UiT Norges arktiske universitet

DTE- 2804-1: E-Health and Systems

Lecture 2. Lossless Compression

Arthur Schuchter
UiT – Campus Bodø

Autumn Term 2023

Content

1. Lossless Compression

Understanding Lossless Compression

Introduction to Lossless Compression:

- Lossless compression is a data compression technique that reduces the size of a file without sacrificing any data.
- Unlike lossy compression, it retains all original information after compression and decompression.

Principles of Lossless Compression:

- Utilizes algorithms to encode data in a more efficient manner.
- Reversible process: Decompression restores the original data exactly as it was before compression.

Applications:

- Commonly used for text files, documents, and program files.
- Ideal for scenarios where data integrity and exact reproduction are crucial.

Advantages:

- Preserves original data quality.
- Suitable for data that can't afford any loss, such as medical records or legal documents.

Disadvantages:

- Typically achieves lower compression ratios compared to lossy compression.
- May not be the most efficient choice for compressing media files.

Lossless Compression Techniques

Run-Length Encoding (RLE):

- Replaces sequences of repeated data with a single data value and a count.
- Effective for compressing simple images and binary data.

Huffman Coding:

- Assigns variable-length codes to characters based on their frequency of occurrence.
- Well-suited for compressing text files and other data with predictable patterns.

Lempel-Ziv-Welch (LZW):

- Replaces repeated sequences of characters with shorter codes.
- Widely used in file formats like GIF and TIFF, as well as UNIX compress utility.

Burrows-Wheeler Transform (BWT):

- Rearranges characters to create runs of similar characters, enhancing compression.
- Often used as a preprocessing step before applying other compression algorithms.

Delta Encoding:

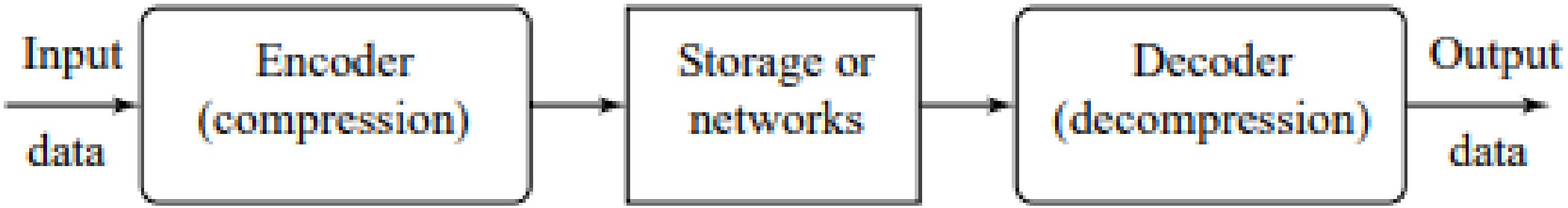
- Represents data as the difference between consecutive values.
- Efficient for compressing data with predictable changes.

Conclusion:

- Lossless compression techniques play a crucial role in maintaining data integrity while reducing storage space.
- The choice of algorithm depends on the type of data and compression requirements.

Basic Idea

- **Compression**: the process of coding that will effectively reduce the total number of bits needed to represent certain information.
- We often exploit redundancy to reduce the overal data size



Basic Idea

- If the compression and decompression processes induce no information loss, then the compression scheme is **lossless**; otherwise it is **lossy**
- **Compression ratio:**

$$compression\ ratio = \frac{B_0}{B_1}$$

B_0 – number of bits before compression
B_1 – number of bits after compression

1.1 Basics of Information Theory

- The entropy η of an information source with the alphabet S = $\{s_1, s_2, \ldots, s_n\}$ is

$$\eta = H(S) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

$$= -\sum_{i=1}^{n} p_i \log_2 p_i$$

p_i – probability that symbol s_i will occur in S.

$\log_2 \frac{1}{p_i}$ – indicates the amount of information (*self-information* as defined by Shannon) contained in s_i, which corresponds to the number of bits needed to encode s_i.

1.2 Run-Length Coding

Understanding Run-Length Encoding (RLE)

Introduction to Run-Length Encoding (RLE):

- Run-length encoding (RLE) is a simple lossless compression technique used to compress sequences of identical or repeated data.

Basic Idea:

- Sequential runs of the same data are replaced with a single data value and a count of the run's length.

Application Areas:

- RLE is efficient for compressing images with large areas of uniform color.
- Used in fax machines, bitmap images, and certain types of graphics.

How RLE Works:

- Identify sequences of identical data.
- Replace each sequence with the data value and the count of occurrences.
- This reduces repetitive data and creates a compressed representation.

Advantages:

- Simple and easy to implement.
- Effective for compressing simple images or binary data.

Run-Length Encoding in Action

Original Data:

- AAAAAABBBCCDAA

RLE Compressed Data:

- 6A2B2C1D2A

Explanation:

- A sequence of 6 consecutive 'A's is replaced by '6A'.
- Followed by 2 'B's replaced by '2B'.
- Then, 2 'C's replaced by '2C'.
- And so on...

Decompression Process:

- During decompression, '6A' is expanded to 'AAAAAA'.
- Similarly, '2B' is expanded to 'BB', and so on.

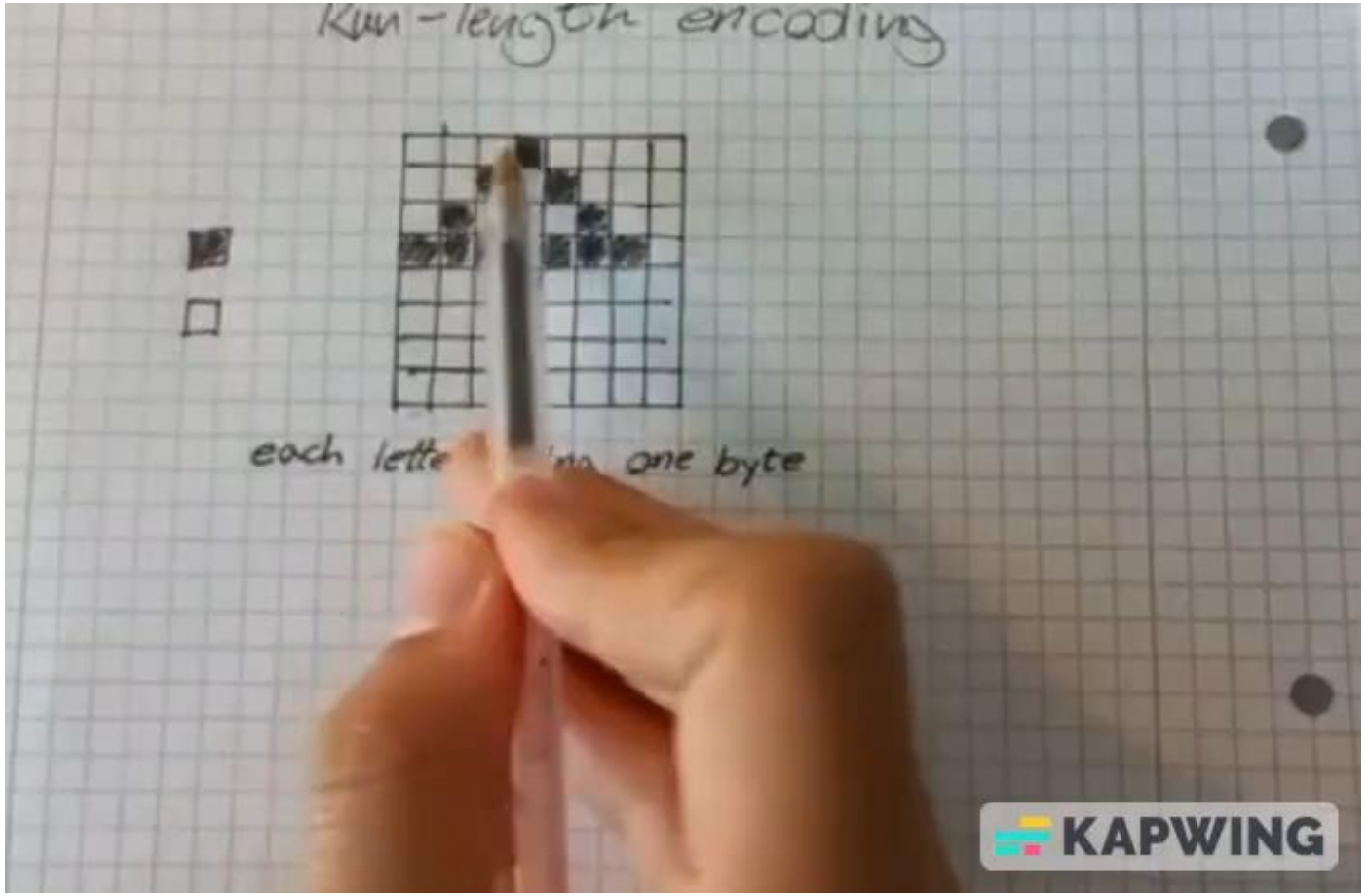
Limitations:

- Most effective on data with long runs of repeated characters.
- Not efficient for data with a diverse range of values.

Conclusion:

- Run-length encoding is a straightforward technique to compress repetitive data sequences, making it useful for certain types of data and applications.

Run-Length Encoding in Action



Run-Length Coding

- **Memoryless Source:** an information source that is independently distributed. Namely, the value of the current symbol does not depend on the values of the previously appeared symbols.
- Instead of assuming a memoryless source, Run-Length Coding (RLC) exploits memory present in the information source.
- **Rationale for RLC:** if the information source has the property that symbols tend to form continuous groups, then such symbol and the length of the group can be coded.

Run-Length-Encoding

- Run-length encoding (RLE) is a very simple form of data compression in which a stream of data is given as the input (i.e. "AAABBCCCC") and the output is a sequence of counts of consecutive data values in a row (i.e. "3A2B4C"). This type of data compression is lossless, meaning that when decompressed, all of the original data will be recovered when decoded. Its simplicity in both the encoding (compression) and decoding (decompression) is one of the most attractive features of the algorithm
- There are many different compression methods. Compression methods can be categorized as being lossless or lossy. With a lossless compression method, the original data can be restored exactly as it was before it was decompressed, whereas when a lossy compression method is used some of the original data is lost during the compression process and cannot be restored when the data is decompressed.
- There are times when lossy compression is the best choice – either when significantly smaller file size is desired or if the data loss is not noticeable, eg there are many sounds that people cannot hear and if compression results in the loss of some of these sounds then the data loss is not important.

Bit Level RLE

- Bit level RLE is effective when one bit is used to represent the colour of each pixel, ie in a monochrome image. Here a single byte represents the value of a pixel and the run length. Within the 8 bits, the left-most bit identifies the colour (eg 0 = white and 1 = black) the next 7 bits identify the run length (runs that are longer than 127 need to be broken down into a number of 127 long runs plus a run to represent the pixels 'left over').

Byte Level RLE

- Each pixel (colour) is represented by a single byte giving 256 possibilities. The colours themselves would be held within a colour palette table which would be stored in the image file. The colour palette is arbitrary and determined by factors such as hardware, image composition or file type. The image data is then represented by pairs of bytes representing the run length and then the index of the run colour in the color palette table

Run-length encoding

- Run-length encoding is probably the simplest method of compression. It can be used to compress data made of any combination of symbols. It does not need to know the frequency of occurrence of symbols and can be very efficient if data is represented as 0s and 1s.
- The general idea behind this method is to replace consecutive repeating occurrences of a symbol by one occurrence of the symbol followed by the number of occurrences.
- The method can be even more efficient if the data uses only two symbols (for example 0 and 1) in its bit pattern and one symbol is more frequent than the other.

Pixel Level RLE

- Each pixel is represented by three bytes, for an RGB bitmap. Each pair would consist of a run-length byte, followed by the three bytes that represent the pixel color.
- The metadata for a bitmap will include the number of rows and number of columns (in this case 8x8), so it is safe for the RLE to 'run over a row, ie reading from the top-left pixel.
- Because of this, RLE is only good for certain types of data and applications. For example, the Pixy camera, which is a robotics camera that helps you easily track objects, uses RLE to compress labeled video data before transferring it from the embedded camera device to an external application. Each pixel is given a label of "no object", "object 1", "object 2", etc. This is the perfect encoding for this application because of its simplicity, speed, and ability to compress the low-entropy label data.

