A Method for Recursive Data Compression

John Archie Gillis

Dept. of R&D at Gillis Intellectual Property Developments

johnarchiegillis@gmail.com

May 10, 2018

(Pat. Pending CA 2,998,336 Mar. 19, 2018)

1 Introduction

Recursive compression of random data is generally deemed to be an impossible process that defies the laws of mathematics/physics. This paper explains why this perception is incorrect and provides a proof that explains how such a compression system may be achieved.

Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data. Lossless compression is used in cases where it is important that the original and the decompressed data be identical, or where deviations from the original data would be unfavorable. Typical examples are executable programs, text documents, and source code. Some image file formats, like PNG or GIF, use only lossless compression.

The following is one of our tricks for creating extra redundancy.

```
00100  11011  00100
 212    212    212
```

The present paper describes methods for compressing *all* forms of data (random and non-random). All of the compression methods used operate on a bit-level and will thus work for compressing any form of data regardless of the program that will execute it or that created it. Accordingly, compressed data can be re-input back into the system again in a recursive nature.

2 Description

The initial program/method that is described in this paper is used to illustrate the strategy that we will use to accomplish the impossible. If the compression and decompression programs are too large however, they will simply not fit onto a standard computer and will likely take much too long to process. The second portion of this paper describes how the program size and time issues are to be addressed so as to present a program that has great utility in a number of situations.

We begin by analyzing a data set. We will execute code to turn a selection (string) of zero's and one's into numerical digits that will represent their run lengths. 00011100101 would become 332111. We will then break each run into a group of selected permutations, which in most cases will provide shorter string sets of data that can be recursively compressed.

We want to discover:
(A) What is the longest consecutive run of either zero (0) or one (1), and
(B) What is the first digit of the selection. Is it a zero (0) or a one (1).

Knowing the length of the longest run of either the sets of 0 or 1, will in some instances assist us with determining the optimal "*sub-set*" algorithm for our program and for each recursive compression. The initial description of the method describes a very large system that assumes access to a super computer. This description is used simply to prove that recursive data compression is possible. We will remedy the requirement for super-computer systems by the end of this paper, thus optimizing our program for speed, processing resources and transmission. This initial overall program, method/algorithm, will include a sub-set number of varying smaller or divisional algorithms (similar to a merkel tree) for the overall system. As an example, when the data is compressed the first time, it may use a specific algorithm for a data set with the longest run of 5-bits, but when the output is sent back through the system, the longest run of the output data may be four 0's or 1's in a row and thus we would use an algorithm that would best fit the four run (4-bit) requirements, and so on.

The present system assists us by finding extra redundancies in *what most would call* random data streams. The data stream will be broken down into a variety of redundant sections by analysis of bit. Each string of repeated bits will be converted to a number and placed into a predetermined group (or coding schematic). When the bit stream changes from *0* to *1* or vice versa, the next string of bits will be converted to a number. For example, if our stream was *11100100 11000110*, the conversion to numbers would be 3, 2, 1, 2, 2, 3, 2, 1.

We then group these numbers into smaller selections (321, 223, 21). Generally, we would multiply the largest number (in this basic example that number is 3) by itself that many times again to get a selected number of possible permutations. It this example we would do 3x3x3=27. We would convert each group of three numbers into one of 27 possible three number permutations. The above example would be 321, 223 and 21 as a remainder. A bit run of 3 or 4 created only a few permutations all of which need to be encoded into our program. With bit runs of higher numbers such as 12, 20 or 30, we will need to modify our approach so that the program size and speed are maintained at a useful level. These approaches will be described shortly.

By incorporating our method and by utilizing strategic variable length codes (see excel sheet below) to represent each permutation, we put the probability of data compression rather than data elongation greatly into our favor, as will be seen in the diagrams below. We can then represent these numbers with fewer binary digits. The longest bit-run provides a constraint to what most believe to be an infinite system that has little or no redundancy. The longer the bit run (especially in random data), the less chance that it has of not flipping from a 0 to a 1 or vice versa, thus an infinite run is not an option.

Figure 1 will assist with the description. The scan **1** in this example is *00010101 11001110 10011100 01010110 11001100 11000110 00101010 11001100*. **2** shows where some of the longest runs of *0* or 1 are located. In this particular stream of data, the longest run of consecutives is three.

Fig. 1

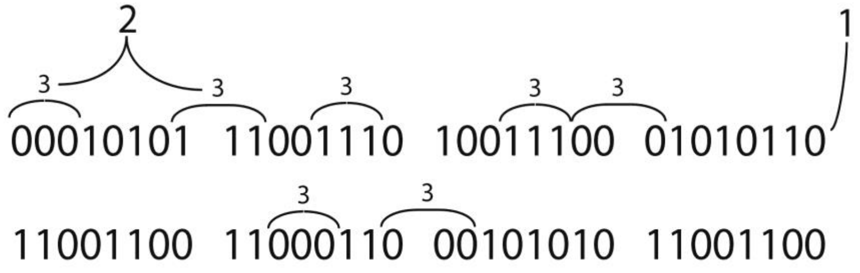


Figure 2 takes the description further by showing **3** which is a prefix that will be added to our data to show if its first number will be a zero or a one. The prefix will be encoded as the opposite of the first bit. Bit representations will be flipped for every new number. In this figure we see the binary data broken up into groups of three numbers (others groupings are possible). Our first set is slightly darker in shade and is blocked off as is shown by **4**. Our second selection of three is lighter in shade and also blocked off. We then go from dark to lighter etc. We see **5** at the end, which is a set of two left over or remainder bits.

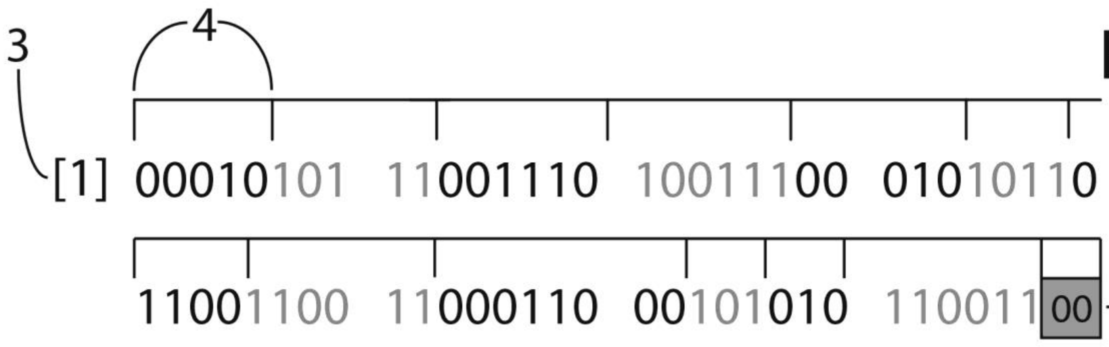


Fig. 2

<u>Figure 3</u> shows that we now convert our groupings into numbers. This example is broken down into three digit numbers. Since there is a maximum of three 0 or 1's in a row, we know that no number will be higher than three and thus in any selection of three numbers we only need to create a possible 27 permutations. By using this method, we constrain our system and it enables us to find redundancy in a seemingly non-redundant data stream. **6** provides the new three-digit number which actually represents a selected permutation rather than a numerical digit.

