タイトル	Zipf 分布における整数符号化の効率について
著者	喜田，拓也；Kida，Takuya
引用	工学研究：北海学園大学大学院工学研究科紀要(25)：33-38
発行日	2025-09-30

研究論文

Zipf 分布における整数符号化の効率について

喜 田 拓 也*

On the Efficiency of Integer Coding under Zipf Distribution

Takuya Kida*

要 旨

既存の代表的な整数符号化について，小さい数ほど出現確率の高い分布における符号の効率について議論する．ここでは，Zipf 分布の下で，ガンマ符号，デルタ符号，フィボナッチ符号，VByte 符号の 4 つの符号の性能を比較する．Zipf 分布に基づく人工データを用いて実験を行った結果，圧縮率ではデルタ符号が，符号化・復号処理の速度では VByte 符号が優れていることが分かった．

1 はじめに

本論文では，整数列に対するデータ圧縮について議論する．特に圧縮後のデータ利活用という観点から，整数の符号化について既存手法を調査し，各手法の特徴をまとめ，圧縮データ利活用に適した整数符号化手法を模索することを目的とする．そのため，実際的な確率分布である Zipf 分布の下での性能比較を行う．今回対象とする符号化手法は，ガンマ符号［1］，デルタ符号［1］，フィボナッチ符号［2］，VByte 符号［3］の 4 つである．これらの符号について，理論的な解析を行うとともに，人工データによる実験を行う．

1.1 整数符号化

データ圧縮の技法のひとつに，整数符号化［4］がある．整数符号化とは，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．実用的には瞬時復号可能な符号化であることが望ましい．瞬時復号可能であれば，符号化後のビット列を前から順に切り出しながら，元の整数列を復元することができる．

取り扱う整数の範囲が決まっている場合，その範囲を表現可能なビット数で固定長符号化するのが簡単である．たとえば，0 から 255 までの整数を符号化するには，その値の 2 進法表現を 8 ビットで固定長符号化すればよい．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ている．

一方，可変長符号化を用いる場合は，0 に近い値ほど出現頻度が高いものとして符号を設計する．なぜなら，整数を符号化する際は，その前段階で差分符号化や Move-to-Front 符号等がしばしば行われるからである［4］．それらの前処理によって，元の整数列は，0 に近い値が多く並んだ整数列へと変換される．また，様々な現象に対して Zipf の法則が当てはまることも理由の一つである［5］．Zipf の法則とは，出現頻度が k 番目の要素の頻度が，最頻する要素の頻度の $\frac{1}{k}$ に比例するという経験則である．Zipf の法則が当てはまるデータに対して差分符号化等を行うと，整数の分布は値の小さいものほど頻度が高いべき乗則の分布となる．

以上のことから，可変長の整数符号化には，通常，小さい値ほど短いビット列が割り当たるような語頭符号が用いられる．語頭符号とは，任意の符号語は他の符号語の語頭（接頭辞）ではないという条件を満たす符号である．語頭符号であれば

* 北海学園大学大学院工学研究科電子情報生命工学専攻
Graduate School of Engineering (Electronics, Information, and Life Science Eng.), Hokkai-Gakuen University

瞬時符号であることは良く知られた事実である．

小さい値により短い符号語を割り当てる整数の可変長符号化としては，ゴロム符号 [6] 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 [2] や VByte 符号 [3]，またそれらの変種 [7, 8, 9] がいくつか提案されている．これらの可変長符号化は，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．

整数符号化は，主にアクセスログや転置インデックス，各種メディアデータの圧縮に用いられている [4, 5]．よって，整数符号化された整数列に対して，集計処理や比較演算による部分抽出，系列マッチングなどが高速にできると利便性が高まる．

圧縮データの利活用のためには，圧縮率の良さはもちろん重要であるが，符号化・復号の処理速度がより重要な事項となる．

1.2 関連研究

値の出現頻度に偏りがある場合，出現頻度の高い値に短い符号語を割り当てることで平均符号長を小さく抑えることができる．この考えに基づいたハフマン符号や算術符号といったエントロピー符号化を，整数の符号化に用いることも可能である [10]．しかし，これらはアルファベットサイズ（要素の種類数）が比較的小さいデータに適した符号化手法であり，範囲の広い整数の符号化には向いていない．

