Data Compression

Department of Computer Engineering
Razi University

Dr. Abdolhossein Fathi

References and Evaluation

- **Textbook:**
  - Khalid Sayood, "Introduction to Data Compression", Fifth Edition, Morgan Kaufmann Publishers, 2017.
  - Ida Mengyi Pu, "Fundamental Data Compression", First Edition, Oxford, 2006.
  - David Salomon and Giovanni Motta, "Handbook of Data Compression", Fifth Edition, Springer, 2010.

- **Score Details:**
  - Presentation 15%
  - Implementation Project 15%
  - Final Exam 70%

Contents

- Basic Concepts in the Information Theory
- Basic Compression Methods
- Statistical Based Compression Methods
- Dictionary Based Compression Methods
- Transform Based Compression Methods
- Image Compression
- Audio Compression
- Video Compression

- Your presentation is explanation of a new methods in the medical image compression (each of yours one new paper last two years ago).

- Your project is analyzing and enhancement of the selected paper along with implementation of enhanced version of it.

Basic Concepts in the Information Theory

Data Compression

- ***Data compression**, in the context of computer science, is the science (and art)* **of representing information in a compact form to:**
  - **Conserve storage space**
  - **Reduce time for transmission**
    - **Faster to encode, send, then decode than to send the original**
  - **Progressive transmission**
    - **Some compression techniques allow us to send the most important bits first so we can get a low resolution version of some data before getting the high fidelity version**
  - **Reduce computation**
    - **Use less data to achieve an approximate answer**

What is Data and Information

- Data and information are not synonymous terms!
- Data is the means by which information is conveyed.
  - Analog data
    - Also called continuous data
    - Represented by real numbers (or complex numbers)
  - Digital data
    - Finite set of symbols {a1, a2, … , am}
    - All data represented as sequences (strings) in the symbol set.
    - Example: {a,b,c,d,r} abracadabra
    - Digital data can be an approximation to analog data

- Data compression aims to reduce the amount of data required to represent a given quantity of information while preserving as much information as possible.

Data ≠ Information

- **The same amount of information can be represented by various amount of data, e.g.:**

Ex1: *Your friend, Ali, will meet you at MehrAbad Airport in Tehran at 5 minutes past 6:00 pm tomorrow night*

Ex2: *Your friend will meet you at MehrAbad Airport at 5 minutes past 6:00 pm tomorrow night*

Ex3: Ali *will meet you at MehrAbad at 6:00 pm tomorrow night*

Symbolic Data

- Language Alphabets like Roman, English, Persian,… .
- ASCII - 256 symbols
- Binary - {0,1}
  - 0 and 1 are called bits
  - All digital information can be represented efficiently in binary
  - {a,b,c,d} fixed length representation
    - 2 bits per symbol

Symbol	a	b	c	d
Binary Codes	00	01	10	11

Why Data Compression is Possible

- **Properties of human perception**

- **Most data from nature has redundancy**
  - **There is more data than the actual information contained in the data.**
  - **Squeezing out the excess data amounts to compression.**
  - **However, unsqueezing is necessary to be able to figure out what the data means.**

- **Information theory is needed to understand the limits of compression and give clues on how to compress well.**

Human Perception Factors

- **Compressed version of digital audio, image, video need not represent the original information exactly.**
- **Perception sensitivities are different for different signal patterns.**
- **Human eye is less sensitive to the higher spatial frequency components than the lower frequencies (transform coding).**
- **<u>Idea:</u> discard data that is perceptually insignificant!**

Psycho Visual Redundancy

Example: quantization

256 gray levels

16 gray levels

C=8/4 = 2:1

16 gray levels

i.e., add to each pixel a small pseudo-random number prior to quantization

Types of Data Redundancy

- **Coding Redundancy**
- **Inter Data Redundancy**
  - Adjacent audio samples are similar (predictive encoding); samples corresponding to silence (silence removal).
  - In digital image, neighboring samples on a scanning line are normally similar (spatial redundancy).
  - In digital video, in addition to spatial redundancy, neighboring images in a video sequence may be similar (temporal redundancy).
- **Compression attempts to reduce one or more of these redundancy types.**

Coding Redundancy

- **Code:** a list of symbols (letters, numbers, bits etc.)
- **Code word:** a sequence of symbols used to represent a piece of information or an event (e.g., gray levels).
- **Code word length:** the number of symbols in each code word

Example: (binary code, symbols: 0,1, length: 3)

0: 000	4: 100
1: 001	5: 101
2: 010	6: 110
3: 011	7: 111

Coding Redundancy

r_k: k-th Event or Code word

$P(r_k)$: probability of r_k

$l(r_k)$: # (the number) of bits for r_k

Expected value: $$E(X) = \sum_x xP(X = x)$$

Average # of bits: $$L_{avg} = E(l(r_k)) = \sum_{k=0}^{L-1} l(r_k)P(r_k)$$

For N symbols total number of bits: NL_{avg}

Coding Redundancy

- $l(r_k)$ = **constant length**
- **Example:** for N x M image with 8 gray level (L=8)

r_k	$p_r(r_k)$	*Code 1*	$l_1(r_k)$
$r_0 = 0$	0.19	000	3
$r_1 = 1/7$	0.25	001	3
$r_2 = 2/7$	0.21	010	3
$r_3 = 3/7$	0.16	011	3
$r_4 = 4/7$	0.08	100	3
$r_5 = 5/7$	0.06	101	3
$r_6 = 6/7$	0.03	110	3
$r_7 = 1$	0.02	111	3

$$\text{Assume } l(r_k) = 3,\ L_{avg} = \sum_{k=0}^{7} 3P(r_k) = 3\sum_{k=0}^{7} P(r_k) = 3 \text{ bits}$$

Total number of bits: $3NM$

Coding Redundancy

- $l(r_k)$ = **variable length**
- **Consider the probability of the gray levels:**

Table 6.1 Variable-Length Coding Example

variable length

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

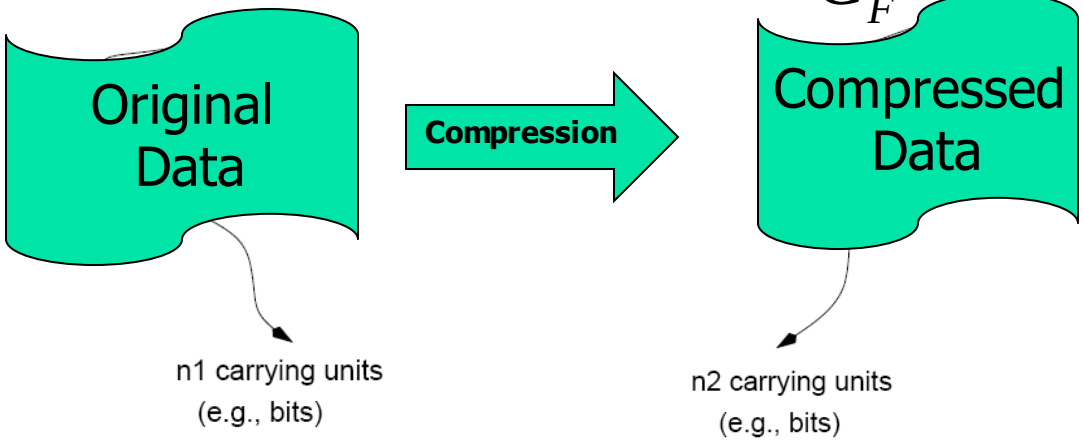
$$L_{avg} = \sum_{k=0}^{7} l(r_k)P(r_k) = 2.7 \text{ bits}$$

Total number of bits: $2.7NM$

Coding Redundancy

- **Compression Factor:** $C_F = \frac{n_1}{n_2}$
- **Relative data redundancy:** $R_D = 1 - \frac{1}{C_F}$

Example:



if $C_F = \frac{10}{1}$ then $R_D = 1 - \frac{1}{10} = 0.9$

(90% of the data in dataset1 is redundant)

if $n_2 = n_1$ then $C_F = 1,\ R_D = 0$

if $n_2 << n_1$ then $C_F \to \infty,\ R_D \to 1$

if $n_2 >> n_1$ then $C_F \to 0,\ R_D \to -\infty$

Coding Redundancy

- $l(r_k)$ = **variable length**
- **Consider the probability of the gray levels:**

Table 6.1 Variable-Length Coding Example

variable length

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_0 = 0$	0.19	000	3	11	2
$r_1 = 1/7$	0.25	001	3	01	2
$r_2 = 2/7$	0.21	010	3	10	2
$r_3 = 3/7$	0.16	011	3	001	3
$r_4 = 4/7$	0.08	100	3	0001	4
$r_5 = 5/7$	0.06	101	3	00001	5
$r_6 = 6/7$	0.03	110	3	000001	6
$r_7 = 1$	0.02	111	3	000000	6

$$L_{avg}^{Code1} = \sum_{k=0}^{7} l_1(r_k) P_r(r_k) = 3\,bits$$

$$\text{Total number of bits(Code1)} : 3NM$$

$$L_{avg}^{Code2} = \sum_{k=0}^{7} l_2(r_k) P_r(r_k) = 2.7\,bits$$

$$\text{Total number of bits(Code2)} : 2.7NM$$

$$C_F = \frac{3}{2.7} = 1.11 \ (about 10\%)$$

$$R_D = 1 - \frac{1}{1.11} = 0.099$$

Inter Data Redundancy

- **Inter data redundancy implies that any data value can be reasonably predicted by its neighbors (i.e., correlated).**

- **To reduce inter data redundancy, the data must be transformed in another format (i.e., through a transformation)**
  - **e.g., thresholding, differences between adjacent data, DFT, DCT, DWT, … .**

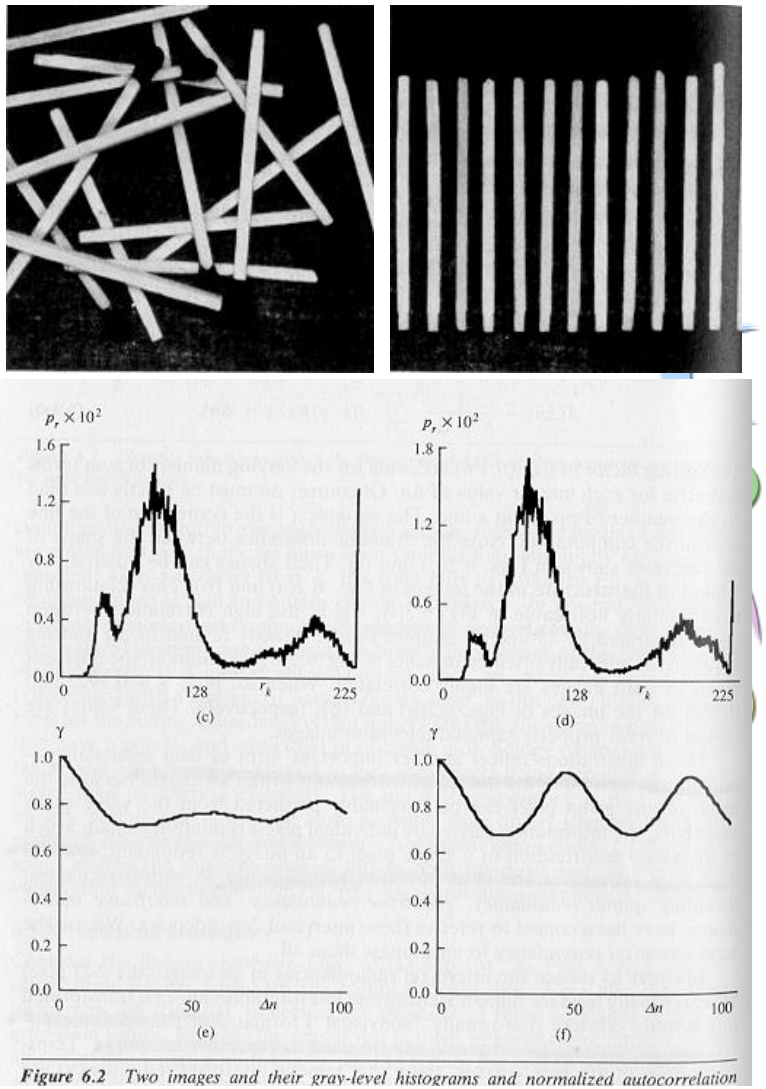


Figure 6.2 Two images and their gray-level histograms and normalized autocorrelation

Information Theory

- **Information theory first was developed by Shannon in the 1940's and 50's.**
  - **It attempts to find a mathematical way of measuring the quantity of information.**
  - **It also explains the limits of communication systems using probability theory.**

- **In the information theory two question are considered:**
  - **What is the information content of a message/file?**
  - **What is the minimum amount of data that is sufficient to describe completely a message/file without loss of information?**

What is the information?

- **The information is something that adds to people's knowledge.**
  - **The more a message conveys what is unknown (so it is new and surprising), the more informative the message is.**
  - **In other words, the element of surprise, unexpectedness or unpredictability is relevant to the amount of information in the message.**
  - **For example, to most final year university students, consider to these two messages:**
    - **'Two thirds of the students passed exams in all subjects'**
    - **'Two thirds of the students got 100% marks for exams in all subjects'**
  - **although the two messages contain a similar number of words, the first message conveys less information than the second message (it states something that would not normally happen and is totally unexpected).**

Modeling Information

- **The expectation of the outcome of an event can be measured by the *probability* of the event.**
  - **The high expectation corresponds to a high probability of the event.**
  - **A rarely happened event means a low probability event.**
  - **Something that never happens has a zero probability.**
  - **Hence the amount of information in a message can also be measured quantitatively according to the unexpectedness or surprise evoked by the event.**
- **Idea: For a random event E with probability P(E) the amount of information (self-information) is:**

$$I(E) = log(\frac{1}{P(E)}) = -log(P(E)) \text{ units of information}$$

  - **Note: I(E)=0 when P(E)=1**

First Order Information

- Suppose there are a set of n events $S=(r_1, r_2, \ldots, r_n)$. S is called an alphabet if each r_k, where $k= 1, \ldots, n$, is a symbol used in a source message.
  - Let the probability of occurrence of event r_k be p_k.
  - These probabilities of the events $P=(p_1, p_2, \ldots, p_n)$ add up to 1, i.e. $\Sigma\, p_k = 1$.
  - Suppose further that the source is memory less, i.e. each symbol occurs independently with no dependence between successive events.
- The amount of one's surprise evoked by the event r_k (is called the self-information) is defined by a simple mathematical formula:

$$I(r_k) = -\log(P(r_k))$$ units of information

First Order Entropy

- **Self-information is useful for measuring individual events or symbols at a time.**
  - **However, we are often more interested in the information of a source where the occurrences of all possible symbols have to be considered.**
  - **In other words, the measure of the information of a source has to be considered over the entire alphabet.**

- **One common way to consider a set of data is to look at its average value of the self-information of all the symbols which is called the first-order entropy.**

$$H = -\sum_{k=0}^{L-1} P(r_k) log(P(r_k))$$

First Order Entropy

- **H is the average number of bits required to code up a symbol, given all we know is the probability distribution of the symbols.**
- **H is the Shannon lower bound on the average number of bits to code a symbol in this "source model".**
- **The amount of redundancy (R) can be measured as:**

$$R = L_{avg} - H$$

- **where:**

$$L_{avg} = E(l(r_k)) = \sum_{k=0}^{L-1} l(r_k)P(r_k)$$

- **Note: of L_{avg}= H, the R=0 (no redundancy)**

First Order Information

- Concider an alphabet {a, b, c} with probability
  - P(a) = 1/8,
  - P(b) = 1/4,
  - P(c) = 5/8
- Then the Information of each alphabet is:
  - $I(a) = \log_2(8) = 3$
  - $I(b) = \log_2(4) = 2$
  - $I(c) = \log_2(8/5) = 0.678$
- Receiving an "a" has more information than receiving a "b" or "c".

First Order Information

- Consider an alphabet {a, b, c} with probability
  - P(a) = 1/8, P(b) = 1/4, P(c) = 5/8
  - The Information of each alphabet is:
    - $I(a) = \log_2(8) = 3$, $I(b) = \log_2(4) = 2$, $I(c) = \log_2(8/5) = 0.678$
- Then the Entropy of alphabet is:
  - H = 1/8 x 3 + 1/4 x 2 + 5/8 x 0.678 = 1.3 bits/symbol

- Consider an alphabet {a, b, c} with probability
  - P(a) = 1/3, P(b) = 1/3, P(c) = 1/3
    - Then $I(a) = I(b) = I(c) = \log_2(3) = 1.6$
- Then the Entropy of alphabet is:(worst case)
  - $H = 3x(1/3)x\log_2(3) = 1.6$ bits/symbol

First Order Entropy Estimation

- **It is not easy to estimate H reliably!**

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243

image

Gray Level	*Count*	*Probability*
21	12	3/8
95	4	1/8
169	4	1/8
243	12	3/8

- **First order estimate of H:**

$$H = -\sum_{k=0}^{3} P(r_k) log(P(r_k)) = 1.81 \text{ bits/pixel}$$

Total bits: 4 x 8 x 1.81 = 58 bits

Second Order Entropy Estimation

- **Second order estimate of H:**
  - **Use relative frequencies of pixel blocks :**

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243

image

Gray Level Pair	*Count*	*Probability*
(21, 21)	8	1/4
(21, 95)	4	1/8
(95, 169)	4	1/8
(169, 243)	4	1/8
(243, 243)	8	1/4
(243, 21)	4	1/8

$$H = -\sum_{k=0}^{5} P(r_k) log(P(r_k)) = 1.25 \text{ bits/pixel}$$

Total bits: $4 \times 8 \times 1.25 = 40$ bits

Estimating Entropy

- The first-order estimate provides only a lower-bound on the compression that can be achieved.

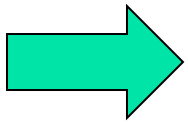
- Differences between higher-order estimates of entropy and the first-order estimate indicate the presence of inter data redundancy!

Need to apply transformations

Estimating Entropy

- **For example, consider differences:**

21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243
21	21	21	95	169	243	243	243



21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0
21	0	0	74	74	74	0	0

Gray Level or Difference	*Count*	*Probability*
0	16	1/2
21	4	1/8
74	12	3/8

Estimating Entropy

- **Entropy of difference image:**

$$H = -\sum_{k=0}^{2} P(r_k) log(P(r_k)) = 1.41 \text{ bits/pixel}$$

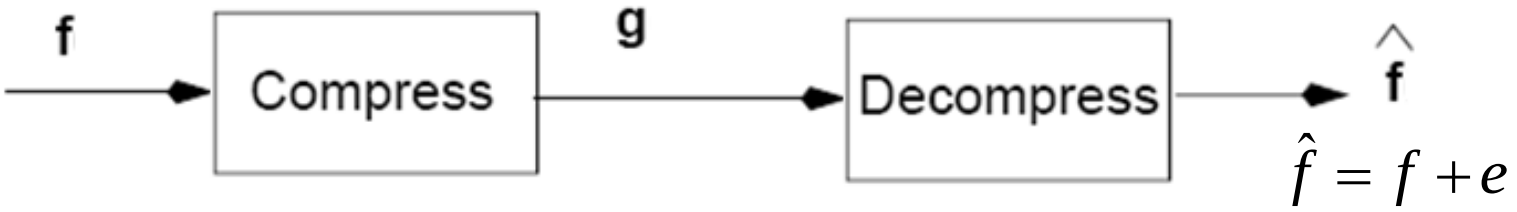
- **Better than before (i.e., H=1.81 for original image)**

- **However, a better transformation could be found since:**

1.41 bits/pixel > 1.25 bits/pixel (from 2nd order estimate of H)

Compression Fidelity Criteria

- **The performance of a compression algorithm can be measured by various criteria depending on the nature of the application.**



- **How $\hat{f}$ is close to f ?**

- **Two types of Criteria can be used:**
  - **Subjective: based on human observers**
  - **Objective: mathematically defined criteria**

Subjective Fidelity Criteria

Value	Rating	Description
1	Excellent	A data of extremely high quality, as good as you could desire.
2	Fine	A data of high quality, providing enjoyable viewing. Interference is not objectionable.
3	Passable	A data of acceptable quality. Interference is not objectionable.
4	Marginal	A data of poor quality; you wish you could improve it. Interference is somewhat objectionable.
5	Inferior	A very poor data, but you could use it. Objectionable interference is definitely present.
6	Unusable	A data so bad that you could not use it.