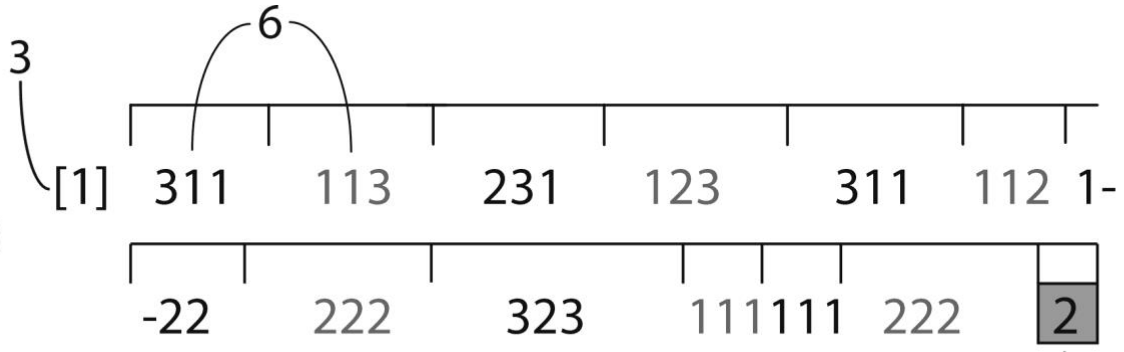
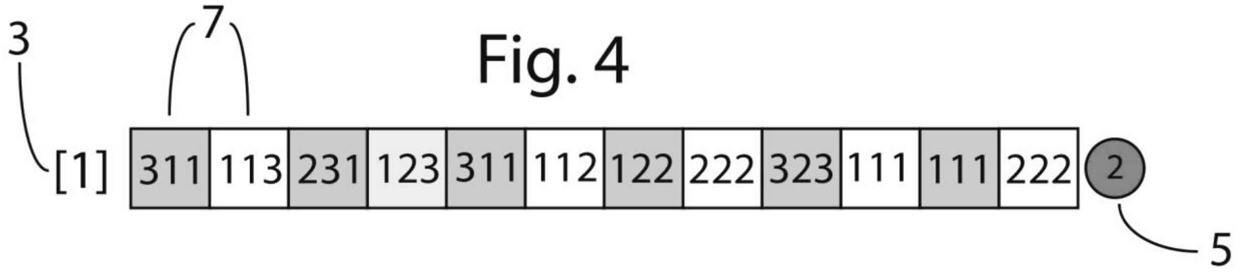


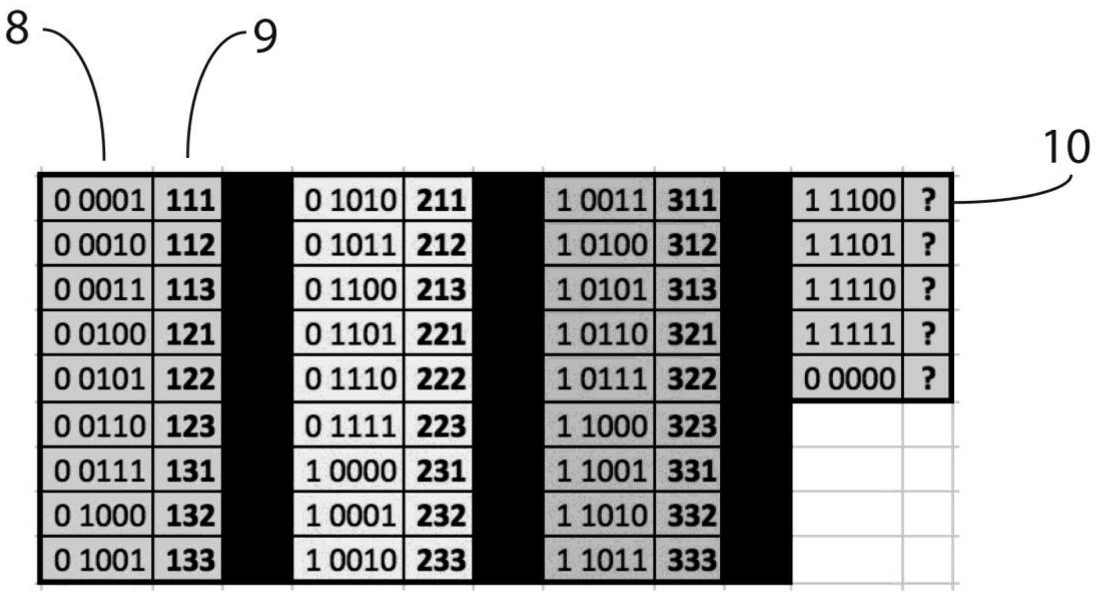
Fig. 3

<u>Figure 4</u> shows our various groupings and how we have now broken up the stream of data into redundant groups. This figure is simply to provide an easy to visualize example. Most data streams will be much longer than the example provided, which will make the remainder(s) **5** mostly insignificant, whereas in figure **4** it seems significant due to the tiny example provided. In a real world scenario, we may have thousands, millions, billions or more bits/bytes of data that will be analyzed. Our groupings will likely also be much larger, but any length and any integer for any number in any of the groups could be a possibility. In one embodiment the integers for each number in each group will not be higher than the longest run of zero's that we had scanned. As an example if the longest run of 0 or 1 was found to be 8 our numbers could be 83263547 or 17263846 or 88888888 or 12345678 or 87878721 etc. The length of the grouping will match the longest run as well, but there may be instances where we wish to use a longer or shorter grouping. As an example we could use 84756 or 27865 etc., even though it could be less effective. Longer runs are in many instances more effective. For example, 876353423453 could work as a grouping for our run of 8 in some instances as well. Any length or any integer is possible and they may also be mixed.

Fig. 4



<u>Figure 5</u> provides a breakdown of all of the 3-run permutations. 3x3x3 tells us that there are 27 possible variations. **9** (the right hand columns display every possibility. **8** shows how they could be encoded into binary form by using 5-bit groupings. 32 numbers can be stored in 5 bits and only 16 numbers in 4 bits, thus we need to use the 5-bit grouping if we wish to capture all 27 various permutations. This will leave us with 5 leftover bits, that could potentially be used for something else, not used, or to represent two or more 1-27 groupings as one of the leftover 5. The 5-bit streams are not really as efficient as our variable length codes however and thus they will be converted into a mixture of lengths as described in the excel sheet below Figure 5.



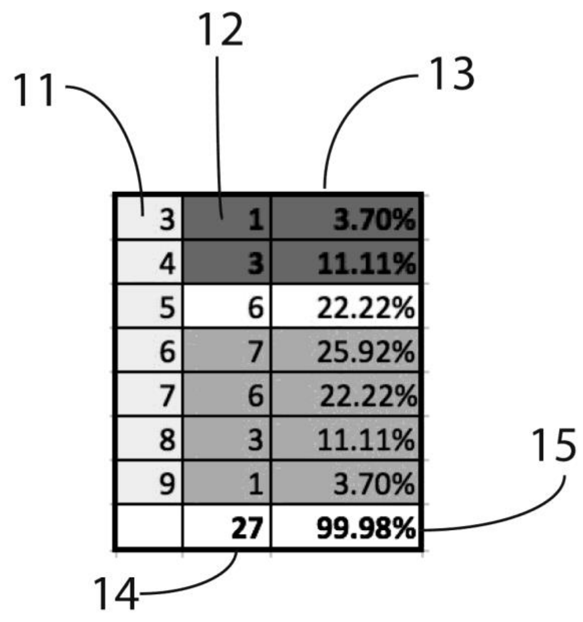
0 0001	111	0 1010	211	1 0011	311	1 1100	?
0 0010	112	0 1011	212	1 0100	312	1 1101	?
0 0011	113	0 1100	213	1 0101	313	1 1110	?
0 0100	121	0 1101	221	1 0110	321	1 1111	?
0 0101	122	0 1110	222	1 0111	322	0 0000	?
0 0110	123	0 1111	223	1 1000	323		
0 0111	131	1 0000	231	1 1001	331		
0 1000	132	1 0001	232	1 1010	332		
0 1001	133	1 0010	233	1 1011	333		