転置インデックスで現れるような，整列済みの整数列，すなわち広義単調増加する非負整数の列に対しては，Elias-Fano 符号 [11, 12] を用いることができる．この符号は，整数の 2 進法表現を上位ビットと下位ビットに分割し，上位ビットの系列を前後で差分を取った系列に変換してそれを 1 進数表現で符号化する．

2 可変長符号化の既存手法

本節では，整数列に対する既存の可変長符号化について概観する．可変長符号化は，ビット単位で可変なビット可変長符号化と，バイト単位で可変なバイト可変長符号化に大別できる．

以降では，符号化の対象とする値 n は自然数（$n \geq 1$）であると仮定する．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 ビット追加することで実現できる．

2.1 ビット可変長符号化

ガンマ符号とデルタ符号はともに Elias によって提案されたビット可変長符号化である．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．

2.1.1. ガンマ符号

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 $K-1$ 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．

2.1.2 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 の長さは $\lfloor \lg n \rfloor + 1$ であり，これをガンマ符号で表現するため，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 $K-1=\lfloor \lg n \rfloor$ 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．

2.1.3　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）を利用して符号化を実行する．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 のようにも分解できる（$F_4+F_3+F_2+F_0=8+5+3+1$）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ことに注意すると，Zeckendorf の定理を満たす分解が得られる．Zeckendorf の定理を満たす整数の分解表現は Zeckendorf 表現と呼ばれる．

フィボナッチ符号の符号化手順は次の通り．

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．繰り返しの中で選択されたフィボナッチ数を昇順に並べて出来上がる 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 $F_0+F_2+F_5$ である．フィボナッチ数の有無を 0 と 1 で表し，$F_0F_1F_2F_3F_4F_5$ の順に並べると 101001 になる．この末尾に 1 を追加した 1010011 が 17 の符号語である．符号の構成の仕方から，符号語の末尾以外では 1 は連続しないことに注意しよう．

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［2］．

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 がある 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．

2.2　バイト可変長符号化

ビット可変長符号化では，符号化・復号処理の際にビット単位でのデータ処理を多用する．そのことが処理速度のボトルネックになっている．データ処理の大半がバイト単位で行えると，処理速度の高速化が望める．特に復号の高速化を指向した整数符号化として，Thiel と Heaps ら［3］が提案した可変バイト符号（VByte 符号）が知られている．

VByte 符号は符号語長をバイト単位に限定することで高速に処理できるように設計された語頭符号である．符号語の境界が 8 ビット = 1 バイト単位なので，ビット可変長符号化と比べて高速に符号語を切り出すことができる．

近年，VByte 符号を元に，対象となる値を 4 バイト整数に限定することで，より高速な復号を可能とする改善手法が提案されている．たとえば，VARINT-GB［7］は，4 つの値をまとめて語頭符号化することで圧縮率と復号速度を改善する手法である．また，VARINT-G8IU［8］は，2 個以上の符号語を 9 バイト固定長にまとめ，SIMD 計算を用いて復号速度を改善する手法である．Stream VByte[9]は，VARINT-GB における符号語長を記述している部分を符号から分離することで，CPU の処理分岐予測の誤りを低減し，かつ VARINT-GB の SIMD 化を実現している．文献［9］によると，復号の速度は Stream VByte，VARINT-G8IU，VARINT-GB の順で良く，圧縮率は VARINT-G8IU が Stream VByte/VARINT-GB に優ることが報告されている．

次節では，今回比較を行う VByte 符号について概説する．

2.2.1　VByte 符号

VByte 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．X を下位から順に 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．各チャンクの最上位ビットを，$x_1, \cdots, x_{k-1}$

については 0 とし，x_k については 1 とする．

たとえば，n=298 のとき X=10 0101010 である．よって，x_1=00101010，x_2=10000010 となる．

復号処理は次のとおり．

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．

VByte 符号の復号では，符号語の切り出しは高速に行える．しかし，チャンクの連結にはビット操作が必要である．

3　符号の性能比較

上述した符号のうち，ガンマ符号，デルタ符号，フィボナッチ符号，VByte 符号について比較を行う．各整数符号化において，符号語長は入力に依存して可変長である．自然数 n に対する各符号の符号語長は**図 1** のようになる．比較のために，32 ビット固定長符号と $\lg n$（$=\log_2 n$）のラインを加えている．整数列の要素が一様分布に出現するのであれば，各符号のラインの下側の面積が小さいものほど効率が良いということになる．図からは，デルタ符号や VByte 符号が有利であることが分かる．