RLE in “C++“

```cpp
// CPP program to implement run length encoding
#include <bits/stdc++.h>
using namespace std;

void printRLE(string str)
{
    int n = str.length();
    for (int i = 0; i < n; i++) {

        // Count occurrences of current character
        int count = 1;
        while (i < n - 1 && str[i] == str[i + 1]) {
            count++;
            i++;
        }
        // Print character and its count
        cout << str[i] << count;
    }
}
//Driver code
int main()
{
    string str = "wwwwaaadexxxxxxywww";
    printRLE(str);
    return 0;
}
```

https://www.geeksforgeeks.org/run-length-encoding/

RLE in “C#“

```
// C# program to implement run length encoding
using System;
class GFG
{
public class RunLength_Encoding
{
public static void printRLE(String str)
{
int n = str.Length;
for (int i = 0; i < n; i++)
{

// Count occurrences of current character
int count = 1;
while (i < n - 1 && str[i] == str[i + 1])
{
count++;
i++;
}

// Print character and its count
Console.Write(str[i]);
Console.Write(count);
}
}

public static void Main(String[] args)
{
String str = "wwwwaaadexxxxxxywww";
printRLE(str);
}
}
}
```

https://www.geeksforgeeks.org/run-length-encoding/

RLE in “Python“

```
# Python3 program to implement
# run length encoding
def printRLE(st):

n = len(st)
i = 0
while i < n- 1:

# Count occurrences of
# current character
count = 1
while (i < n - 1 and
st[i] == st[i + 1]):
count += 1
i += 1
i += 1

# Print character and its count
print(st[i - 1] +
str(count),
end = "")

# Driver code
if __name__ == "__main__":

st = "wwwwaaadexxxxxxywww"
printRLE(st)
```

https://www.geeksforgeeks.org/run-length-encoding/

1.3 Arithmetic Coding

Understanding Arithmetic Coding

Introduction to Arithmetic Coding:

- Arithmetic coding is a powerful and efficient entropy coding technique used in data compression.

Basic Idea:

- Unlike traditional methods that encode fixed-size symbols, arithmetic coding encodes entire messages as single fractional numbers.

How Arithmetic Coding Works:

- Divides the interval [0, 1) into sub-intervals based on symbol probabilities.
- The entire message is represented by a single number within the final sub-interval.
- The fractional number is then converted into a binary representation.

Application Areas:

- Used in various compression algorithms, including image, audio, and video compression.
- Widely used in lossless data compression standards.

Arithmetic Coding in Action

Steps of Arithmetic Coding:

- **Initialization:** Define initial interval [0, 1) and assign sub-intervals for symbols.
- **Encoding:** Update interval based on the probability of the next symbol.
- **Normalization:** Scale the interval to [0, 1) and output bits if needed.
- **Decoding:** Reverse the process to retrieve the original message.

Advantages:

- High compression efficiency due to flexible sub-interval representation.
- Adaptable to dynamic changes in symbol probabilities.
- Better compression for symbols with higher probabilities.

Challenges:

- Requires accurate symbol probabilities for optimal compression.
- More complex compared to simpler coding techniques.

Conclusion:

- Arithmetic coding is a versatile and efficient technique for lossless data compression, providing high compression ratios with accurate probability models.

Arithmetic Coding

- Arithmetic coding is a more modern coding method that usually outperforms Huffman coding.
- Huffman coding assigns each symbol a codeword that has an integral bit length. Arithmetic coding can treat the whole message as one unit.
- A message is represented by a half-open interval [a, b) where a and b are real numbers between 0 and 1. Initially, the interval is [0, 1). When the message becomes longer, the length of the interval shortens, and the number of bits needed to represent the interval increases.

Arithmetic Coding Encoder

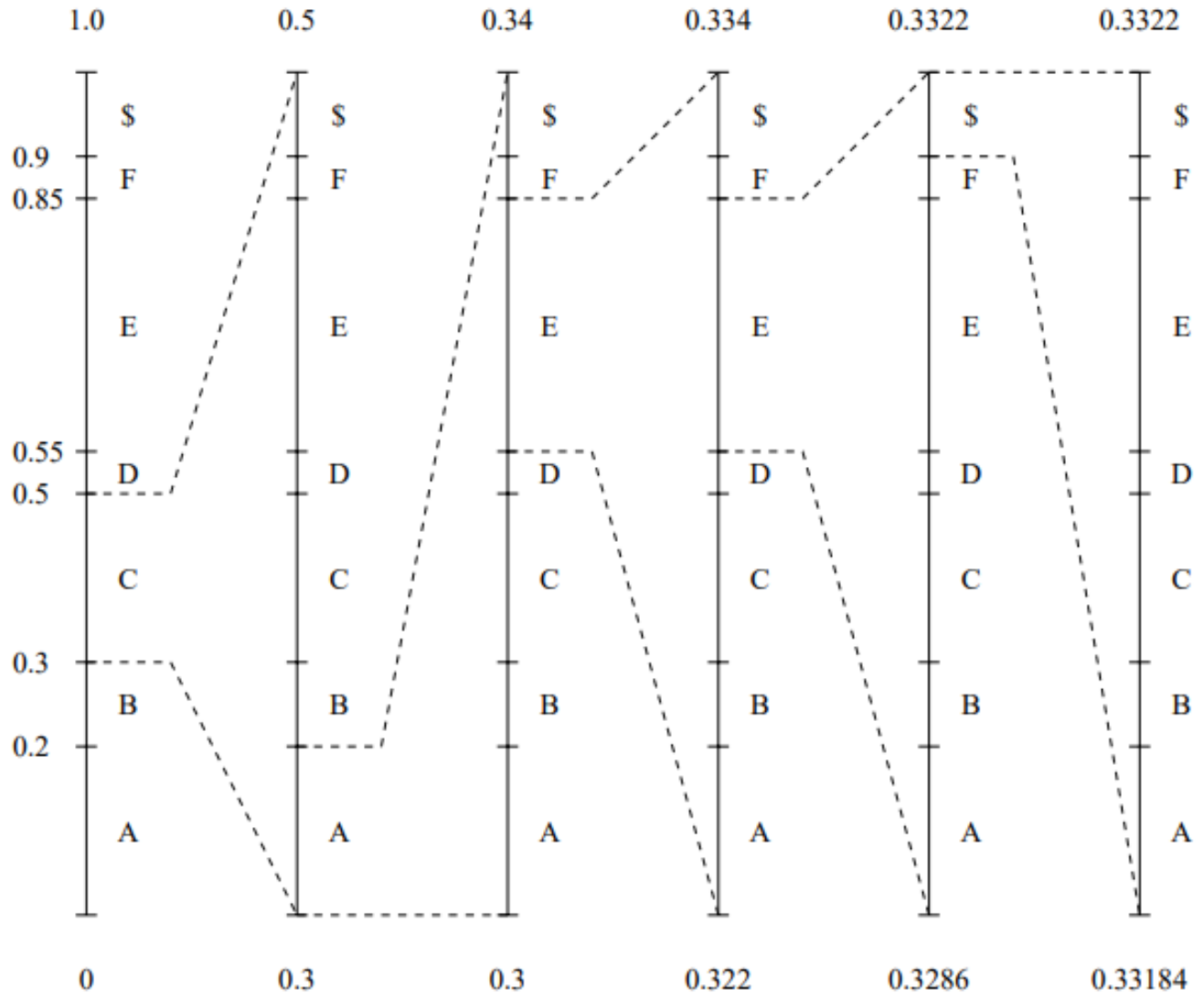
```
BEGIN
    low = 0.0;    high = 1.0;    range = 1.0;

    while (symbol != terminator)
        {
          get (symbol);
          low =  low + range * Range_low(symbol);
          high = low + range * Range_high(symbol);
          range = high - low;
        }

    output a code so that low <= code < high;
END
```

Arithmetic Coding: Encode Symbols “CAEE$”

Symbol	Probability	Range
A	0.2	[0, 0.2)
B	0.1	[0.2, 0.3)
C	0.2	[0.3, 0.5)
D	0.05	[0.5, 0.55)
E	0.3	[0.55, 0.85)
F	0.05	[0.85, 0.9)
$	0.1	[0.9, 1.0)

(a) Probability distribution of symbols.

Graphical display of shrinking ranges.



Arithmetic Coding: Encode Symbols “CAEE$”

Symbol	low	high	range
	0	1.0	1.0
C	0.3	0.5	0.2
A	0.30	0.34	0.04
E	0.322	0.334	0.012
E	0.3286	0.3322	0.0036
$	0.33184	0.33220	0.00036

(c) New *low*, *high*, and *range* generated.

Generating Codeword for Encoder

```
BEGIN
    code = 0;
    k = 1;
    while (value(code) < low)
        { assign 1 to the kth binary fraction bit
          if (value(code) > high)
             replace the kth bit by 0

          k = k + 1;
        }
END
```

- The final step in Arithmetic encoding calls for the generation of a number that falls within the range [*low*, *high*). The above algorithm will ensure that the shortest binary codeword is found.

Arithmetic Coding Decoder

```
BEGIN
    get binary code and convert to
        decimal value = value(code);
    Do
        { find a symbol s so that
                Range_low(s) <= value < Range_high(s);
          output s;
          low = Rang_low(s);
          high = Range_high(s);
          range = high - low;
          value = [value - low] / range;
        }
    Until symbol s is a terminator
END
```

Arithmetic coding: decode symbols “CAEE$”

value	Output Symbol	low	high	range
0.33203125	C	0.3	0.5	0.2
0.16015625	A	0.0	0.2	0.2
0.80078125	E	0.55	0.85	0.3
0.8359375	E	0.55	0.85	0.3
0.953125	$	0.9	1.0	0.1

Arithmetic Coding in “C++“

```
#include<iostream>
#include<unordered_map>
#include<vector>
using namespace std;

struct node{
double prob, range_from, range_to;
};

double encoding(unordered_map<char, node> arr, string s){
cout<<"\nEncoding\n";
double low_v=0.0, high_v=1.0, diff= 1.0;
cout<<"Symbol\tLow_v\tHigh_v\tdiff\n";
for(int i=0; i<s.size(); i++){
high_v= low_v+ diff* arr[s[i]].range_to;
low_v= low_v+ diff* arr[s[i]].range_from;
diff= high_v- low_v;
cout<<s[i]<<"\t"<<low_v<<"\t"<<high_v<<"\t"<<diff<<endl;
}
return low_v;
}

string decoding(unordered_map<char, node> arr, double code_word, int len){
cout<<"\nDecoding: \n";
char ch;
string text= "";
int j=0;
unordered_map<char, node>:: iterator it;
cout<<"Code\tOutput\tRange_from\tRange_to\n";
while(j<len){
cout<<code_word<<"\t";
for(it= arr.begin(); it!=arr.end(); it++){
char i= (*it).first;
if(arr[i].range_from<= code_word && code_word< arr[i].range_to){
ch= i;
code_word= (code_word-arr[i].range_from)/(arr[i].range_to- arr[i].range_from);
break;
}
}
cout<<ch<<"\t"<<arr[ch].range_from<<"\t\t"<<arr[ch].range_to<<endl;
text+= ch;
j++;
}
return text;
}
```

```
int main(){
int n;
cout<<"Enter number of characters: ";
cin>>n;
unordered_map<char, node> arr;
vector<char> ar;
double range_from= 0;
cout<<"Enter probability of each character:\n";
for(int i=0; i<n; i++){
char ch;
cin>>ch;
ar.push_back(ch);
cin>>arr[ch].prob;
arr[ch].range_from= range_from;
arr[ch].range_to= range_from+ arr[ch].prob;
range_from= arr[ch].range_to;
}
cout<<"Symbol\tProbability\tRange_from\tRange_to\n";
cout<<"-----------------------------------------------------\n";
for(int i=0; i<ar.size(); i++){
char ch= ar[i];
cout<<ch<<"\t"<<arr[ch].prob<<"\t\t"<<arr[ch].range_from<<"\t\t"<<arr[ch].range_to<<endl;
}
cout<<endl;
string s;
cout<<"Enter text: ";
cin>>s;
double code_word= encoding(arr, s);
cout<<"Code word for "<<s<<" is: "<<code_word<<endl;
string text= decoding(arr, code_word, s.size());
cout<<"Text for "<<code_word<<" is: "<<text<<endl;

}
```


https://rosettacode.org/wiki/Arithmetic_coding/As_a_generalized_change_of_radix

Arithmetic Coding in “C#“

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;
using System.Text;

namespace ArithmeticCoding {
    using Freq = Dictionary<char, long>;
    using Triple = Tuple<BigInteger, int, Dictionary<char,
long>>;

    class Program {
        static Freq CumulativeFreq(Freq freq) {
            long total = 0;
            Freq cf = new Freq();
            for (int i = 0; i < 256; i++) {
                char c = (char)i;
                if (freq.ContainsKey(c)) {
                    long v = freq[c];
                    cf[c] = total;
                    total += v;
                }
            }
            return cf;
        }

        static Triple ArithmeticCoding(string str, long radix) {
            // The frequency of characters
            Freq freq = new Freq();
            foreach (char c in str) {
                if (freq.ContainsKey(c)) {
                    freq[c] += 1;
                } else {
                    freq[c] = 1;
                }
            }

