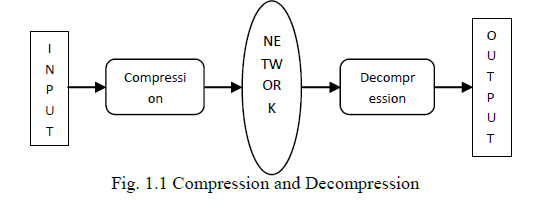
CHAPTER 1

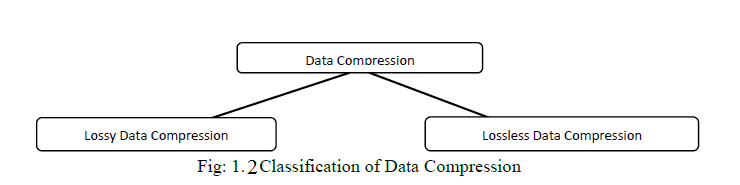
**INTRODUCTION**

* 1. **Introduction**

In respect to signal transmission, **data compression**, or **bit-rate reduction** can be referred to as encoding of provided data using less [bits](https://en.wikipedia.org/wiki/Bit) than the original pattern. It is broadly divided as either [lossy](https://en.wikipedia.org/wiki/Lossy_compression) or [lossless](https://en.wikipedia.org/wiki/Lossless_compression). In case of lossy compression, data bits are reduced by removing unnecessary or less important information which makes it impossible to recover original data. On the other hand, data bits are reduced by recognizing and discarding [statistical redundancy](https://en.wikipedia.org/wiki/Redundancy_(information_theory)) in [lossless compression](https://en.wikipedia.org/wiki/Lossless_compression) which means that the information can be recovered to its original form in lossless compression. Thus, data compression can be estimated as the procedure of minimizing the size of a data file.

Compression can be very useful as it greatly reduces resources and effort required to transmit and store data. As we know, both compression and decompression involves consumption of heavy amount of computational resources. Hence, data compression gets subjected to a trade-off between [space–time complexity](https://en.wikipedia.org/wiki/Time/space_complexity). During the modelling of data compression techniques, trade-offs between numerous aspects, including the compression ratio, the proportion of alteration introduced (while employing [lossy data compression](https://en.wikipedia.org/wiki/Lossy_data_compression) schemes), and the amount of computational resources needed for compression and decompression of the data are generally involved. For example, [a compression technique for video](https://en.wikipedia.org/wiki/Video_compression) file may be in need of expensive [hardware](https://en.wikipedia.org/wiki/Electronic_hardware) and software for the video to be decoded fast enough in order to be streamed as it is being decoded, and the alternative to decode the video fully before watching it may tend to be discomforting or require additional storage.



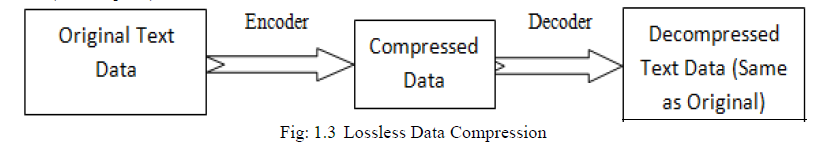
As depicted in above diagram, a compression engine is used in conversion of data from an easy-to-use format to the one optimized for space-compactness. Similarly, a decompression engine is used to return the compressed information to its original form while making data transmission speedy.

* 1. **Lossless Data Compression**

[Lossless data compression](https://en.wikipedia.org/wiki/Lossless_data_compression) [techniques](https://en.wikipedia.org/wiki/Algorithm) usually derive benefit from [statistical redundancy](https://en.wikipedia.org/wiki/Redundancy_(information_theory)) to represent data in lesser bits. This transformation is totally reversible since it is done without losing any [information](https://en.wikipedia.org/wiki/Self-information). Since most real-world data contains statistical redundancy in the presented information, application of lossless data compression techniques is sound in such cases. For instance, use of run length encoding in case of image representation can lead to better compactness in file size with full recovery of the information. There exist several schemes that reduce file size with the principle of eliminating data redundancy.

Text compression is viewed as a significant area for application of lossless compression. Here, it is required that the reconstruction of compressed text is identical to the original text, as minute differences may result in statements with categorically different meanings. The significance of no loss of data is more prevalent when dealing with sensitive and important information such as bank records etc.