Fig. 5

1st permutation possibility is	1	1	1	&	can be converted to a variable length code as	[00]0	wherein the represented	3	bits	can be encoded with	3	bits
2nd permutation possibility is	2	1	1	&	can be converted to a variable length code as	[00]1	wherein the represented	4	bits	can be encoded with	3	bits
3rd permutation possibility is	3	1	1	&	can be converted to a variable length code as	[01]01	wherein the represented	5	bits	can be encoded with	4	bits
4th permutation possibility is	1	2	1	&	can be converted to a variable length code as	[01]00	wherein the represented	4	bits	can be encoded with	4	bits
5th permutation possibility is	2	2	1	&	can be converted to a variable length code as	[01]10	wherein the represented	5	bits	can be encoded with	4	bits
6th permutation possibility is	1	1	2	&	can be converted to a variable length code as	[01]01	wherein the represented	4	bits	can be encoded with	4	bits
7th permutation possibility is	1	3	1	&	can be converted to a variable length code as	[10]100	wherein the represented	5	bits	can be encoded with	5	bits
8th permutation possibility is	3	2	1	&	can be converted to a variable length code as	[10]011	wherein the represented	6	bits	can be encoded with	5	bits
9th permutation possibility is	2	1	2	&	can be converted to a variable length code as	[10]000	wherein the represented	5	bits	can be encoded with	5	bits
10th permutation possibility is	3	1	2	&	can be converted to a variable length code as	[10]101	wherein the represented	6	bits	can be encoded with	5	bits
11th permutation possibility is	1	2	2	&	can be converted to a variable length code as	[10]001	wherein the represented	5	bits	can be encoded with	5	bits
12th permutation possibility is	2	2	2	&	can be converted to a variable length code as	[10]110	wherein the represented	6	bits	can be encoded with	5	bits
13th permutation possibility is	1	1	3	&	can be converted to a variable length code as	[10]010	wherein the represented	5	bits	can be encoded with	5	bits
14th permutation possibility is	1	3	2	&	can be converted to a variable length code as	[10]111	wherein the represented	6	bits	can be encoded with	5	bits
15th permutation possibility is	2	3	1	&	can be converted to a variable length code as	[11]0000	wherein the represented	6	bits	can be encoded with	5	bits
16th permutation possibility is	3	3	1	&	can be converted to a variable length code as	[11]0001	wherein the represented	7	bits	can be encoded with	6	bits
17th permutation possibility is	2	3	2	&	can be converted to a variable length code as	[11]0010	wherein the represented	7	bits	can be encoded with	6	bits
18th permutation possibility is	3	3	2	&	can be converted to a variable length code as	[11]0011	wherein the represented	8	bits	can be encoded with	6	bits
19th permutation possibility is	3	2	2	&	can be converted to a variable length code as	[11]0100	wherein the represented	7	bits	can be encoded with	6	bits
20th permutation possibility is	2	1	3	&	can be converted to a variable length code as	[11]0101	wherein the represented	6	bits	can be encoded with	6	bits
21st permutation possibility is	3	1	3	&	can be converted to a variable length code as	[11]0110	wherein the represented	7	bits	can be encoded with	6	bits
22nd permutation possibility is	1	2	3	&	can be converted to a variable length code as	[11]0111	wherein the represented	6	bits	can be encoded with	6	bits
23rd permutation possibility is	2	2	3	&	can be converted to a variable length code as	[11]1000	wherein the represented	7	bits	can be encoded with	6	bits
24th permutation possibility is	3	2	3	&	can be converted to a variable length code as	[11]1001	wherein the represented	8	bits	can be encoded with	6	bits
25th permutation possibility is	1	3	3	&	can be converted to a variable length code as	[11]1010	wherein the represented	7	bits	can be encoded with	6	bits
26th permutation possibility is	2	3	3	&	can be converted to a variable length code as	[11]1011	wherein the represented	8	bits	can be encoded with	6	bits
27th permutation possibility is	3	3	3	&	can be converted to a variable length code as	[11]1100	wherein the represented	9	bits	can be encoded with	6	bits
					NOT USED	[11]1101		162			139	
					NOT USED	[11]1110						
					NOT USED	[11]1111						

Figure 6 gives us a break-down of our potential data savings by using our new method of data compression and transmission. We see that the 27 grouping or permutation groupings represent between 3-9 bits. Only four groupings are less than 5 bits, which would make our data longer. Six groupings stay the same length of five, but seventeen make our data shorter. Odds are in our favor that the data will be compressed rather than elongated. Over a long stream of data, this will almost certainly be the case, but because the smaller additions (3 and for in this example) will be more likely to occur in general, we will want to utilize our mixed length coding scheme to guarantee that this is the outcome.

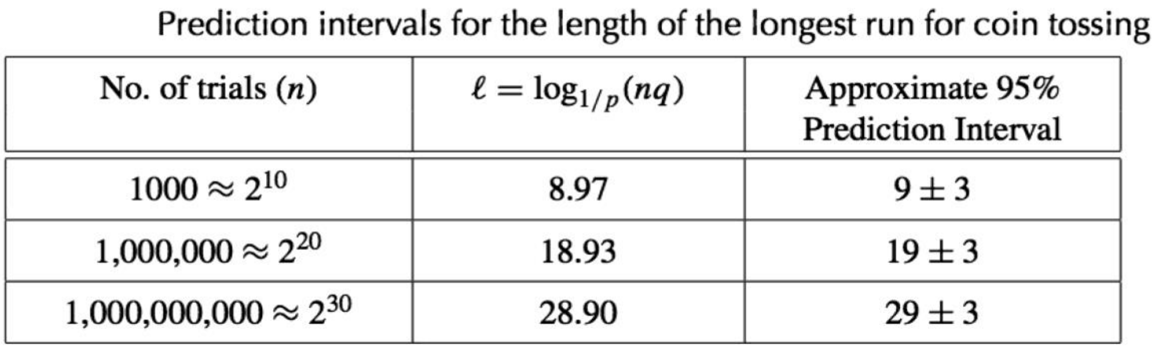
...go to next page...


11
12
13
15
14

3	1	3.70%
4	3	11.11%
5	6	22.22%
6	7	25.92%
7	6	22.22%
8	3	11.11%
9	1	3.70%
	27	**99.98%**

Fig. 6

Figure 7 provides a table that predicts intervals for the longest run for coin tossing. This is likely nearly identical for random runs of 0's and 1's. For example, 1000 bits would likely have a longest run of about 9 zero's or one's in a row or between six and twelve. We can see the it would be nineteen and then twenty- nine, (give or take three) for 1,000,000 and 1,000,000,000 bits etc. This is key to telling us that it is so unlikely that we will run into a string of say 1000 of the same bit in a row, that we likely do not even have to consider the possibility.