            // The cumulative frequency
            Freq cf = CumulativeFreq(freq);

            // Base
            BigInteger @base = str.Length;

            // Lower bound
            BigInteger lower = 0;
```

```
            // Product of all frequencies
            BigInteger pf = 1;

            // Each term is multiplied by the product of the
            // frequencies of all previously occuring symbols
            foreach (char c in str) {
                BigInteger x = cf[c];
                lower = lower * @base + x * pf;
                pf = pf * freq[c];
            }

            // Upper bound
            BigInteger upper = lower + pf;

            int powr = 0;
            BigInteger bigRadix = radix;

            while (true) {
                pf = pf / bigRadix;
                if (pf == 0) break;
                powr++;
            }

            BigInteger diff = (upper - 1) / (BigInteger.Pow(bigRadix, powr));
            return new Triple(diff, powr, freq);
        }

        static string ArithmeticDecoding(BigInteger num, long radix, int pwr, Freq freq) {
            BigInteger powr = radix;
            BigInteger enc = num * BigInteger.Pow(powr, pwr);
            long @base = freq.Values.Sum();

            // Create the cumulative frequency table
            Freq cf = CumulativeFreq(freq);

            // Create the dictionary
            Dictionary<long, char> dict = new Dictionary<long, char>();
            foreach (char key in cf.Keys) {
                long value = cf[key];
                dict[value] = key;
            }

            // Fill the gaps in the dictionary
            long lchar = -1;
            for (long i = 0; i < @base; i++) {
                if (dict.ContainsKey(i)) {
                    lchar = dict[i];
                } else if (lchar != -1) {
                    dict[i] = (char)lchar;
                }
            }
```

```
            // Decode the input number
            StringBuilder decoded = new StringBuilder((int)@base);
            BigInteger bigBase = @base;
            for (long i = @base - 1; i >= 0; --i) {
                BigInteger pow = BigInteger.Pow(bigBase, (int)i);
                BigInteger div = enc / pow;
                char c = dict[(long)div];
                BigInteger fv = freq[c];
                BigInteger cv = cf[c];
                BigInteger diff = enc - pow * cv;
                enc = diff / fv;
                decoded.Append(c);
            }

            // Return the decoded output
            return decoded.ToString();
        }

        static void Main(string[] args) {
            long radix = 10;
            string[] strings = { "DABDDB", "DABDDBBDDBA", "ABRACADABRA", "TOBEORNOTTOBEORTOBEORNOT"
};
            foreach (string str in strings) {
                Triple encoded = ArithmeticCoding(str, radix);
                string dec = ArithmeticDecoding(encoded.Item1, radix, encoded.Item2,
encoded.Item3);
                Console.WriteLine("{0,-25}=> {1,19} * {2}^{3}", str, encoded.Item1, radix,
encoded.Item2);
                if (str != dec) {
                    throw new Exception("\tHowever that is incorrect!");
                }
            }
        }
    }
}
```

UiT / NORGES ARKTISKE UNIVERSITET

https://rosettacode.org/wiki/Arithmetic_coding/As_a_generalized_change_of_radix

Arithmetic Coding in “Python“

```python
from collections import Counter

def cumulative_freq(freq):
    cf = {}
    total = 0
    for b in range(256):
        if b in freq:
            cf[b] = total
            total += freq[b]
    return cf

def arithmethic_coding(bytes, radix):

    # The frequency characters
    freq = Counter(bytes)

    # The cumulative frequency table
    cf = cumulative_freq(freq)

    # Base
    base = len(bytes)

    # Lower bound
    lower = 0

    # Product of all frequencies
    pf = 1

    # Each term is multiplied by the product of
the
    # frequencies of all previously occurring
symbols
    for b in bytes:
        lower = lower*base + cf[b]*pf
        pf *= freq[b]

    # Upper bound
    upper = lower+pf

    pow = 0
    while True:
        pf //= radix
        if pf==0: break
        pow += 1

    enc = (upper-1) // radix**pow
    return enc, pow, freq
```

```python
def arithmethic_decoding(enc, radix, pow, freq):

    # Multiply enc by radix^pow
    enc *= radix**pow;

    # Base
    base = sum(freq.values())

    # Create the cumulative frequency table
    cf = cumulative_freq(freq)

    # Create the dictionary
    dict = {}
    for k,v in cf.items():
        dict[v] = k

    # Fill the gaps in the dictionary
    lchar = None
    for i in range(base):
        if i in dict:
            lchar = dict[i]
        elif lchar is not None:
            dict[i] = lchar

    # Decode the input number
    decoded = bytearray()
    for i in range(base-1, -1, -1):
        pow = base**i
        div = enc//pow

        c  = dict[div]
        fv = freq[c]
        cv = cf[c]

        rem = (enc - pow*cv) // fv

        enc = rem
        decoded.append(c)

    # Return the decoded output
    return bytes(decoded)

radix = 10      # can be any integer greater or equal with 2

for str in b'DABDDB DABDDBBDDBA ABRACADABRA
TOBEORNOTTOBEORTOBEORNOT'.split():
    enc, pow, freq = arithmethic_coding(str, radix)
    dec = arithmethic_decoding(enc, radix, pow, freq)

    print("%-25s=> %19s * %d^%s" % (str, enc, radix, pow))

    if str != dec:
    raise Exception("\tHowever that is incorrect!")
```

https://rosettacode.org/wiki/Arithmetic_coding/As_a_generalized_change_of_radix

1.4 Variable-Length coding (VLC)

Variable-Length Coding (VLC)

Shannon-Fano Algorithm – a top-down approach

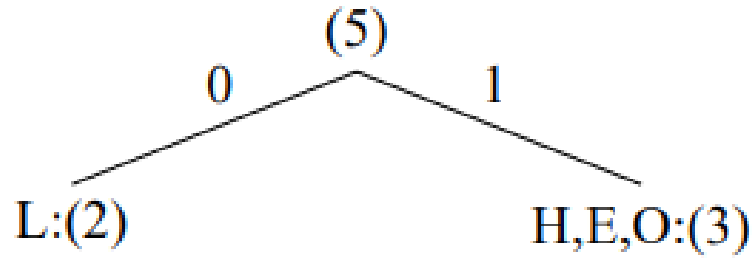
1. Sort the symbols according to the frequency count of their occurrences.
2. Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.

An Example: coding of "HELLO"

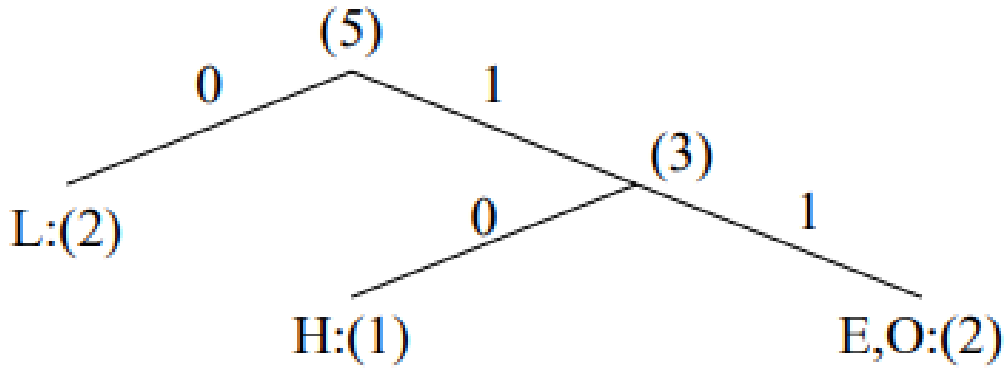
Symbol	H	E	L	O
Count	1	1	2	1

Frequency count of the symbols in "HELLO".

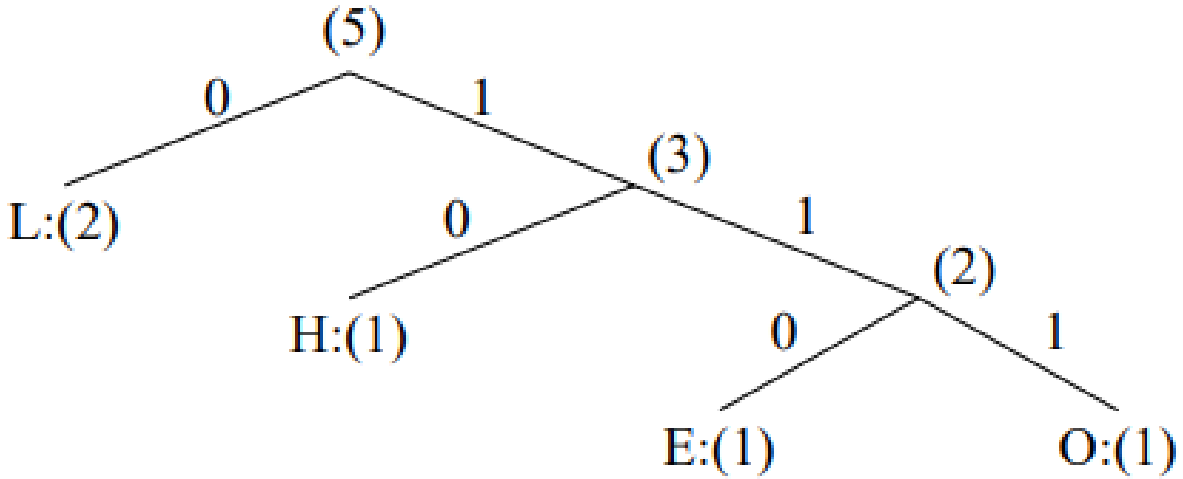
Coding Tree for HELLO by Shannon-Fano



(a)



(b)

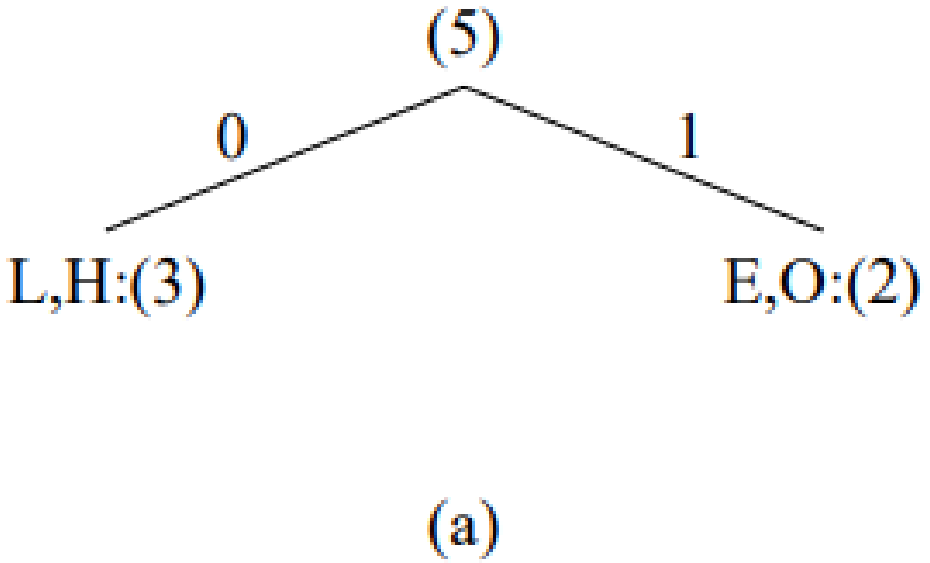


(c)

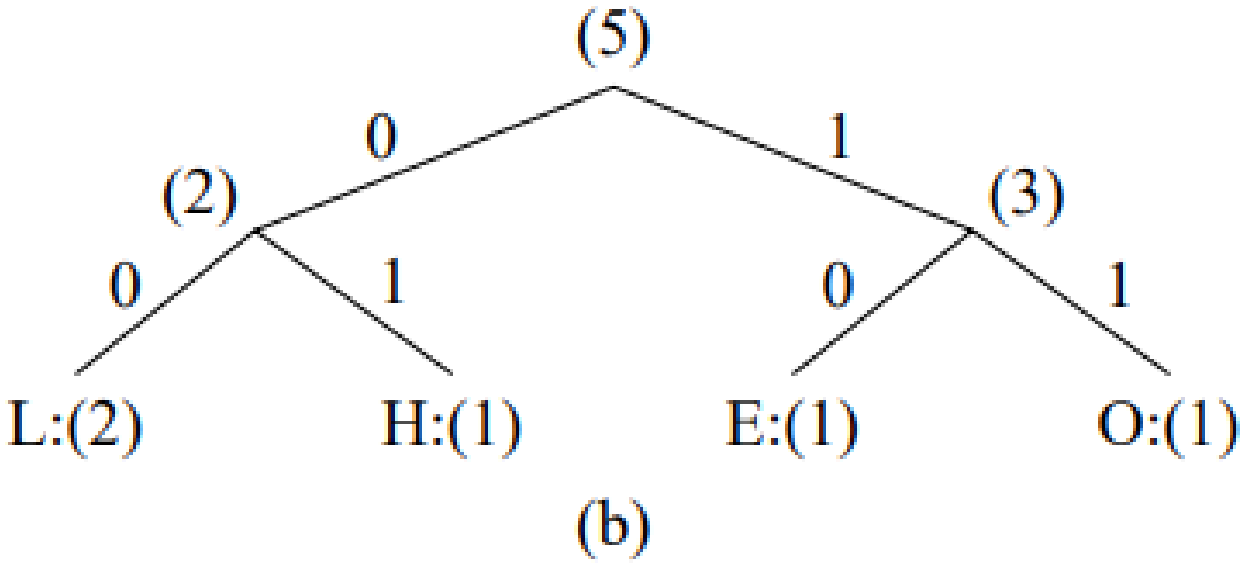
Result of Performing Shannon-Fano on HELL

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
TOTAL number of bits:				10

Another coding tree for HELLO by Shannon-Fano.



(a)



(b)

Another Result of Performing Shannon-Fano on HELLO

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
TOTAL number of bits:				10

VLC: Shannon-Fano in “C++“