The best modern lossless compression engine now make application of probabilistic models, i.e. prediction of data by partial matching. Another example of indirect form of statistical modelling is the [Burrows–Wheeler transform](https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform).

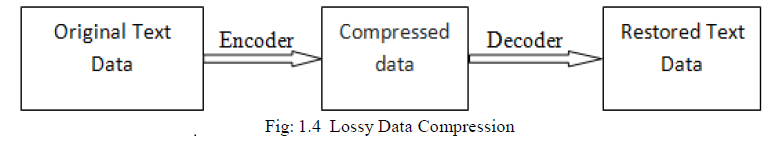
****

Lossless compression can be referred to as a simple case of redundancy reduction in input message. Since redundancy reduction depends upon the repeating sequences in the message, it doesn’t operate well on messages containing random text. Thus, lossless data compression is beneficial when we don’t want the received uncompressed file to differ in content with original file.

* 1. **Lossy Data Compression**

[Lossy data compression](https://en.wikipedia.org/wiki/Lossy_data_compression) is the scheme where inexact approximations are used along with partial data discarding to get compact data file while encoding. Loss of information is acceptable in these techniques. This technique follows the principle that dropping of non-essential details from the input data file can preserve storage space. These compression techniques are developed through analysis on how people represent and discern the data in probe. Therefore, there exists corresponding [trade-off](https://en.wikipedia.org/wiki/Trade-off) between maintaining information and minimizing size. Many popular compression techniques make use of these perceptual distinctions, such as [those employed in](https://en.wikipedia.org/wiki/Psychoacoustics) audio files, images, and video. For instance, the human eye is more responsive towards subtle imbalances in [luminance](https://en.wikipedia.org/wiki/Luminance) than the changes in color.

In lossy [audio compression](https://en.wikipedia.org/wiki/Audio_compression_(data)) scheme, non-audible components of the [audio](https://en.wikipedia.org/wiki/Audio_signal_processing) file are eliminated via methods of [psychoacoustics](https://en.wikipedia.org/wiki/Psychoacoustics). Audio compression is used in ripping of CD which is then decoded and streamed by the audio players. [JPEG](https://en.wikipedia.org/wiki/JPEG) [image compression](https://en.wikipedia.org/wiki/Image_compression) operates in part by rounding off non-essential bits of data file.

****

The benefit of using lossless compression techniques over lossy compression techniques is that the lossless compression results maintain the integrity of original input data after a compression/decompression cycle is completed. On the other hand, lossy compression schemes have large compression ratio as compared to lossless compression schemes. The performance of both algorithms can be differentiated using the parameters like compression ratio and Space savings.

* 1. **Theory**

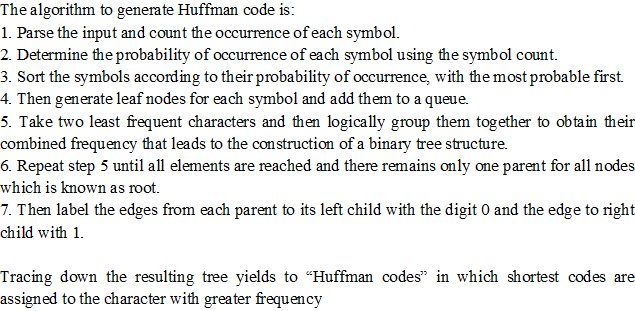
Some of the most used compression techniques are listed below:

***A. Huffman Coding:***

Huffman coding handles with compressing data based on their ASCII character. It makes a binary tree by following top-down approach to generate best outcome. In Huffman coding, most frequent character has the shortest bit length and the least frequent character has highest bit length. Leaves of the tree denote the character present in data that are to be encoded.

Edges are labelled as 0 or 1.