Objective Fidelity Criteria

- **Root mean square error (RMS):**

$$e_{rms} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(\hat{f}_i - f_i)^2}$$

$$e_{rms} = \sqrt{\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}(\hat{f}(x,y) - f(x,y))^2}$$

- **Mean-square signal-to-noise ratio (SNR):**

$$SNR_{ms} = \frac{\sum_{i=0}^{N-1}(\hat{f}_i)^2}{\sum_{i=0}^{N-1}(\hat{f}_i - f_i)^2}$$

$$SNR_{ms} = \frac{\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}(\hat{f}(x,y))^2}{\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}(\hat{f}(x,y) - f(x,y))^2}$$

Objective Fidelity Criteria

- **Compression Ratio:** $$C_R = \frac{Size(g)}{Size(f)}$$

- **Compression Factor:** $$C_F = \frac{Size(f)}{Size(g)}$$

- **Saving Percentage:** $$S_P = \frac{Size(f) - Size(g)}{Size(f)}\%$$

Objective Fidelity Criteria

- **Computation Complexity:**
  - **This can be adopted from well-established algorithm analysis techniques. We may, for example, use the O-notation for the time efficiency and storage requirement. However, compression algorithms' behavior can be very inconsistent. Nevertheless, it is possible to use past empirical results.**

- **Compression Time:**
  - **We normally consider the time for encoding and decoding separately. In some applications, decoding time is more important than encoding time. In other applications, they are equally important.**

Objective Fidelity Criteria

- **Entropy :**
  - **If the compression algorithm is based on statistical results, then entropy can be used as a theoretical bound to the source to help make a useful quantity judgment. It also provides a theoretical guidance as to how much compression can be achieved.**

- **Redundancy :**
  - **In certain areas of compression, the difference between the average code length and the entropy of the source can be regarded as redundancy. In some other areas, the difference between a normal and uniform probability distribution is identified as redundancy. The larger the gap, the greater amount of the redundancy in the code. When the gap is zero, the code is said to be *optimal.***

Objective Fidelity Criteria

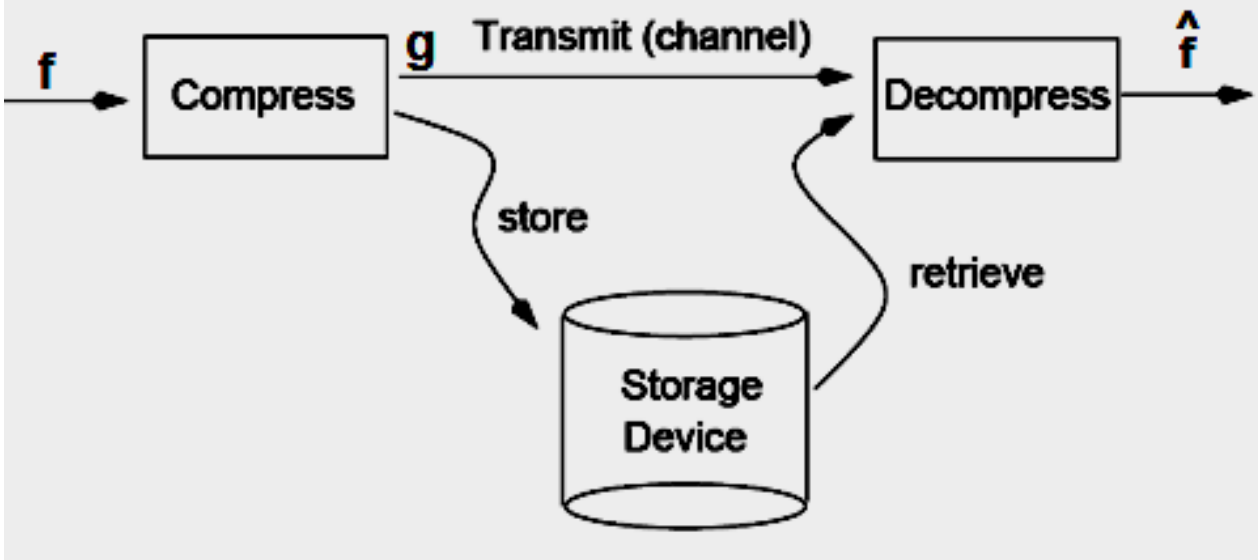
- **Empirical testing:**
  - Measuring the performance of a compression scheme is difficult if not impossible. There is perhaps no better way than simply testing the performance of a compression algorithm by implementing the algorithm and running the programs with sufficiently rich test data. Canterbury Corpus provides a good test bed for testing compression programs. For details refer to: http://corpus.canterbury.ac.nz.

- **Overhead:**
  - This measure is used often by the information technology industry. Overhead is the amount of extra data added to the compressed version of the data for decompression later. Overhead can sometimes be large although it should be much smaller than the space saved by compression.

Basic Methods

Goal of Compression

- **As we mentioned the goal of each compression system is to:**
  - **Conserve storage space**
  - **Reduce time for transmission**
  - **Progressive transmission**
  - **Reduce computation**



Basic Data Types

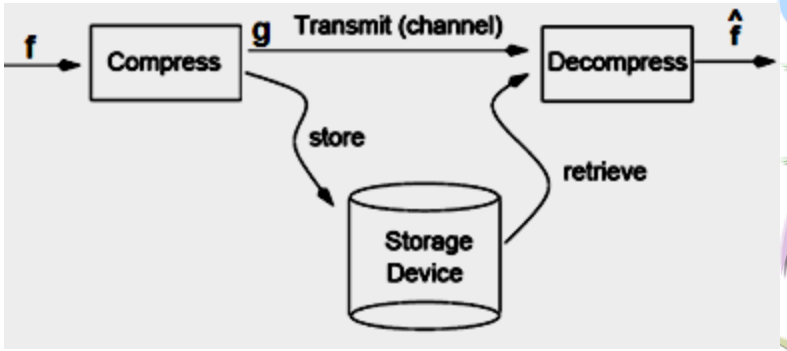
- The word *data in the context of data compression includes any digital form* of factual information that can be processed by a computer program.
- Hence data may also be classified as:
  - text,
  - binary,
  - graphic,
  - image ,
  - audio,
  - video,
  - and multimedia
- while the real digital data format consists of 0s and 1s in a binary format.

Basic Data Types

- **Text** data are usually represented by 8 bit extended ASCII code.
  - They appear in files with the extension .txt or .tex (or other coding system readable files like .doc) using an editor.
- **Binary** data include database files, spreadsheet data, executable files, and program codes.
  - These files have the extension .bin.
- **Image** data are represented often by a two-dimensional array of *pixels* in which each pixel is associated with its color code.
  - Extension .bmp represents a type of bitmap image file in Windows, and .psd for Adobe Photoshop's native file format.
- **Graphics** data are in the form of vectors or mathematical equations.
  - An example of the data format is .png which stands for Portable Network Graphics.
- **Sound** data are represented by a wave (periodic) function.
  - A common example is sound files in .way format.

Basic Data Compression Concepts

- **Lossless** compression $(f = \hat{f})$
  - Also called entropy coding, reversible coding.
  - Information preserving.
  - Low compression ratios.
- **Lossy** compression $(f \neq \hat{f})$
  - Also called irreversible coding.
  - Not information preserving.
  - High compression ratios.



- Compression ratio = $\frac{|g|}{|f|}$
  - $|f|$ is number of bits in f.
- **Trade-off**: Quality vs Compression ratio

Lossless Compression

- Data is not lost - the original is really needed.
  - text compression
  - compression of computer binary files
- Compression ratio typically no better than 4:1
- Statistical-Based Techniques
  - Huffman coding
  - Arithmetic coding
  - Golomb coding
- Dictionary-Based techniques
  - LZW, LZ77
  - Sequitur
  - Burrows-Wheeler Method
- Existing Standards –
  - Morse code, Braille, gzip, zip, bzip, GIF, JBIG, Lossless JPEG

Lossy Compression

- **Data is lost, but not too much.**
  - **audio**
  - **video**
  - **still images, medical images, photographs**
- **Compression ratios of 10:1 often yield quite high fidelity results.**
- **Major techniques include**
  - **Vector Quantization**
  - **Wavelets**
  - **Block transforms**
  - **Standards - JPEG, MPEG**

Compression Algorithm Design

- **There are many ways to design algorithms. A systematic approach including eight stages can be summarized as follows:**
  - **1. Description of the problem**
  - **2. Mathematical modeling**
  - **3. Design of the algorithm**
  - **4. Verification of the algorithm**
  - **5. Estimation of the computational complexity**
  - **6. Implementation**
  - **7. Program testing**
  - **8. Documentation.**

Compression Algorithm Design

- **Description of the problem:**
  - A compression problem, from the algorithmic point of view, is to find an effective and efficient algorithm to remove various redundancy from certain types of data.
  - In this stage, we want to understand the problem precisely and ask the right questions. For example, we want to know what the format of the data is and what the restrictions to the output might be. We need to understand the problem well enough to be able to write a precise statement of the problem.
  - If the problem is vague, we may use methods such as *divide and conquer* to divide the problem into sub-problems, and to divide the sub-problems into their sub-problems repeatedly until every sub-problem is manageable.
  - The deliverables of this stage are a specification of the problem which should include details about the input and output of the algorithm.

Compression Algorithm Design

- **Mathematical modeling**
  - **Modeling is a process of setting up an environment to allow the interested variables to be observed or certain behavior of a system to be explored. It is a formalization and extension to the description of a problem. Rules and relationships are often set in mathematical formula.**
  - **Modeling is a critical stage of algorithm design. A model can sometimes decide immediately the approaches of the algorithm.**
  - **In data compression, models are used to describe a source data. For compression problems, modeling can be viewed as a process of identifying the redundant characteristics of the source data, and finding an effective way to describe them.**
  - **The model is the embodiment of what the compression algorithm knows about the source. It is like a platform on which every compression algorithm has to make use of some knowledge in order to perform.**

Compression Algorithm Design

- **Mathematical modeling is an important branch in mathematics, statistics and computer science as a popular research area in its own right.**
- **In data compression, as one of its application areas, commonly used mathematical models can be classified as follows:**
  - **1. Physical model: using known physics of the source such as certain data generation processes or empirical observation**
  - **2. Probability models: using probability theory to describe the source**
  - **3. Markov model: using Markov chain theory to model the source**
  - **4. Composite model: describing the source as a combination of several different kinds and using a switch to activate one type at a time.**

Compression Algorithm Design

- **Design of the algorithm**
  - **In this stage, we may apply literally all the algorithm knowledge and techniques we have.**
  - **Design of the algorithms is an interesting and challenging task. The techniques depend highly upon the choice of the mathematical models.**
  - **We may add further details to the model, consider feedback to realize the model, using standard techniques in algorithm design.**
    - **For example, we may decide to use certain data structures, abstract data types or various off-the-shelf tools to help us with algorithmic solutions.**
  - **We may take top-down approaches, and identify existing efficient algorithms to achieve partial solutions to the problem.**

Compression Algorithm Design

- **Most data compression problems are data oriented. There is unlikely to be an efficient algorithm to a general question about all sorts of data.**
- **We may then have to adjust the data range, or add more restrictions to the specific type of data.**
- **We may even return to the previous stage and experiment with alternative models if serious flaws are found in the approach.**
- **The deliverables of this stage are the correct algorithmic solutions to our problem. This includes algorithms in pseudo code and convincing consideration on data structures.**

Compression Algorithm Design

- **Verification of the algorithm**
  - **This is sometimes the most difficult task. We may have seen or heard of a common practice in software development in which people tend to leave this until the program testing stage.**
  - **The drawback in that approach is that we may have wasted an enormous amount of energy and time on programming before realizing the algorithm has fundamental flaws.**
  - **In this stage, we check the correctness of the algorithm, the compression quality, and efficiency of the coder.**
    - **It is relatively easy to check the compression quality. For example, we can use *compression ratio or saving percentage to see* how effective the compression is achieved.**
    - **The coder efficiency is defined as the difference between the average length of the codewords and the entropy.**
  - **The deliverables at the end of this stage can be correctness proofs for the algorithmic solutions, or other correctness insurance such as reasoning or comparisons.**

Compression Algorithm Design

- **Estimation of the computational complexity**
  - **Similar to algorithm verification, time and money have been regarded as well spent for analyzing and estimating the efficiency of algorithms.**
  - **A good analysis can save a lot of time and energy being wasted on faulty and disastrous software products.**
  - **In this stage, it is possible to estimate and predict the behaviors of the software to be developed using a minimum amount of resources, for example using just a pen and a piece of paper.**
  - **We should therefore try to compare at least two candidate algorithms in terms of efficiency, usually time efficiency.**
  - **The more efficient algorithm should also be checked to ensure that it has met certain theoretical bounds before being selected and implemented.**

Compression Algorithm Design

- **Implementation**
  - There is no restriction on what high-level computer language you use and how you would like to implement procedures or functions. But the MATLAB is one of the most popular tools that you can use.
- **Program testing**
  - This is a huge topic in its own right. There are formal methods that are dedicated to the work at this stage, like using standard test benches.
- **Documentation**
  - This is another important stage of work that encouraged to find an effective and consistent way to include this stage into your study activities.

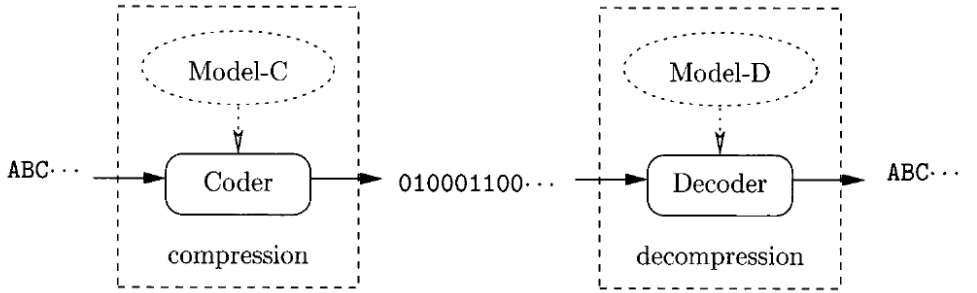
Compression Algorithms

- **Any compression methods involve essentially two types of work:**
  - **Modeling: The model represents our knowledge about the source domain and the manipulation of the source redundancy.**
  - **coding. The device that is used to fullfil the task of coding is usually called *coder meaning encoder.***
    - ***Based on the model and* some calculations, the coder is used to derive a code and encode (compress) the input.**
    - **A coder can be independent of the model.**
- **A similar structure applies to decoding algorithms. There is again a *model* and a *decoder for any decoding algorithm.***
- **Such a model-coder structure at both the compression and decompression ends.**

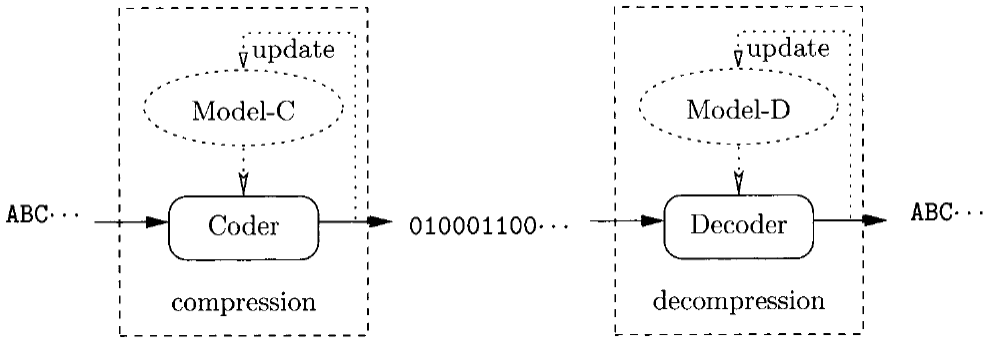
Compression Algorithms

- **Conceptually, we can distinguish *two types of compression algorithms* :**

- **Static (non-adaptive) system:** **The model (Model-C or Model-D) remains unchanged during the compression or decompression process.**



- **Adaptive system:** **The model may be changed during the compression or decompression process according to the change of input (or feedback from the output). Some adaptive algorithms actually build the model based on the input starting from an empty model.**



Compression Algorithms

- **In terms of the length of codewords used *before or after compression*, compression algorithms can be classified into the following categories:**
  - **Fixed-to-fixed: each symbol before compression is represented by a fixed number of bits (for example, 8 bits in ASCII format) and is encoded as a sequence of bits of a fixed length after compression. Example *A:00, B:01, C:10, D:11***
  - **Fixed-to-variable: each symbol before compression is represented by a fixed number of bits and is encoded as a sequence of bits of different length. Example *A:O; B:1O; C:101; D:0101.***
  - **Variable-to-fixed: a sequence of symbols represented in a different number of bits before compression is encoded as a fixed-length sequence of bits. Example *ABCD:O0; ABCDE:01; BC:11.***
  - **Variable-to-variable: a sequence of symbols represented in a different number of bits before compression is encoded as a variable-length sequence of bits. Example *ABCD:O; ABCDE:01; BC:1; BBB:0001.***

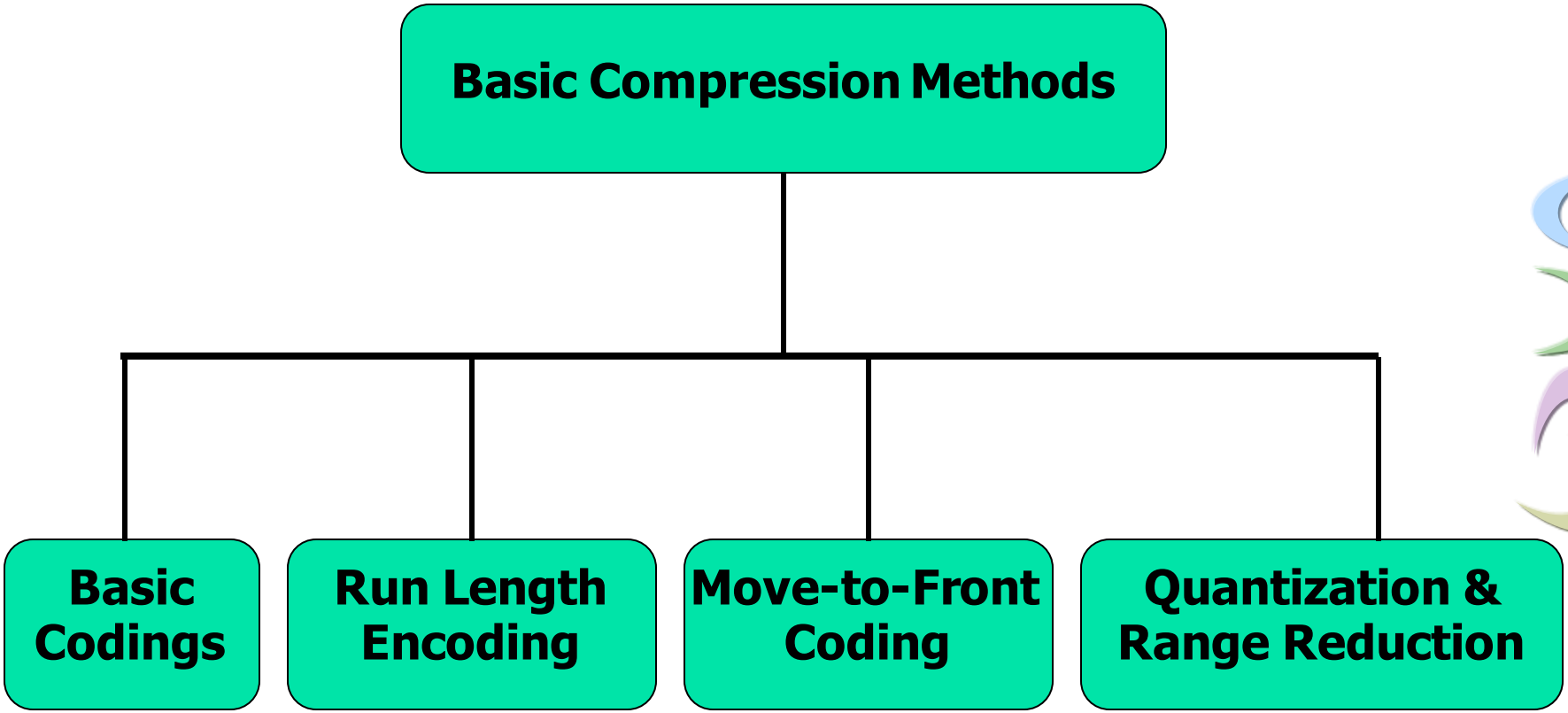
Compression Algorithms

- we often use the term *symbol*, *or character*, *to mean a symbolic* representation of input data into a compression algorithm.
- Under this notation, a symbol can be an audio sample, or an image pixel value as well as a letter, special character or a group of letters in a text.
- A text, image, audio or video file can then be considered as a one-dimensional, or multi-dimensional sequence of symbols.
- The process of assigning code words to each symbol in a source is called encoding. The reverse process, i.e. to reconstruct the sequence of symbols in the source, is called decoding.
  - Suppose the alphabet of a source is $S=(r_1, r_2, \ldots, r_n)$. The digital representation of the symbol set is called the code $S=(c_1, c_2, \ldots, c_n)$ and the representation c_j of each symbol is called the codeword for symbol r_j, where $j = 1, 2, \ldots, n$.
- Clearly, compression can be viewed as encoding and decompression as decoding in this sense.