Prediction intervals for the length of the longest run for coin tossing

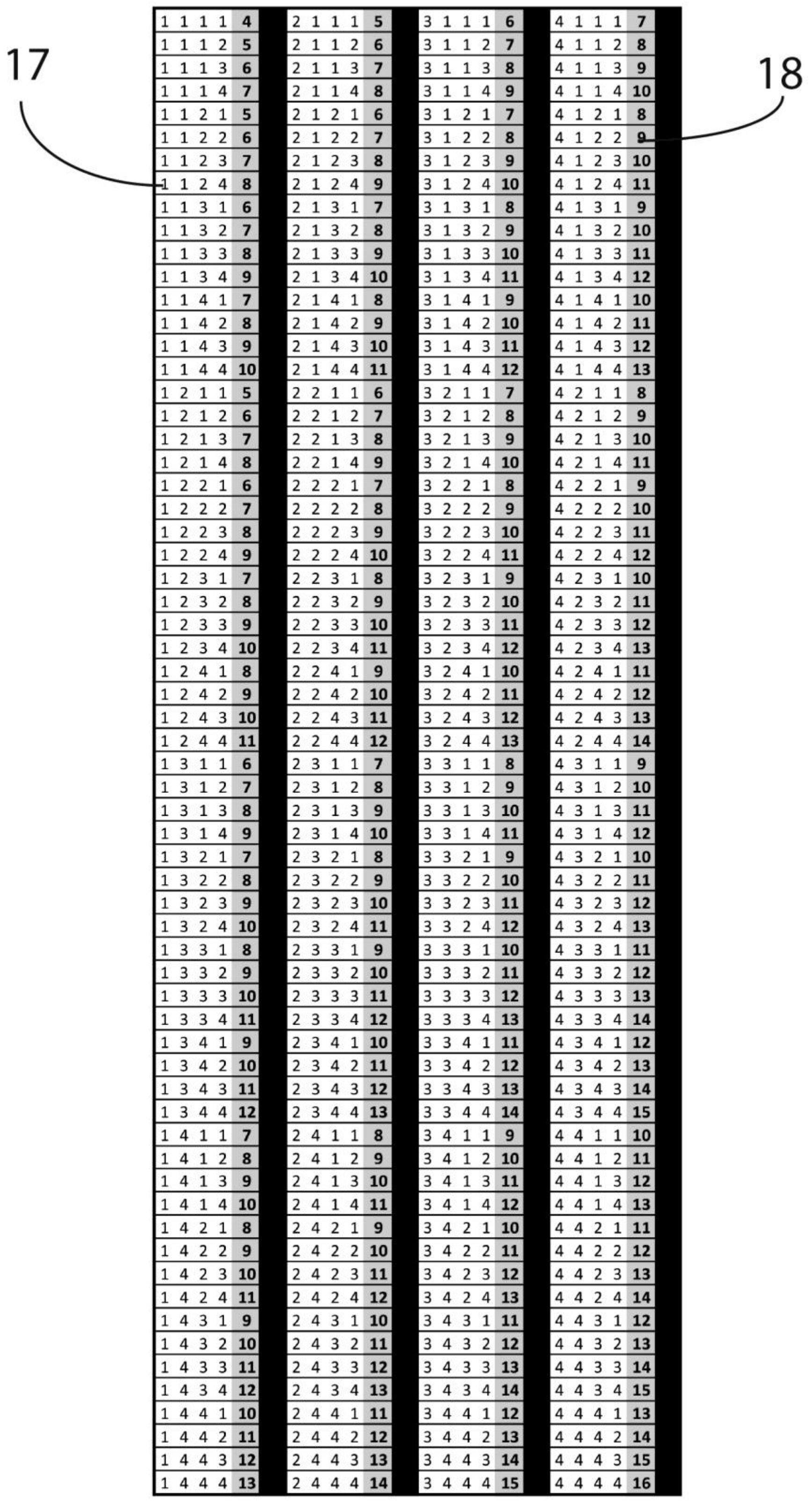
No. of trials (n)	$\ell = \log_{1/p}(nq)$	Approximate 95% Prediction Interval
$1000 \approx 2^{10}$	8.97	9 ± 3
$1{,}000{,}000 \approx 2^{20}$	18.93	19 ± 3
$1{,}000{,}000{,}000 \approx 2^{30}$	28.90	29 ± 3

Fig. 7

Figure 8 provides a great example of one scenario wherein there is no option for the process to lengthen the data stream that is to be compressed. In every scenario the data selected will either stay the same length or get shorter. This is achieved by using a variable length coding process.

We could use the famous HUFFMAN coding system for shorter codes, but in the present embodiment an alternate system invented by the present inventor is used. In this example the first two digits which are bolded with brackets **[00]** **[01]** **[10]** and **[11]** will represent how long the next set of data will be. In this case they will stand for 1, 2, 3, and 4. So after **[00]** we will have a *"one bit run"* wherein only a 0 or a 1 are possible which will give two possibilities **[00]**0 or **[00]**1. After the **[01]** we will have a "*two bit run",* which will give us four possibilities **[01]**00, **[01]**01, **[01]**10, and **[01]**11. The "three bit run" will give us eight possibilities and the "four bit run" will give us sixteen possibilities. Each variable length code will be assigned one of the selected permutations. The permutation of 111 will be matched with **[00]**0 so that when decoded it will be the same length (and no longer). We start saving bits right away as we then move on to the next smallest permutation of 112 which represents four bits, but can be encoded into **[00]**1 three bits, thus saving one bit on every occurrence of either 010 or 101. It's like a two for one deal! 16 provides a breakdown of the savings that will occur. We see that **[01]**nn has four options that would be best suited for two permutations that when the integers are added up are equal to four, and two permutations that equal 5. Adding up the integers is key to knowing how to encode the permutations for maximum compressibility.

To provide an example the permutations could be 121, 211, 211(add up to 4) or 221, 122, 212 (add up to 5). Maximum benefit is seen when we get into the higher numbered permutations, such as those that add up to 7, 8 and 9. For example a permutation like 333 adds up to 9, but can be represented by [11]0111 which is a 6-bit string. In this example the bit savings are 3.