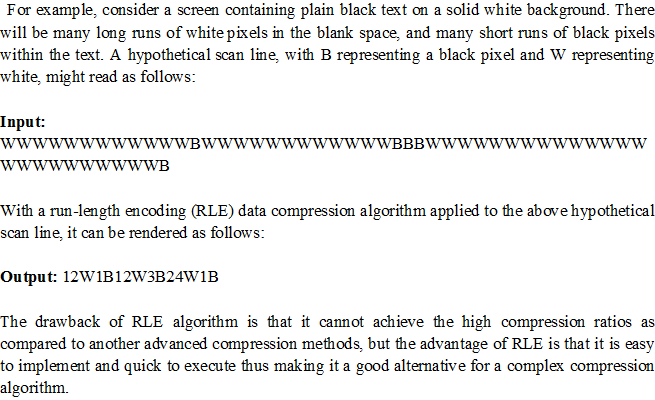
Huffman coding follows the following algorithm:



***B. Run length Encoding:***

Run length encoding deals with files which has data as a sequence of similar bytes. It is an easy data compression technique that is mainly employed by bitmap file like BMP. RLE compresses the file by minimizing the total length of repeating characters. In RLE we use two bytes to denote the occurrence of similar bytes. First byte denotes the length of character and second byte denote that character. The length of repeating character is called run.

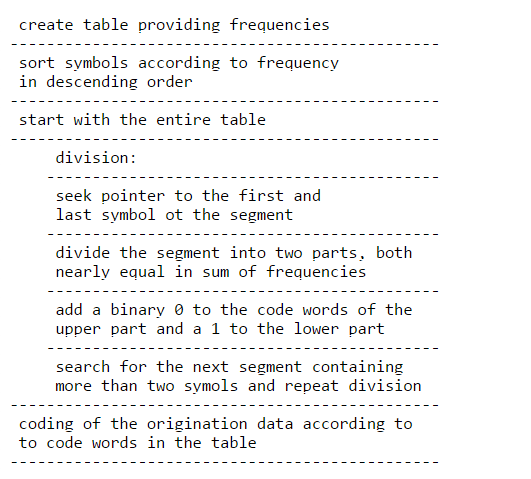
Run length encoding is generalization of zero suppression. Blank space in text file is just a symbol. Single space or pair of space is simply ignored. Single occurring character has run 1



***C. Shannon-Fano coding:***

This technique follows the binary tree generation representing the possibility of occurrence of each character. Ordering of characters is done based on decreasing order of probability. It means the symbols with least likely frequency appear at the bottom of the tree and symbol having highest probability (i.e. the most frequent symbol) comes at the top position of the tree.

Its pseudocode can be stated as:

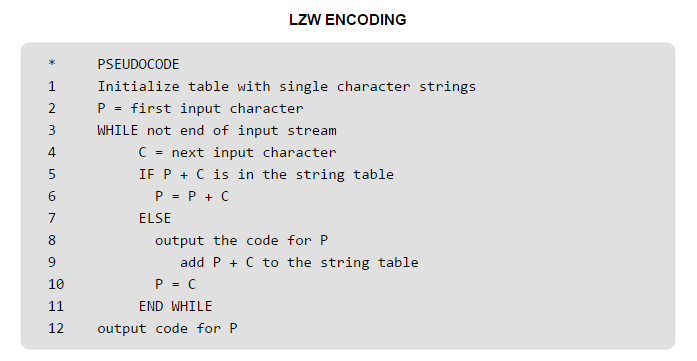


In general, the optimal code generation is not guaranteed by Shannon-Fano coding. This algorithm is efficient in the case when the probabilities of symbols are much closer to inverses of powers of 2.

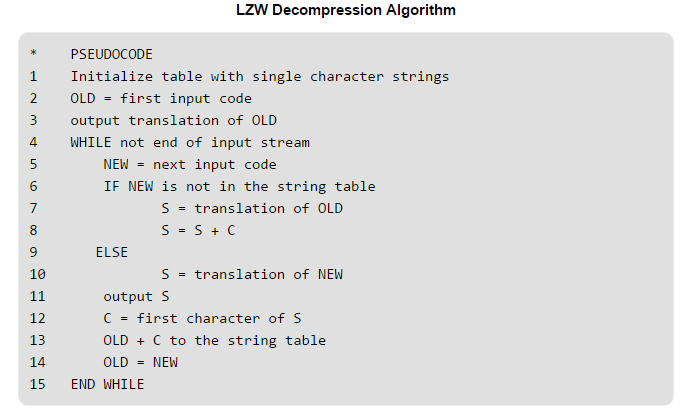
***D. Lempel–Ziv–Welch****(****LZW****)*

LZW compression operates by scanning a sequence of symbols followed by grouping the symbols into different strings, and thereafter converting the strings into binary codes. Since the codes hold up less space than the strings they replace, we get compressed data. Important features of LZW are as follows:

* It employs a code table which has 4096 table entries. Codes from 0-255 in the code table are always used to represent single bytes from the input text.
* When compression cycle begins, the table holds the first 256 entries only, with the remaining table entries being blanks. Compression is performed by employing codes from 256 through 4095 to constitute sequences of bytes.
* As encoding process follows, LZW recognizes repeated patterns in the data, and further adds them to the table.