Basic Methods

- Data is compressed by reducing its redundancy, but this also makes the data less reliable, more prone to errors.
- Increasing the integrity of data, on the other hand, is done by adding check bits and parity bits, a process that increases the size of the data, thereby increasing redundancy.
- Data compression and data reliability are therefore opposites, and it is interesting to note that the latter is a relatively recent field, whereas the former existed even before the advent of computers.
  - The sympathetic telegraph, the Braille code and the Morse code use simple, intuitive forms of compression.

Basic Methods



Basic Codings

- **The fundamental representation of digital data can be performed with two types of codewords:**
  - ***Fixed length codewords:*** ***the length of codes of all symbols is same*****.**
    - **CDC display code.**
    - **ASCII codes.**
    - **Braille coding.**
    - **Baudot code.**
    - **Tunstall code**
  - ***Variable length codewords:*** ***It is possible to*** **represent an alphabet by a set of** ***codes with variable length codes.***
    - ***Self-delimiting coding.***
    - ***Suffix coding.***
    - ***Prefix coding.***
    - **Unary coding*.***
    - ***Phase in coding.***
    - ***Golomb coding.***

Fixed Length Coding
CDC Display Coding

- **If the text includes just uppercase letters, digits, and some punctuation marks, the old 6-bit CDC display code may be used.**
  - **This code was commonly used in second-generation computers (and even a few third-generation ones).**
  - **These computers did not need more than 64 characters because they did not have any display monitors and they sent their output to printers that could print only a limited set of characters.**

Bits	Bit positions 210							
543	0	1	2	3	4	5	6	7
0		A	B	C	D	E	F	G
1	H	I	J	K	L	M	N	O
2	P	Q	R	S	T	U	V	W
3	X	Y	Z	0	1	2	3	4
4	5	6	7	8	9	+	-	*
5	/	(	)	$	=	sp	,	.
6	≡	[	]	:	≠	_	∨	∧
7	↑	↓	<	>	≤	≥	¬	;

Fixed Length Coding
Braille Coding

- **The well-known Braille code for the blind was developed by Louis Braille in the 1820s and is still in common use today.**
  - **It consists of groups (or cells) of 3 *× 2 dots each, embossed on thick paper.***
  - ***Each of the six dots*** **in a group may be flat or raised, implying that the information content of a group is equivalent to six bits, resulting in 64 possible groups.**
  - **The letters, digits, and common punctuation marks do not require all 64 codes, which is why the remaining groups may be used to code common words—such as and, for, and of—and common strings of letters —such as ound, ation, and th.**

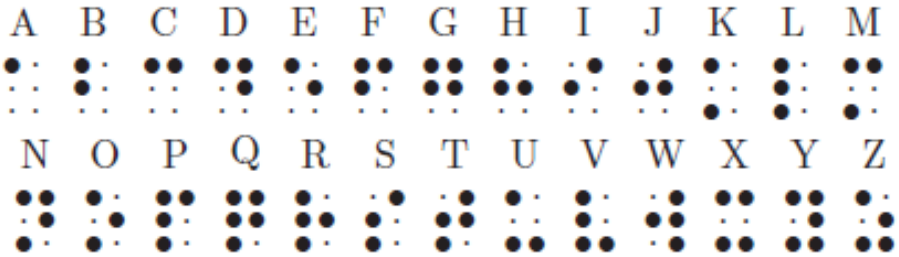


Table 1.1: The 26 Braille Letters.

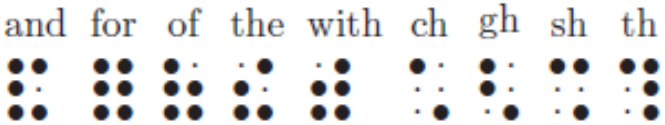


Table 1.2: Some Words and Strings in Braille.

Fixed Length Coding
Baudot Coding

- **The Baudot code is a 5-bit code developed by J. M. E. Baudot in about 1880 for telegraph communication.**
  - **It became popular and by 1950 was designated the International Telegraph Code No. 1.**
  - **It was used in many first- and second-generation computers. The code uses 5 bits per character but encodes more than 32 characters.**
  - **Each 5-bit code can be the code of two characters, a letter and a figure. The "letter shift" and "figure shift" codes are used to shift between letters and figures.**
  - **The Baudot code can represent $32 \times 2 - 2 = 62$ *characters* (each code can have two meanings except the LS and FS codes).**

Letters	Code	Figures	Letters	Code	Figures
A	10000	1	Q	10111	/
B	00110	8	R	00111	-
C	10110	9	S	00101	SP
D	11110	0	T	10101	na
E	01000	2	U	10100	4
F	01110	na	V	11101	'
G	01010	7	W	01101	?
H	11010	+	X	01001	,
I	01100	na	Y	00100	3
J	10010	6	Z	11001	:
K	10011	(	LS	00001	LS
L	11011	=	FS	00010	FS
M	01011	)	CR	11000	CR
N	01111	na	LF	10001	LF
O	11100	5	ER	00011	ER
P	11111	%	na	00000	na

LS, Letter Shift; FS, Figure Shift.
CR, Carriage Return; LF, Line Feed.
ER, Error; na, Not Assigned; SP, Space.

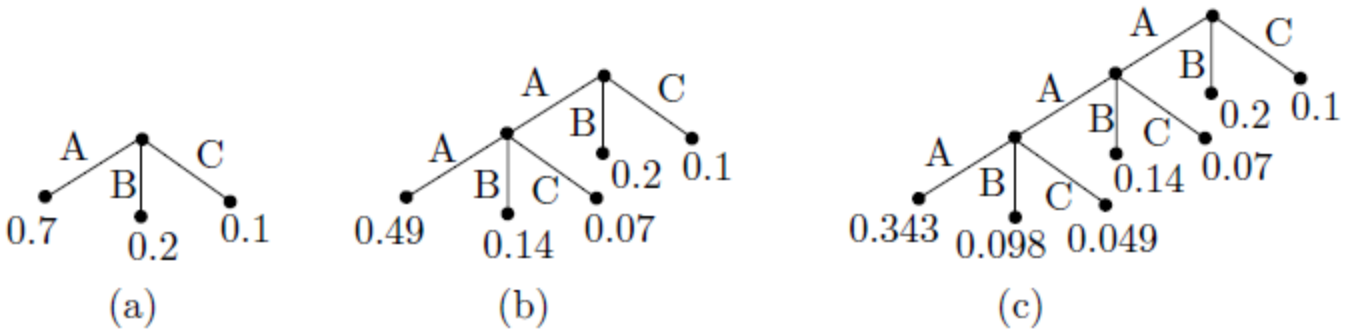
Fixed Length Coding
Tunstall Coding

- **The Tunstall is a variable-to-fixed length code that try to minimize the number of bits per symbol.**

- **Suppose there are m initial symbols. Choose a target output length n where $2^n > m$.**
  - **1. Form a tree with a root and m children with edges labeled with the symbols.**
  - **2. Find the leaf with highest probability and expand it to have m children.****
    - ** The probability is the product of the probabilities of the symbols on the root to leaf path.
  - **3. If the number of leaves is > $2^n - m$ then halt*, else Go to 2.**
    - * In the next step we will add m-1 more leaves.

Fixed Length Coding
Tunstall Coding (Example)

- **Given an alphabet with the three symbols *A, B, and C , with probabilities* 0.7, 0.2, and 0.1, respectively, we decide to try to construct a set of 3-bit Tunstall codes (thus, *n = 3).***
  - ***We start our code table as a tree with a root and three children*** **(Fig. a).**
  - **In the first iteration, we select *A and turn it into the root of a subtree* with children *AA, AB, and AC with probabilities 0.49, 0.14, and 0.07, respectively* (Fig. b).**
  - **The largest probability in the tree is that of node *AA, so the second* iteration converts it to the root of a subtree with nodes *AAA, AAB, and AAC with* probabilities 0.343, 0.098, and 0.049, respectively (Fig. c).**



Fixed Length Coding
Tunstall Coding

- After each iteration we count the number of leaves of the tree and compare it to $2^3 = 8$.
- After the second iteration there are seven leaves in the tree, so the loop stops.
- Seven 3-bit codes are arbitrarily assigned to elements *AAA, AAB, AAC, AB, AC, B, and C.*
- *The eighth* available code should be assigned to a substring that has the highest probability and also satisfies the prefix property.
- The average bit length of this code is easily computed as:

$$n/\sum_{i=1}^{m} p_i l_i = \frac{3}{3(0.343+0.098+0.049)+2(0.14+0.07)+0.2+0.1} = 1.37 \text{ bits/symbol}$$

- The entropy of our alphabet is :

$$-(0.7\bar{} \times \log_2 0.7\bar{} + 0.2 \times \log_2 0.2 + 0.1 \times \log_2 0.1) = 1.156,$$

- *so the Tunstall* codes do not provide the best compression.

Variable Length Coding (VLC)

- **Encoding** a string of symbols a_i with **Variable Length Codes (VLCs)** is **easy**. The software reads the original symbols a_i one by one and replaces each a_i with its binary, variable-length code c_i . The codes are concatenated to form one (normally long) bitstring.
  - The encoder either includes a table with all the pairs (a_i , c_i) or it executes a procedure to compute code c_i from the bits of symbol a_i .
- **Decoding** is slightly **more complex**, because of the different lengths of the codes. When the decoder reads the individual bits of VLCs from a bitstring, **it has to know either how long each code is** or **where each code ends**.
  - This is why a set of variable-length codes has to be carefully selected and why the decoder has to be taught about the codes. The decoder either has to have a table of all the valid codes, or it has to be told how to identify valid codes (using **self-delimiting (suffix)** codes or **Prefix-free (Prefix)** codes).

Variable Length Coding
Self-Delimiting and Suffix Coding

- **Self-delimiting codes** are codes that have variable lengths and can be decoded unambiguously. In these methods we add some extra bits to end of data (or beginning of data) to indicate its length or end point.
- **Suffix codes** (codes that end with a special flag). Such codes limit the propagation of an error and are therefore robust.
  - An error in a codeword affects at most that codeword and the one or two codewords following it.
  - Most other variable-length codes sacrifice data integrity to achieve short codes, and are fragile because a single error can propagate indefinitely through a sequence of concatenated codewords.
  - Flag code is also included in this category.
- Note. The term "**suffix code**" is ambiguous. It may refer to codes that end with a special bit pattern, but it also refers to codes where no codeword is the suffix of another codeword (the opposite of prefix codes).

Variable Length Coding
Definition of prefix codes

- **A prefix is the first few consecutive bits of a codeword. When two codewords are of different lengths, it is possible that the shorter codeword is identical to the first few bits of the longer codeword. In this case, the shorter codeword is said to be a prefix of the longer one.**
- **A prefix code c is a code with the property that for all different symbols x and y, c(x) is not a prefix of c(y).**
- **The prefix property of a code guarantees only the correctness and efficiency of decoding. To achieve a good compression, the length of the codewords are required to be as short as possible.**
  - **the length of codes used for symbols is different and symbols with high occurrence probability have codes with lower length(such as {0,11,10} codes for {a,b,c}).**
  - **The most common example of prefix codes are the Huffman codes. The other example is the unary code.**
- **Prefix codes are also referred to as prefix-free codes, prefix condition codes, or instantaneous codes.**

Variable Length Coding
Uniquely Decodable

- **A code is uniquely decodable (UD) if there is only one possible way to decode encoded messages.**
- **Consider the codes for a1, a2, a3, and a4:**

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

- **Code 1 and Code 2 are not uniquely decodable.**
  - In Code 1 both a1 and a2 have been assigned the codeword 0
  - In Code 2 string **100** can be decoded as ***a2 a1 a1***, or as ***a2 a3***
- **Code 3 and Code 4 are uniquely decodable.**
  - There is a slight difference between Code 3 and Code 4. In the case of Code 3, the decoder knows the moment a code is complete. In Code 4, we have to wait till the beginning of the next codeword before we know that the current codeword is complete. Because of this property, Code 3 is called an ***instantaneous*** code.

Variable Length Coding
Uniquely Decodable

- A prefix code (a set of codewords that satisfy the prefix property) is uniquely decodable (UD).

- Such a code is also complete if adding any codeword to it turns it into a non-UD code. A complete code is the largest UD code, but it also has a downside; it is less robust.
  - If even a single bit is accidentally modified or deleted (or if a bit is somehow added) during storage or transmission, the decoder will lose synchronization and the rest of the transmission will be decoded incorrectly.

- A UD code that consists of n codewords of lengths L_i must satisfy the Kraft inequality , but this inequality does not require a prefix code. Thus, if a code satisfies the Kraft inequality it is UD, but if it is also a prefix code, then it is instantaneous.

Kraft-McMillan Inequality

- The **Kraft–McMillan inequality** is concerned with the existence of a uniquely decodable (UD) code. It establishes the relation between such a code and the lengths L_i *of its* codewords.

- **Theorem**: There exists a prefix binary code C =($c_1, c_2, \ldots, c_n$) with *n* codewords of lengths ($L_1, L_2, \ldots, L_n$) respectively if and only if:

$$k(C) = \sum_{i=1}^{n} 2^{-L_i} \leq 1$$

- **This inequality is known as the *Kraft –McMillan inequality.***

- **Kraft–McMillan's** theorem provides a useful guide on the minimum requirements to the codeword lengths of prefix codes.
  - Knowing the limit, we can avoid looking for a prefix code when it in fact does not exist.
  - If the lengths do not satisfy the *Kraft–McMillan inequality,* we can conclude that it is not possible to find a prefix code consisting of these lengths

Kraft-McMillan Inequality

- **This inequality states that given a set of *n* positive integers ($L_1, L_2, \ldots, L_n$) that satisfy *Kraft –McMillan inequality*, there exists an prefix code such that the L_i are the lengths of its individual codewords.**

  - **Example1:** ***Discuss the possibility of finding a prefix code with codeword lengths 1, 2, 3, 2.***

$$k(C) = \sum_{i=1}^{4} 2^{-L_i} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^2} > 1$$

  - **it is impossible to find a prefix code with codeword lengths 1, 2, 3, 2.**

  - **Example2:** ***Discuss the possibility of finding a prefix code with codeword lengths 1, 2, 3, 3.***

$$k(C) = \sum_{i=1}^{4} 2^{-L_i} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^3} = 1$$

  - **the lengths of the codewords satisfy the Kraft inequality, it is possible to find a prefix code with these codeword lengths.**

Kraft-McMillan Inequality

- The **Kraft–McMillan inequality** sets requirements to the lengths of a prefix code. If the lengths do not satisfy the **Kraft–McMillan inequality**, we know there is no chance of finding a prefix code with these lengths.
- The **Kraft–McMillan inequality** does not tell us how to construct a prefix code, nor the form of the code. Hence it is possible to find prefix codes in different forms and a prefix code can be transformed to another by swapping the position of 0s and ls.
- The **Kraft–McMillan inequality** can tell that a given code is *not a prefix code but it* cannot be used to decide if a given code is a prefix code.
- The **Kraft–McMillan inequality** can tell us whether the lengths of a prefix code can be shortened, but it cannot make any change to the lengths.

Shannon's Theorem

- **Shannon showed that the best a lossless symbolic compression scheme using binary prefix codes can encode a source with an average number of bits equal to the entropy of the source.**
- **Theorem: For any binary prefix code with the average codeword length $\bar{l}$, we have:**

$$H(P) \le \bar{l}(P,L) \le H(P)+1$$

- **H(P) is the entropy of *n* independent symbols *with length* $L=(L_1,L_2,\ldots,L_n)$ and occurrence probabilities $P=(p_1,p_2,\ldots,p_n)$.**

$$\bar{l}(P,L)=\sum_{i=1}^{n} p_i L_i \qquad H(P)=-\sum_{i=1}^{n} p_i \log_2(p_i)$$

- **The ratio in percentage of the entropy of a source to the average code length is a useful measure of the *code efficiency for the source.***

$$E(P,L)=\frac{H(P)}{\bar{l}(P,L)}\times 100\%$$

- **A code is *optimum if the code efficiency reaches 100%.***

Variable Length Coding
Unary Coding

- **The unary code is simplest variable-length code for integers. The unary code of the positive integer *n is constructed from n- 1 one's followed by a* single 0, or alternatively as *n−1 zeros followed by a single 1*.**
  - **The unary code is simple to construct and is employed in many applications. Stone-age people indicated the integer *n by marking n adjacent vertical bars on a stone, which* is why the unary code is sometimes known as a stone-age binary and each of its *n or (n−1) 1's [or n or (n−1) zeros] is termed a stone-age bit.***
  - **The length of the unary code for the integer *n is therefore n bits. The two* rightmost columns of Table 1.2 show how the unary code can be extended to encode the nonnegative integers (which makes the codes more useful but also one bit longer).**

n	Code	Reverse	Alt. code	Alt reverse
0	–	–	0	1
1	0	1	10	01
2	10	01	110	001
3	110	001	1110	0001
4	1110	0001	11110	00001
5	11110	00001	111110	000001

Variable Length Coding
Phase-In Coding

- The phased-in codes consists of codewords of two lengths and may contribute something (although not much) to the compression of data where symbols have roughly equal probabilities.
  - Data where symbols have equal probabilities cannot be compressed by VLCs and is normally assigned fixed-length codes.
- Given *n data symbols,*
  - *Where* $n = 2^m$ *(implying that* $m = \log_2(n)$*), we can assign* them *m-bit codewords.*
  - If $2^m < n < 2^{m+1}$, *then* $\log_2(n)$ *is not an integer.* If we assign fixed-length codes to the symbols, each codeword would be $\log_2(n)$ *bits long,* but not all the codewords would be used.
    - The case *n = 1000 is a good example. In* this case, each fixed-length codeword is $\log_2(1000) = 10$ *bits long, but only 1000 out* of the 1024 possible codewords are used.
- In the phase-in coding, we try to assign two sets of codes to the *n symbols:*
  - the codewords of one set are *m bits long and may have several prefixes*
  - the codewords of the other set are *m+1 bits long and have different prefixes.*
  - *The average* length of such a code is between *m and m+1 bits and is shorter when there are more* short codewords.