1 1 1 1	**4**	2 1 1 1	**5**	3 1 1 1	**6**	4 1 1 1	**7**
1 1 1 2	**5**	2 1 1 2	**6**	3 1 1 2	**7**	4 1 1 2	**8**
1 1 1 3	**6**	2 1 1 3	**7**	3 1 1 3	**8**	4 1 1 3	**9**
1 1 1 4	**7**	2 1 1 4	**8**	3 1 1 4	**9**	4 1 1 4	**10**
1 1 2 1	**5**	2 1 2 1	**6**	3 1 2 1	**7**	4 1 2 1	**8**
1 1 2 2	**6**	2 1 2 2	**7**	3 1 2 2	**8**	4 1 2 2	**9**
1 1 2 3	**7**	2 1 2 3	**8**	3 1 2 3	**9**	4 1 2 3	**10**
1 1 2 4	**8**	2 1 2 4	**9**	3 1 2 4	**10**	4 1 2 4	**11**
1 1 3 1	**6**	2 1 3 1	**7**	3 1 3 1	**8**	4 1 3 1	**9**
1 1 3 2	**7**	2 1 3 2	**8**	3 1 3 2	**9**	4 1 3 2	**10**
1 1 3 3	**8**	2 1 3 3	**9**	3 1 3 3	**10**	4 1 3 3	**11**
1 1 3 4	**9**	2 1 3 4	**10**	3 1 3 4	**11**	4 1 3 4	**12**
1 1 4 1	**7**	2 1 4 1	**8**	3 1 4 1	**9**	4 1 4 1	**10**
1 1 4 2	**8**	2 1 4 2	**9**	3 1 4 2	**10**	4 1 4 2	**11**
1 1 4 3	**9**	2 1 4 3	**10**	3 1 4 3	**11**	4 1 4 3	**12**
1 1 4 4	**10**	2 1 4 4	**11**	3 1 4 4	**12**	4 1 4 4	**13**
1 2 1 1	**5**	2 2 1 1	**6**	3 2 1 1	**7**	4 2 1 1	**8**
1 2 1 2	**6**	2 2 1 2	**7**	3 2 1 2	**8**	4 2 1 2	**9**
1 2 1 3	**7**	2 2 1 3	**8**	3 2 1 3	**9**	4 2 1 3	**10**
1 2 1 4	**8**	2 2 1 4	**9**	3 2 1 4	**10**	4 2 1 4	**11**
1 2 2 1	**6**	2 2 2 1	**7**	3 2 2 1	**8**	4 2 2 1	**9**
1 2 2 2	**7**	2 2 2 2	**8**	3 2 2 2	**9**	4 2 2 2	**10**
1 2 2 3	**8**	2 2 2 3	**9**	3 2 2 3	**10**	4 2 2 3	**11**
1 2 2 4	**9**	2 2 2 4	**10**	3 2 2 4	**11**	4 2 2 4	**12**
1 2 3 1	**7**	2 2 3 1	**8**	3 2 3 1	**9**	4 2 3 1	**10**
1 2 3 2	**8**	2 2 3 2	**9**	3 2 3 2	**10**	4 2 3 2	**11**
1 2 3 3	**9**	2 2 3 3	**10**	3 2 3 3	**11**	4 2 3 3	**12**
1 2 3 4	**10**	2 2 3 4	**11**	3 2 3 4	**12**	4 2 3 4	**13**
1 2 4 1	**8**	2 2 4 1	**9**	3 2 4 1	**10**	4 2 4 1	**11**
1 2 4 2	**9**	2 2 4 2	**10**	3 2 4 2	**11**	4 2 4 2	**12**
1 2 4 3	**10**	2 2 4 3	**11**	3 2 4 3	**12**	4 2 4 3	**13**
1 2 4 4	**11**	2 2 4 4	**12**	3 2 4 4	**13**	4 2 4 4	**14**
1 3 1 1	**6**	2 3 1 1	**7**	3 3 1 1	**8**	4 3 1 1	**9**
1 3 1 2	**7**	2 3 1 2	**8**	3 3 1 2	**9**	4 3 1 2	**10**
1 3 1 3	**8**	2 3 1 3	**9**	3 3 1 3	**10**	4 3 1 3	**11**
1 3 1 4	**9**	2 3 1 4	**10**	3 3 1 4	**11**	4 3 1 4	**12**
1 3 2 1	**7**	2 3 2 1	**8**	3 3 2 1	**9**	4 3 2 1	**10**
1 3 2 2	**8**	2 3 2 2	**9**	3 3 2 2	**10**	4 3 2 2	**11**
1 3 2 3	**9**	2 3 2 3	**10**	3 3 2 3	**11**	4 3 2 3	**12**
1 3 2 4	**10**	2 3 2 4	**11**	3 3 2 4	**12**	4 3 2 4	**13**
1 3 3 1	**8**	2 3 3 1	**9**	3 3 3 1	**10**	4 3 3 1	**11**
1 3 3 2	**9**	2 3 3 2	**10**	3 3 3 2	**11**	4 3 3 2	**12**
1 3 3 3	**10**	2 3 3 3	**11**	3 3 3 3	**12**	4 3 3 3	**13**
1 3 3 4	**11**	2 3 3 4	**12**	3 3 3 4	**13**	4 3 3 4	**14**
1 3 4 1	**9**	2 3 4 1	**10**	3 3 4 1	**11**	4 3 4 1	**12**
1 3 4 2	**10**	2 3 4 2	**11**	3 3 4 2	**12**	4 3 4 2	**13**
1 3 4 3	**11**	2 3 4 3	**12**	3 3 4 3	**13**	4 3 4 3	**14**
1 3 4 4	**12**	2 3 4 4	**13**	3 3 4 4	**14**	4 3 4 4	**15**
1 4 1 1	**7**	2 4 1 1	**8**	3 4 1 1	**9**	4 4 1 1	**10**
1 4 1 2	**8**	2 4 1 2	**9**	3 4 1 2	**10**	4 4 1 2	**11**
1 4 1 3	**9**	2 4 1 3	**10**	3 4 1 3	**11**	4 4 1 3	**12**
1 4 1 4	**10**	2 4 1 4	**11**	3 4 1 4	**12**	4 4 1 4	**13**
1 4 2 1	**8**	2 4 2 1	**9**	3 4 2 1	**10**	4 4 2 1	**11**
1 4 2 2	**9**	2 4 2 2	**10**	3 4 2 2	**11**	4 4 2 2	**12**
1 4 2 3	**10**	2 4 2 3	**11**	3 4 2 3	**12**	4 4 2 3	**13**
1 4 2 4	**11**	2 4 2 4	**12**	3 4 2 4	**13**	4 4 2 4	**14**
1 4 3 1	**9**	2 4 3 1	**10**	3 4 3 1	**11**	4 4 3 1	**12**
1 4 3 2	**10**	2 4 3 2	**11**	3 4 3 2	**12**	4 4 3 2	**13**
1 4 3 3	**11**	2 4 3 3	**12**	3 4 3 3	**13**	4 4 3 3	**14**
1 4 3 4	**12**	2 4 3 4	**13**	3 4 3 4	**14**	4 4 3 4	**15**
1 4 4 1	**10**	2 4 4 1	**11**	3 4 4 1	**12**	4 4 4 1	**13**
1 4 4 2	**11**	2 4 4 2	**12**	3 4 4 2	**13**	4 4 4 2	**14**
1 4 4 3	**12**	2 4 4 3	**13**	3 4 4 3	**14**	4 4 4 3	**15**
1 4 4 4	**13**	2 4 4 4	**14**	3 4 4 4	**15**	4 4 4 4	**16**

Fig. 9

<u>Figure 9</u> takes things a step further and introduces a *"4-run permutation"*. In this scenario we have four digit groupings that each can be one of four numbers, 1, 2, 3 or 4. 4x4x4x4=256 tells us that there are 256 different groups that we can code to. So if we ran a computer program that scanned a set of data and that scan told us that the longest run of consecutive 0's or 1's happened to be four, (such as 1111 or 0000), then we would use (or the computer would use) our four-run algorithm. 11110000 11110000 could be coded to 4444 or permutation number 256 which then could be encoded into one byte of data as 11111111, which is half of its previous size. We could alternatively use one of any number of variable length encoding mechanisms to make the savings even greater. Again these are just examples to show a few scenarios out of almost limitless possibilities. We could decide to encode the 4-run data as 44444444 if we so desired and use a much larger number of permutation possibilities (in the case of 44444444 we would have 65536 possibilities). Also if our scan had shown a longer run than four we would likely include larger integers in our system as well, such as 123456 etc. The length of the permutation can be varied.

