の最上位ビットの 1 を取り除いた長さ $K - 1$ のビット列を出力した結果がデルタ符号となる。たとえば、9 を符号化する場合、X は 1001 で、$K = 4$ である。よって、4 のガンマ符号(00 100)を出力し、X の最上位ビットを取り除いた 001 を出力する。したがって、9 のデルタ符号は 00 100 001 となる。

桁数 K はガンマ符号で表現されるため、そのビット長は $2\lfloor \lg(\lfloor \lg n \rfloor + 1) \rfloor + 1$ である。その後ろに $\lfloor \lg n \rfloor$ ビットを出力するため、ガンマ符号の符号長は $\lfloor \lg n \rfloor + 2\lfloor \lg(\lfloor \lg n \rfloor + 1) \rfloor + 1$ ビットである。

復号処理は次のようになる。まず、ビット列の先頭からガンマ符号で符号化された桁数 K を読み取る。続く $K - 1$ 個分のビット列の手前に 1 を追加した系列を元の値 n の 2 進法表現とする。

2.3 フィボナッチ符号

フィボナッチ符号は、次で定義されるフィボナッチ数列を利用した語頭符号である。

$$F_0 = 1, F_1 = 2,$$
$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2 \text{ のとき})$$

任意の正の整数は、連続しないフィボナッチ数の和で表現出来るという定理（Zeckendorf の定理）を利用した符号化手法である。たとえば、17 は $F_5 + F_2 + F_0 = 13 + 3 + 1$ と分解できる。もちろん、F_5 の代わりに $F_4 + F_3$ とも分解できるので、分解の仕方は一意ではない。しかし、連続するフィボナッチ数の和は一つ後ろのフィボナッチ数に一致する。よって、次に述べる符号化処理によって Zeckendorf の定理を満たす分解（Zeckendorf 表現と呼ばれる）が得られる。

まず対象の値を n とする。n 以下の中で最も大きいフィボナッチ数 F_m を選択する。そして、n から F_m を引いて出来た数を新たな n とする。この処理を、n が 0 になるまで繰り返す。選択されたフィボナッチ数を並べて出来上がる Zeckendorf 表現を $m+1$ ビットのビット列で表し、末尾に 1 を付け足した結果がフィボナッチ符号となる。たとえば、先の例でいうと、$n=17$ に対する Zeckendorf 表現は $F_5+F_2+F_0$ であるので、100101 に 1 を追加した 100101 1 が 17 のフィボナッチ符号となる。

フィボナッチ符号の符号長は、$\phi=1+\sqrt{5}/2$ （黄金比）とすると、フィボナッチ数の一般項の形から $\lfloor \log_\phi(\sqrt{5}n)+1 \rfloor$ であることが導かれる[3]。

復号処理は次のようになる。まず、ビット列の先頭から 1 が 2 個連続するところまでを読み込み、末尾の 1 を除いて符号語を切り出す。切り出したビット列中にある 1 のある i 番目の桁に対応したフィボナッチ数 F_i の総和が元の数 n である。

2.4 可変バイト符号

可変バイト符号は、バイト単位で可変長にすることで、復号処理を高速化することを目的としている。符号語の境界が 8 ビット＝1 バイト単位なので、他の可変長ビットの符号化と比べて高速に符号語を切り出すことができる。

対象の値 n の 2 進法表現を X とする。符号化のアイデアは、下位から順に 7 ビット毎のチャンク $x_1, x_2, \cdots, x_k$ に分割し、各チャンクを 1 バイトの下位ビットに配置することである。各チャンクの最上位ビットは、$x_1, x_2, \cdots, x_{k-1}$ については 0 とし、x_k については 1 とする。このように符号語の境界を明示することで語頭条件を満たしている。

可変バイトの符号長は、値 n の桁数が $\lfloor \lg n \rfloor + 1$ であることから、$8\lceil (\lfloor \lg n \rfloor + 1)/7 \rceil$ ビットとなる。

復号処理は次のようになる。まず、$y \leftarrow 0$ とする。次にビット列の先頭からバイト単位でチャンク $b_1, b_2, \cdots, b_k$ を切り出し、その値が $b_k \geq 128$ となるところまで読み込む。このとき、$b_i < 128$ を読むたびに $y \leftarrow (y+b_i)\cdot 2^7$ とする。最後に $b_k \geq 128$ を読んだら $n = y + b_k - 128$ とすることで復号できる。