Variable Length Coding
Phase-In Coding

- To construct the two sets of codes given a set of n data symbols (or simply the integers 0 through n−1) where $2^m \le n < 2^{m+1}$
  - For some integer m, we denote $n=2^m+p$ where $0 \le p < 2^m-1$ and also define $P=2^m-p$.
  - We construct 2p long (i.e., (m+1)-bit) codewords and P short, m-bit codewords.
  - The total number of codewords is always $2p+P=2p+2^m-p=(n-2^m)+2^m=n$
  - and there is an even number of long codewords.
  - The first P integers (symbols) 0 through P−1 receive the short codewords (m bits) and the remaining 2p integers (symbols) P through n−1 are assigned the long codewords.
  - The long codewords consist of p pairs, where each pair starts with the m-bit value P, P+1, . . . , P+p−1, followed by an extra bit, 0 or 1, to distinguish between the two codewords of a pair.

Variable Length Coding
Phase-In Coding

- **The left table lists the values of *m, p, and P for n=7, 8, 9, 15, 16,17. The right table lists* the actual codewords of them.**

n	m	$p = n - 2^m$	$P = 2^m - p$
7	2	3	1
8	3	0	8
9	3	1	7
15	3	7	1
16	4	0	16
17	4	1	15

i	$n = 7$	8	9	15	16	17
0	00	000	000	000	0000	0000
1	010	001	001	0010	0001	0001
2	011	010	010	0011	0010	0010
3	100	011	011	0100	0011	0011
4	101	100	100	0101	0110	0100
5	110	101	101	0110	0101	0101
6	111	110	110	0111	0110	0110
7		111	1110	1000	0111	0111
8			1111	1001	1000	1000
9				1010	1001	1001
10				1011	1010	1010
11				1100	1011	1011
12				1101	1100	1100
13				1110	1101	1101
14				1111	1110	1110
15					1111	11110
16						11111

Variable Length Coding
Golomb Coding

- The Golomb codes belong to a family of codes designed to encode integers with the assumption that the larger an integer, the lower its probability of occurrence.
  - The simplest code for this situation is the *unary* code.
- The Golomb code is actually a family of codes parameterized by an integer m > 0.
- In the Golomb code with parameter m, we represent an integer n > 0 using two numbers q and r, where:

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \quad r = n - qm, \text{ and } c = \lceil \log_2 m \rceil$$

  - q is the quotient and can take on values 0, 1, 2, ... and is represented by the unary code of q.
  - r is the remainder and can take on the values 0, 1, 2, ..., m−1
    - The first $2^c - m$ values of r are coded as unsigned integers, in $c-1$ bits each,
    - and the rest are coded in c bits each (encoding the value $r+2^c-m$ instead of r).

Variable Length Coding
Golomb Coding

Golomb code for $m = 5$

n	q	r	Codeword	n	q	r	Codeword
0	0	0	0 00	8	1	3	10 110
1	0	1	0 01	9	1	4	10 111
2	0	2	0 10	10	2	0	110 00
3	0	3	0 110	11	2	1	110 01
4	0	4	0 111	12	2	2	110 10
5	1	0	10 00	13	2	3	110 110
6	1	1	10 01	14	2	4	110 111
7	1	2	10 10	15	3	0	1110 00

- **The first 8−5 = 3 values of r (that is, r = 0, 1, 2) will be represented by the 2-bit binary representation of r, and the next two values (that is, r = 3, 4) will be represented by the3-bit representation of r +3.**
- **The quotient q is always represented by the unary code for q.**

Variable Length Coding
Golomb Coding

- **Decoding:** The Golomb codes are designed in this special way to facilitate their decoding. We first demonstrate the decoding for the simple case m=16 (m is a power of 2).
  - To decode, start at the left end of the code and count the number A of 1s preceding the first 0. The length of the code is A+c+1 bits (for m=16, this is A+5 bits). If we denote the rightmost five bits of the code by R, then the value of the code is 16A+R. This simple decoding reflects the way the code was constructed. To encode n with m=16, start by dividing it by 16 to get n=16A+R, then write A 1s followed by a single 0, followed by the 4-bit representation of R.
- In summary, given a binary string, we can employ the method of run-length encoding to compress it with Golomb codes in the following steps:
  - (1) count the number of zeros and ones,
  - (2) compute the probability p of a zero,
  - (3) compute m using Equation :

$$m = \left\lceil -\frac{\log_2(1+p)}{\log_2 p} \right\rceil$$

  - (4) construct the family of Golomb codes for m, and
  - (5) for each run-length of n zeros, write the Golomb code of n on the compressed stream.

Variable Length Coding
Golomb Coding

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
c	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
$2^c - m$	0	1	0	3	2	1	0	7	6	5	4	3	2	1	0

m/n	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0|0	0|1	10|0	10|1	110|0	110|1	1110|0	1110|1	11110|0	11110|1	111110|0	111110|1	1111110|0
3	0|0	0|10	0|11	10|0	10|10	10|11	110|0	110|10	110|11	1110|0	1110|10	1110|11	11110|0
4	0|00	0|01	0|10	0|11	10|00	10|01	10|10	10|11	110|00	110|01	110|10	110|11	11110|00
5	0|00	0|01	0|10	0|110	0|111	10|00	10|01	10|10	10|110	10|111	110|00	110|01	110|10
6	0|00	0|01	0|100	0|101	0|110	0|111	10|00	10|01	10|100	10|101	10|110	10|111	110|00
7	0|00	0|010	0|011	0|100	0|101	0|110	0|111	10|00	10|010	10|011	10|100	10|101	10|110
8	0|000	0|001	0|010	0|011	0|100	0|101	0|110	0|111	10|000	10|001	10|010	10|011	10|100
9	0|000	0|001	0|010	0|011	0|100	0|101	0|110	0|1110	0|1111	10|000	10|001	10|010	10|011
10	0|000	0|001	0|010	0|011	0|100	0|101	0|1100	0|1101	0|1110	0|1111	10|000	10|001	10|010
11	0|000	0|001	0|010	0|011	0|100	0|1010	0|1011	0|1100	0|1101	0|1110	0|1111	10|000	10|001
12	0|000	0|001	0|010	0|011	0|1000	0|1001	0|1010	0|1011	0|1100	0|1101	0|1110	0|1111	10|000
13	0|000	0|001	0|010	0|0110	0|0111	0|1000	0|1001	0|1010	0|1011	0|1100	0|1101	0|1110	0|1111

Some Golomb Codes for $m = 2$ Through 13.

Run-Length Encoding (RLE)
(inter data redundancy)

- The consecutive recurrent symbols are usually called ***runs*** in a sequence of symbols.
- **run-length algorithm** used to reduce the size of a repeating string of symbols (i.e., runs):
- Encodes a run of symbols into two bytes:

(count,symbol)

111110000001 → (5,1)(6,0)(1,1)

aaabbbbbbcc → (3,a)(6,b)(2,c)

- **RLE** can compress any type of data but cannot achieve high compression ratios compared to other compression methods.

Run Length Encoding
Hardware data compression (HDC)

- **The so-called HDC (hardware data compression) algorithm is a simple run-length algorithm.**
  - **used by tape drives connected to IBM computer systems,**
  - **similar algorithm used in the IBM System Network Architecture (SNA) standard for data communications are still in use today.**

- **In this form of run-length coding two types of control characters are exist:**
  - **One is a flag for run sequences $R=(r_2, r_3, \ldots, r_k)$.**
  - **The other is a flag for non-run sequences $N=(n_1, n_2, \ldots, n_k)$.**
  - **We assume each run or the non-repeating symbol sequence contains no more than k symbols (i.e. k=64).**
  - **For the non-run parts, repeating control flags, depends on the run-length, followed by the repeating symbol are used .**
    - **For example, VVVV can be replaced by r_4V**
  - **For the non-run parts, non-repeating control flags are used which *n* are followed by the length of the longest non-repeating symbols.**
    - **For example, ABCDEFG will be replaced by n_7ABCDEFG**

Move-to-Front Coding

- The basic idea of this method is to maintain the alphabet A of symbols as a list where frequently-occurring symbols are located near the front.

- A symbol *s* is encoded as the number of symbols that precede it in this list.
  - Thus if A=(*t,h,e,s,...*) and the next symbol in the input stream to be encoded is the *e*, it will be encoded as 2, since it is preceded by two symbols.

- There are several possible variants to this method;

Move-to-Front Coding

- **The most basic of them adds one more step:**
  - **After symbol *s* is encoded, it is moved to the front of list A. Thus, after encoding the *e*, the alphabet is modified to A=(*e,t,h,s,…*).**
  - **This move-to-front step reflects the expectation that once *e* has been read from the input stream, it will be read many more times and will, at least for a while, be a common symbol.**
  - **The move-to-front method is locally adaptive, since it adapts itself to the frequencies of symbols in local areas of the input stream.**
  - **The method produces good results if the input stream satisfies this expectation, i.e., if it contains concentrations of identical symbols (if the local frequency of symbols changes significantly from area to area in the input stream). We call this *the concentration property*.**

Move-to-Front Coding

- Here is an example that illustrates the move-to-front idea. It assumes the alphabet is A=(a, b, c, d, m, n, o, p) and the input stream is abcddcbamnopponm:
  - With move-to-front it is encoded as C=(0,1,2,3,0,1,2,3,4,5,6,7,0,1,2,3) (Table a).
  - Without move-to-front it is encoded as C′=(0,1,2,3,3,2,1,0,4,5,6,7,7,6,5,4) (Table b).
  - Both C and C′ contain codes in the same range [0,7], but the elements of C are smaller on average, since the input starts with a concentration of abcd and continues with a concentration of mnop. The average value of C is 2.5, while C′ is 3.5

a	abcdmnop	0
b	abcdmnop	1
c	bacdmnop	2
d	cbadmnop	3
d	dcbamnop	0
c	dcbamnop	1
b	cdbamnop	2
a	bcdamnop	3
m	abcdmnop	4
n	mabcdnop	5
o	nmabcdop	6
p	onmabcdp	7
p	ponmabcd	0
o	ponmabcd	1
n	opnmabcd	2
m	nopmabcd	3
	mnopabcd	

(a)

a	abcdmnop	0
b	abcdmnop	1
c	abcdmnop	2
d	abcdmnop	3
d	abcdmnop	3
c	abcdmnop	2
b	abcdmnop	1
a	abcdmnop	0
m	abcdmnop	4
n	abcdmnop	5
o	abcdmnop	6
p	abcdmnop	7
p	abcdmnop	7
o	abcdmnop	6
n	abcdmnop	5
m	abcdmnop	4

(b)

Move-to-Front Coding

- **Here is an example that illustrates the move-to-front idea. It assumes the alphabet is A=(a, b, c, d, m, n, o, p) and the input stream is abcdmnopabcdmnop:**

- **With move-to-front it is encoded as C=(0,1,2,3,4,5,6,7,7,7,7,7,7,7,7,7) (Table c).**
- **Without move-to-front it is encoded as C′=(0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7) (Table d).**
- **The average of C is now 5.25, greater than that of C′, which is 3.5.**
- **The move-to-front rule creates a worse result in this case, because the input does not contain concentrations of identical symbols (it does not satisfy the concentration property).**

a	abcdmnop	0
b	abcdmnop	1
c	bacdmnop	2
d	cbadmnop	3
m	dcbamnop	4
n	mdcbanop	5
o	nmdcbaop	6
p	onmdcbap	7
a	ponmdcba	7
b	aponmdcb	7
c	baponmdc	7
d	cbaponmd	7
m	dcbaponm	7
n	mdcbapon	7
o	nmdcbapo	7
p	onmdcbap	7
	ponmdcba	

(c)

a	abcdmnop	0
b	abcdmnop	1
c	abcdmnop	2
d	abcdmnop	3
m	abcdmnop	4
n	abcdmnop	5
o	abcdmnop	6
p	abcdmnop	7
a	abcdmnop	0
b	abcdmnop	1
c	abcdmnop	2
d	abcdmnop	3
m	abcdmnop	4
n	abcdmnop	5
o	abcdmnop	6
p	abcdmnop	7

(d)

Quantization

- The dictionary definition of the term "quantization" is "to restrict a variable quantity to discrete values rather than to a continuous set of values."
- In the field of data compression, quantization is employed in two ways:
  - 1) If the data to be compressed is in the form of large numbers, quantization is used to convert it to small numbers.
    - Small numbers take less space than large ones, so quantization generates compression.
    - On the other hand, small numbers generally contain less information than large numbers, so quantization results in lossy compression.
  - 2) If the data to be compressed is analog (i.e., a voltage that changes with time) quantization is used to digitize it into small numbers.
    - The smaller the numbers the better the compression, but also the greater the loss of information.
    - This aspect of quantization is exploited by several speech compression methods.

Scalar Quantization

- In the discussion here we assume that the data to be compressed is in the form of numbers.
- The first example is naive discrete quantization of an input stream of 8-bit numbers:
  - We can simply delete the least-significant four bits of each data item.
    - This is one of those rare cases where the compression factor (=2) is known in advance and does not depend on the data.
  - The input data consists of 256 different symbols, while the output data consists of just 16 different symbols.
  - This method is simple but not very practical because too much information is lost in order to get the unimpressive compression factor of 2.

Scalar Quantization

- **In order to develop a better approach we assume again that the data consists of 8-bit numbers, and that they are unsigned.**
  - **Thus, input symbols are in the range [0, 255] (if the input data is signed, input symbols have values in the range [−128,+127]).**
  - **We select a spacing parameter *s* and compute the sequence of uniform quantized values *0, s, 2s,. . . ,ks* such that (k+1)s>255 and ks ≤255.**
  - **Each input symbol n is quantized by converting it to the nearest value in this sequence.**
    - **Selecting s = 3 produces the uniform sequence 0,3,6,9,12,…,252,255.**
    - **Selecting s = 4 produces the uniform sequence 0,4,8,12,…,252,255.**

Scalar Quantization

- A similar approach is to select quantized values such that any number in the range [0, 255] will be no more than d units distant from one of the data values that are being quantized.
  - This is done by dividing the range into segments of size 2d+1 and centering them on the range [0, 255].
    - If d=16, then the range [0, 255] is divided into 7 segments of size 33 each, with 25 numbers remaining.
    - We can thus start the first segment 12 numbers from the start of the range, which produces the 10-number sequence 12, 33, 45, 78, 111, 144, 177, 210, 243, and 255.
    - Any number in the range [0, 255] is at most 16 units distant from any of these numbers.
    - If we want to limit the quantized sequence to just eight numbers (so each can be expressed in 3 bits) we can apply this method to compute the sequence 8, 41, 74, 107, 140, 173, 206, and 239.
- The quantized sequences above make sense in cases where each symbol appears in the input data with equal probability (cases where the source is memoryless).

Scalar Quantization

- **If the input data is not uniformly distributed, the sequence of quantized values should be distributed in the same way as the data.**
- **Imagine an input stream of 8-bit unsigned data items most of which are zero or close to zero and few are large. A good sequence of quantized values for such data should have the same distribution, many small values and few large ones.**
- **One way of computing such a sequence is to select a value for the length parameter *L* and to construct a "window" of the form $1\underbrace{b\ldots bb}_{L}$**
- **where each *b* is a bit, and place it under each of the 8 bit positions of a data item. If the window sticks out on the right, some of the *L* bits are truncated. As the window is shifted to the left, bits of 0 are appended to it.**
- **The numbers 0 and 255 should be manually added to such a quasi-logarithmic sequence (for L=2) to make it more general.**

bbbbbbbb		bbbbbbbb	
1	1	.	
10	2	.	
11	3	.	
100	4	100|000	32
101	5	101|000	40
110	6	110|000	48
111	7	111|000	56
100|0	8	100|0000	64
101|0	10	101|0000	80
110|0	12	110|0000	96
111|0	14	111|0000	112
100|00	16	100|00000	128
101|00	20	101|00000	160
110|00	24	110|00000	192
111|00	28	111|00000	224

Vector Quantization

- **Vector quantization is a generalization of the scalar quantization method.**
  - **It is used for both image and audio compression.**
  - **In practice, vector quantization is commonly used to compress data that has been digitized from an analog source, such as audio samples and scanned images. Such data is called *digitally sampled analog data* (DSAD).**
- **Vector quantization is based on the fact that compression methods that compress strings (or blocks) rather than individual symbols can produce better results.**
  - **Such methods are called *block coding* *and, as the block length* increases, compression methods based on block coding can approach the entropy when compressing stationary information sources**

Vector Quantization

- **We start with a simple, intuitive vector quantization method for image compression.**
  - **Given an image, we divide it into small blocks of pixels, typically 2 *× 2 or 4×4.***
  - ***Each* block is considered as a vector.**
  - **The encoder maintains a list (called a *codebook) of vectors* and compresses each block by writing on the compressed stream a pointer to the block in the codebook.**
  - **The decoder has the easy task of reading pointers, following each pointer to a block in the codebook, and appending the block to the image-so-far.**

- **Thus, vector quantization is an asymmetric compression method.**

Vector Quantization

- In the case of 2×2 blocks, each block (vector) consists of four pixels. If each pixel is one bit, then a block is four bits long and there are only $2^4 = 16$ different blocks.
- It is easy to store such a small, permanent codebook in both encoder and decoder.
- However, a pointer to a block in such a codebook is, of course, four bits long, so there is no compression gain by replacing blocks with pointers.
- If each pixel is k bits, then each block is 4k bits long and there are 2^{4k} different blocks.
- The codebook grows very fast with k (for k = 8, for example, it is $256^4 = 2^{32} = 4$ Giga entries) but the point is that we again replace a block of 4k bits with a 4k-bit pointer, resulting in no compression gain. This is true for blocks of any size.

Vector Quantization

- **Once it becomes clear that this simple method does not work, the next thing that comes to mind is that any given image may not contain every possible block.**
  - **Given 8-bit pixels, the number of 2×2 *blocks is* $2^{2\times2\times8} = 2^{32} \approx$ *4.3 billion, but any particular* image may contain only a few million pixels and a few thousand different blocks.**

- **Thus, our next version of vector quantization starts with an empty codebook and scans the image block by block. The codebook is searched for each block.**
  - **If the block is already in the codebook, the encoder outputs a pointer to the block in the (growing) codebook.**
  - **If the block is not in the codebook, it is added to the codebook and a pointer is output.**

Vector Quantization

- The problem with this simple method is that each block added to the codebook has to be written on the compressed stream.
  - This greatly reduces the effectiveness of the method and may lead to low compression and even to expansion.
- There is also the small added complication that the codebook grows during compression, so the pointers get longer, but this is not difficult for the decoder to handle.
- These problems are the reason why image vector quantization is lossy. If we accept lossy compression, then the size of the codebook can be greatly reduced.

Vector Quantization

- **Here is an intuitive lossy method for image compression by vector quantization.**
  - **Analyze a large number of different "training" images and find the *B most-common blocks.***
  - ***Construct a* codebook with these *B blocks and embed it into both encoder and decoder.***
  - ***Each entry* of the codebook is a block.**
  - **To compress an image, scan it block by block, and for each block find the codebook entry that best matches it, and output a pointer to that entry.**
- **The size of the pointer is, of course, $\log_2 B$, *so the compression ratio (which is known* in advance) is:**

$$C_R = \frac{\lceil \log_2 B \rceil}{blocksize}$$

Vector Quantization

- **One problem with this approach is how to match image blocks to codebook entries. Here are a few common similarity or distance measures.**
  - **The "distance" between a block of image pixels (B) and a codebook entry (C) by d(B,C) can be measured in different ways as follows:**

$$d_1(B,C) = \sum_{i=1}^{n} |b_i - c_i|$$

$$d_2(B,C) = \sum_{i=1}^{n} (b_i - c_i)^2$$

$$d_3(B,C) = MAX_{i=1}^{n} |b_i - c_i|$$

  - **each of block of image pixels B and codebook entry C is a vector of size n bits (B=(b_1, b_2, . . . , b_n) and C=(c_1, c_2, . . . , c_n)).**

Vector Quantization

- The third measure $d_3(B,C)$ is easy to interpret. It finds the component where B and C differ most, and it returns this difference.

- The first two measures are easy to visualize in the case n = 3.
  - Measure $d_1(B,C)$ becomes the distance between the two 3D (three-dimensional) vectors B and C when we move along the coordinate axes.
  - Measure $d_2(B,C)$ becomes the Euclidean (straight line) distance between the two vectors.