During the LZW decompression process, compressed data is decoded by picking each code from the compressed output and then translation is done through the code table to evaluate which character it represents.



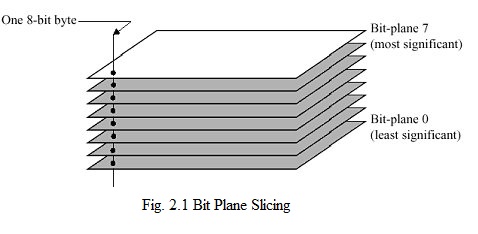
CHAPTER 2

**BIT-SLICING ALGORITHM**

**2.1. Bit Plane Slicing**

The proposed algorithm is derived from the principle of bit-plane slicing done in Digital Image Processing. In Bit Plane Slicing, we follow these principles:

* The contribution towards image appearance done by specific bit is highlighted instead of highlighting intensity ranges.
* Suppose that the image constitutes of eight 1-bit planes, varying from plane 7 as most significant bit to plane 0 as least significant bit.
* As per bit-plane slicing, it is revealed that the five highest order bits carry visually significant data. In this case, plane 7 correlates exactly with an image threshold at gray-level 128. Thus, the unimportant part is discarded that makes the largest contribution.



**2.2. Proposed Compression Algorithm**

The compression algorithm reduces standard 8-bit data into 6-bit and then rearranges it to get corresponding ASCII character.

**Constraint:** This Algorithm works only for alpha numeric data only.

**Mapping: [**0-63**]**

**A : 0 a : 26 0 : 52 space : 62**

**- : - - : - - : - new line : 63**

**- : - - : - - : -**

**Z : 25 z : 51 9 : 61**

**Steps**: -

1. Map the alphabets and numbers with the integers starting from 0.
2. Obtain the corresponding binary code of mapped values present in INPUT FILE.
3. Put these binary codes into an array of bytes (8-bit array), chopping extra 2-bit from left.
4. Rearrange the bits to get 8 bits again as follows:

* Left 2 bits of second character to first character.
* Left 4 bits of third character to second character.
* Right 2 bits of third character to fourth character.

1. Put the ASCII- Code of so obtained 8 bits in OUTPUT FILE.

**Example: -**

Suppose our text message is “AnTo”

1. Binary representation of mapped values of given characters are:

A : 00000000 n : 00101000

T : 00010011 o : 00101001

2. After removing two bits from left, we have   
 A : 000000 n : 101000

T : 010011 o : 101001

3. Now rearrange as follows:

* Left 2 bits of second character to first character.
* Left 4 bits of third character to second character.
* Right 2 bits of third character to fourth character.

So, we have :   
 000000 10**|**1000 0100**|**11 101001

4. Put the ASCII Code of formed 8-bit characters into a new file.

**2.3. Proposed Decompression Algorithm**

**Steps:-**

1. Map the alphabets and numbers with the integers starting from 0 in reverse order as we did in compression.
2. Obtain the corresponding binary code of characters present in COMPRESSED FILE.
3. Put these binary codes into an array of bytes (8-bit array).
4. Perform slicing and rearrangements of bits as follows:
   * Remove first 6 bits of first character.
   * Combine right 2 bits of first character and first 4 bits of second character.
   * Combine right 4 bits of second character and first 2 bits of third character.
   * Remove left 6 bits of third character.
5. Put the character of so formed bit after rearranging using map in Output file.

**Example:-**

Suppose our binary equivalent of characters in compressed file is

00000010 10000100 11101001

1. Perform the slicing of bits and rearrangement as given below:

* + Remove first 6 bits of first character.
  + Combine right 2 bits of first character and first 4 bits of second character.
  + Combine right 4 bits of second character and first 2 bits of third character.
  + Remove left 6 bits of third character.