<u>Figure 10</u> provides us with a breakdown of the various permutations and what their numbers add up to. This distribution looks like a bell and is found quite commonly in numerous statistical situations and scenarios. Here we see that the middle or median number is the most common and is found 43 times in our 256 possible permutations. Some examples would include 4231, 1234, 2224, 3331, etc. By classifying our bit scan information into these limited permutation groups we are essentially finding redundancy in what would have previously been deemed random data. **19** shows what each of the 256 permutations would add up to if each integer were added to a single number. **20** shows the amount of times out of the 256 possibilities that a permutation adds up to that number. If we were to encode these permutations into bytes of data 0000 0000 or groups of 8 bits, we would see that the first four sections of the bell (4, 5, 6, 7) would get longer (8) would stay the same and 9-16 would get shorter. This of course is a probability distribution chart and so we can never know with 100% certainty what the distribution will look like, but the odds are greatly in our favor that we will be able to compress the original data stream into a much smaller code. We will then be able to analyze the output of that code and repeat the process again until the original code is even more compressed. Then we repeat that process again and so on. Permutations that add up to smaller numbers will likely occur more often than those that add up to larger numbers, but this will soon be addressed in our next section.

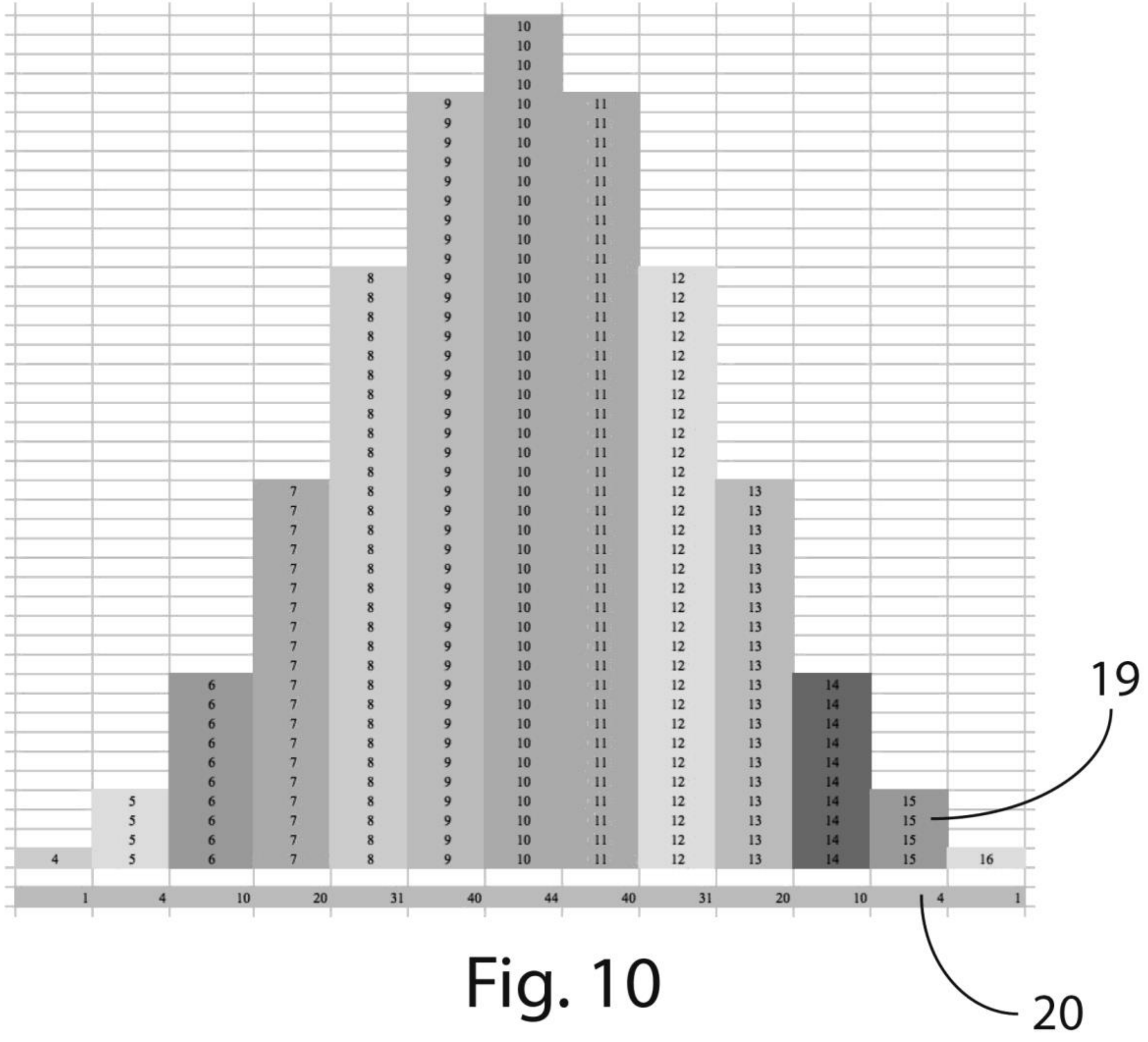
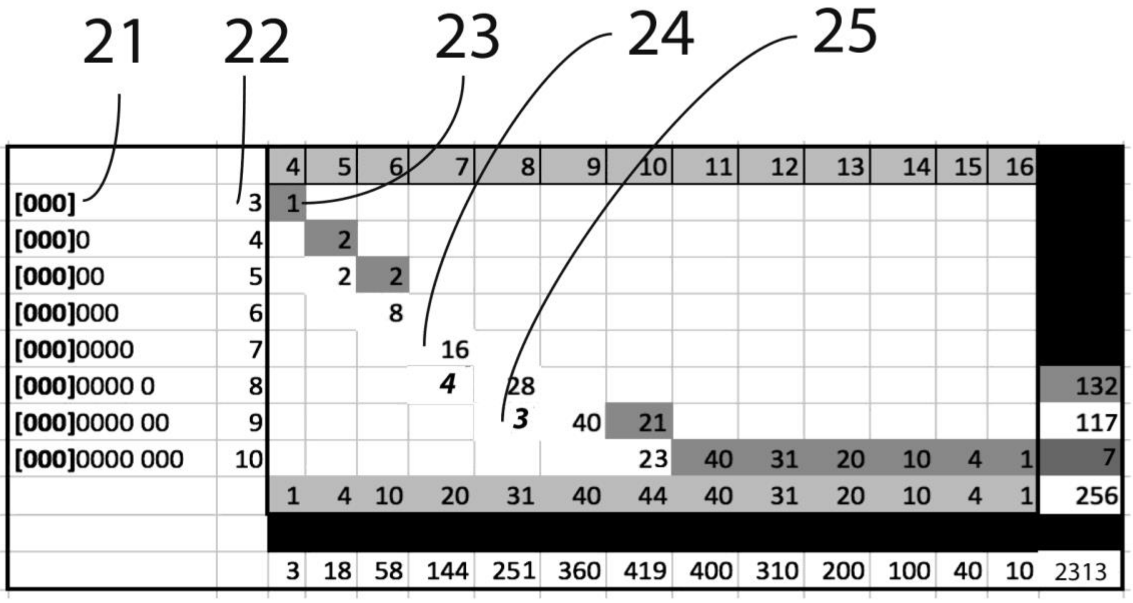


Fig. 10

<u>Figure 11</u> is a chart that assists in explaining how we might convert our code to a variable length code to get even better results. **21** shows how many place holders would be reserved to encode our data in this method. [000] only has the one option of no number whereas [000]0 has the option of two numbers and would likely be encoded as **[001]**0 and **[001]**1. **[000]**00 would have four options and would likely be encoded as **[011]**00, **[011]**01, **[011]**10 and **[011]**11. This process would continue until we ran out of unique options. **22** tells us how long the new variable length code will be.

23 tells us that all the dark numbers inside the chart provide savings (1, 2, 2, 21, 40, 31, 20, 10, 4, 1). We look at the heading (such as the 4) at the top and below it we see the number 1. This means that the number 4 happens only once in our permutation count and that we will thus be able to encode this representation of four bits into a three-bit code system. **24** are a set of numbers surrounded by a lighter background (2, 8, 16, 28, 40, 23) and shows where the code length will stay the same.