- The quantities $d_i(B,C)$ can also be considered measures of distortion.

Vector Quantization

- **Another problem with this approach is the quality of the codebook.**
  - **In the case of 2×2 *blocks with 8-bit pixels the total number of blocks is* $2^{32} \approx 4.3$ *billion.***
  - ***If we decide* to limit the size of our codebook to, say, a million entries, it will contain only 0.023% of the total number of blocks (and still be 32 million bits, or about 4 MB long).**
  - **Using this codebook to compress an "atypical" image may result in a large distortion regardless of the distortion measure used.**
  - **When the compressed image is decompressed, it may look so different from the original as to render our method useless.**
- **A natural way to solve this problem is to modify the original codebook entries in order to *adapt them to the* particular image being compressed.**

Vector Quantization

- Such an algorithm has been developed by Linde, Buzo, and Gray and is known as the **LBG algorithm**.
  - The final codebook will have to be included in the compressed stream, but since it has been adapted to the image, it may be small enough to yield a good compression ratio, yet close enough to the image blocks to produce an acceptable decompressed image.
  - It is the basis of many vector quantization methods for the compression of images and sound.

- Scalar/Vector quantization is an example of a lossy compression method, where it is easy to control the trade-off between compression ratio and the amount of loss. However, because it is so simple, its use is limited to cases where much loss can be tolerated.

Recursive Range Reduction

- The recursive range reduction (3R) method described here generates decent compression, is easy to implement, and its performance is independent of the amount of data to be compressed.
  - These features make it an attractive candidate for compression in embedded systems, low-cost microcontrollers, and other applications where space is limited or resources are constrained.

- The compression efficiency depends on the data, but not on the amount of data available for compression. The method does not improve by applying it to vast quantities of data.

Recursive Range Reduction

- **The method is described first for a sorted list of integers, where its application is simplest and no recursion is required.**
  - **Given a list of integers, we first eliminate the effects of the sign bit.**
    - **We either rotate each integer to move the sign bit to the least-significant position (a folding) or add an offset to all the integers, so they become nonnegative.**
  - **The list is then sorted in nonincreasing order. The first element of the list is the largest one, and we assume that its most-significant bit is a 1 (i.e., it has no extra zeros on the left).**
  - **It is obvious that the integers that follow the first element may have some mostsignificant zero bits, and the heart of this method is to eliminate most of those bits while leaving enough information for the decoder to restore them.**

Recursive Range Reduction

- **The first item in the compressed stream is a header with the length *L* of the largest integer. This length is stored in a fixed-size field whose size is sufficient for any data that may be encountered.**
  - **In the examples here, this size is four bits, allowing for integers up to 16 bits long. Once the decoder inputs *L*, it knows the length of the first integer. The MSB of this integer is a *1*, so this bit can be eliminated and the first integer can be emitted in *L*−*1* bits.**

- **The next integer is written on the output as an *L-bit* number, thereby allowing the decoder to read it unambiguously. The decoder then checks the most-significant bits of the second integer:**
  - **If the leftmost k bits are zeros, then the decoder knows that the next integer (i.e., the third one) was written without its k leftmost zeros and thus occupies the next *L-k* bits in the compressed stream.**
  - **This is how range reduction compresses a sorted list of nonnegative integers.**

Recursive Range Reduction

- **An example is:**
  - **The data consists of ten 7-bit integers.**
  - **The 4-bit length field is set to 6, indicating 7-bit integers (notice that L cannot be zero, so 6 indicates a length of seven bits).**
  - **The first integer is emitted minus its leftmost bit.**
  - **And the next integer is output in its entirety.**
  - **The third integer is also emitted as is, but its leftmost zero (listed in bold) indicates to the decoder that the following (fourth) integer will have only six bits.**

Data	RR code
	$0110 = 6$
1011101	011101
1001011	1001011
0110001	0110001
0101100	101100
0001110	001110
0001101	1101
0001100	1100
0001001	1001
0000010	0010
0000001	01
70	53

- **The total size of the ten integers is 70 bits, and this should be compared with the 53 bits created by Range Reduction codes.**

Recursive Range Reduction

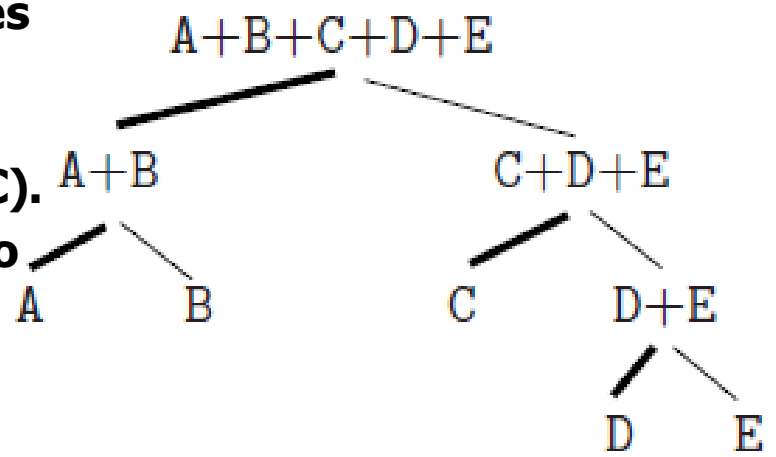
- **Now for unsorted data:** Given an unsorted list of integers,
  - we can sort it,
  - compress it with Range Reduction ,
  - and then include side information about the sort,
- So that the decoder can unsort the data correctly after decompressing it.

Recursive Range Reduction

- **An efficient method is required to specify the relation between a list of numbers and its sorted version, and the method described here is referred to as recursive range reduction or 3R:**
  - **The 3R algorithm involves the creation of a binary tree where each path from the root to a leaf is a non-increasing list of integers.**
  - **Each path is encoded with 3R separately, which requires a recursive algorithm (hence the word "recursive" in the algorithm's name). Each pair of consecutive integers becomes a subtree at the lowest level.**
  - **Each paths from the root to a leaf is a sorted list, and it is compressed with 3R.**
  - **The rule is to follow every path from the root and select the nodes along edges that go to the left (or only those that go to the right).**

Recursive Range Reduction

- **Given the five unsorted integers A through E, a binary tree is constructed. Each of the five paths from the root to a leaf is a sorted list. Thus, we write the following values on the output:**
  - **1) a header with the length of the root,**
  - **2) the 3R-encoded path (root, A+B, A),**
  - **3) the value of node C [after the path (root, C+D+E, C) is encoded], and**
  - **4) the value of node D [after the path (root, C+D+E, D+E, D) is encoded].**
- **The decoder reads the header and first path (root, A+B, A).**
  - **It decodes the path to obtain the three values and subtracts (A + B) *– A to obtain B.***
  - ***It then subtracts* Root *–(A+B) to obtain C+D+E and decode the path (root, C+D+E,* C).**
  - **Next, the decoder subtracts C from C+D+E to obtain D+E, it inputs D, and decodes path (root, C+D+E, D+E, D).**
  - **One more subtraction yields E**



Statistical Methods

Statistical Methods

- **Statistical data compression methods employ variable-length codes, with the shorter codes assigned to symbols or groups of symbols that appear more often in the data (have a higher probability of occurrence).**

- **The statistical compression methods try assign codes with the minimum average size by employing different algorithms such as:**
  - **Shannon-Fano Coding**
  - **Huffman Coding**
  - **MNP Coding (MNP5 and MNP7)**
  - **Arithmetic Coding**
  - **The QM Coder**
  - **PPM**
  - **PAQ**

Shannon-Fano Coding

- **Shannon-Fano coding, named after Claude Shannon and Robert Fano, was the first algorithm to construct a set of the best variable-length codes.**
- **We start with a set of *n symbols with known probabilities (or frequencies) of occurrence.***
  - **The symbols are first arranged in descending order of their probabilities.**
  - **The set of symbols is then divided into two subsets that have the same (or almost the same) probabilities.**
  - **All symbols in one subset get assigned codes that start with a 0, while the codes of the symbols in the other subset start with a 1.**
  - **Each subset is then recursively divided into two subsubsets of roughly equal probabilities, and the second bit of**
  - **all the codes is determined in a similar way.**
  - **When a subset contains just two symbols, their codes are distinguished by adding one more bit to each.**
  - **The process continues until no more subsets remain.**

Shannon-Fano Coding

- This Example illustrates the Shannon-Fano algorithm for a seven-symbol alphabet. Notice that the symbols themselves are not shown, only their probabilities.
- The first step splits the set of seven symbols into two subsets:
  - one with two symbols and a total probability of 0.45:
    - The two symbols in the first subset are assigned codes that start with 1, so their final codes are 11 and 10.
  - the other with the remaining five symbols and a total probability of 0.55.
    - The second subset is divided, in the second step, into two symbols (with total probability 0.3 and codes that start with 01) and three symbols (with total probability 0.25 and codes that start with 00).
    - Step three divides the last three symbols into 1 (with probability 0.1 and code 001) and 2 (with total probability 0.15 and codes that start with 000).

	Prob.	Steps					Final
1.	0.25	1	1				:11
2.	0.20	1	0				:10
3.	0.15	0		1	1		:011
4.	0.15	0		1	0		:010
5.	0.10	0		0		1	:001
6.	0.10	0		0		0	1 :0001
7.	0.05	0		0		0	0 :0000

- The average size of this code is $0.25\times2+0.20\times2+0.15\times3+0.15\times3+0.10\times3+0.10\times4+0.05\times4 = 2.7$ bits/symbol. This is a good result because the entropy is 2.6.

Huffman Coding

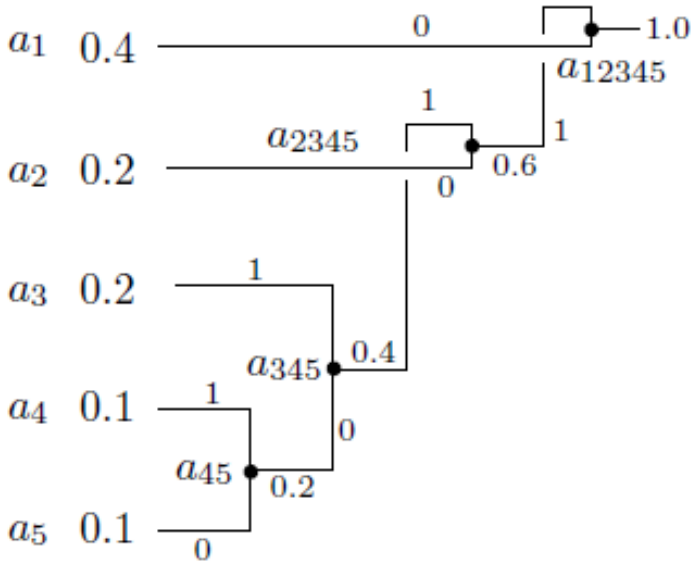
- **Huffman coding is a popular method for data compression. It serves as the basis for several popular programs run on various platforms.**
  - **Some programs use just the Huffman method, while others use it as one step in a multistep compression process.**
  - **The Huffman method is somewhat similar to the Shannon-Fano method.**
  - **It generally produces better codes, and like the Shannon-Fano method, it produces the best code when the probabilities of the symbols are negative powers of 2.**
  - **The main difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the codes from right to left).**
  - **Since its development, in 1952, by D. Huffman, this method has been the subject of intensive research into data compression.**

Huffman Coding

- The Huffman algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities.
- It then constructs a tree, with a symbol at every leaf, from the bottom up.
- This is done in steps, where at each step the two symbols with smallest probabilities are selected,
  - added to the top of the partial tree,
  - deleted from the list,
  - and replaced with an auxiliary symbol representing the two original symbols.
- When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete.
- The tree is then traversed to determine the codes of the symbols.

Huffman Coding

- **This process is best illustrated by an example. Given five symbols with probabilities as shown in Figure, they are paired in the following order:**
  - **1. a_4 is combined with a_5 and both are replaced by the combined symbol a_{45}, whose probability is 0.2.**
  - **2. There are now four symbols left, a1, with probability 0.4, and a_2, a_3, and a_{45}, with probabilities 0.2 each. We arbitrarily select a_3 and a_{45}, combine them, and replace them with the auxiliary symbol a_{345}, whose probability is 0.4.**
  - **3. Three symbols are now left, a_1, a_2, and a_{345}, with probabilities 0.4, 0.2, and 0.4, respectively. We arbitrarily select a_2 and a_{345}, combine them, and replace them with the auxiliary symbol a_{2345}, whose probability is 0.6.**
  - **4. Finally, we combine the two remaining symbols, a_1 and a_{2345}, and replace them with a_{12345} with probability 1.**

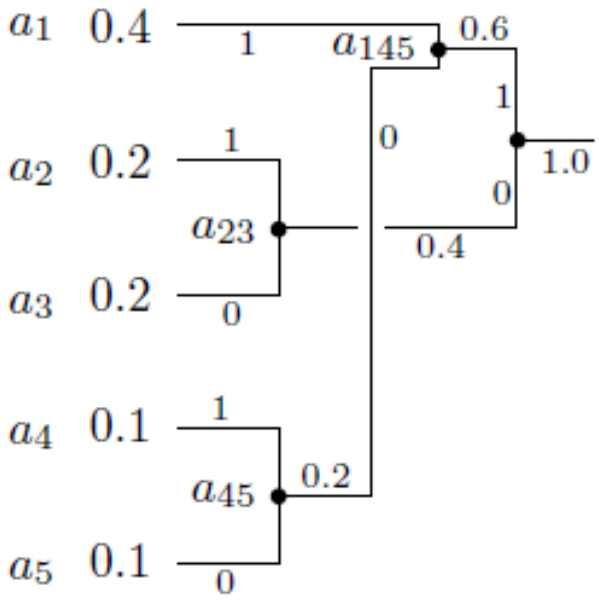


- **To assign the codes, we arbitrarily assign a bit of 1 to the top edge, and a bit of 0 to the bottom edge, of every pair of edges.**

Huffman Coding

- **This results in the codes 0, 10, 111, 1101, and 1100. The assignments of bits to the edges is arbitrary.The average size of this code is 0*.4 × 1 + 0.2 × 2 + 0.2 × 3 + 0.1 × 4 + 0.1 ×4 = 2.2***

  - **The Huffman code is not unique.**
  - **Some of the steps above were chosen arbitrarily, since there were more than two symbols with smallest probabilities.**
  - **Figure shows how the same five symbols can be combined differently to obtain a different Huffman code (11, 01, 00, 101, and 100).**
  - **The average size of this code is 0.4×2+0.2×2 + 0.2×2+0.1×3+0.1×3=2.2 bits/symbol, the same as the previous code.**



- **which of the different Huffman codes for a given set of symbols is best?**
  - **The answer, while not obvious, is simple: The best code is the one with the smallest variance. The variance of a code measures how much the sizes of the individual codes deviate from the average size. (the first code is better)**

Huffman Decoding

- **Before starting the compression of a data stream, the compressor (encoder) has to determine the codes.**

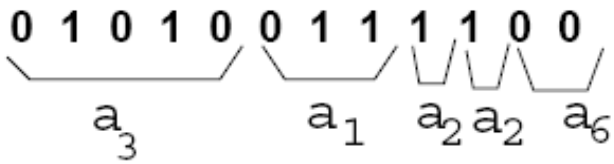
- **It means that the probabilities (or frequencies of occurrence) of the symbols have to be written, as side information, on the compressed stream, so that any Huffman decompressor (decoder) will be able to decompress the stream.**
  - **This is easy, since the frequencies are integers and the probabilities can be written as scaled integers.**
  - **It normally adds just a few hundred bytes to the compressed stream.**
  - **It is also possible to write the variable-length codes themselves on the stream, but this may be awkward, because the codes have different sizes.**
  - **It is also possible to write the Huffman tree on the stream, but this may require more space than just the frequencies.**

Huffman Decoding

- **In any case, the decoder must know what is at the start of the stream, read it, and construct the Huffman tree for the alphabet.**
  - **Only then can it read and decode the rest of the stream.**
  - **The algorithm for decoding is simple.**
  - **Start at the root and read the first bit off the compressed stream.**
  - **If it is zero, follow the bottom edge of the tree; if it is one, follow the top edge.**
  - **Read the next bit and move another edge toward the leaves of the tree.**
  - **When the decoder gets to a leaf, it finds the original, uncompressed code of the symbol (normally its ASCII code), and that code is emitted by the decoder.**
  - **The process starts again at the root with the next bit.**

Huffman Coding/Decoding

- **After the code has been created, *coding/decoding* can be implemented using a look-up table.**
- **Table should be included in the compressed data.**
- **Note that decoding is done unambiguously.**



Original source		
Sym.	Prob.	Code
a_2	0.4	1
a_6	0.3	00
a_1	0.1	011
a_4	0.1	0100
a_3	0.06	01010
a_5	0.04	01011

Fast Huffman Decoding

- **Decoding a Huffman-compressed file by sliding down the code tree for each symbol is conceptually simple, but slow.**
  - **The compressed file has to be read bit by bit and the decoder has to advance a node in the code tree for each bit.**
- **The fast Huffman decoding uses preset partial-decoding tables.**
  - **These tables depend on the particular Huffman code used, but not on the data to be decoded.**
- **The compressed file is read in chunks of k bits**
  - **(where k is normally 8 or 16 but can have other values)**
- **the current chunk is used as a pointer to a table.**
  - **The table entry that is selected in this way can decode several symbols and it also points the decoder to the table to be used for the next chunk.**
- **This process continues until the end of the encoded input.**

```
i←0; output←null;
repeat
   j←input next chunk;
   (s,i)←Table_i[j];
   append s to output;
until end-of-input
```

Fast Huffman Decoding Example

- As an example, consider the Huffman code, where the five codewords are 0, 10, 111, 1101, and 1100.
  - The string of symbols $a_1a_1a_2a_4a_3a_1a_5$ ... is compressed by this code to the string 0|0|10|1101|111|0|1100 ....
  - We select k= 3 and read this string in 3-bit chunks 001|011|011|110|110|0 ....
  - Examining the first chunk, it is easy to see that it should be decoded into a_1a_1 followed by the single bit 1 which is the prefix of another codeword.
  - The first chunk is 001 = 1_{10}, so we set entry 1 of the first table (table 0) to the pair (a_1a_1, 1).
  - When chunk 001 is used as a pointer to table 0, it points to entry 1, which immediately provides the decoder with the two decoded symbols a_1a_1 and also directs it to use table 1 for the next chunk.
  - Table 1 is used when a partially-decoded chunk ends with the single-bit prefix 1. The next chunk is $011=3_{10}$, so entry 3 of table 1 corresponds to the encoded bits 1|011.
  - Again, it is easy to see that these should be decoded to a_2 and there is the prefix 11 left over. Thus, entry 3 of table 1 should be (a_2, 2).
  - It provides the decoder with the single symbol a_2 and also directs it to use table 2 next (the table that corresponds to prefix 11).
  - The next chunk is again $011=3_{10}$, so entry 3 of table 2 corresponds to the encoded bits 11|011. It is again obvious that these should be decoded to a_4 with a prefix of 1 left over.