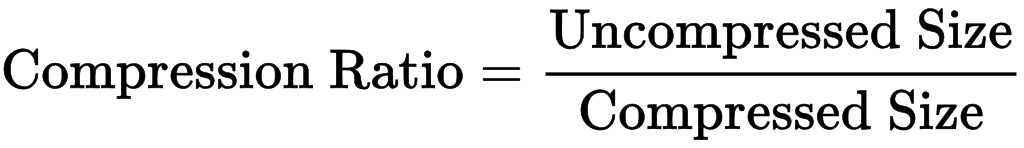
We will get following sequence of 6-bit data:

000000**|**10 1000**|**0100 11**|**101001

2. Print the mapped value of so obtained 6-bit data respectively.  
3. We will get back our input data same as before.

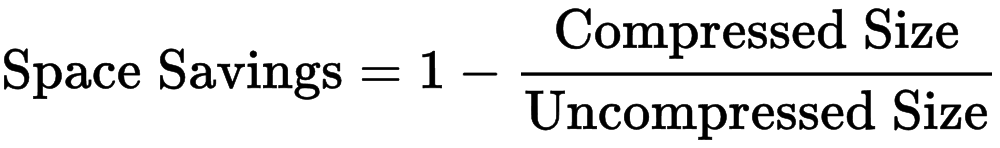
**2.4. Compression ratio and Space saving**

Compression ratio is referred to as the ratio between the uncompressed file size and compressed file size.

{\displaystyle {\rm {Compression\;Ratio}}={\frac {\rm {Uncompressed\;Size}}{\rm {Compressed\;Size}}}}

Therefore, a characterization of data that can compress a 20 MB file to 4 MB evaluates a compression ratio of 20/4 = 5, generally recorded as an explicit ratio, 5:1. Note that this formula is applicable equally for compression as well as for decompression.

Further, sometimes instead of compression ratio, the **space savings** parameter is used, which is referred to as the minimization in file size with respect to the uncompressed file size.

{\displaystyle {\rm {Space\;Savings}}=1-{\frac {\rm {Compressed\;Size}}{\rm {Uncompressed\;Size}}}}

Hence, a characterization of data that compresses a 20MB file to 4MB would generate a space savings of (1 - 4/20) = 0.8, generally recorded as a percentage of 80%.

For proposed algorithm, the compression ratio comes about **1.31** approx. and space saving is **38%** approximately on given input text.

**2.5. Data Encryption and Avalanche effect**

Data Encryption can be understood as the technique of representing an information in such a way that only authorized users can access it. Encrypting data doesn’t only prevent tampering, but also denies the access of intelligible information to an unauthorized entity. During encryption, the sender sends a message or information, known as plaintext, which is then encrypted using an encryption algorithm, producing ciphertext that can only be accessible by the receiver if it is decrypted. For introducing technical complexity in decryption, an encryption technique usually employs a pseudo-random encryption key generated by using hash functions. An authorized recipient may decode the received message using the [key](https://en.wikipedia.org/wiki/Key_(cryptography)) given by the sender to recipients, known as public key.”

In [cryptology](https://en.wikipedia.org/wiki/Cryptography), the **avalanche effect** is considered as one of the desirable aspects of cryptographic [algorithms](https://en.wikipedia.org/wiki/Algorithm). This is prevalent typically for [cryptographic hash functions](https://en.wikipedia.org/wiki/Cryptographic_hash_function) and [block ciphers](https://en.wikipedia.org/wiki/Block_cipher) where if there is any slight change in input (for instance, flipping of a single bit) then the output will change significantly (i.e. half the output bits may be flipped). In case of block ciphers which are of high-quality, small changes done in either the [plaintext](https://en.wikipedia.org/wiki/Plaintext) or the [key](https://en.wikipedia.org/wiki/Key_(cryptography)) would produce a substantial change in the [ciphertext](https://en.wikipedia.org/wiki/Ciphertext). If a cryptographic hash function or block cipher doesn’t display any avalanche effect, then it is said to contain poor randomization, and hence a [cryptanalyst](https://en.wikipedia.org/wiki/Cryptanalyst) may be capable to make guesses about the input, provided only some of the output cases. This knowledge may be adequate to break the algorithm partially or completely. Hence, the avalanche effect is a prudent condition as per the developer of the cryptographic schemes or device.”