3. 比較実験

上述した、ガンマ符号、デルタ符号、フィボナッチ符号、可変バイト符号を C 言語で実装し、圧縮率、符号化速度、復号速度を計測し、比較する。

入力に用いるデータは、Zipf 分布（パラメータ値 1.1）に従って $1 \sim 2^{32}-1$ の範囲の自然数をランダムに 100 万個生成したファイルを用いる。このデータ 1 行に 1 つの整数が記述されたテキストファイルとして保存した。これはある意味、素朴な可変長符号化と見ることができる。対象のテキストファイルのサイズは 4,528,934 バイトであり、整数 1 つ当たりのビット数（bpi）は約 36.23 ビットである。32 ビットの 2 進法による固定長符号化した場合と比べると 4 ビット以上大きい。全体のデータ量で言うと、$4{,}528{,}934\ /4{,}000{,}000 - 1 \fallingdotseq 13\%$ の増加である。

実験環境は、AMD Ryzen 5 3600 6-Core Processor 3.59 GHz, 16.0GB RAM, Ubuntu 20.04.4 LTS on WSL(Windows 10) である。使用した C コンパイラは gcc version 9.4.0 で、最適化オプション O3 を付けてコンパイルしている。時間計測には Ubuntu 上のビルトイン time コマンドを利用し、符号化時間、復号時間ともに、５回の経過時間（real）の平均値を求めた。

表１　各符号のパフォーマンス測定結果

	圧縮率（bpi）	符号化時間（秒）	復号時間（秒）
ガンマ符号	19.92	0.068	0.124
デルタ符号	**15.34**	0.071	0.102
可変バイト符号	15.89	**0.065**	**0.086**
フィボナッチ符号	15.52	0.167	0.163

実験の結果は表１のとおりである。

4. 考察

符号化時間は、処理が軽い順に順当な結果であることが見て取れる。一方、復号時間に関しては、ガンマ符号とデルタ符号で逆転している。これは圧縮率の差によって、復号時に入力となる圧縮ファイルのサイズが影響を及ぼしていると考えられる。すなわち、ガンマ符号のほうが復号処理は軽いが、処理すべき入力ファイルサイズが大きい分、時間がかかっているものと推測される。

この観点から言うと、可変長バイト符号が、Zipf 分布の元では想定していたよりも良好な圧縮率を得ており、復号処理の軽さがそのまま復号時間の短さにつながっていると考えられる。フィボナッチ符号は、圧縮率の面ではデルタ符号に次いで優秀であるが、復号処理が重く、時間がかかっていることが分かる。これは、符号語のすべてのビット毎にフィボナッチ数の和を計算しなければならないことが原因であると思われる。

5. おわりに

今回、整数の符号化について、既存の手法４つ（ガンマ符号、デルタ符号、可変バイト符号、フィボナッチ符号）の性能比較実験を行った。可変バイト符号の圧縮率が、ビット単位で可変な符号であるデルタ符号やフィボナッチ符号と遜色ないものであること、またその復号処理が他の符号よりも高速であることが分かった。

可変バイト符号にはより高速な変種[5]も提案されており、そうした新しい符号についても網羅的にパフォーマンスの実証を行うことが今後の課題として挙げられる。

謝辞

本研究は JSPS 科研費 JP21K11758, JP 20H00595 の助成を受けたものです。

参考文献

[1] 定兼邦彦, “簡潔データ構造”, アルゴリズム・サイエンス シリーズ 8 数理技法編, 共立出版, 2018 年 2 月.
[2] Ian H. Witten, Alistair Moffat, and Timothy C. Bell, “Managing Gigabytes”, second edition, Morgan Kaufmann Publishers, 1999.
[3] Aviezri S. Fraenkel and Shmuel T. Klein, “Robust universal complete codes for transmission and compression”, Discrete Applied Mathematics, Volume 64, Issue 1, pp. 31-55, 1996.
[4] Gonzalo Navarro, “Compact Data Structures: A Practical Approach”, Cambridge University Press, 2016. （邦訳：”コンパクトデータ構造　実践的アプローチ”, 定兼邦彦 訳, 2023 年 7 月）
[5] Daniel Lemire, Nathan Kurz, and Christoph Rupp, “Stream VByte: Faster byte-oriented integer compression”, Information Processing Letters, Volume 130, pp. 1-6, 2018.