Fast Huffman Decoding Example

- **This table lists the four tables required to decode this code.**
  - **It is easy to see that they correspond to the prefixes Λ (null), 1, 11, and 110.**
  - **A quick glance at Figure shows that these correspond to the root and the four interior nodes of the Huffman code tree.**
  - **Thus, each partial-decoding table corresponds to one of the four prefixes of this code.**
- **The number of partial-decoding tables (m) equals the number of interior nodes (plus the root) which is one less than the number of symbols of the alphabet (m=N-1).**

$T_0 = \Lambda$			$T_1 = 1$			$T_2 = 11$			$T_3 = 110$		
000	$a_1a_1a_1$	0	1|000	$a_2a_1a_1$	0	11|000	a_5a_1	0	110|000	$a_5a_1a_1$	0
001	a_1a_1	1	1|001	a_2a_1	1	11|001	a_5	1	110|001	a_5a_1	1
010	a_1a_2	0	1|010	a_2a_2	0	11|010	a_4a_1	0	110|010	a_5a_2	0
011	a_1	2	1|011	a_2	2	11|011	a_4	1	110|011	a_5	2
100	a_2a_1	0	1|100	a_5	0	11|100	$a_3a_1a_1$	0	110|100	$a_4a_1a_1$	0
101	a_2	1	1|101	a_4	0	11|101	a_3a_1	1	110|101	a_4a_1	1
110	–	3	1|110	a_3a_1	0	11|110	a_3a_2	0	110|110	a_4a_2	0
111	a_3	0	1|111	a_3	1	11|111	a_3	2	110|111	a_4	2

Adaptive Huffman Coding

- In static Huffman coding, the probability distribution remains unchanged during the process of encoding and decoding.
- An alphabet and probability distribution is often applied based on the previous experience.
- Such an estimated model can compromise the compression quality substantially.
- The amount of the loss in compression quality depends very much on how much the probability distribution of the source differs from the estimated probability distribution.

Adaptive Huffman Coding

- **Adaptive Huffman coding algorithms improve the compression ratio by applying to the model the statistics based on the source content seen from the immediate past.**

- **In the adaptive Huffman coding, an alphabet and its frequency table are dynamically adjusted after reading each symbol during the process of compression or decompression.**

- **Compared to static Huffman coding, the adaptive model is much more close to the real situation of the source after initial steps.**

Adaptive Huffman Coding

- The Huffman tree is also updated based on the alphabet and frequencies dynamically.
- When the encoder and decoder are at different locations, both maintain an identical Huffman tree for each step independently. Therefore, there is no need transferring the Huffman tree.
- During the compression process, the Huffman tree is updated each time after a symbol is read. The codeword(s) for the symbol is output immediately.
- For convenience of discussion, the frequency of each symbol is called the weight of the symbol to reflect the change of the frequency count at each stage.

Adaptive Huffman Coding

- The output of the adaptive Huffman encoding consists of Huffman codewords as well as fixed length codewords.
  - For each input symbol, the output can be a Huffman codeword based on the Huffman tree in the previous step or a codeword of a fixed length code (such as ASCII) for new symbol.
- Using a fixed length codeword as the output is necessary when a new symbol is read for the first time.
  - In this case, the Huffman tree does not include the symbol yet.
  - It is therefore reasonable to output the uncompressed version of the symbol.
  - In the encoding process if the input symbol has been seen befor, it outputs a Huffman codeword.
- However, a mixture of the fixed length and variable length codewords can cause problems in the decoding process.
  - The decoder needs to know whether the codeword should be decoded according to a Huffman tree or by a fixed length codeword before taking a right approach.
  - A special symbol as a flag, therefore, is used to signal a switch from one type of codeword to another.

Adaptive Huffman Encoding Algorithm

1. Initialize the Huffman tree T containing the only node SHIFT.
2. While more characters remain do
3. *S <– next_symbol_in_text ( )*
4. If *s* has been seen before then
   output *h(s)*
5. Else
   output h(SHIFT) followed by *g(s)*
6. End If
7. *T <– update_tree (T)*
8. End While

- *next_symbol_in_text()* is a function that reads one symbol from the input sequence.
- *update_tree ()* is a function which update the weight of symbols and then recompute the Huffman tree for the new set of weights or symbols.

Adaptive Huffman Decoding Algorithm

1. Initialize the Huffman tree T containing the only node SHIFT.
2. While more characters remain do
3. *S <– huffman_next_symbol( )*
4. If *s =SHIFT* then
   s <– *read_unencoded_symbol()*
5. Else
   output*(s)*
6. End If
7. *T <– update_tree (T)*
8. End While

- The function *huffman_next_symbol()* reads bits from the input until it reaches a leaf node and returns the symbol with which that leaf is labelled.
- The function *read_unencoded_symbol()* simply reads the next unencoded symbol from the input.

Huffman Coding Advantages

- **Adaptive Huffman coding has the advantage of requiring no preprocessing and the low overhead of using the uncompressed version of the symbols only at their first occurrence.**
- **The algorithms can be applied to other types of files in addition to text files.**
- **The symbols can be objects or bytes in executable files.**
- **Huffman coding, either static or adaptive, has two disadvantages that remain unsolved:**

Huffman Coding Disadvantages

- **Disadvantage 1:** It is not optimal unless all probabilities are negative powers of 2. This means that there is a gap between the average number of bits and the entropy in most cases.
  - Recall the particularly bad situation for binary alphabets. Although by grouping symbols and extending the alphabet, one may come closer to the optimal, the blocking method requires a larger alphabet to be handled. Sometimes, extended Huffman coding is not that effective at all.
- **Disadvantage 2:** Despite the availability of some clever methods for counting the frequency of each symbol reasonably quickly, it can be very slow when rebuilding the entire tree for each symbol. This is normally the case when the alphabet is big and the probability distributions change rapidly with each symbol.

The MNP Coding (MNP5, MNP7)

- Microcom Inc., a maker of modems, has developed a protocol (called MNP, for Microcom Networking Protocol) for use in its modems.
- The MNP protocol specifies:
  - how to unpack bytes into individual bits before they are sent by the modem,
  - how to transmit bits serially in the synchronous and asynchronous modes,
  - what modulation techniques should be used.
- Each specification is called a class, and classes 5 and 7 (called MNP5 and MNP7) specify some methods for data compression.
- These methods (especially MNP5) have become very popular and were used by many modems in the 1980s and 1990s.

The MNP5 Coding

- **The MNP5 method is a two-stage process:**
  - **starts with run-length encoding,**
    - **When three or more identical consecutive bytes are found in the source stream, the compressor emits three copies of the byte onto its output stream, followed by a repetition count.**
  - **followed by adaptive frequency encoding.**
    - **It is similar to the adaptive Huffman method, but, it is simpler to implement and also faster, because it has to swap rows in a table, rather than update a tree, when updating the frequencies of the symbols.**
    - **In the Table, the first column, the frequency counts, is initialized to all zeros.**
    - **The second column is initialized to variable-length codes, called tokens, that vary from a short "000|0" to a long "111|11111110". Each token starts with a 3-bit header, followed by some code bits.**
    - **When the next byte B is read, the corresponding token is written to the output stream, and the frequency of entry B is incremented by 1. Following this, tokens may be swapped to ensure that table entries with large frequencies always have the shortest tokens.**

The MNP7 Coding

- The MNP7 combines run-length encoding with a two-dimensional variant of adaptive Huffman coding and is more complex and sophisticated than MNP5.
  - Stage 1 identifies runs and emits three copies of the run character, followed by a 4-bit count of the remaining characters in the run.
    - A count of zero implies a run of length 3, and a count of 15 (the largest possible in a 4-bit nibble), a run of length 18.
  - Stage 2 starts by assigning to each character a complete table with many variable-length codes.
    - When a character C is read, one of the codes in its table is selected and output, depending on the character preceding C in the input stream.
    - If the preceding character is P, then the pair PC (digram) is incremented by 1, and table rows may be swapped, using the same algorithm as for MNP5, to move the pair to a position in the table that has a shorter code.

Arithmetic (or Range) Coding (Reducing coding redundancy)

- Arithmetic compression is based on interpreting a character-string as a single real number.
- No assumption on encoded source symbols one at a time.
  - Sequences of source symbols are encoded together.
  - There is no one-to-one correspondence between source symbols and code words.
- A sequence of symbols is assigned to a single arithmetic code word which corresponds to a sub-interval in [0,1).
- As the number of symbols in the message increases, the interval used to represent it becomes smaller.
- Smaller intervals require more information units (i.e., bits) to be represented.
- Slower than Huffman coding but typically achieves better compression.

Arithmetic Coding

- **The first step is calculating, or at least to estimating, the frequencies of occurrence of each symbol.**
  - **For best results, the exact frequencies are calculated by reading the entire input file in the first pass of a two-pass compression job.**
  - **However, if the program can get good estimates of the frequencies from a different source, the first pass may be omitted.**
- **The second step is generating subintervals for all symbols, subintervals depend uniquely on the string's symbols and their frequencies.**
  - **The interval [0, 1) is divided among all symbols by assigning each a subinterval proportional in size to its probability.**
  - **Interval [x, y) has width w=y–x,**
  - **The interval [p, q) for symbol a_i, which has width (or probability) w, is p=y and q=p+w; where y is the upper band of the previous symbol.**
  - **The order of the subintervals is immaterial.**

Arithmetic Coding

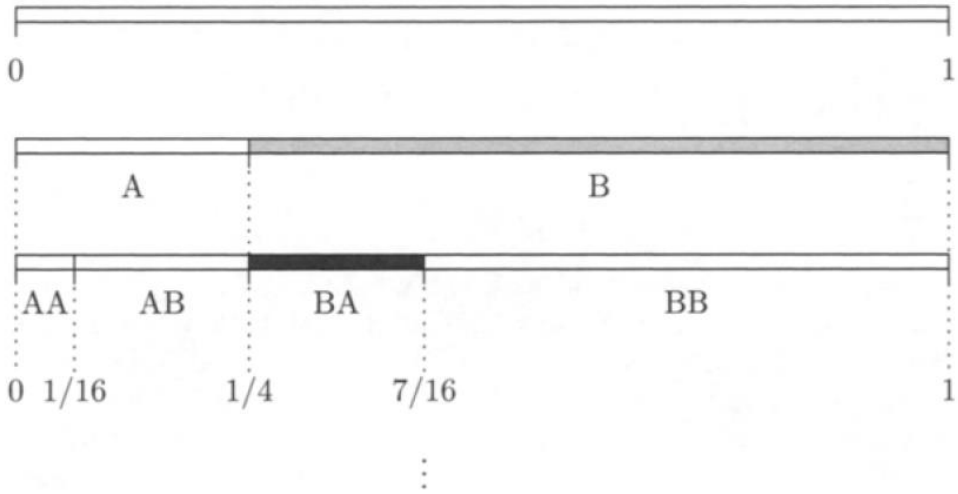
- The third step is encoding the input stream of symbols as below:
  - 1. Start by defining the "current interval" as [0, 1).
  - 2. Repeat the following two steps for each symbol a_i in the input stream:
    - Divide the current interval into subintervals whose sizes are proportional to the symbols' probabilities.
    - Select the subinterval for a_i and define it as the new current interval.
  - 3. When the entire input stream has been processed in this way, the output should be any number that uniquely identifies the current interval.
    - (i.e., any number inside the current interval).

Arithmetic Decoding

- To extract the input stream of symbols from the encoded value we set the "current interval" to [0, 1) and perform the following steps:
  - 1. Divide the "current interval" into subintervals whose sizes are proportional to the symbols' probabilities.
  - 2. Compare the encoded value with the "current interval" and select the sun-interval that the encoded value falls into it.
  - 3. The corresponding symbol of the selected sub-interval is appended to the other outputs.
  - 4. If the number of symbols in the output stream is equal to the size of compressed stream or if the last symbol is the End Of File (EOF) symbol, we finish the decompressing process,
  - 5. Otherwise we set the selected sub-interval as the "current interval" and go to step 1.

Arithmetic Coding (Example 1)

- **Consider a binary source alphabet (A, B) with a probability distribution (0.25, 0.75).**
  - **The initial interval is [0, 1].**
  - **Suppose that an input string contains only 1 symbol. The current interval [0, 1) can be divided into two subintervals according to the probability distribution (P_A,P_B), where a symbol A corresponds to the left subinterval and B to the right.**
  - **Suppose that an input string contains only 2 symbols. The current interval can be divided into two subintervals further according to the probability distribution (P_A,P_B), where a symbol A corresponds to the left subinterval and B to the right.**
  - **For instance, sequence BA corresponds to the highlighted interval.**



Arithmetic Coding (Example 2)

- **Generating subintervals of the string's characters A, B, C, D and E with their occurrence frequencies as below:**

Symbol	Frequency (%)	Subinterval [p, q)
A	25	[0, 0.25)
B	15	[0.25, 0.40)
C	10	[0.40, 0.50)
D	20	[0.50, 0.70)
E	30	[0.70, 1)

Arithmetic Coding (Example 2)

Encoding 'CABAC'

- **Step 1: 'C' 0.....0.4.....0.5.....1.0**
  - **based on p = 0.4, q = 0.5**
- **Step 2: 'A' 0.4.....0.425.....0.5**
  - **based on p = 0.0, q = 0.25**
- **Step 3: 'B' 0.4.....0.40625.....0.41.....0.425**
  - **based on p = 0.25, q = 0.4**
- **Step 4: 'A' 0.40625.....0.4071875.....0.41**
  - **based on p = 0.0, q = 0.25**
- **Step 5: 'C' 0.40625.....0.406625..... 0.4067187.....0.4071875**
  - **based on p = 0.4, q = 0.5**

Final representation? **Sub-Interval** **(or its midpoint)**

[0.406625..... 0.4067187) **0.40667185**

Arithmetic Decoding (Example 2)

- **Extracting 'CABAC' from encoded value 0.4067**

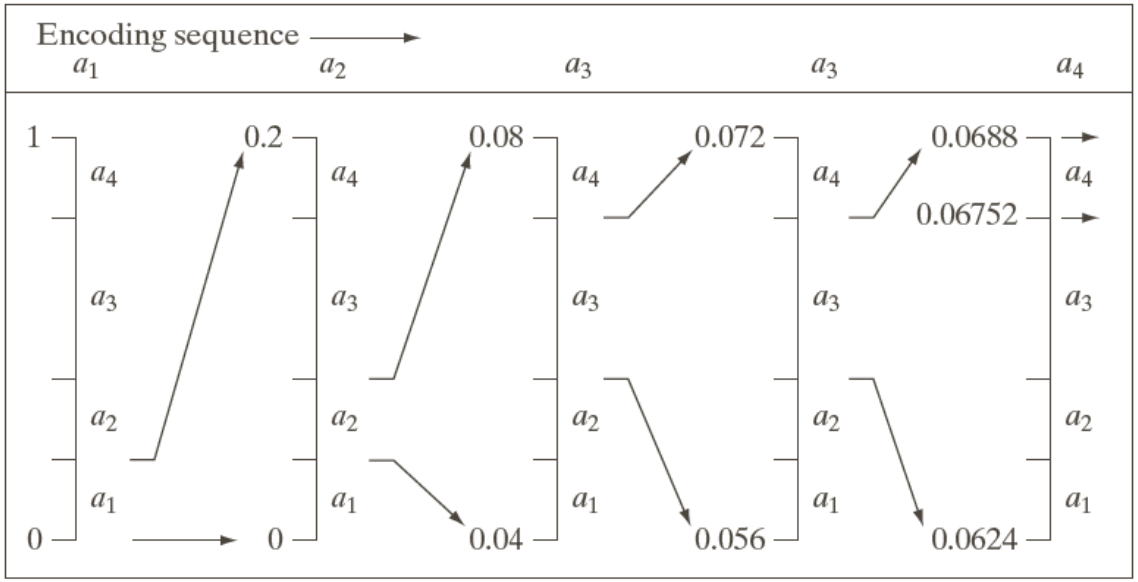
N	Interval[p, q]	Width	Character	N-p	(N-p)/width
0.4067	0.4 – 0.5	0.1	C	0.0067	0.067
0.067	0.0 – 0.25	0.25	A	0.067	0.268
0.268	0.25 – 0.4	0.15	B	0.018	0.12
0.12	0.0 – 0.25	0.25	A	0.12	0.48
0.48	0.4 – 0.5	0.1	C	0.08	0.8
?					

When decompression process stops?

- **Once we reach to the terminal character (EOF)**
  - **The terminal character is added to the original character set and encoded.**
- **The number of extracted symbols is equal to the size of original stream.**
  - **The size of original data stream should be added to compressed version.**

Arithmetic Coding (Example 3)

Source Symbol	Probability	Initial Subinterval
a_1	0.2	[0.0, 0.2)
a_2	0.2	[0.2, 0.4)
a_3	0.4	[0.4, 0.8)
a_4	0.2	[0.8, 1.0)



Encode

$a_1\ a_2\ a_3\ a_3\ a_4$

[0.06752, 0.0688)

or,

0.068

Arithmetic Coding (Example 3)

- **The message $a_1\ a_2\ a_3\ a_3\ a_4$ is encoded using 3 decimal digits or 3/5 = 0.6 decimal digits per source symbol.**

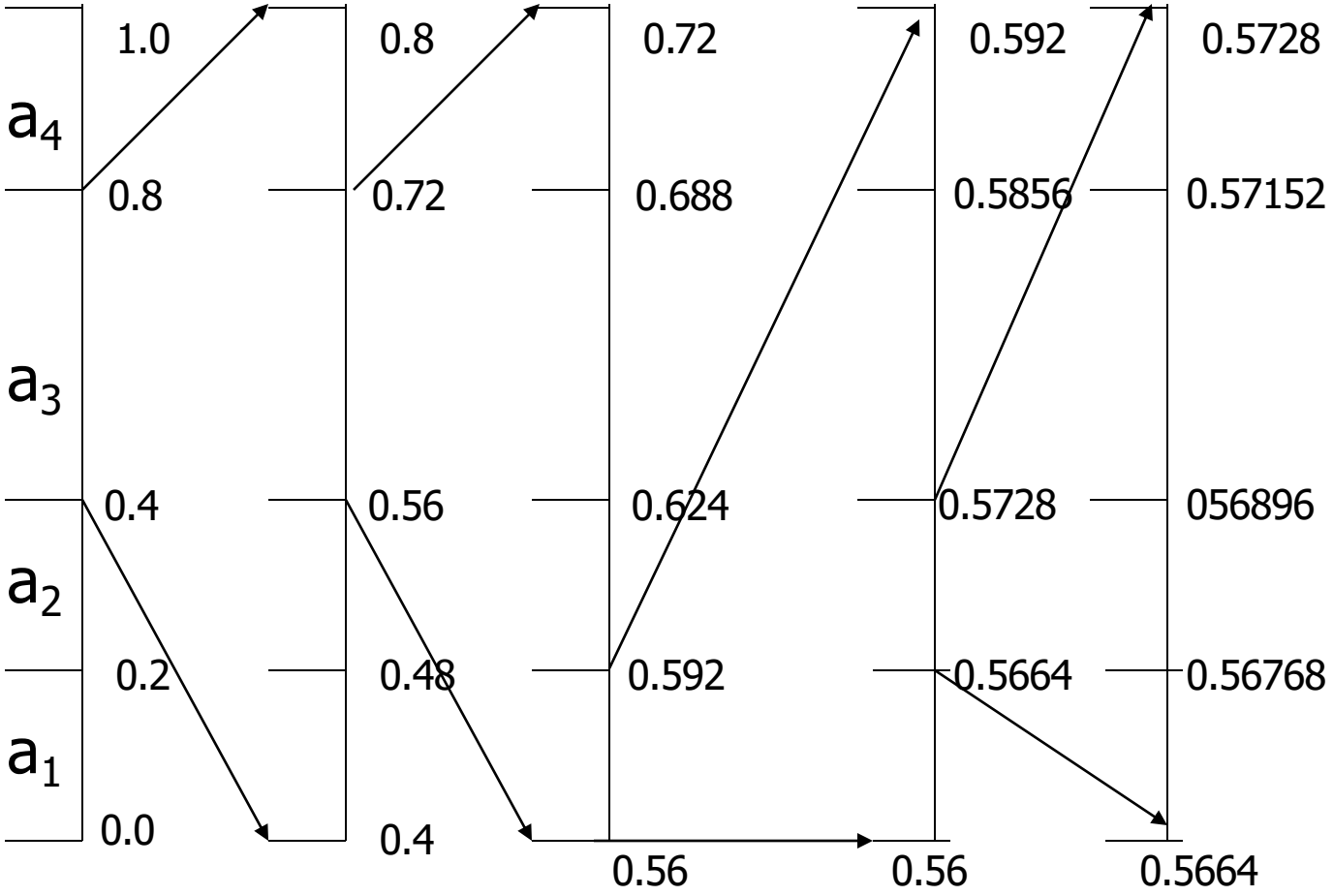
- **The entropy of this message is:**

$$H = -\sum_{k=0}^{3} P(r_k) log(P(r_k))$$

$-(3 \times 0.2\log_{10}(0.2)+0.4\log_{10}(0.4))=0.5786$ digits/symbol

Note: finite precision arithmetic might cause problems due to truncations (underflow problem)!