```cpp
// C++ program for Shannon Fano Algorithm
// include header files
#include <bits/stdc++.h>
using namespace std;
// declare structure node
struct node {
    // for storing symbol
    string sym;
    // for storing probability or frequency
    float pro;
    int arr[20];
    int top;
} p[20];
typedef struct node node;
// function to find shannon code
void shannon(int l, int h, node p[])
{
    float pack1 = 0, pack2 = 0, diff1 = 0, diff2 = 0;
    int i, d, k, j;
    if ((l + 1) == h || l == h || l > h) {
        if (l == h || l > h)
            return;
        p[h].arr[++(p[h].top)] = 0;
        p[l].arr[++(p[l].top)] = 1;
        return;
    }
    else {
        for (i = l; i <= h - 1; i++)
            pack1 = pack1 + p[i].pro;
        pack2 = pack2 + p[h].pro;
        diff1 = pack1 - pack2;
        if (diff1 < 0)
            diff1 = diff1 * -1;
        j = 2;
        while (j != h - l + 1) {
            k = h - j;
            pack1 = pack2 = 0;
            for (i = l; i <= k; i++)
                pack1 = pack1 + p[i].pro;
            for (i = h; i > k; i--)
                pack2 = pack2 + p[i].pro;
            diff2 = pack1 - pack2;
            if (diff2 < 0)
                diff2 = diff2 * -1;
            if (diff2 >= diff1)
                break;
            diff1 = diff2;
            j++;
        }
        k++;
        for (i = l; i <= k; i++)
            p[i].arr[++(p[i].top)] = 1;
        for (i = k + 1; i <= h; i++)
            p[i].arr[++(p[i].top)] = 0;

        // Invoke shannon function
        shannon(l, k, p);
        shannon(k + 1, h, p);
    }
}
```

```cpp
// Function to sort the symbols
// based on their probability or frequency
void sortByProbability(int n, node p[])
{
    int i, j;
    node temp;
    for (j = 1; j <= n - 1; j++) {
        for (i = 0; i < n - 1; i++) {
            if ((p[i].pro) > (p[i + 1].pro)) {
                temp.pro = p[i].pro;
                temp.sym = p[i].sym;

                p[i].pro = p[i + 1].pro;
                p[i].sym = p[i + 1].sym;

                p[i + 1].pro = temp.pro;
                p[i + 1].sym = temp.sym;
            }
        }
    }
}

// function to display shannon codes
void display(int n, node p[])
{
    int i, j;
    cout << "\n\n\n\tSymbol\tProbability\tCode";
    for (i = n - 1; i >= 0; i--) {
        cout << "\n\t" << p[i].sym << "\t\t" << p[i].pro << "\t";
        for (j = 0; j <= p[i].top; j++)
            cout << p[i].arr[j];
    }
}
```

```cpp
// Driver code
int main()
{
    int n, i, j;
    float total = 0;
    string ch;
    node temp;

    // Input number of symbols
    cout << "Enter number of symbols\t: ";
    n = 5;
    cout << n << endl;

    // Input symbols
    for (i = 0; i < n; i++) {
        cout << "Enter symbol " << i + 1 << " : ";
        ch = (char)(65 + i);
        cout << ch << endl;

        // Insert the symbol to node
        p[i].sym += ch;
    }

    // Input probability of symbols
    float x[] = { 0.22, 0.28, 0.15, 0.30, 0.05 };
    for (i = 0; i < n; i++) {
        cout << "\nEnter probability of " << p[i].sym << " : ";
        cout << x[i] << endl;

        // Insert the value to node
        p[i].pro = x[i];
        total = total + p[i].pro;

        // checking max probability
        if (total > 1) {
            cout << "Invalid. Enter new values";
            total = total - p[i].pro;
            i--;
        }
    }

    p[i].pro = 1 - total;

    // Sorting the symbols based on
    // their probability or frequency
    sortByProbability(n, p);

    for (i = 0; i < n; i++)
        p[i].top = -1;

    // Find the shannon code
    shannon(0, n - 1, p);

    // Display the codes
    display(n, p);
    return 0;
}
```

https://www.geeksforgeeks.org/shannon-fano-algorithm-for-data-compression/

VLC: Shannon-Fano in “C#“

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.Json;
namespace SHANON_FANO
{
    class Program
    {
        static  IDictionary<char, int> ReturnWordCount(string word)
        {
            Dictionary<char, int> obj = new();
            word.ToList().ForEach(i =>
           {
               var found = obj.TryGetValue(i, out int value) ? obj[i]++ : obj[i] = 1;
           });
            Dictionary<char, int> sorted = obj.OrderByDescending(i => i.Value).ToDictionary(i => i.Key, i
=> i.Value);
            return sorted;
        }
        static Dictionary<char, string> answer = new();
        static  IDictionary<char, string> CreateTree(string word)
        {
            IDictionary<char, int> probClass = ReturnWordCount(word);
            probClass.Keys.ToList().ForEach(i => { answer[i] = ""; });
            if (probClass.Keys.Count > 1)
            {
                RecursiveMethod(group: probClass);
            }
            else answer[probClass.Keys.ToList()[0]] = "0";
            return answer;
        }
```

```
        static  void RecursiveMethod(
            IDictionary<char, int> group = null,
            string accumulation = ""
           )
        {
            group.Keys.ToList().ForEach(i => answer[i] = accumulation);
            List<char> keys = group.Keys.ToList();
            List<int> val = group.Values.ToList();
            List<int> diff = new();
            if (val.Count > 1)
            {
                diff = val.Select((_, index) => Math.Abs(val.ToArray()[0 ..(index + 1)].Sum() -
val.ToArray()[(index + 1)..].Sum())).ToList().GetRange(0, val.Count - 1);
                var indexOfMin = diff.IndexOf(diff.Min());
                Dictionary<char, int> leftParameter = new();
                Dictionary<char, int> rightParameter = new();
                for (int i = 0; i < indexOfMin + 1; i++)
                {
                    leftParameter[keys[i]] = val[i];
                }
                for (int i = indexOfMin + 1; i < val.Count(); i++)
                {
                    rightParameter[keys[i]] = val[i];
                }
                if (leftParameter.Values.Count > 0)
                {
                    RecursiveMethod(group: leftParameter, accumulation: accumulation + "0");
                }
                if (rightParameter.Values.Count > 0)
                {
                    RecursiveMethod(group: rightParameter, accumulation: accumulation + "1");
                }
            }

        }

        static void Main(string[] args)
        {

            if (args.Length == 0)
            {
                Console.ForegroundColor = ConsoleColor.DarkYellow;
                Console.WriteLine("Empty string argument");
                Console.ForegroundColor = ConsoleColor.Blue;
                Console.WriteLine($"example of a command: \nshannon hello");
                Console.ResetColor();
            }
            else
            {
                var answer = JsonSerializer.Serialize(CreateTree(args[0]));
                Console.ForegroundColor = ConsoleColor.DarkYellow;
                Console.WriteLine(answer);
                Console.ResetColor();
            }
        }
    }
}
```

https://github.com/sammychinedu2ky/shannon-fano-encoding/blob/master/Program.cs

VLC: Shannon-Fano in “Python“

```python
# Python3 program for Shannon Fano Algorithm
# declare structure node
class node :
    def __init__(self) -> None:
        # for storing symbol
        self.sym=''
        # for storing probability or frequency
        self.pro=0.0
        self.arr=[0]*20
        self.top=0
p=[node() for _ in range(20)]

# function to find shannon code
def shannon(l, h, p):
    pack1 = 0; pack2 = 0; diff1 = 0; diff2 = 0
    if ((l + 1) == h or l == h or l > h) :
        if (l == h or l > h):
            return
        p[h].top+=1
        p[h].arr[(p[h].top)] = 0
        p[l].top+=1
        p[l].arr[(p[l].top)] = 1

        return

    else :
        for i in range(l,h):
            pack1 = pack1 + p[i].pro
        pack2 = pack2 + p[h].pro
        diff1 = pack1 - pack2
        if (diff1 < 0):
            diff1 = diff1 * -1
        j = 2
        while (j != h - l + 1) :
            k = h - j
            pack1 = pack2 = 0
            for i in range(l, k+1):
                pack1 = pack1 + p[i].pro
            for i in range(h,k,-1):
                pack2 = pack2 + p[i].pro
            diff2 = pack1 - pack2
            if (diff2 < 0):
                diff2 = diff2 * -1
            if (diff2 >= diff1):
                break
            diff1 = diff2
            j+=1

        k+=1
        for i in range(l,k+1):
            p[i].top+=1
            p[i].arr[(p[i].top)] = 1

        for i in range(k + 1,h+1):
            p[i].top+=1
            p[i].arr[(p[i].top)] = 0


        # Invoke shannon function
        shannon(l, k, p)
        shannon(k + 1, h, p)
```

```python
# Function to sort the symbols
# based on their probability or frequency
def sortByProbability(n, p):
    temp=node()
    for j in range(1,n) :
        for i in range(n - 1) :
            if ((p[i].pro) > (p[i + 1].pro)) :
                temp.pro = p[i].pro
                temp.sym = p[i].sym

                p[i].pro = p[i + 1].pro
                p[i].sym = p[i + 1].sym

                p[i + 1].pro = temp.pro
                p[i + 1].sym = temp.sym


# function to display shannon codes
def display(n, p):
    print("\n\n\n\tSymbol\tProbability\tCode",end='')
    for i in range(n - 1,-1,-1):
        print("\n\t", p[i].sym, "\t\t", p[i].pro,"\t",end='')
        for j in range(p[i].top+1):
            print(p[i].arr[j],end='')
```

https://www.geeksforgeeks.org/shannon-fano-algorithm-for-data-compression/

```python
# Driver code
if __name__ == '__main__':
    total = 0

    # Input number of symbols
    print("Enter number of symbols\t: ",end='')
    n = 5
    print(n)
    i=0
    # Input symbols
    for i in range(n):
        print("Enter symbol", i + 1," : ",end="")
        ch = chr(65 + i)
        print(ch)

        # Insert the symbol to node
        p[i].sym += ch


    # Input probability of symbols
    x = [0.22, 0.28, 0.15, 0.30, 0.05]
    for i in range(n):
        print("\nEnter probability of", p[i].sym, ": ",end="")
        print(x[i])

        # Insert the value to node
        p[i].pro = x[i]
        total = total + p[i].pro

        # checking max probability
        if (total > 1) :
            print("Invalid. Enter new values")
            total = total - p[i].pro
            i-=1


    i+=1
    p[i].pro = 1 - total
    # Sorting the symbols based on
    # their probability or frequency
    sortByProbability(n, p)

    for i in range(n):
        p[i].top = -1

    # Find the shannon code
    shannon(0, n - 1, p)

    # Display the codes
    display(n, p)
```

1.5 Huffman Coding

Huffman Coding

Huffman Coding Algorithm — a bottom-up approach

1. Initialization: Pu all symbols on a list sorted according to their frequency counts.
2. Repeat until the list has only one symbol left:
    1. From the list pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node
    2. Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
    3. Delete the children from the list.
3. Assign a codeword for each leaf based on the path from the root.