次に，Zipf 分布のもとでの符号長について議論する．Zipf 分布の一般的な確率の式は次の通りである．

$$\mathrm{Zipf}(k;s,N)=\frac{\frac{1}{k^s}}{\sum_{n=1}^{N}\frac{1}{n^s}}$$

ここで，k は順位，s はパラメータ，N は全要素の数である．パラメータ s は元のジップの法則では s=1 である．**図 2** に示すように，s を増やすとより偏りが大きくなる．この Zipf の法則に基づいて，小さい自然数ほど頻出する確率分布（Zipf 分布）の下での平均符号長 L は次のように書ける．

$$L=\sum_{k=1}^{N}\mathrm{Zipf}(k;s,N)\times \mathrm{code\_len}(k)$$

ここで，code_len(k) は自然数 k に対する符号語長である．**図 3** は，和の中身のラインを図示したものである．すなわち，各符号の平均符号長 L は

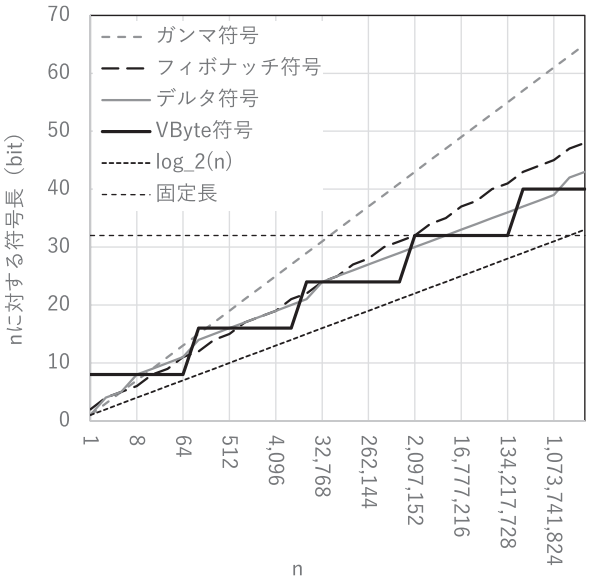

図 1　自然数 n に対する符号語長

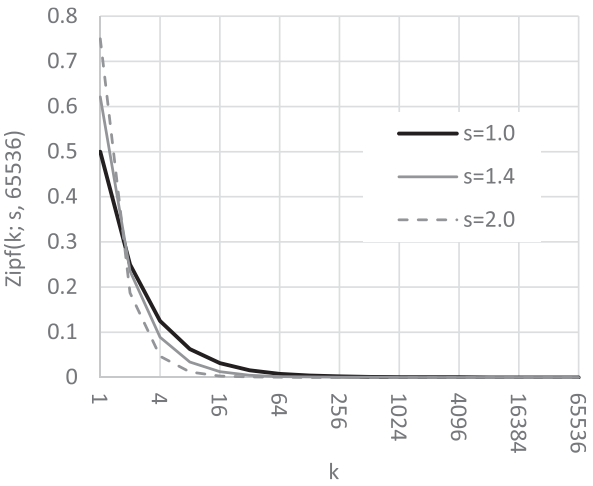

図 2　Zipf$(k;s,2^{16})$ のグラフ

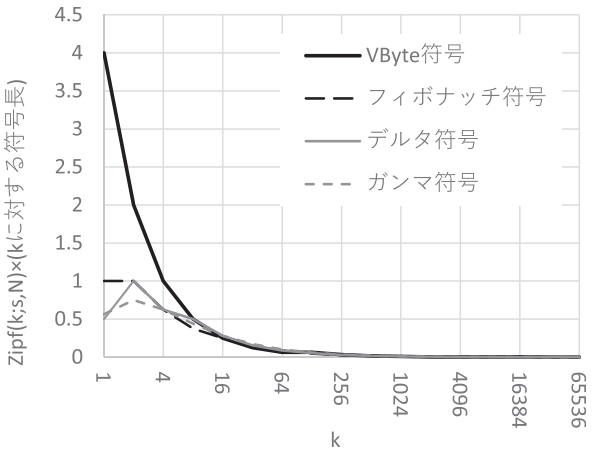

図 3　Zipf$(k;s,N)\times$(k に対する符号長)