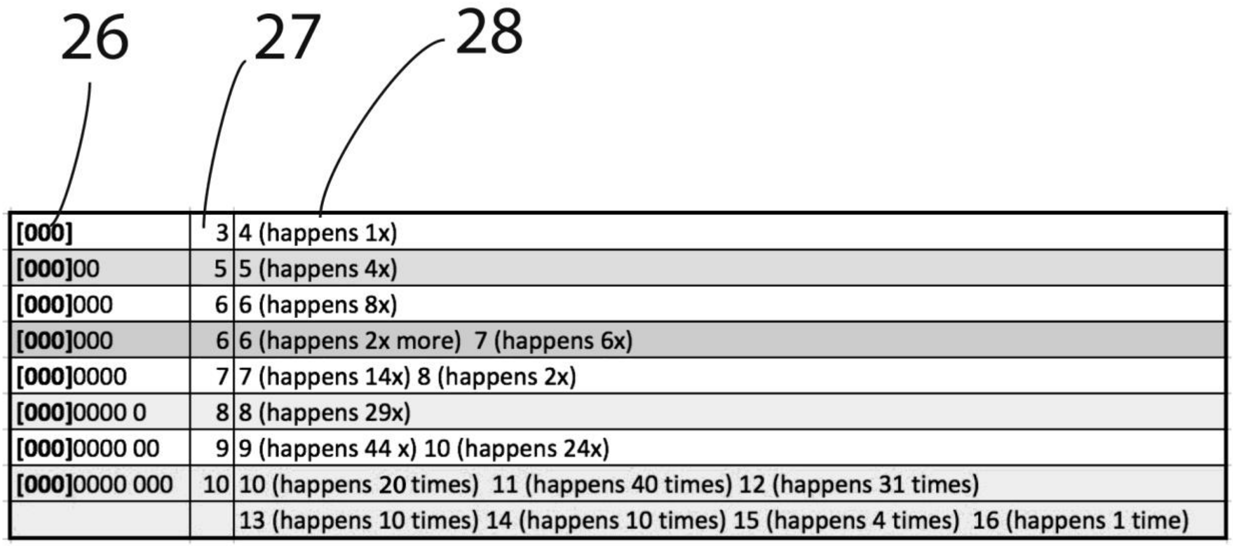
25 are bold italic numbers of 4 and 3 that show where our system would fail in those two scenarios by making the re-coding longer. To the right of the chart we see that 132 permutations would benefit, 117 would remain the same and 7 would elongate for our total of 256. The number at the bottom of the chart are how many bits would be counted if each permutation was counted once. Obviously this would be a rare occurrence, but it does provide us with a number to measure our potential efficiency with. Thus if every permutation happened once in a selected data set we would have had an original data set of 2560. If we converted this into one-byte (or 8 bit) permutation representations, we would cut our bits down to 2048 for a savings of 512 bits (64 bytes). By using the variable length coding method described we would get 2313, which is not as efficient as the 8- bit system, but may have other benefits in some scenarios.


21
22
23
24
25

		4	5	6	7	8	9	10	11	12	13	14	15	16	
[000]	3	1													
[000]0	4		2												
[000]00	5		2	2											
[000]000	6			8											
[000]0000	7				16										
[000]0000 0	8				***4***	28									132
[000]0000 00	9					***3***	40	21							117
[000]0000 000	10							23	40	31	20	10	4	1	7
		1	4	10	20	31	40	44	40	31	20	10	4	1	256
		3	18	58	144	251	360	419	400	310	200	100	40	10	2313

Fig. 11

Figure 12 provides an alternate way of using variable length code. In this example no selected permutation will ever be made longer. This may be a benefit in some cases where entropy is acting funny. By this method (if every permutation were to happen once), the data of 2560 would be compressed to 2235. Other forms of variable-length coding, such as Huffman coding can also be used.



[000]	3	4 (happens 1x)
[000]00	5	5 (happens 4x)
[000]000	6	6 (happens 8x)
[000]000	6	6 (happens 2x more) 7 (happens 6x)
[000]0000	7	7 (happens 14x) 8 (happens 2x)
[000]0000 0	8	8 (happens 29x)
[000]0000 00	9	9 (happens 44 x) 10 (happens 24x)
[000]0000 000	10	10 (happens 20 times) 11 (happens 40 times) 12 (happens 31 times)
		13 (happens 10 times) 14 (happens 10 times) 15 (happens 4 times) 16 (happens 1 time)

Fig. 12

To provide some additional examples and to address the elephant in the room regarding the potential size of such a program and duration of time to execute such a task as that described, alternative codes will now be presented. If one were to scan a set of data with a string of thirteen zeros' that were found in a row, we could use 13x13x13x13x13x13x13x13x13x13x13x13x13 = (or) 302,875,106,592,253 various permutations which could be represented in 49-bit groupings (with many bits left over) to compress the data, (Note: *this is MUCH too large*).

Once that data is output from the initial compression we would again run the scan and again use which ever permutation set would best compress the newly scanned data. If the scan found a longest run of 12, we would use the possible permutations from 12x12x12x12x12x12x12x12x12x12x12x12 = (or) 8,916,100,448,256 to compress this new data and so on. As the numbers get larger we can use hexadecimal or make sure to use clear separators for our groupings. For example, [1,6,8,4,6,12,2,8,9,11,10,13,4] would be considered a 13-run permutation, even though it has double digit numbers. Alternatively, we could write it as [1, 6, 8, 4, 6, C, 2, 8, 9, B, A, D, 4] wherein A=10, B=11, C=12 and D=13.

These number are very large and thus would require a powerful super computer to be able to represent the total 8,916,100,448,256 twelve-set permutations each consisting of 5.5 Bytes of data.

In one instance we can always shorten our permutation numbers to fewer digits (i.e. [2,4,6,13,5,11] for a-13 run or use more digits (1-20) than the run requires to achieve a variety of results, but this is somewhat less efficient. A more efficient system would be to use a scan that showed that a 5-run of 5 zeros or 5 ones in a row could use a ten run permutation of only five (1-5) unique integers, such as [5,1,3,2,4,3,2,4,5,1]. Any mixture or variation is possible.

It is also possible to mix different lengths of permutations into our encoding/decoding scheme. For example, we may have 111, 112, 113, 121 etc. mixed with groupings of 11, 12, 13, etc. or even groupings of single digits such as 1, 2, 3, 4, 5, 6 etc. Some groupings may also include a "don't flip" command. For example, we may have a permutation that would be 221x wherein x represents that we do not flip the next permutation to its opposite 0 or 1, but rather add the digit after the x to the digit preceding the x, thus giving us a larger number and allowing us to keep our program down to a smaller size. 123x, 4323, could be interpreted as 011000000011100111 or 100111111100011000. The x or don't flip command in this instance would be included in one or more of the permutations. By mixing **(a)** the size of the integers, (n=1, 2, 3, etc.) **(b)** the number of integers (nnnnnn, etc.) in each grouping and **(c)** a don't flip command x, we can create a variety of programs that can compress data in a variety of ways, while still controlling our program size.

Remainders may be dealt with by simply sending an addition note in the program regarding what the remainder is, or by making x (or some other variable) notate an end. For example, 212xx, or 2345x or 212xx or 2xxxx could all be endings of a data set or ##xxx could represent delete the last two numbers of the previous code (which defaulted to 1 upon discovering a high number).