Pertaining to above conditions, the proposed algorithm for data compression also possesses the property of data encryption along with it. It is evident from explained examples. Also, the proposed algorithm retains the avalanche effect which makes it a suitable enciphering algorithm to be used in messaging applications over network.

CHAPTER-3

**CONCLUSION**

**3.1. Conclusion**

The proposed algorithm works efficiently on messages containing alphanumeric characters. It can be viewed as a bit-slicing operation implemented on binary code of the mapped values for respective alphanumeric characters in the given message. This algorithm will always return constant compression ratio of 1.38 approx. and the space savings will be about 38% approx. Therefore, this algorithm can be considered suitable in streaming of message data across messaging devices and applications, provided they follow the constraint of including alphanumeric characters only. Furthermore, it also achieves encryption along with required avalanche effect while compression.

**REFERENCES**

1. Wade, Graham (1994). [Signal coding and processing](https://books.google.com/books?id=CJswCy7_W8YC) (2 ed.). Cambridge University Press. p. 34. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [978-0-521-42336-6](https://en.wikipedia.org/wiki/Special:BookSources/978-0-521-42336-6). Retrieved 2011-12-22. The broad objective of source coding is to exploit or remove 'inefficient' redundancy in the [PCM](https://en.wikipedia.org/wiki/Pulse-code_modulation) source and thereby achieve a reduction in the overall source rate R.
2. Mahdi, O.A.; Mohammed, M.A.; Mohamed, A.J. (November 2012). ["Implementing a Novel Approach and Convert Audio Compression to Text Coding via Hybrid Technique"](http://ijcsi.org/papers/IJCSI-9-6-3-53-59.pdf)(PDF). International Journal of Computer Science Issues. **9** (6, No. 3): 53–59. Retrieved 6 March 2013.
3. Pujar, J.H.; Kadlaskar, L.M. (May 2010). ["A New Lossless Method of Image Compression and Decompression Using Huffman Coding Techniques"](http://www.jatit.org/volumes/research-papers/Vol15No1/3Vol15No1.pdf). Journal of Theoretical and Applied Information Technology. **15** (1): 18–23.
4. Salomon, David (2008). A Concise Introduction to Data Compression. Berlin: Springer. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [9781848000728](https://en.wikipedia.org/wiki/Special:BookSources/9781848000728).
5. S. Mittal and J. Vetter, "[A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems](http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7110612)", IEEE Transactions on Parallel and Distributed Systems, 2015.
6. Tank, M.K. (2011). Implementation of Limpel-Ziv algorithm for lossless compression using VHDL. Thinkquest 2010: Proceedings of the First International Conference on Contours of Computing Technology. Berlin: Springer. pp. 275–283.
7. Navqi, Saud; Naqvi, R.; Riaz, R.A.; Siddiqui, F. (April 2011). ["Optimized RTL design and implementation of LZW algorithm for high bandwidth applications"](http://pe.org.pl/articles/2011/4/68.pdf). Electrical Review. **2011** (4): 279–285.
8. Mahmud, Salauddin (March 2012). ["An Improved Data Compression Method for General Data"](http://www.ijser.org/researchpaper%5CAn-Improved-Data-Compression-Method-for-General-Data.pdf). International Journal of Scientific & Engineering Research. **3** (3): 2. Retrieved 6 March 2013.
9. Arcangel, Cory. ["On Compression"](http://www.coryarcangel.com/downloads/Cory-Arcangel-OnC.pdf). Retrieved 6 March 2013.
10. S. Porwal, Y. Chaudhary, J. Joshi and M. Jain, “Data Compression Methodologies for Lossless Data and Comparison between Algorithms” International Journal of Engineering Science and Innovative Technology (IJESIT) Volume 2, Issue 2, March 2013
11. S. Shanmugasundaram and R. Lourdusamy, “A Comparative Study of Text Compression Algorithms” International Journal of Wisdom Based Computing, Vol. 1 (3), December 2011
12. S.R. Kodituwakku and U. S. Amarasinghe, "Comparison of Lossless Data Compression Algorithms for Text Data" Indian Journal of Computer Science and Engineering Vol1No. 4 416-425
13. [Welch, Terry](https://en.wikipedia.org/wiki/Terry_Welch) (1984). ["A Technique for High-Performance Data Compression"](http://www.cs.duke.edu/courses/spring03/cps296.5/papers/welch_1984_technique_for.pdf). *Computer*. **17** (6): 8–19. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1109/MC.1984.1659158](https://dx.doi.org/10.1109%2FMC.1984.1659158)