図 3 におけるラインの下側の面積である．この図から分かるように，Zipf 分布の下では VByte 符号よりも他の符号のほうが平均符号長は有利であることが分かる．

次に，各符号を C 言語で実装し，その圧縮率，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～$2^{32}-1$ 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≒13%の増加である．

実験環境は，AMD Ryzen 5 3600 6-Core Processor 3.59 GHz，16.0 GB 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．

ガンマ符号やデルタ符号においては，32 ビットで表現された整数の最上位ビット（MSB）を取得する必要がある．この処理は，理論的には値 n に対して $O(\lg n)$ 時間を要する．しかし，ビット長が固定であると考えるならば定数時間である．実際，gcc では「__builtin_clz()」というビルトイン関数を用いることで，定数時間で高速に計算することができる．

実験の結果は**表 1** のとおりである．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，VByte 符号が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考えられる．

表 1　各符号の比較実験の結果

	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
VByte 符号	15.89	**0.065**	**0.086**
フィボナッチ符号	15.52	0.167	0.163

4　おわりに

今回，整数の符号化について，4 つの既存の手法（ガンマ符号，デルタ符号，フィボナッチ符号，VByte 符号）が Zipf 分布のもとでどのような性能を示すか調査を行った．4 バイト整数の範囲で，Zipf 分布に従う人工データを用いた実験により，圧縮率ではデルタ符号が最も優れており，符号化および復号処理は VByte 符号が最も優れていることが分かった．加えて，VByte 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も分かった．

以上のことから，実用的には，VByte 符号がバランスのよい整数符号化であることが分かる．

2.2 節で述べたように，VByte 符号には，より高速な変種がいくつか提案されており，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．また，圧縮データの利活用という観点からは，符号化された系列を元に戻さずに大小比較できるような技術の開発も今後の課題である．

謝辞

本研究は，JSPS 科研費 JP21K11758，および JP20H00595 の助成を受けたものです．

参考文献

[1] Peter Elias, “Universal codeword sets and representations of the integers,” *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[2] Aviezri S. Fraenkel and Shmuel T. Klein, “Robust universal complete codes for transmission and compression,” *Discrete Applied Mathematics*, vol. 64, no. 1, pp. 31–55, 1996.

[3] Larry H. Thiel and H. S. Heaps, “Program design for retrospective searches on large data bases,” *Information Storage and Retrieval*, vol. 8, no. 1, pp. 1–20, 1972.

[4] 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.

[5] Ian H. Witten, Alistair Moffat and Timothy C. Bel, Managing Gigabytes, second edition, Morgan Kaufmann Publishers, 1999.

[6] Solomon W. Golomb, “Run-length encodings,” *IEEE Trans. Info. Theory.*, vol. 12, no. 3, p. 399, 1966.

[7] Jeff Dean, “Challenges in building large-scale information retrieval systems: invited talk,” in *Proc. the Second ACM Int. Conf. on Web Search and Data Mining*, 2009.

[8] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst and Paramjit S. Oberoi, “SIMD-based decoding of posting lists,” in *Proc. the 20th ACM Int. Conf. on Information and Knowledge Management*, 2011.

[9] Daniel Lemire, Nathan Kurz and Christoph Rupp, “Stream VByte: Faster byte-oriented integer compression,” *Information Processing Letters*, vol. 130, pp. 1–6, 2018.

[10] Gonzalo Navarro, Compact Data Structures: A Practical Approach（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），Cambridge University Press, 2016.

[11] Peter Elias, “Efficient Storage and Retrieval by Content and Address of Static Files,” *J. ACM*, vol. 21, no. 2, p. 246–260, 1974.

[12] Robert M. Fano, “On the number of bits required to implement an associative memory,” *Computer Structures Group, MIT*, p. Memorandum 61, 1971.