Huffman Coding in Data Compression

Introduction:

- Huffman coding is a popular data compression technique that assigns variable-length codes to characters based on their frequencies.

Application: Text File Compression

- Huffman coding is widely used in compressing text files such as documents, eBooks, and source code.
- It replaces frequently occurring characters with shorter codes and less frequent characters with longer codes.

Efficiency in Text Compression:

- Huffman coding excels in compressing text files with repetitive patterns.
- It achieves higher compression ratios compared to fixed-length encoding methods.

Benefits:

- Reduced storage space: Compressed files require less storage space.
- Faster transmission: Compressed data can be transmitted faster over networks.
- Efficient archiving: Huffman-coded files are suitable for archiving and data backup.

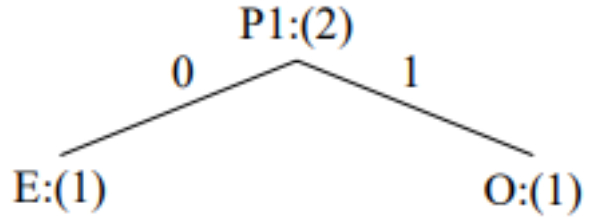
Drawbacks:

- Huffman coding requires building and transmitting the codebook along with the compressed data, adding some overhead.
- Decompression requires access to the original Huffman tree or codebook.

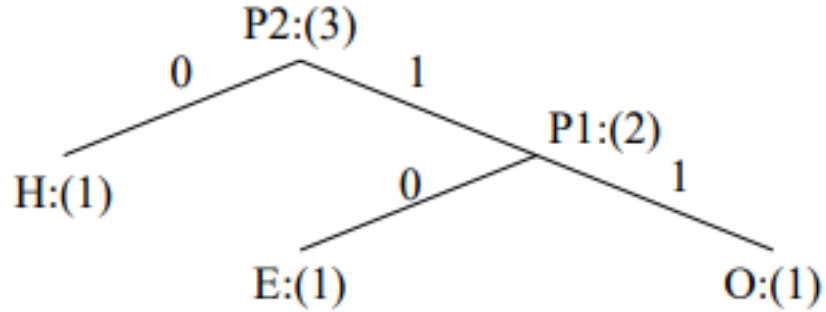
Real-world examples:

- ZIP and GZIP file formats use Huffman coding in their compression algorithms.
- Web servers use Huffman coding to compress HTML, CSS, and JavaScript files for faster webpage loading.

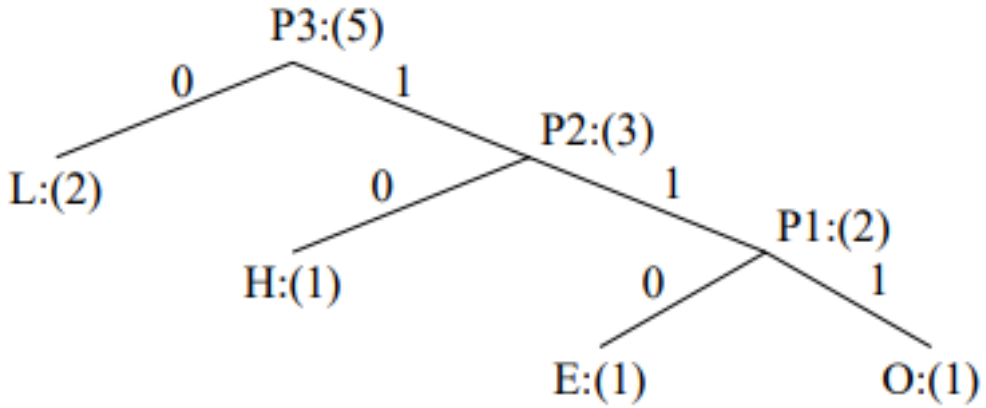
Coding Tree for “HELLO” using the Huffman Algorithm



(a)



(b)



(c)

Huffman Coding

In Figure of slide 16, new symbols P1, P2, P3 are created to refer to the parent nodes in the Huffman coding tree. The contents in the list are illustrated below:

After initialization: L H E O

After iteration (a): L P1 H

After iteration (b): L P2

After iteration (c): P3

Properties of Huffman Coding

1. **Unique Prefix Property:** No Huffman code is a prefix of any other Huffman code - precludes any ambiguity in decoding.
2. **Optimality:** minimum redundancy code - proved optimal for a given data model (i.e., a given, accurate, probability distribution):
   - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
   - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
   - The average code length for an information source S is strictly less than η + 1. Combined with the Figure on slide 16, we have:

$$\bar{l} < \eta + 1$$

Extended Huffman Coding

- Motivation: All codewords in Huffman coding have integer bit lengths. It is wasteful when p_i is very large and hence $\log_2 \frac{1}{p_i}$ is close to 0

  Why not group several symbols together and assign a single codeword to the group as a whole?

- Extended Alphabet: For alphabet S = { s 1, s 2,...,s n }, if k symbols are grouped together, then the extended alphabet is:

$$S^{(k)} = \{\overbrace{s_1 s_1 \dots s_1}^{k \ symbols}, s_1 s_1 \dots s_2, \dots, s_1 s_1 \dots s_n, s_1 s_1 \dots s_2 s_1, \dots, s_n s_n \dots s_n\}.$$

— the size of the new alphabet $S^{(k)}$ is n^k.

Extended Huffman Coding

- It can be proven that the average # of bits for each symbol is:

$$\eta \leq \bar{l} < \eta + \frac{1}{k}$$

  An improvement over the original Huffman coding, but not much.

- Problem: If k is relatively large (e.g., k ≥ 3), then for most practical applications where n >> 1, n^k implies a huge symbol table — impractical.

Adaptive Huffman Coding

- **Adaptive Huffman Coding:** statistics are gathered and updated dynamically as the data stream arrives.

```
ENCODER
-------

Initial_code();
while not EOF
{
    get(c);
    encode(c);
    update_tree(c);
}
```

```
DECODER
-------

Initial_code();
while not EOF
{
    decode(c);
    output(c);
    update_tree(c);
}
```

Adaptive Huffman Coding

- Initial code assigns symbols with some initially agreed upon codes, without any prior knowledge of the frequency counts.
- update tree constructs an Adaptive Huffman tree.

  It basically does two things:

  a) Increments the frequency counts for the symbols (including any new ones)
  b) updates the configuration of the tree.

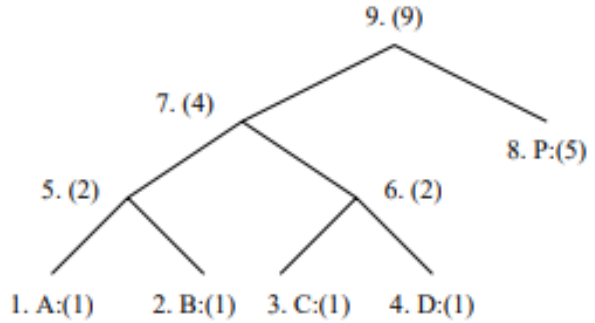
- The encoder and decoder must use exactly the same *initial_code* and *update_tree* routines.

Notes on Adaptive Huffman Tree Updating

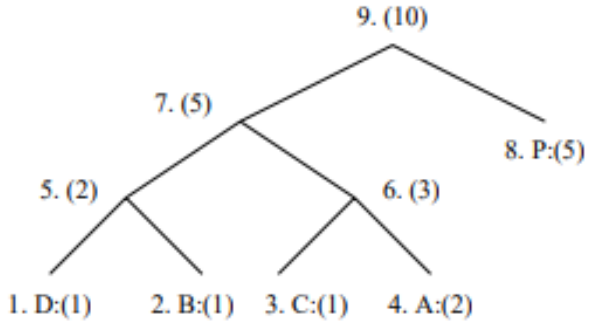
- Nodes are numbered in order from left to right, bottom to top. The numbers in parentheses indicate the count.
- The tree must always maintain its sibling property, i.e., all nodes (internal and leaf) are arranged in the order of increasing counts.

  If the sibling property is about to be violated, a swap procedure is invoked to update the tree by rearranging the nodes.
- When a swap is necessary, the farthest node with count N is swapped with the node whose count has just been increased to N + 1.

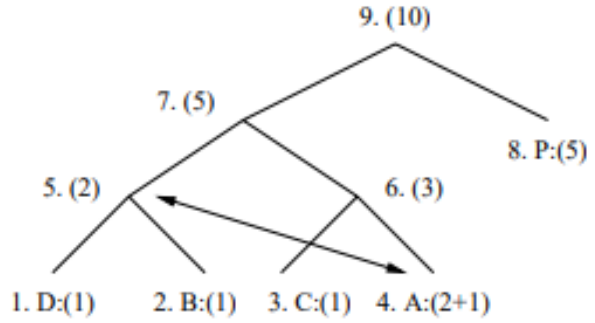
Node Swapping for Updating an Adaptive Huffman Tree



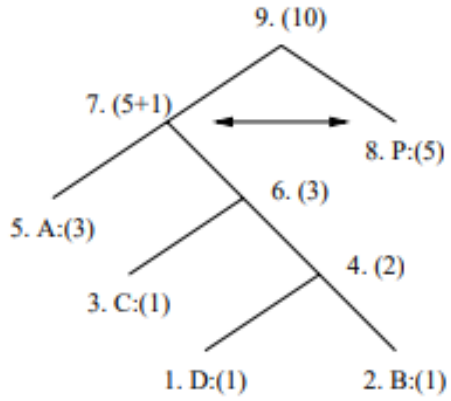
(a) A Huffman tree



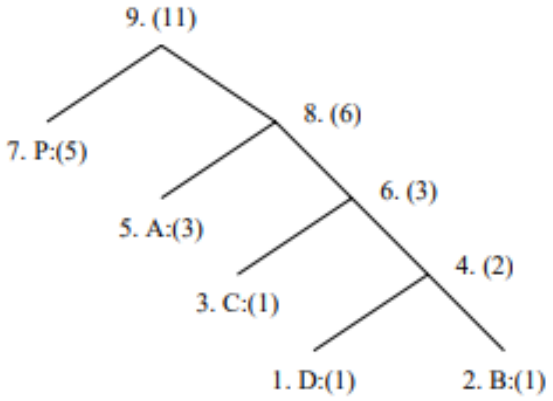
(b) Receiving 2nd 'A' triggered a swap



(c−1) A swap is needed after receiving 3rd 'A'



(c−2) Another swap is needed



(c−3) The Huffman tree after receiving 3rd 'A'

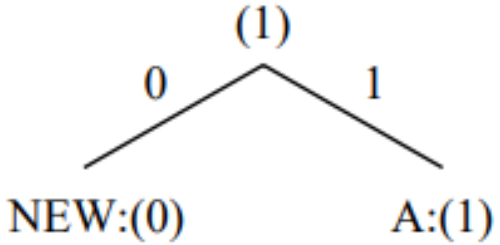
Another Example: Adaptive Huffman Coding

- This is to clearly illustrate more implementation details. We show exactly what bits are sent, as opposed to simply stating how the tree is updated.
- An additional rule: if any character/symbol is to be sent the first time, it must be preceded by a special symbol, NEW. The initial code for NEW is 0. The count for NEW is always kept as 0 (the count is never increased); hence it is always denoted as NEW:(0) in Figure on slide 27

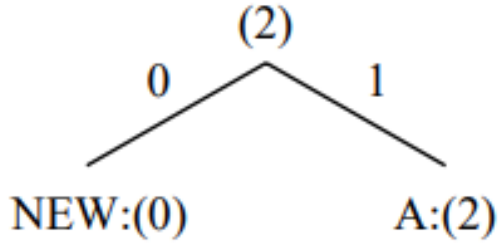
Initial code assignment for AADCCDD using adaptive Huffman coding.