Arithmetic Decoding (Example 3)



Decode 0.572

$a_3\ a_3\ a_1\ a_2\ a_4$

Adaptive Arithmetic Coding

- **Two features of arithmetic coding make it easy to extend:**

1. **One of the main encoding or decoding steps are updating the low and high bands of each sub-interval.**
   - **This means that in order to encode symbol X, the encoder should be given the cumulative frequencies of the symbol and of the one above it.**
   - **This also implies that the frequency of X (or, equivalently, its probability) could be changed each time it is encoded, provided that the encoder and the decoder agree on how to do this.**

2. **The order of the symbols is unimportant.**
   - **They can even be swapped in the table during the encoding process as long as the encoder and decoder do it in the same way.**

Speedup Arithmetic Coding Range Encoding

- The integers is used in any practical implementation of arithmetic coding, but it results in slow encoding because of the need for frequent renormalizations.
- The main steps in any integer-based arithmetic coding implementation are:
  - 1) proportional range reduction.
  - 2) range expansion (renormalization).
- Range encoding (or range coding) is an improvement to arithmetic coding that reduces the number of renormalizations and thereby speeds up integer-based arithmetic coding by factors of up to 2.
- The main idea is to treat the output not as a binary number, but as a number to another base (256 is commonly used as a base, implying that each digit is a byte).

Speedup Arithmetic Coding Range Encoding

- This requires fewer renormalizations and no bitwise operations. The following analysis may shed light on this method. At any point during arithmetic coding, the output consists of four parts as follows:
  - 1. The part already written on the output. This part will not change.
  - 2. One digit (bit, byte, or a digit to another base) that may be modified by at most one carry when adding to the lower end of the interval.
    - There cannot be two carries because when this digit was originally determined, the range was less than or equal to one unit. Two carries require a range greater than one unit.
  - 3. A (possibly empty) block of digits that passes on a carry (1 in binary, 9 in decimal,255 for base-256, etc.) and are represented by a counter counting their number.
  - 4. The low variable of the encoder.

The QM Coder

- **JPEG, which is an important image compression method, uses arithmetic coding, but not in the way described here.**

- **The arithmetic coder of JPEG is called the QM-coder and is designed for simplicity and speed up, so it is limited to input symbols that are single bits and it uses an approximation instead of a multiplication.**

- **It also uses fixed-precision integer arithmetic, so it has to resort to renormalization of the probability interval from time to time, in order for the approximation to remain close to the true multiplication.**

The PPM Coding

- The **PPM** method is a state of the art compression method that maintains a statistical model of the input stream.
- The encoder inputs the next symbol S, assigns it a probability P, and sends S to an adaptive arithmetic encoder, to be encoded with probability P.
- The simplest statistical model counts the number of times each symbol has occurred in the past and assigns the symbol a probability based on that.
  - Assume that 1217 symbols have been input and encoded so far, and 34 of them were the letter q.
  - If the next symbol is a q, it is assigned a probability of 34/1217 and its count is incremented by 1.
  - The next time q is seen, it will be assigned a probability of 35/t, where t is the total number of symbols input up to that point (not including the last q).

The PPM Coding

- **The next model up is a context-based statistical model.**
  - **The idea is to assign a probability to symbol S depending not only on the frequency of the symbol but also on the contexts in which it has occurred so far.**

- **A static context-based modeler always uses the same probabilities.**
  - **It contains static tables with the probabilities of all the possible digrams (or trigrams) of the alphabet and uses the tables to assign a probability to the next symbol S depending on the symbol (or, in general, on the context) C preceding it. We can imagine S and C being used as indices for a row and a column of a static frequency table.**
  - **The table itself can be constructed by accumulating digram or trigram frequencies from large quantities of text.**

The PPM Coding

- **Such a modeler is simple and produces good results on average, but has two problems.**
  - **The first is that some input streams may be statistically very different from the data originally used to prepare the table.**
    - **A static encoder may create considerable expansion in such a case.**
  - **The second problem is zero probabilities. The probability based models (static or adaptive) such as arithmetic/Huffman encoders, requires all symbols to have nonzero probabilities. Two simple solutions are traditionally adopted for this problem:**
    1. **At the beginning (for adaptive model) or after analyzing a large quantity of data and counting frequencies (for static models), go over the frequency table and assign a frequency count of 1 to empty cell .**
    2. **Add 1 to the total count and divide this single 1 among all the empty cells.**

The PPM Encoding

- An adaptive context-based modeler also maintains tables to assign a probability to the next symbol S depending on a few symbols immediately preceding it (its context C).
  - The tables are updated all the time as more data is being input, which adapts the probabilities to the particular data being compressed.
- The PPM encoder reads the next symbol S, and starts an order-N context C (the last N symbols read), and searches its data structure for a previous occurrence of the current context C followed by the next symbol S
  - Based on input data that has been seen in the past, it determines the probability P that S will appear following the particular context C.
  - If it finds no such occurrence (i.e., if the probability of this particular C followed by this S is 0), it switches to order N−1 and tries the same thing.
- PPM tries to use smaller and smaller parts of the context C, which is the reason for its name "prediction with partial string matching."

The PPM Coding (PPMC)

- The algorithm needs an additional feature that will make it possible for the decoder to stay in lockstep with the encoder.
  - The feature used by PPM is to reserve one symbol of the alphabet as an escape symbol.
  - When the encoder decides to switch to a shorter context, it first writes the escape symbol (arithmetically encoded) on the output stream. After decoding an escape, the decoder also switches to a shorter context.
- The particular method for assigning escape probabilities is called PPMC:
  - Initially, the escape probability should be high, but it should drop as more symbols are input and decoded and more information is collected by the modeler about contexts in the particular data being compressed.

The PPM Coding Variants

- **Four more methods, entitled as:**
  - **PPMA,**
  - **PPMB,**
  - **PPMP,**
  - **PPMX,**
- **have also been developed in attempts to assign precise escape probabilities in PPM.**
- **All five methods have been selected based on the vast experience that the developers had with data compression.**

The PPMA Coding

- **PPMA assigns the escape symbol a probability of 1/(n+1).**
  - **Suppose that a group of contexts has total frequencies n (excluding the escape symbol).**
  - **This is equivalent to always assigning it a count of 1.**

- **The other members of the group are still assigned their original probabilities of x/n, and these probabilities add up to 1 (not including the escape probability).**

The PPMB Coding

- **PPMB is similar to PPMC with one difference.**
  - **It assigns a probability to symbol S following context C only after S has been seen twice in context C.**
  - **This is done by subtracting 1 from the frequency counts.**
    - **If, for example, context abc was seen three times, twice followed by x and once by y, then x is assigned probability (2−1)/3, and y (which should be assigned probability (1−1)/3 = 0) is not assigned any probability.**
    - **Instead, the escape symbol "gets" the two counts subtracted from x and y, and it ends up being assigned probability 2/3.**
- **This method is based on the belief that "seeing twice is believing."**

The PPMP and PPMX Coding

- The PPMP and PPMX are based on Poisson distribution [Witten and Bell 91], which is the reason for the "P" in PPMP (the "X" comes from "approximate," since PPMX is an approximate variant of PPMP).

- PPMP is based on a different principle. It considers the appearance of each symbol a separate Poisson process.
  - Suppose that there are q different symbols in the input stream.
  - At a certain point during compression, n symbols have been read, and symbol i has been input c_i times (so $\Sigma\, c_i = n$).
  - Some of the c_i s are zero (this is the zero-probability problem).
  - PPMP is based on the assumption that symbol i appears according to a Poisson distribution with an expected value (average) λ_i.

The PPMP and PPMX Coding

- The statistical problem considered by PPMP is to estimate q by extrapolating from the n-symbol sample input so far to the entire input stream of N symbols (or, in general, to a larger sample).
- If we express N in terms of n in the form $N = (1+\theta)n$, then a lengthy analysis shows that the number of symbols that haven't appeared in our n-symbol sample is given by $t_1\theta - t_2\theta^2 + t_3\theta^3 - \ldots$,
  - where t_1 is the number of symbols that appeared exactly once in our sample, t_2 is the number of symbols that appeared twice, and so on.

- PPMX uses the approximate value t_1/n (the first term of the sum) as the escape probability.
  - This expression also breaks down when t_1 happens to be 0 or n, so in these cases PPMX is modified to PPMXC, which uses the same escape probability as PPMC.

The PAQ Coding

- **PAQ is an open-source high-performance compression algorithm that combined sophisticated prediction (modeling) with adaptive arithmetic encoding.**
  - **The original idea for PAQ is due to Mahoney (2008).**

- **PAQ prediction (or modeling) pays special attention to the difference between stationary and nonstationary data.**
  - **In the stationary type, the distribution of symbols (the statistics of the data), remains the same throughout the different parts of the input file.**
  - **In the nonstationary type, different regions of the input file discuss different topics and consequently feature different symbol distributions.**

The PAQ Coding

- Modeling **stationary** binary data is straightforward. The predictor initializes the two counts n_0 *and* n_1 *to zero.*
  - *If the next bit to be encoded is b, then count* n_b *is* incremented by 1. At any point, the probabilities for a 0 and a 1 are $p_0 = n_0/(n_0+n_1)$ and $p_1 = n_1/(n_0+n_1)$.
- Modeling **nonstationary** data is more complex and can be done in various ways. **The approach considered by PAQ is perhaps the simplest.**
  - If the next bit to be encoded is *b, then increment count* n_b *by 1 and clear the other counter.*
    - *Thus, a* sequence of consecutive zeros will result in higher and higher values of n_0 *and in* $n_1 = 0$.
    - Essentially, this is the same as predicting that the last input will repeat. Once the next bit of 1 appears in the input, n_0 *is cleared,* n_1 *is set to 1, and we expect (or predict) a* sequence of consecutive 1's.

Summary

- Statistical encoding exploits the fact that not all symbols in the source information occur with equal probability
- Variable length codewords are used with the shortest ones used to encode symbols that occur most frequently
- Static coding: text type is pre-defined and codewords are derived once and used for all subsequent transfers
- Dynamic coding: type of text may vary from one transfer to another and same set of codewords are generated at the transmitter and the receiver as the transfer takes place

Dictionary Based Methods

Dictionary Based Coding (interdata redundancy reduction)

- **Dictionary based coding does not use statistical knowledge or priori knowledge of data.**

- **It assigns fixed length code words to variable length sequences.**
  - **Encoder: As the input is processed develop a dictionary and transmit the index of strings found in the dictionary.**
  - **Decoder: As the code is processed reconstruct the dictionary to invert the process of encoding.**

- **Examples:LZ77, LZ78, LZW, Sequitur, ACB**
- **Patented Algorithm US 4,558,302**
- **Included in GIF and TIFF and PDF file formats**

Dictionary Based Coding

- The goal of dictionary compression is to eliminate the redundancy of storing repetitive strings for words and phrases repeated within the data in order to achieve a better compression.
- The dictionary is used to store the string patterns seen before and the indexes are used to encode the repeated patterns.
- The coder keeps a record of the most common words or phrases in a document called a dictionary and uses their indices in the dictionary as output tokens.
- Ideally, the tokens are much shorter in comparison with the words or phrases themselves and the words and phrases are frequently repeated in the document.

Dictionary Based Coding

- The encoder reads the input string, identifies those recurrent words, and outputs their indices in the dictionary.
- A new word is output in the uncompressed form and added into the dictionary as an new entry.
- The main operations involve in dictionary-based method are:
  - The comparison of strings.
  - Dictionary maintenance.
  - Efficient way of encoding.

Dictionary Based Coding

- **Compressors and decompressors both maintain a dictionary by themselves.**

- **The dictionary holds strings of symbols, and it may be static or dynamic (adaptive):**
  - **The former is permanent, sometimes permitting the addition of strings but no deletions.**
  - **The latter holds strings previously found in the input stream, allowing for additions and deletions of strings as new input is being read.**

- **The dictionary-based algorithms are normally faster than entropy-based ones.**

Dictionary Based Coding

- The simplest example of a static dictionary is a dictionary of the English language used to compress English text.
  - Imagine a dictionary containing perhaps half a million words (without their definitions).
  - A word (a string of symbols terminated by a space or a punctuation mark) is read from the input stream and the dictionary is searched.
  - If a match is found, an index to the dictionary is written into the output stream.
  - Otherwise, the uncompressed word itself is written. (This is an example of logical compression.)
- As a result, the output stream contains indexes and raw words, and it is important to distinguish between them.

Dictionary Based Coding

- One way to achieve this is to reserve an extra bit in every item written.
  - In principle, a 19-bit index is sufficient to specify an item in a 2^{19}= 524,288-word dictionary.
  - Thus, when a match is found, we can write a 20-bit token that consists of a flag bit (perhaps a zero) followed by a 19-bit index.
  - When no match is found, a flag of 1 is written, followed by the size of the unmatched word, followed by the word itself.
- Example:
  - Assuming that the word ***bet*** is found in dictionary entry 1025, it is encoded as the 20-bit number 0|00000001000000001.
  - Assuming that the word ***xet*** is not found, it is encoded as 1|0000011|01111000|01100101|01110100.
    - This is a 4-byte number where the 7-bit field 0000011 indicates that three more bytes follow.

Dictionary Based Coding Performance Evaluation

- **Thus, we have to answer the question how many matches are needed in order to have overall compression?**
  - **Assuming that the size is written as a 7-bit number, and that an average word size is five characters, an uncompressed word occupies, on average, six bytes (= 48 bits) in the output stream.**
    - **Compressing 48 bits into 20 is excellent, provided that it happens often enough.**
    - **We denote the probability of a match (the case where the word is found in the dictionary) by P.**
    - **After reading and compressing N words, the size of the output stream will be $N[20P + 48(1-P)] = N[48-28P]$ bits.**
    - **The size of the input stream is (assuming five characters per word) 40N bits.**
  - **Compression is achieved when $N[48-28P] < 40N$, which implies $P>0.29$.**
  - **We need a matching rate of 29% or better to achieve compression.**

Dictionary Based Coding

- As long as the input stream consists of English text, most words will be found in a 500,000-word dictionary. Other types of data, however, may not do that well.
  - A file containing the source code of a computer program may contain "words" such as cout, xor, and malloc that may not be found in an English dictionary.
  - A binary file normally contains gibberish when viewed in ASCII, so very few matches may be found, resulting in considerable expansion instead of compression.
- This shows that a static dictionary is not a good choice for a general-purpose compressor.
  - It may, however, be a good choice for a special-purpose one.
  - Special-purpose compression software may benefit from a small, specialized dictionary containing, just a fewer words.

Dictionary Based Coding

- **Adaptive dictionary-based** method can start with an empty dictionary or with a small, default dictionary, add words to it as they are found in the input stream, and delete old words because a big dictionary slows down the search.
- Such a method consists of a loop where each iteration starts by reading the input stream and breaking it up (parsing it) into words or phrases.
- It then should search the dictionary for each word and,
  - if a match is found, write a token on the output stream.
  - Otherwise, the uncompressed word should be written and also added to the dictionary.
- The last step in each iteration checks whether an old word should be deleted from the dictionary.

Dictionary Based Coding

- This may sound complicated, but it has two advantages:
  - 1. It involves string search and match operations, rather than numerical computations. Many programmers prefer that.
  - 2. The decoder is simple (this is an asymmetric compression method).
- In an adaptive dictionary-based method, however, the decoder has to read its input stream, determine whether the current item is a token or uncompressed data, use tokens to obtain data from the dictionary, and output the final, uncompressed data.
- It does not have to parse the input stream in a complex way, and it does not have to search the dictionary to find matches. Many programmers like that, too.

Lempel-Ziv (LZ77) Coding

- LZ77 was proposed by Ziv and Lempel, in 1977.
- The principle of LZ77 (sometimes also referred to as LZ1) is to use part of the previously-seen input stream as the dictionary.

- An important source of redundancy in digital data is repeating phrases.
- This type of redundancy exists in all types of data—audio, still images, and video—but is easiest to visualize for text.
- The phrases that occur again and again in the data constitute the basis of all the dictionary-based compression algorithms.

Lempel-Ziv (LZ77) Coding

- The LZ77 method is based on a *sliding window*.
- The encoder maintains a window to the input stream and shifts the input in that window from right to left as strings of symbols are being encoded.
- The window below is divided into two parts.
  - The part on the left is the *search buffer.*
    - This is the current dictionary, and it includes symbols that have recently been input and encoded.
  - The part on the right is the *look-ahead buffer*, *containing text yet to be* encoded.

Lempel-Ziv (LZ77) Coding

- The encoder scans the search buffer backwards (from right to left) looking for a match for the first symbol in the look-ahead buffer (X).
  - If encoder finds the symbol X, the distance or offset (D) of it from the end of the search buffer is held.
  - The encoder then matches other symbols following it as possible.
  - The number of symbols which are matched is also held as the length of the match (L).
- The encoder then continues the backward scan, trying to find longer matches.
  - The encoder selects the longest match or, if they are all the same length, the last one found, and prepares the token (D,L,X).

Lempel-Ziv (LZ77) Coding

- In general, an LZ77 token has three parts:
  - (offset, length, next symbol in the look-ahead buffer)
- The offset can also be thought of as the distance between the previous and the current occurrences of the string being compressed.
- This token is written on the output stream, and the window is shifted to the right (or, alternatively, the input stream is moved to the left) L+1 positions:
  - L positions for the matched string
  - and one position for the next symbol.

Lempel-Ziv (LZ77) Coding

- **If the backward search yields no match, an LZ77 token with zero offset and length and with the unmatched symbol is written.**
- **This is also the reason a token has a third component.**
- **Tokens with zero offset and length are common at the beginning of any compression job, when the search buffer is empty or almost empty.**

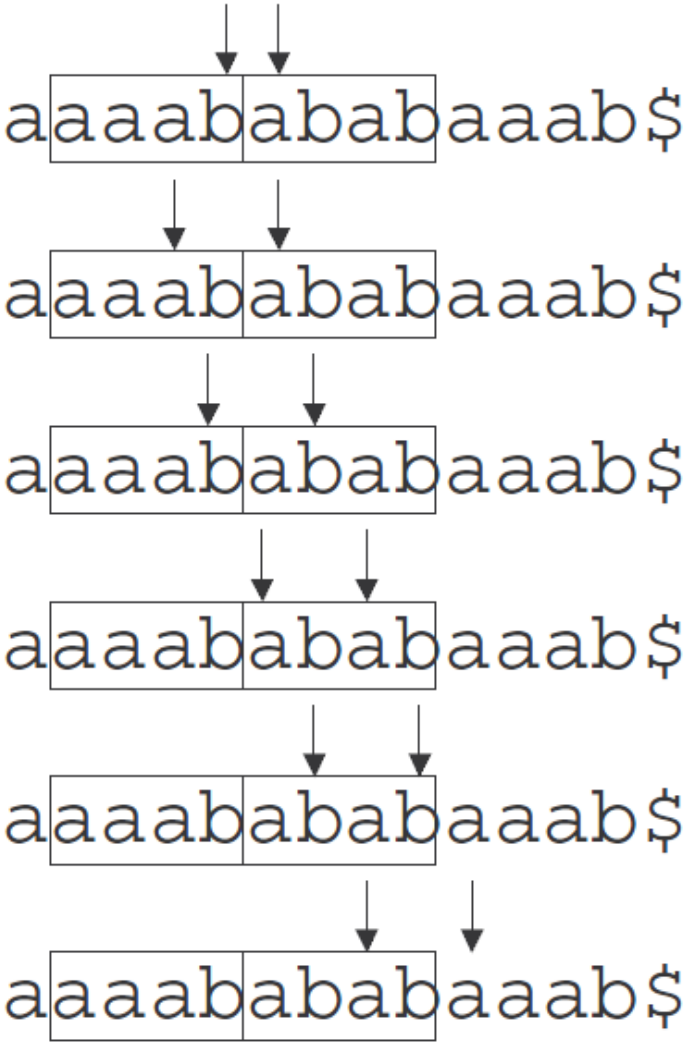
	sir␣sid␣eastman␣	⇒	(0,0,“s”)
s	ir␣sid␣eastman␣e	⇒	(0,0,“i”)
si	r␣sid␣eastman␣ea	⇒	(0,0,“r”)
sir	␣sid␣eastman␣eas	⇒	(0,0,“␣”)
sir␣	sid␣eastman␣easi	⇒	(4,2,“d”)

Lempel-Ziv (LZ77) Coding

s = 4, t = 4, a = 3

	tuple
aaaababababaaab$	<0,0,a>
aaaaababababaaab$	<1,3,b>
aaaababababaaab$	<2,5,a>
aaaababababaaab$	<4,2,$>

Lempel-Ziv (LZ77) Coding



offset	length
1	0
2	1
2	2
2	3
2	4
2	5

tuple
<2,5,a>

Lempel-Ziv (LZ77) Coding

- Common Search Window size
  - Search buffer 32KB
  - Look-ahead buffer 258 Bytes
- How to store such a large dictionary
  - Hash table that stores the starting positions for all three byte sequences.
  - Hash table uses chaining with newest entries at the beginning of the chain.
- Variable length code using adaptive Huffman or arithmetic code on Tuples can be used for more compression.
  - LZPP proposed by Pylak in 2003.
- Coding done in blocks to avoid disk accesses.
- LZ77 used in Zip and Gzip.