A longest run scan is not a crucial aspect of the invention. A data scan still needs to be completed to determine how to encode the data, but not to determine the longest run of zeros or ones for selecting a specific program.

In this scenario and in this type of system (i.e. 212xx) a "don't flip" command and/or don't use a number here command may be used, as in this type of scenario not all strings of data will necessarily fit perfectly into our permutation sets. For example, if we have groupings of three permutations of up to three integers, but there is a need for the number 4, 5, 6 etc. (and if we did not do a longest run scan), then our "don't flip" command will provide the means for us to show necessary larger numbers. 212, 123, 12x, 321, would be 11011011 00010000 0110 or 00100100 11101111 1001. In this scenario the x would need to be encoded into the set of permutations as required. We see in this example how we have achieved the run of five 00000 or 11111.

The present invention seems to work best if the number of integers (horizontal) exceeds the size of the integers (vertical) used. As an example; A (4 4 4 4 4) or (n n n n n) grouping of five with up to four (n=1, 2, 3 or 4) provides a better and more useful system than (4 4 4 4) or (n n n n) four grouped sets of up to four. This is particularly useful when utilizing variable length codes.

The present system in essence requires that the number of permutations of the system be encoded with what would be equal to or more than enough bit pattern coding options to represent each unique permutation, but only up to a reasonable point, such as six by six. In a 6x6 setting we would have access to 46,656 permutations and 18,880 alternative codes, which could represent everything from xxxxx to 010101010101010 or 1000 in a row, etc., yet keep our program relatively small and fast. The possibilities are vast.

In another scenario 0000 0000 1111 1111 could = 10 or 11 or 12 or 13 etc. Alternatively, it could represent a permutation of less than ten numbers, such as 2 4 3 5 7 2 x x x x, or 9 8 7 6 x x x x x x, wherein the x simply states that no number is located here and we can precede to the next grouping, end the system or use a code of choice. This way we can use these methods for remainders or preceding larger numbers such as 10, 11, 12, 13, 14, 15 etc. By doing this we can keep our program smaller. For example, if a longest run scan showed a longest run of 30 zeros or one's, our using 30 permutations of 30 would get extremely large and somewhat impractical (impossible) for a non-super computer, but since a run of 30 in a row is not very common, our alternative codes should be able to manage.

Variable length codes will also assist us in optimizing our compression so that more frequently occurring permutations are made as short as possible with the goal being to make them at least the same length or shorter than the data converted from the original data set. Longer permutation sets will be used for the permutation sets that add up to the smallest numbers and that have higher probability of occurrence than other sets. As an example in (Fig. 8) our variable length encoding system provides 30 possible patterns for use, but we utilize only 27. If we deleted the permutation set of 111 See. (Fig. 5) represented by **[00]**0 in Fig. 8, and replaced it with 1111, 1112 and 1113 we could eliminate the permutation set that only adds up to three, which in some instances (with larger data sets) could be troublesome. We in essence skew our data. We may decide to use longer variable length codes in this embodiment as we are no longer tied to the length of ***three***. We could now start with [00]00 rather than only [00]0. Again, it doesn't seem super relevant in such a small example as that in the figures provided, but it can become more significant in larger sets of permutations and data sets.

Our variable length codes may also be designed so that we have a number of extra bit patterns that we can encode a variety of useful options into. Elimination of smaller permutation sums, such as 111=3 in Fig. 5 is simply one example. We may decide that 112=4 is also too small. Alternatively, some of these additional bit patterns could be used to encode operators, especially if a longest run scan in not employed. For example, [11]1111 could be encoded to mean (Plus, Do Nothing, Plus) or (+ N/A +), wherein two permutations can be added. 323 (+ N/A +) 323 would now become 626 which provides us with a larger number than was available in our three sets of three permutations. 123 (N/A + +) 133 would become 156. For a longer example **87684645**

(+N/A++++N/A+) **87486545** would become 16 7 10 16 10 11 4 10. Other operators could also be employed for creative uses as well. These operators will be highly useful in sets of data where larger numbers (long consecutive strings of zeros and ones) are found. If these longer consecutive runs are infrequent the fact that they may come at the expense of some longer encodings is negligible to the compression of the entire system.

In some instances, particularly with highly redundant data, it may be beneficial to first run the data to be compressed through an existing or traditional compression system.

Appendix A

I submitted the following question to Quora (a platform wherein both experts and those with various skills answer questions), ***"If a lossless compression algorithm for random data was found (even though many state that it is impossible), what implications would it have? What cool things would be possible?"*** I received eleven answers that all stated that a recursive random data program is a scientific impossibility and that it would be unwise to believe otherwise. The top up-voted response (with 252 up-votes) is as follows. The other response are located below/

> "Compression of random data isn't just impossible in the colloquial sense of the word. That is, compression of random data is not just a problem that seems too hard to us at the moment.
>
> Compression of random data is really impossible in the strict mathematical sense of the word. We know for a fact that it cannot be done. We have proofs of that fact.
>
> It's actually quite easy to explain.
>
> There are some obvious problems: If you really could compress any file, you could just keep compressing the result again and again, shrinking it more and more until… what

exactly? You get an empty file? But how do you decompress that? But that's not the main reason why compression, in general, isn't possible.

How to get around this? Maybe compression only works when the files are large enough?

Nope.

Suppose we set ourselves a much, much more modest goal: instead of wanting to compress everything, we just want an algorithm that can shrink any file that has exactly 1,000,000,000 bytes by just ten bytes. Sounds doable?

Well, even this tiny goal is still impossible. The reason? Compression has to be injective: when you compress two different files, you must again obtain two different files — otherwise you wouldn't know which one of the two was the original. However, the number of files that have exactly 1G bytes is much, much bigger than the number of files that are 10 bytes smaller than that: the first batch is exactly 1,208,925,819,614,629,174,706,176 times bigger than the second one.

Essentially, compression can only work in the presence of redundancy — that is, if the amount of information contained in the original file happens to be smaller than its size. This is often the case with real-life data (as we are the ones who store it in those redundant ways), but it's virtually never the case with random data."

- **Michal Forišek,**

(Works at Comenius University in Bratislava)

Nothing … quite literally. If random data was compressible, it would then mean the compressed version is also compressible, meaning you'd end up being able to compress, and then compress and then compress until you're left with … Nothing!

Other than that … nothing. That's it.

Though: "many state it is impossible" … nope everyone who actually know what they're talking about also know that it has been irrefutably proven to be impossible. The same way all mathematical proofs have been proven.

I.e. you could actually more easily state that 1+1 is only 2 for some people. Others can believe it's actually 24, or 5 million, or green, or hate. It boils down to the same premise of "yes but if random could be compressed" … exactly the same as saying "yes but if 1+1 is a dog".

- **Irné Barnard,**

Programming since the 80s

The question is like asking "can we make a white black object"

Bibliography

1. Huffman, D. *(1952).* "A Method for the Construction of Minimum-Redundancy Codes"*(PDF).* Proceedings of the IRE*.* ***40*** *(9): 1098–1101.* doi*:*10.1109/JRPROC.1952.273898
2. Shannon, C.E. (1948), "A Mathematical Theory of Communication", *Bell System Technical Journal*, 27, pp. 379–423 & 623–656, July & October, 1948. PDF.
3. R.V.L. Hartley, "Transmission of Information", *Bell System Technical Journal*, July 1928
4. Andrey Kolmogorov (1968), "Three approaches to the quantitative definition of information" in International Journal of Computer Mathematics.