Initial	*Code*
NEW:	0
A:	00001
B:	00010
C:	00011
D:	00100
.	.
.	.
.	.

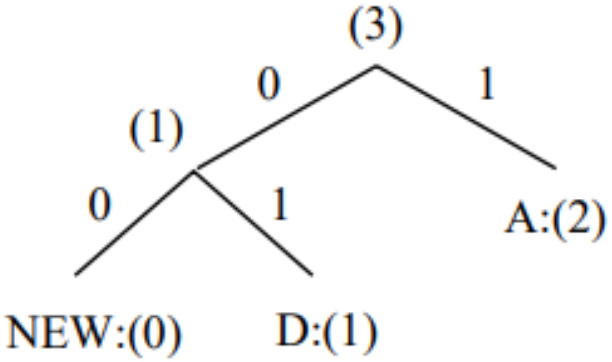
Adaptive Huffman tree for AADCCDD.



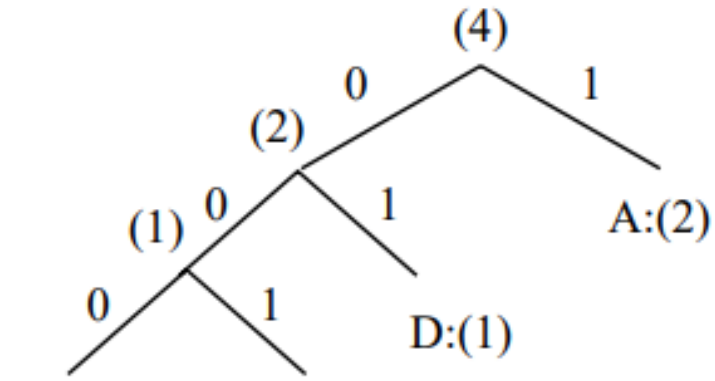
"A"



"AA"

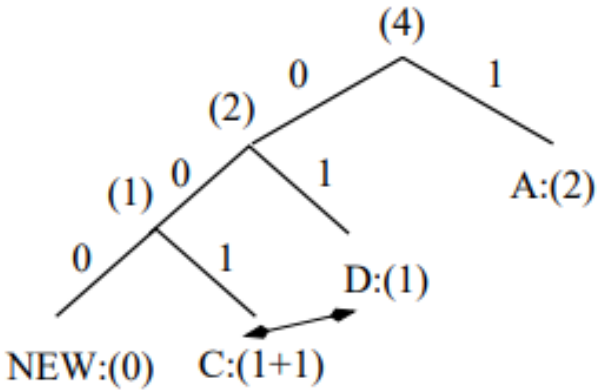


"AAD"

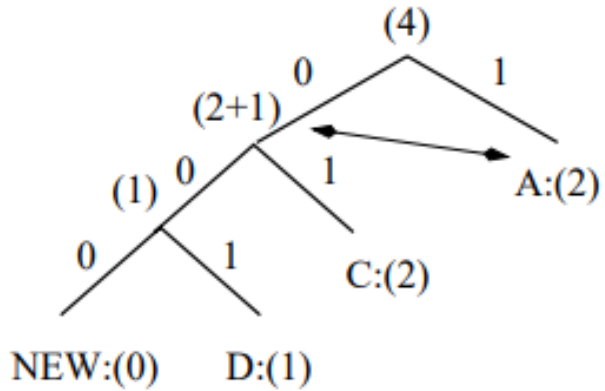


"AADC"

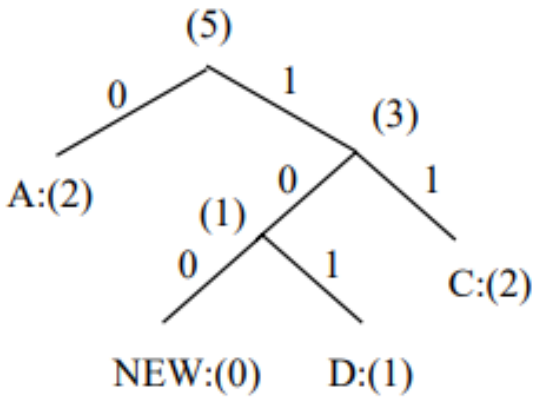
(cont'd) Adaptive Huffman tree for AADCCD



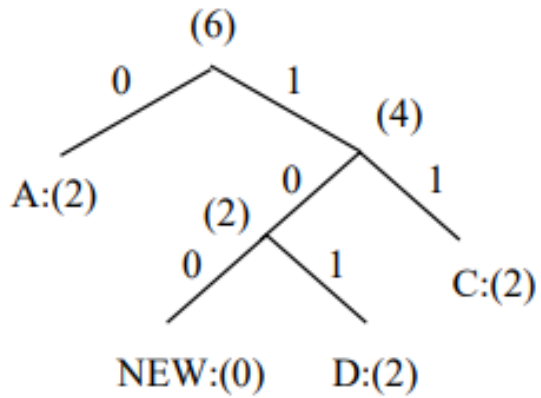
"AADCC" Step 1



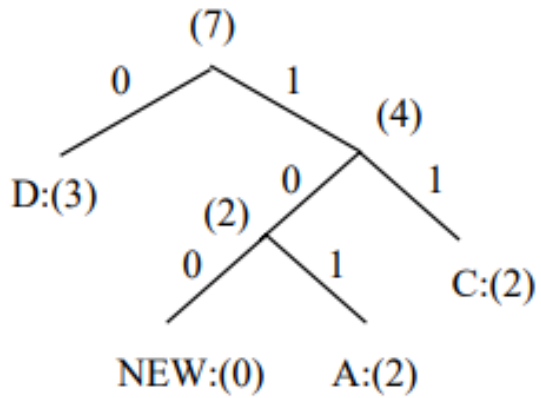
"AADCC" Step 2



"AADCC" Step 3



"AADCCD"



"AADCCDD"

A sequence of symbols and codes sent to the decoder

Symbol	NEW	A	A	NEW	D	NEW	C	C	D	D
Code	0	00001	1	0	00100	00	00011	001	101	101

- It is important to emphasize that the code for a particular symbol changes during the adaptive Huffman coding process.

  For example, after AADCCDD, when the character D overtakes A as the most frequent symbol, its code changes from 101 to 0.
- The “Squeeze Page” on this book’s web site provides a Java applet for adaptive Huffman coding.

VLC: Shannon-Fano in “C#“

```
private const int MAX_TREE_NODES = 511;

public class BitStream
{
    public byte[] BytePointer;
    public uint BitPosition;
    public uint Index;
}

public struct Symbol
{
    public int Sym;
    public uint Count;
    public uint Code;
    public uint Bits;
}

public class EncodeNode
{
    public EncodeNode ChildA;
    public EncodeNode ChildB;
    public int Count;
    public int Symbol;
}

private static void initBitstream(ref BitStream stream, byte[] buffer)
{
    stream.BytePointer = buffer;
    stream.BitPosition = 0;
}

private static void writeBits(ref BitStream stream, uint x, uint bits)
{
    byte[] buffer = stream.BytePointer;
    uint bit = stream.BitPosition;
    uint mask = (uint)(1 << (int)(bits - 1));

    for (uint count = 0; count < bits; ++count)
    {
        buffer[stream.Index] = (byte)((buffer[stream.Index] & (0xff ^ (1 << (int)(7 - bit)))) +
((Convert.ToBoolean(x & mask) ? 1 : 0) << (int)(7 - bit)));
        x <<= 1;
        bit = (bit + 1) & 7;

        if (!Convert.ToBoolean(bit))
        {
            ++stream.Index;
        }
    }

    stream.BytePointer = buffer;
    stream.BitPosition = bit;
}

private static void histogram(byte[] input, Symbol[] sym, uint size)
{
    int i;
    int index = 0;

    for (i = 0; i < 256; ++i)
    {
        sym[i].Sym = i;
        sym[i].Count = 0;
        sym[i].Code = 0;
        sym[i].Bits = 0;
    }

    for (i = (int)size; Convert.ToBoolean(i); --i, ++index)
    {
        sym[input[index]].Count++;
    }
}
```

```
private static void storeTree(ref EncodeNode node, Symbol[] sym, ref BitStream stream, uint code, uint
bits)
{
    uint symbolIndex;

    if (node.Symbol >= 0)
    {
        writeBits(ref stream, 1, 1);
        writeBits(ref stream, (uint)node.Symbol, 8);

        for (symbolIndex = 0; symbolIndex < 256; ++symbolIndex)
        {
            if (sym[symbolIndex].Sym == node.Symbol)
                break;
        }

        sym[symbolIndex].Code = code;
        sym[symbolIndex].Bits = bits;
        return;
    }
    else
    {
        writeBits(ref stream, 0, 1);
    }

    storeTree(ref node.ChildA, sym, ref stream, (code << 1) + 0, bits + 1);
    storeTree(ref node.ChildB, sym, ref stream, (code << 1) + 1, bits + 1);
}

private static void makeTree(Symbol[] sym, ref BitStream stream)
{
    EncodeNode[] nodes = new EncodeNode[MAX_TREE_NODES];

    for (int counter = 0; counter < nodes.Length; ++counter)
    {
        nodes[counter] = new EncodeNode();
    }

    EncodeNode node1, node2, root;
    uint i, numSymbols = 0, nodesLeft, nextIndex;

    for (i = 0; i < 256; ++i)
    {
        if (sym[i].Count > 0)
        {
            nodes[numSymbols].Symbol = sym[i].Sym;
            nodes[numSymbols].Count = (int)sym[i].Count;
            nodes[numSymbols].ChildA = null;
            nodes[numSymbols].ChildB = null;
            ++numSymbols;
        }
    }

    root = null;
    nodesLeft = numSymbols;
    nextIndex = numSymbols;

    while (nodesLeft > 1)
    {
        node1 = null;
        node2 = null;

        for (i = 0; i < nextIndex; ++i)
        {
            if (nodes[i].Count > 0)
            {
                if (node1 == null  || (nodes[i].Count <= node1.Count))
                {
                    node2 = node1;
                    node1 = nodes[i];
                }
                else if (node2 == null || (nodes[i].Count <= node2.Count))
                {
                    node2 = nodes[i];
                }
            }
        }

        root = nodes[nextIndex];
        root.ChildA = node1;
        root.ChildB = node2;
        root.Count = node1.Count + node2.Count;
        root.Symbol = -1;
        node1.Count = 0;
        node2.Count = 0;
        ++nextIndex;
        --nodesLeft;
    }

    if (root != null)
    {
        storeTree(ref root, sym, ref stream, 0, 0);
    }
    else
    {
        root = nodes[0];
        storeTree(ref root, sym, ref stream, 0, 1);
    }
}
```

```
1  public static int Compress(byte[] input, byte[] output, uint inputSize)
2  {
3      Symbol[] sym = new Symbol[256];
4      Symbol temp;
5      BitStream stream = new BitStream();
6      uint i, totalBytes, swaps, symbol;
7
8      if (inputSize < 1)
9          return 0;
10
11     initBitstream(ref stream, output);
12     histogram(input, sym, inputSize);
13     makeTree(sym, ref stream);
14
15     do
16     {
17         swaps = 0;
18
19         for (i = 0; i < 255; ++i)
20         {
21             if (sym[i].Sym > sym[i + 1].Sym)
22             {
23                 temp = sym[i];
24                 sym[i] = sym[i + 1];
25                 sym[i + 1] = temp;
26                 swaps = 1;
27             }
28         }
29     } while (Convert.ToBoolean(swaps));
30
31     for (i = 0; i < inputSize; ++i)
32     {
33         symbol = input[i];
34         writeBits(ref stream, sym[symbol].Code, sym[symbol].Bits);
35     }
36
37     totalBytes = stream.Index;
38
39     if (stream.BitPosition > 0)
40     {
41         ++totalBytes;
42     }
43
44     return (int)totalBytes;
45 }
```