**APPENDIX**

**A1. Code for Proposed Data Compression algorithm**

#include<iostream>

#include<stdio.h>

#include<vector>

#include<map>

#include<algorithm>

#include<fstream>

#define ll int

using namespace std;

vector< pair<ll,ll> > A;

ll T[5];

map<ll, ll> mp;

int main()

{char ch;

long double count=0;

ll num=0;

ll f1=48;

ll f2=15;

ll s1=60;

ll s2=3;

ll temp1,temp2,temp3,temp4,lelo;

ll size=0;

FILE \*fin;

ofstream fout;

fin=fopen("INPUT.txt", "rb");

fout.open("OUTPUT.txt");

for(ch='A';ch<='Z';ch++)

{ll lol=ch;

mp[lol]=num ;

num++;

}

for(ch='a';ch<='z';ch++)

{ll lol=ch;

mp[lol]=num;

num++;

}

for(ch='0';ch<='9';ch++)

{ll lol=ch;

mp[lol]=num;

num++;

}

mp[32]=62;

mp[10]=63;

size=ftell(fin);

num=1;

while(1)

{

ch=fgetc(fin);

count++;

if(ch==EOF)

break;

T[num]=ch;

T[num]=mp[T[num]];

if(T[num]==0&&ch!='A')

continue;

num++;

if(num==5)

{num=1;

lelo=T[2]&32;

if(lelo==0)

temp1=0;

else

temp1=2;

lelo=T[2]&16;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+1;

lelo=T[2]&8;

if(lelo==0)

temp2=0;

else

temp2=8;

lelo=T[2]&4;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+4;

lelo=T[2]&2;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+2;

lelo=T[2]&1;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+1;

lelo=T[3]&32;

if(lelo==0)

temp3=0;

else

temp3=8;

lelo=T[3]&16;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+4;

lelo=T[3]&8;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+2;

lelo=T[3]&4;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+1;

lelo=T[3]&2;

if(lelo==0)

temp4=0;

else

temp4=2;

lelo=T[3]&1;

if(lelo==0)

temp4=temp4+0;

else

temp4=temp4+1;

char ch1,ch2,ch3;

ch1= (T[1]<<2) + temp1;

ch2= (temp2<<4) + temp3;

ch3= (temp4<<6) + T[4];

fout<<ch1<<ch2<<ch3;

T[1]=-1;

T[2]=-1;

T[3]=-1;

T[4]=-1;

}

}

for(ll i=num;i<=4;i++)

{T[i]=62;count++;}

if(num!=1)

{

lelo=T[2]&32;

if(lelo==0)

temp1=0;

else

temp1=2;

lelo=T[2]&16;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+1;

lelo=T[2]&8;

if(lelo==0)

temp2=0;

else

temp2=8;

lelo=T[2]&4;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+4;

lelo=T[2]&2;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+2;

lelo=T[2]&1;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+1;

lelo=T[3]&32;

if(lelo==0)

temp3=0;

else

temp3=8;

lelo=T[3]&16;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+4;

lelo=T[3]&8;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+2;

lelo=T[3]&4;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+1;

lelo=T[3]&2;

if(lelo==0)

temp4=0;

else

temp4=2;

lelo=T[3]&1;

if(lelo==0)

temp4=temp4+0;

else

temp4=temp4+1;

char ch1,ch2,ch3;

ch1= (T[1]<<2) + temp1;

ch2= (temp2<<4) + temp3;

ch3= (temp4<<6) + T[4];

fout<<ch1<<ch2<<ch3;

T[1]=-1;

T[2]=-1;

T[3]=-1;

T[4]=-1;

}

//cout<<"INPUT FILE SIZE : "<<(count)/1024<<"kb\n\n";

return 0;

}

**A2. Code for Proposed Data Decompression algorithm**

#include<iostream>

#include<stdio.h>

#include<vector>

#include<map>

#include<algorithm>

#include<fstream>

#define ll int

using namespace std;

vector< pair<ll,ll> > A;

ll T[5];

ll call(char);

map<ll, ll> mp;

int main()