https://www.programmingalgorithms.com/algorithm/huffman-compress/

VLC: Shannon-Fano in “C++“

```c
// Huffman Coding in C

#include <stdio.h>
#include <stdlib.h>

#define MAX_TREE_HT 50

struct MinHNode {
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
};

struct MinHeap {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};

// Create nodes
struct MinHNode *newNode(char item, unsigned freq) {
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
  temp->item = item;
  temp->freq = freq;

  return temp;
}

// Create min heap
struct MinHeap *createMinH(unsigned capacity) {
  struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode *));
  return minHeap;
}

// Function to swap
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}

// Heapify
void minHeapify(struct MinHeap *minHeap, int idx) {
  int smallest = idx;
  int left = 2 * idx + 1;
  int right = 2 * idx + 2;

  if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
    smallest = left;

  if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
    smallest = right;

  if (smallest != idx) {
    swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}
```

```c
// Check if size if 1
int checkSizeOne(struct MinHeap *minHeap) {
  return (minHeap->size == 1);
}

// Extract min
struct MinHNode *extractMin(struct MinHeap *minHeap) {
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];

  --minHeap->size;
  minHeapify(minHeap, 0);

  return temp;
}

// Insertion function
void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
  ++minHeap->size;
  int i = minHeap->size - 1;

  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
  int n = minHeap->size - 1;
  int i;

  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
  struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);

  minHeap->size = size;
  buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
  struct MinHNode *left, *right, *top;
  struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);

    top = newNode('$', left->freq + right->freq);

    top->left = left;
    top->right = right;

    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}
```

```c

void printHCodes(struct MinHNode *root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
    printf("  %c   | ", root->item);
    printArray(arr, top);
  }
}

// Wrapper function
void HuffmanCodes(char item[], int freq[], int size) {
  struct MinHNode *root = buildHuffmanTree(item, freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}

// Print the array
void printArray(int arr[], int n) {
  int i;
  for (i = 0; i < n; ++i)
    printf("%d", arr[i]);

  printf("\n");
}

int main() {
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};

  int size = sizeof(arr) / sizeof(arr[0]);

  printf(" Char | Huffman code ");
  printf("\n--------------------\n");

  HuffmanCodes(arr, freq, size);
}
```

https://www.programiz.com/dsa/huffman-coding

VLC: Shannon-Fano in “Python“

```python
# Huffman Coding in python

string = 'BCAADDDCCACACAC'


# Creating tree nodes
class NodeTree(object):

    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self):
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)

# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d


# Calculating frequency
freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

nodes = freq

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])

print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

https://www.programiz.com/dsa/huffman-coding

1.6 Dictionary-Based Coding

Dictionary-Based Coding

- LZW uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together, e.g., words in English text.
- the LZW encoder and decoder build up the same dictionary dynamically while receiving the data.
- LZW places longer and longer repeated entries into a dictionary, and then emits the code for an element, rather than the string itself if the element has already been placed in the dictionary.

Introduction to LZW Compression

LZW Compression Technique

- LZW stands for Lempel-Ziv-Welch, a widely used compression technique in data and file compression.
- Developed by Abraham Lempel, Jacob Ziv, and Terry Welch, LZW is known for its efficiency in reducing data size.
- It works by replacing repetitive patterns in data with shorter codes, effectively compressing the information.

How LZW Compression Works

The Process of LZW Compression

1. Dictionary Initialization:

1. The technique begins by initializing a dictionary containing individual characters.
2. As the algorithm progresses, it dynamically adds new patterns to the dictionary.

2. Pattern Recognition:

1. LZW scans the input data, identifying recurring patterns.
2. These patterns can be strings of characters that appear frequently.

3. Code Assignment:

1. Each identified pattern is assigned a unique code that represents it in the dictionary.
2. The codes are shorter than the patterns they replace, achieving compression.

4. Output Encoding:

1. During compression, the original data is replaced with the corresponding codes from the dictionary.
2. The compressed data consists of these codes.

Benefits and Applications

Advantages and Use Cases of LZW Compression

- **Space Efficiency:** LZW achieves high compression ratios, effectively reducing the size of data or files.
- **Text and Image Compression:** LZW is suitable for compressing both text and image data, making it versatile for various applications.
- **Transmission and Storage:** Compressed data requires less storage space and can be transmitted faster, optimizing network bandwidth.
- **Archival and Communication:** LZW compression is often used for archiving large datasets or transmitting data over the internet.
- **Trade-offs:** While LZW excels in compression, its decompression process may be slower than other algorithms.

BABAABAAA

P=A
C = empty

Encoder Output		String	Table
Output Code	representing	codeword	string
66	B	256	BA

LZW compression step 1

BABAABAAA

P=B
C = empty

Encoder Output		String	Table
Output Code	representing	codeword	string
66	B	256	BA
65	A	257	AB

LZW compression step 2

BABAABAAA

P=A
C = empty

Encoder Output		String	Table
Output Code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA

LZW compression step 3

BABAABAAA

P=A
C = empty

Encoder Output		String	Table
Output Code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA

LZW compression step 4

BABAABAAA

P=A
C = A

Encoder Output		String	Table
Output Code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA
65	A	260	AA

LZW compression step 5

BABAABAAA

P=AA
C = empty

Encoder Output		String	Table
Output Code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA
65	A	260	AA
260	AA		

LZW compression step 6

LZW Compression

```
BEGIN
   s = next input character;
   while not EOF
       { c = next input character;

         if s + c exists in the dictionary
            s = s + c;
         else
           { output the code for s;
             add string s + c to the dictionary with a new code;
             s = c;
           }
       }
   output the code for s;
END
```

Example: LZW compression for string “ABABBABCABABBA”

- Let’s start with a very simple dictionary (also referred to as a “string table”), initially containing only 3 characters, with codes as follows:

code	string
1	A
2	B
3	C

- Now if the input string is “ABABBABCABABBA”, the LZW compression algorithm works as follows:

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	A	4	10	ABA
A	B			
AB	B			
ABB	A	6	11	ABBA
A	EOF	1		

- The output codes are: 1 2 4 5 2 3 4 6 1. Instead of sending 14 characters, only 9 codes need to be sent (compression ratio = 14/9 = 1.56).

LZW Decompression

```
BEGIN
    s = NIL;
    while not EOF
        {
          k = next input code;
          entry = dictionary entry for k;
          output entry;
          if (s != NIL)
             add string s + entry[0] to dictionary with a new code;
          s = entry;
        }
END
```

LZW decompression for the string “ABABBABCABABBA”.

Input codes to the decoder are 1 2 4 5 2 3 4 6 1.

The initial string table is identical to what is used by the encoder.

The LZW decompression algorithm then works as follows:

s	k	entry/output	code	string
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	4	AB	9	CA
AB	6	ABB	10	ABA
ABB	1	A	11	ABBA
A	EOF			

Apparently, the output string is “ABABBABCABABBA”, a truly lossless result!

Dictionary Coding (LZW) in “C#“

```
using System;
using System.Collections.Generic;
using System.Text;

namespace LZW
{
    public class Program
    {
        public static void Main(string[] args)
        {
            List<int> compressed = Compress("TOBEORNOTTOBEORTOBEORNOT");
            Console.WriteLine(string.Join(", ", compressed));
            string decompressed = Decompress(compressed);
            Console.WriteLine(decompressed);
        }
```

```
        public static List<int> Compress(string uncompressed)
        {
            // build the dictionary
            Dictionary<string, int> dictionary = new Dictionary<string, int>
();          for (int i = 0; i < 256; i++)
                dictionary.Add(((char)i).ToString(), i);

            string w = string.Empty;
            List<int> compressed = new List<int>();

            foreach (char c in uncompressed)
            {
                string wc = w + c;
                if (dictionary.ContainsKey(wc))
                {
                    w = wc;
                }
                else
                {
                    // write w to output
                    compressed.Add(dictionary[w]);
                    // wc is a new sequence; add it to the dictionary
                    dictionary.Add(wc, dictionary.Count);
                    w = c.ToString();
                }
            }

            // write remaining output if necessary
            if (!string.IsNullOrEmpty(w))
                compressed.Add(dictionary[w]);

            return compressed;
        }

        public static string Decompress(List<int> compressed)
        {
            // build the dictionary
            Dictionary<int, string> dictionary = new Dictionary<int, string>
();          for (int i = 0; i < 256; i++)
                dictionary.Add(i, ((char)i).ToString());

            string w = dictionary[compressed[0]];
            compressed.RemoveAt(0);
            StringBuilder decompressed = new StringBuilder(w);

            foreach (int k in compressed)
            {
                string entry = null;
                if (dictionary.ContainsKey(k))
                    entry = dictionary[k];
                else if (k == dictionary.Count)
                    entry = w + w[0];

                decompressed.Append(entry);

                // new sequence; add it to the dictionary
                dictionary.Add(dictionary.Count, w + entry[0]);

                w = entry;
            }

            return decompressed.ToString();
        }
    }
}
```

https://rosettacode.org/wiki/LZW_compression#

Dictionary Coding (LZW) in “C++“

```cpp
#include <string>
#include <map>

// Compress a string to a list of output symbols.
// The result will be written to the output iterator
// starting at "result"; the final iterator is returned.
template <typename Iterator>
Iterator compress(const std::string &uncompressed, Iterator result)
{ // Build the dictionary.
  int dictSize = 256;
  std::map<std::string,int> dictionary;
  for (int i = 0; i < 256; i++)
    dictionary[std::string(1, i)] = i;

  std::string w;
  for (std::string::const_iterator it = uncompressed.begin();
       it != uncompressed.end(); ++it) {
    char c = *it;
    std::string wc = w + c;
    if (dictionary.count(wc))
      w = wc;
    else {
      *result++ = dictionary[w];
      // Add wc to the dictionary.
      dictionary[wc] = dictSize++;
      w = std::string(1, c);
    }
  }

  // Output the code for w.
  if (!w.empty())
    *result++ = dictionary[w];
  return result;
}
```

```cpp
// Decompress a list of output ks to a string.
// "begin" and "end" must form a valid range of ints
template <typename Iterator>
std::string decompress(Iterator begin, Iterator end) {
  // Build the dictionary.
  int dictSize = 256;
  std::map<int,std::string> dictionary;
  for (int i = 0; i < 256; i++)
    dictionary[i] = std::string(1, i);

  std::string w(1, *begin++);
  std::string result = w;
  std::string entry;
  for ( ; begin != end; begin++) {
    int k = *begin;
    if (dictionary.count(k))
      entry = dictionary[k];
    else if (k == dictSize)
      entry = w + w[0];
    else
      throw "Bad compressed k";

    result += entry;

    // Add w+entry[0] to the dictionary.
    dictionary[dictSize++] = w + entry[0];

    w = entry;
  }
  return result;
}

#include <iostream>
#include <iterator>
#include <vector>

int main() {
  std::vector<int> compressed;
  compress("TOBEORNOTTOBEORTOBEORNOT", std::back_inserter(compressed));
  copy(compressed.begin(), compressed.end(), std::ostream_iterator<int>(std::cout, ",
");}std::cout << std::endl;
  std::string decompressed = decompress(compressed.begin(), compressed.end());
  std::cout << decompressed << std::endl;

  return 0;
}
```

https://rosettacode.org/wiki/LZW_compression#

Dictionary Coding (LZW) in “C#“

```python
def compress(uncompressed):
    """Compress a string to a list of output symbols."""
    # Build the dictionary.
    dict_size = 256
    dictionary = dict((chr(i), i) for i in range(dict_size))
    # in Python 3: dictionary = {chr(i): i for i in range(dict_size)}

    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            # Add wc to the dictionary.
            dictionary[wc] = dict_size
            dict_size += 1
            w = c

    # Output the code for w.
    if w:
        result.append(dictionary[w])
    return result
```

```python
def decompress(compressed):
    """Decompress a list of output ks to a string."""
    from io import StringIO

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((i, chr(i)) for i in range(dict_size))
    # in Python 3: dictionary = {i: chr(i) for i in range(dict_size)}

    # use StringIO, otherwise this becomes O(N^2)
    # due to string concatenation in a loop
    result = StringIO()
    w = chr(compressed.pop(0))
    result.write(w)
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError('Bad compressed k: %s' % k)
        result.write(entry)

        # Add w+entry[0] to the dictionary.
        dictionary[dict_size] = w + entry[0]
        dict_size += 1

        w = entry
    return result.getvalue()

# How to use:
compressed = compress('TOBEORNOTTOBEORTOBEORNOT')
print (compressed)
decompressed = decompress(compressed)
print (decompressed)
```

https://rosettacode.org/wiki/LZW_compression#

1.7 ATRAC

ATRAC Overview

ATRAC (Adaptive Transform Acoustic Coding):

- Developed by Sony in the early 1990s.
- Audio compression algorithm for efficient data storage and transmission.

Key Features:

- Adaptive Coding: Adjusts compression according to audio content complexity.
- Transform Coding: Converts audio signals into frequency domain for compression.

Versions:

- ATRAC1: First version, used in MiniDisc players.
- ATRAC3: Improved compression and sound quality.
- ATRAC3plus: Further enhancement with better compression ratios.
- ATRAC Advanced Lossless (ATRAC AL): Lossless version for archival purposes.

ATRAC Technical Details and Decoding

Technical Details:

- Used in various Sony devices: MiniDisc players, Walkman, PlayStation, etc.
- Compression ratios range from 5:1 to 10:1.
- ATRAC3plus achieved better sound quality at lower bitrates.

Advantages:

- Efficient Compression: High compression ratios with good audio quality.
- Suitable for Portable Players: Ideal for devices with limited storage.

Challenges:

- Proprietary Format: Limited cross-platform compatibility.
- Quality vs. Compression: Balancing audio quality and compression ratio.

Legacy and Impact:

- Used in the MiniDisc era and some Sony devices.
- Evolved into ATRAC Advanced Lossless for archival use.

1.8 WMA

Windows Media Losless (WMA) Overwiew

Windows Media Audio Lossless (WMA Lossless):

- Lossless audio codec by Microsoft.
- Competes with other lossless codecs like FLAC, Apple Lossless, etc.
- Designed for archival purposes, and maintains audio quality during compression.

Features:

- Supports 5.1 surround sound for an immersive audio experience.
- Supports up to 24-bit/96 kHz lossless audio.
- Compression ratios range between 1.7:1 and 3:1 for music.

Hardware Support:

- Available on various devices including Cowon A3, Sony Walkman, Zune, Xbox 360, etc.
- Used for online music distribution by select stores.

Technical Details and Decoding

Technical Details:

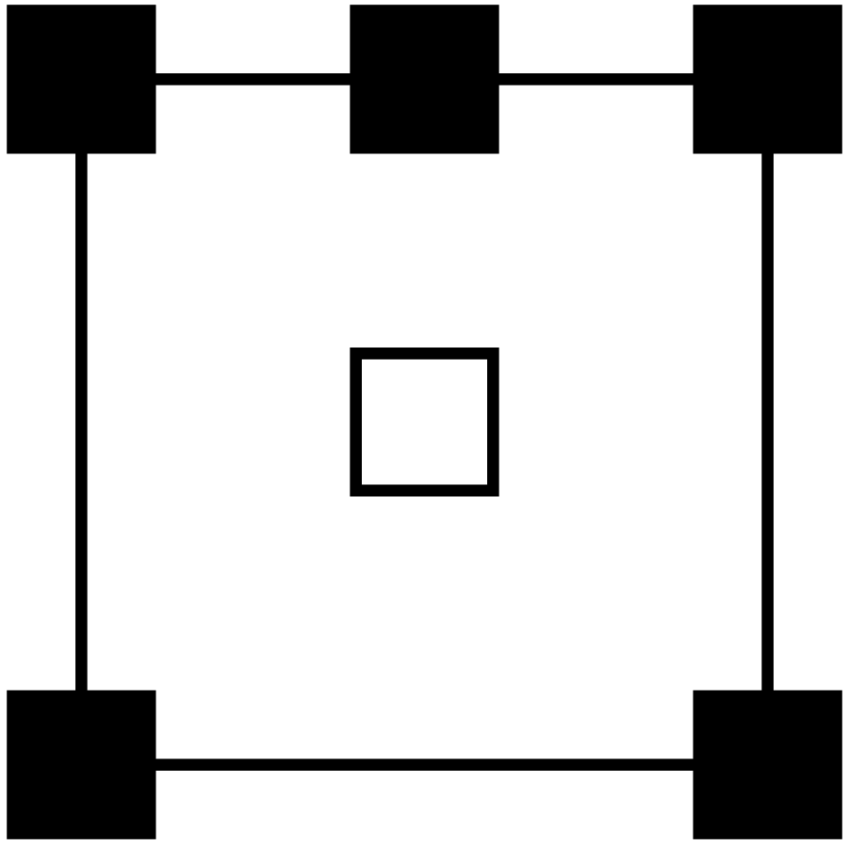
- WMA Lossless compresses audio signals without loss of quality using VBR.
- Decompressed signal is an exact replica of the original.
- Supports up to 96 kHz, 24-bit audio for 5.1 channel surround.

Decoding and Compatibility:

- Hardware and software support available for various devices.
- Libav and ffmpeg projects provide open-source WMA Lossless decoders.
- FFMPEG currently supports decoding of 16-bit WMA files.

Advantages:

- Maintains audio quality, useful for archival purposes.
- Supports surround sound and high-resolution audio.
- Wide compatibility across devices.



surround sound, the maximum channel configuration for Windows Media Audio Lossless.

1.9 BWT

Burrows-Wheeler Transform Overview

Burrows-Wheeler Transform (BWT):

- Data compression algorithm named after its inventors Michael Burrows and David Wheeler.
- Used in various data compression applications, including file compression and data transmission.

Algorithm Steps:

1. Rearrange: Rearranges characters in the input data to create a matrix.
2. Sorting: Sorts the rows of the matrix lexicographically.
3. Extraction: Extracts the last column of the sorted matrix.

Properties:

- Lossless Compression: BWT retains original data integrity after compression.
- Preprocessing Step: Often used as a preprocessing step before applying further compression methods.

BWT and Compression

Compression Process:

1. Original data is transformed using the BWT.
2. Transformed data often exhibits runs of identical characters.
3. Run-Length Encoding (RLE) is applied to replace runs with a single character and a count.

Advantages:

- Effective for repetitive data: BWT reduces repetition, enhancing compression efficiency.
- Suitability for Different Data Types: Works well with text, DNA sequences, and more.

Limitations:

- No Bit-Level Compression: BWT doesn't perform bit-level compression.
- Decoding: Requires additional methods to reconstruct the original data.

Applications:

- Used in file compression algorithms like BZIP2.
- Commonly used in data transmission for efficient bandwidth usage**.**

1.10 Rendundancy

Introduction to Redundancy

The Concept of Redundancy

- Redundancy in technology refers to the presence of backup systems or comparable components alongside the primary system.
- These additional components may not be active at all times but significantly enhance system reliability and availability.
- Examples of redundancy strategies include backups, data safety measures, business continuity plans, and recovery solutions.
- Redundancy is essential to mitigate the risk of a "Single Point of Failure," where the failure of one element leads to the system's complete breakdown.

How Redundancy Works

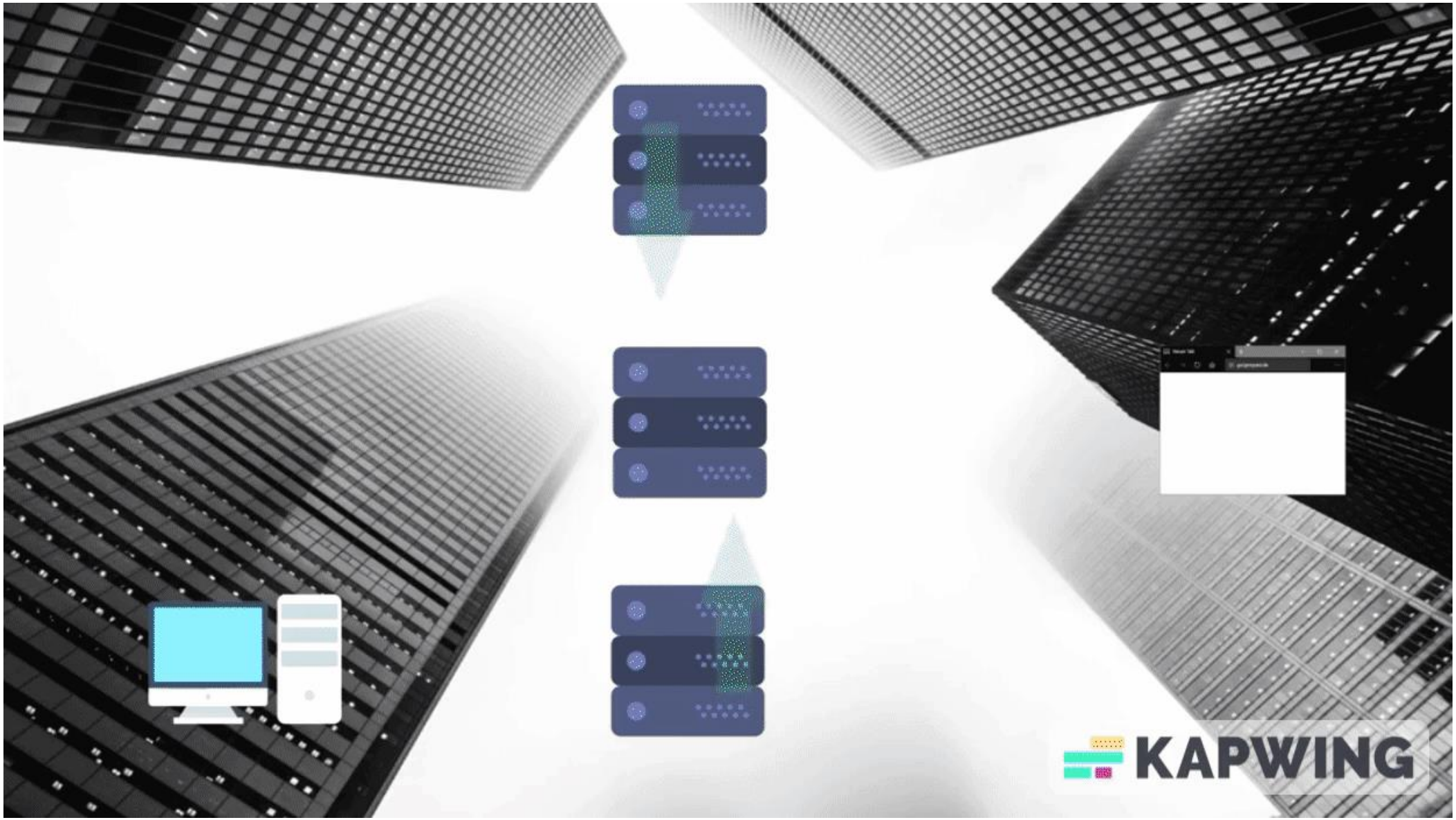
Ensuring Continuous Availability

- Redundancy works by setting up duplicate systems that mirror the primary system's functionality.
- These duplicates are synchronized with the primary system, ensuring that data and configurations are up-to-date.
- In case of a failure or disruption in the primary system, the redundant systems automatically take over.
- Users are often unaware of these transitions due to the seamless and rapid nature of redundancy implementation.
- Redundant systems ensure uninterrupted service and data access even during hardware failures, cyberattacks, or maintenance

Real-world Application of Redundancy

Applying Redundancy in Technology

- Redundancy finds applications in various technological contexts, including web servers, networking, and data storage.
- In the case of web servers, redundant systems ensure continuous online presence even during cyberattacks or system failures.
- Networking systems use redundancy to maintain connectivity in case of router or link failures.
- Redundant data storage systems prevent data loss by replicating information across multiple drives or locations.
- Embracing redundancy is crucial to guaranteeing system availability, performance, and data integrity in the face of unpredictable events.


KAPWING