{char ch;

ll num=0;

ll f1=48;

ll f2=15;

ll s1=60;

ll s2=3;

ll temp1,temp2,temp3,temp4,temp5,temp6,lelo;

long long count=0;

FILE \*fin;

ofstream fout;

fin=fopen("OUTPUT.txt", "rb");

fout.open("OUTPUT2.txt");

for(ch='A';ch<='Z';ch++)

{ll lol=ch;

mp[num]=lol ;

num++;

}

for(ch='a';ch<='z';ch++)

{ll lol=ch;

mp[num]=lol;

num++;

}

for(ch='0';ch<='9';ch++)

{ll lol=ch;

mp[num]=lol;

num++;

}

mp[62]=32;

mp[63]=10;

num=1;

while(1)

{ count++;

ch=fgetc(fin);

T[num]=call(ch);

num++;

if(num==4)

{num=1;

if(T[1]==255&&T[2]==255&&T[3]==255)

break;

lelo=T[1]&128;

if(lelo==0)

temp1=0;

else

temp1=32;

lelo=T[1]&64;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+16;

lelo=T[1]&32;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+8;

lelo=T[1]&16;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+4;

lelo=T[1]&8;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+2;

lelo=T[1]&4;

if(lelo==0)

temp1=temp1+0;

else

temp1=temp1+1;

lelo=T[1]&2;

if(lelo==0)

temp2=0;

else

temp2=2;

lelo=T[1]&1;

if(lelo==0)

temp2=temp2+0;

else

temp2=temp2+1;

lelo=T[2]&128;

if(lelo==0)

temp3=0;

else

temp3=8;

lelo=T[2]&64;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+4;

lelo=T[2]&32;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+2;

lelo=T[2]&16;

if(lelo==0)

temp3=temp3+0;

else

temp3=temp3+1;

lelo=T[2]&8;

if(lelo==0)

temp4=0;

else

temp4=8;

lelo=T[2]&4;

if(lelo==0)

temp4=temp4+0;

else

temp4=temp4+4;

lelo=T[2]&2;

if(lelo==0)

temp4=temp4+0;

else

temp4=temp4+2;

lelo=T[2]&1;

if(lelo==0)

temp4=temp4+0;

else

temp4=temp4+1;

lelo=T[3]&128;

if(lelo==0)

temp5=0;

else

temp5=2;

lelo=T[3]&64;

if(lelo==0)

temp5=temp5+0;

else

temp5=temp5+1;

lelo=T[3]&32;

if(lelo==0)

temp6=0;

else

temp6=32;

lelo=T[3]&16;

if(lelo==0)

temp6=temp6+0;

else

temp6=temp6+16;

lelo=T[3]&8;

if(lelo==0)

temp6=temp6+0;

else

temp6=temp6+8;

lelo=T[3]&4;

if(lelo==0)

temp6=temp6+0;

else

temp6=temp6+4;

lelo=T[3]&2;

if(lelo==0)

temp6=temp6+0;

else

temp6=temp6+2;

lelo=T[3]&1;

if(lelo==0)

temp6=temp6+0;

else

temp6=temp6+1;

char ch1,ch2,ch3,ch4;

ch1= mp[temp1];

ch2= mp[ (temp2<<4) + temp3];

ch3= mp[ (temp4<<2) + temp5];

ch4= mp[temp6];

fout<<ch1<<ch2<<ch3<<ch4;

T[1]=-1;

T[2]=-1;

T[3]=-1;

T[4]=-1;

}

}

return 0;

}

ll call(char ch)

{ll num=ch,temp,lelo;

lelo=num&128;

if(lelo==0)

temp=0;

else

temp=128;

lelo=num&64;

if(lelo==0)

temp=temp+0;

else

temp=temp+64;

lelo=num&32;

if(lelo==0)

temp=temp+0;

else

temp=temp+32;

lelo=num&16;

if(lelo==0)

temp=temp+0;

else

temp=temp+16;

lelo=num&8;

if(lelo==0)

temp=temp+0;

else

temp=temp+8;

lelo=num&4;

if(lelo==0)

temp=temp+0;

else

temp=temp+4;

lelo=num&2;

if(lelo==0)

temp=temp+0;

else

temp=temp+2;

lelo=num&1;

if(lelo==0)

temp=temp+0;

else

temp=temp+1;

return temp;

}