Dictionary Based Coding (Bounded Size Dictionary)

- **Bounded Size Dictionary**
  - **n bits of index allows a dictionary of size 2^n**
  - **Doubtful that long entries in the dictionary will be useful.**

- **Strategies when the dictionary reaches its limit.**
  - **1. Don't add more, just use what is there.**
  - **2. Throw it away and start a new dictionary.**
  - **3. Double the dictionary, adding one more bit to indices.**
  - **4. Throw out the least recently visited entry to make room for the new entry.**

Lempel-Ziv (LZ78) Coding

- LZ78 was proposed by Ziv and Lempel, in 1978.
- The LZ78 method does not use any search buffer, look-ahead buffer, or sliding window.
- Instead, there is a dictionary of previously encountered strings.
- This dictionary starts empty (or almost empty), and its size is limited only by the amount of available memory.
- The encoder outputs two-field tokens:
  - The first field is a pointer to the dictionary;
  - the second is the code of a symbol.
- Tokens do not contain the length of a string, since this is implied in the dictionary.

Lempel-Ziv (LZ78) Coding

- Each token corresponds to a string of input symbols, and that string is added to the dictionary after the token is written on the compressed stream.
- Nothing is ever deleted from the dictionary, which is both an advantage over LZ77.
- The dictionary starts with the null string at position zero.
  - As symbols are input and encoded, strings are added to the dictionary at positions 1, 2, and so on.
  - When the next symbol x is read from the input stream, the dictionary is searched for an entry with the one-symbol string x.
  - If none are found, x is added to the next available position in the dictionary, and the token (0, x) is output.
    - This token indicates the string "null x" (a concatenation of the null string and x).

Lempel-Ziv (LZ78) Coding

- If an entry with x is found (at, say, position K), the next symbol y is read, and the dictionary is searched for an entry containing the two-symbol string xy.
- If none are found, then string xy is added to the next available position in the dictionary, and the token (K, y) is output.
  - This token indicates the string xy, since K is the dictionary position of string x.
- The process continues until the end of the input stream is reached.

- **At a certain point the string is not found in the dictionary, so the encoder adds it to the dictionary and outputs a token with:**
  - the last dictionary match as its first field, and
  - the last symbol of the string (the one that caused the search to fail) as its second field.

Lempel-Ziv (LZ78) Coding

- **This table shows the first 14 steps in encoding the string:**

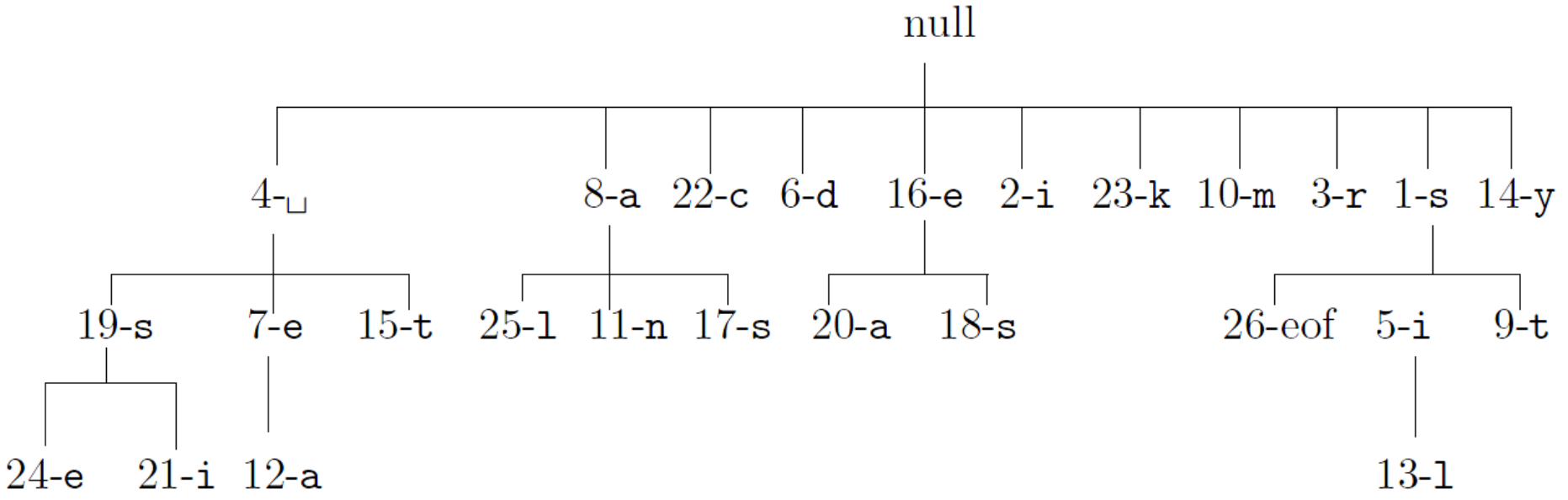
`sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals`

- **In each step, the string added to the dictionary is the one being encoded, minus its last symbol.**

Dictionary		Token	Dictionary		Token
0	null				
1	“s”	(0, “s”)	8	“a”	(0, “a”)
2	“i”	(0, “i”)	9	“st”	(1, “t”)
3	“r”	(0, “r”)	10	“m”	(0, “m”)
4	“␣”	(0, “␣”)	11	“an”	(8, “n”)
5	“si”	(1, “i”)	12	“␣ea”	(7, “a”)
6	“d”	(0, “d”)	13	“sil”	(5, “l”)
7	“␣e”	(4, “e”)	14	“y”	(0, “y”)

Lempel-Ziv (LZ78) Coding

- A good data structure for the dictionary is a tree, but not a binary tree. The tree starts with the null string as the root.
- All the strings that start with the null string (strings for which the token pointer is zero) are added to the tree as children of the root.
- In the above example those are s, i, r, , d, a, m, y, e, c, and k. Each of them becomes the root of a subtree.



Lempel-Ziv-Welch (LZW) Coding

- The LZW is a popular variant of LZ78, developed by Terry Welch in 1984.
- Its main feature is eliminating the second field of a token, thus the LZW token consists of just a pointer to the dictionary.
- The LZW method starts by initializing the dictionary to all the symbols in the alphabet.
  - In the common case of 8-bit symbols, the first 256 entries of the dictionary (entries 0 through 255) are occupied before any data is input.
  - Because the dictionary is initialized, the next input character will always be found in the dictionary.
- This is why an LZW token can consist of just a pointer and does not have to contain a character code as in LZ77 and LZ78.

Lempel-Ziv-Welch (LZW) Coding

- The principle of LZW is that the encoder inputs symbols one by one and accumulates them in a string *I*.
- After each symbol is input and is concatenated to *I*, the dictionary is searched for string *I*.
- As long as *I* is found in the dictionary, the process of reading new symbol is continued.
- At a certain point, adding the next symbol x causes the search to fail;
  - string I is in the dictionary but string Ix (symbol x concatenated to I) is not.
- At this point the encoder
  - 1) outputs the dictionary pointer that points to string I,
  - 2) saves string Ix *in the next available dictionary entry, and*
  - *3) initializes* string I to symbol x.

LZW Coding Example

- To illustrate this process, we again use the text string:

sir␣sid␣eastman␣easily␣teases␣sea␣sick␣seals

- The steps are as follows:
  - 0. Initialize entries 0–255 of the dictionary to all 256 8-bit bytes.
  - 1. The first symbol s is input and is found in the dictionary (in entry 115, since this is the ASCII code of s).
    - The next symbol i is input, but si is not found in the dictionary. The encoder performs the following:
      - 1) outputs 115,
      - 2) saves string si in the next available dictionary entry (entry 256)
      - 3) initializes I to the symbol i.
  - 2. The r of sir is input, but string ir is not in the dictionary.
    - 1) The encoder outputs 105 (the ASCII code of i),
    - 2) saves string ir in the dictionary (entry 257), and
    - 3) initializes I to the symbol r.

LZW Coding Example

I	in dict?	new entry	output
s	Y		
si	N	256-si	115 (s)
i	Y		
ir	N	257-ir	105 (i)
r	Y		
r␣	N	258-r␣	114 (r)
␣	Y		
␣s	N	259-␣s	32 (␣)
s	Y		
si	Y		
sid	N	260-sid	256 (si)
d	Y		
d␣	N	261-d␣	100 (d)
␣	Y		
␣e	N	262-␣e	32 (␣)
e	Y		
ea	N	263-ea	101 (e)
a	Y		
as	N	264-as	97 (a)

I	in dict?	new entry	output
s	Y		
st	N	265-st	115 (s)
t	Y		
tm	N	266-tm	116 (t)
m	Y		
ma	N	267-ma	109 (m)
a	Y		
an	N	268-an	97 (a)
n	Y		
n␣	N	269-n␣	110 (n)
␣	Y		
␣e	Y		
␣ea	N	270-␣ea	262 (␣e)
a	Y		
as	Y		
asi	N	271-asi	264 (as)
i	Y		
il	N	272-il	105 (i)
l	Y		
ly	N	273-ly	108 (l)

LZW Coding Example

I	in dict?	new entry	output	I	in dict?	new entry	output
y	Y			␣	Y		
y␣	N	274-y␣	121 (y)	␣s	Y		
␣	Y			␣si	N	283-␣si	259 (␣s)
␣t	N	275-␣t	32 (␣)	i	Y		
t	Y			ic	N	284-ic	105 (i)
te	N	276-te	116 (t)	c	Y		
e	Y			ck	N	285-ck	99 (c)
ea	Y			k	Y		
eas	N	277-eas	263 (ea)	k␣	N	286-k␣	107 (k)
s	Y			␣	Y		
se	N	278-se	115 (s)	␣s	Y		
e	Y			␣se	Y		
es	N	279-es	101 (e)	␣sea	N	287-␣sea	281 (␣se)
s	Y			a	Y		
s␣	N	280-s␣	115 (s)	al	N	288-al	97 (a)
␣	Y			l	Y		
␣s	Y			ls	N	289-ls	108 (l)
␣se	N	281-␣se	259 (␣s)	s	Y		
e	Y			s,eof	N		115 (s)
ea	Y						
ea␣	N	282-ea␣	263 (ea)				

LZW Coding Example

- **The complete output stream is (only the numbers are output, not the strings in parentheses) as follows:**
  - **115 (s), 105 (i), 114 (r), 32 (), 256 (si), 100 (d), 32 (), 101 (e), 97 (a), 115 (s), 116(t), 109 (m), 97 (a), 110 (n), 262 (e), 264 (as), 105 (i), 108 (l), 121 (y),32 (), 116 (t), 263 (ea), 115 (s), 101 (e), 115 (s), 259 (s), 263 (ea), 259 (s), 105 (i), 99 (c), 107 (k), 280 (se), 97 (a), 108 (l), 115 (s), eof.**

0	NULL	110	n	262	␣e	276	te
1	SOH	...		263	ea	277	eas
...		115	s	264	as	278	se
32	SP	116	t	265	st	279	es
...		...		266	tm	280	s
97	a	121	y	267	ma	281	␣se
98	b	...		268	an	282	ea␣
99	c	255	255	269	n␣	283	␣si
100	d	256	si	270	␣ea	284	ic
101	e	257	ir	271	asi	285	ck
...		258	r␣	272	il	286	k␣
107	k	259	␣s	273	ly	287	␣sea
108	l	260	sid	274	y␣	288	al
109	m	261	d␣	275	␣t	289	ls

Table 6.24: An LZW Dictionary.

LZW Encoding Algorithm


```
for i:=0 to 255 do
    append i as a 1-symbol string to the dictionary;
append λ (NULL) to the dictionary;
di:=dictionary index of λ;
repeat
    read(ch);
    if <<di,ch>> is in the dictionary then
        di:=dictionary index of <<di,ch>>;
    else
        output(di);
        append <<di,ch>> to the dictionary;
        di:=dictionary index of ch;
    endif;
until end-of-input;
```

LZW Decoding

- To understand how the LZW decoder works, we recall the three steps the encoder performs each time:
  - 1) it outputs the dictionary pointer that points to string I,
  - 2) it saves string Ix in the next available entry of the dictionary,
  - 3) it initializes string I to symbol x.

- The decoder starts with the first entries of its dictionary initialized to all the symbols of the alphabet (normally 256 symbols).
  - It then reads its input stream (which consists of pointers to the dictionary) and uses each pointer to retrieve uncompressed symbols from its dictionary and write them on its output stream.
  - It also builds its dictionary in the same way as the encoder (this fact is usually expressed by saying that the encoder and decoder are synchronized, or that they work in lockstep).

LZW Decoding

- In the first decoding step, the decoder inputs the first pointer and uses it to retrieve a dictionary item I.
  - This is a string of symbols, and it is written on the decoder's output.
  - String Ix needs to be saved in the dictionary, but symbol x is still unknown;
    - it will be the first symbol in the next string retrieved from the dictionary.
- In each decoding step after the first, the decoder inputs the next pointer,
  - retrieves the next string J from the dictionary,
  - writes it on the output,
  - isolates its first symbol x,
  - and saves string Ix in the next available dictionary entry (after checking to make sure string Ix is not already in the dictionary).
- The decoder then moves J to I and is ready for the next step.

LZW Variant Coding

- LZW is an adaptive data compression method, but it is slow to adapt to its input, since strings in the dictionary become only one character longer at a time.
- The LZMW method is a variant of LZW that overcomes this problem.
  - Its main principle is this: Instead of adding I plus one character of the next phrase to the dictionary, add I plus the entire next phrase to the dictionary.
- The LZAP method is another variant based on this idea:
  - Instead of just concatenating the last two phrases and placing the result in the dictionary, place all prefixes of the concatenation in the dictionary.
  - More specifically, if S and T are the last two matches, add St to the dictionary for every nonempty prefix t of T, including T itself.

RAR and WinRAR Software

- The popular RAR software is the creation of Eugene Roshal, who started it as his university doctoral dissertation.
  - RAR is an acronym that stands for Roshal Archive (or ARchiver).
  - RAR is currently available as shareware, for Windows , Pocket PC, Macintosh OS X, Linux, DOS, and FreeBSD.
  - WinRAR provides complete support for RAR and ZIP archives and can decompress (but not compress) CAB, ARJ, LZH, TAR, GZ, ACE, UUE, BZ2, JAR, ISO, 7Z, and Z archives.
  - In addition to compressing data, WinRAR can encrypt data with the advanced encryption standard (AES-128).
  - RAR has two compression modes: general and special.
    - The general mode employs an LZSS-based algorithm and literals, offsets, and match lengths are compressed further by a Huffman coder.
    - The special compression mode is based on the PPMD algorithm.

Sequitur Coding

- **Proposed by Nevill-Manning and Witten, 1996.**
- **Uses a context-free grammar (without recursion) to represent a string.**
  - **The grammar is inferred from the string.**
  - **If there is structure and repetition in the string then the grammar may be very small compared to the original string.**
- **Clever encoding of the grammar yields impressive compression ratios.**
- **Compression plus structure!**

Sequitur Coding
Context-Free Grammars

- **Context-free grammar was invented by Chomsky in 1959 to explain the grammar of natural languages.**
- **Also it was invented by Backus in 1959 to generate and parse Fortran.**
- **Example:**
  - **– terminals: b, e**
  - **– non-terminals: S, A**
  - **– Production Rules:**

    S → SA,

    S → A,

    A → bSe,

    A → be
  - **– S is the start symbol**

Sequitur Coding
Context-Free Grammars Example

$S \rightarrow SA,$

$S \rightarrow A,$

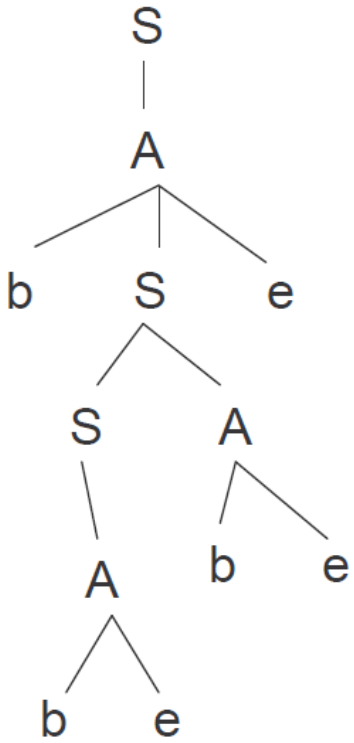
$A \rightarrow bSe,$

$A \rightarrow be$

derivation of bbebee

S
A
bSe
bSAe
bAAe
bbeAe
bbebee

Hierarchical parse tree



Example: b and e matched as parentheses

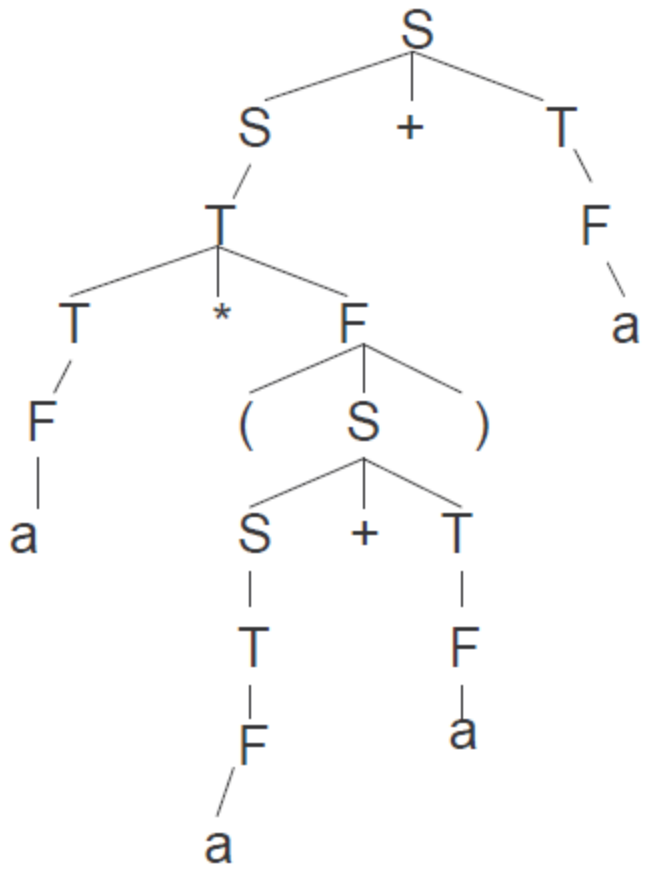
Context-Free Grammars Example
Arithmetic Expressions

$S \rightarrow S + T$
$S \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow a$
$F \rightarrow (S)$

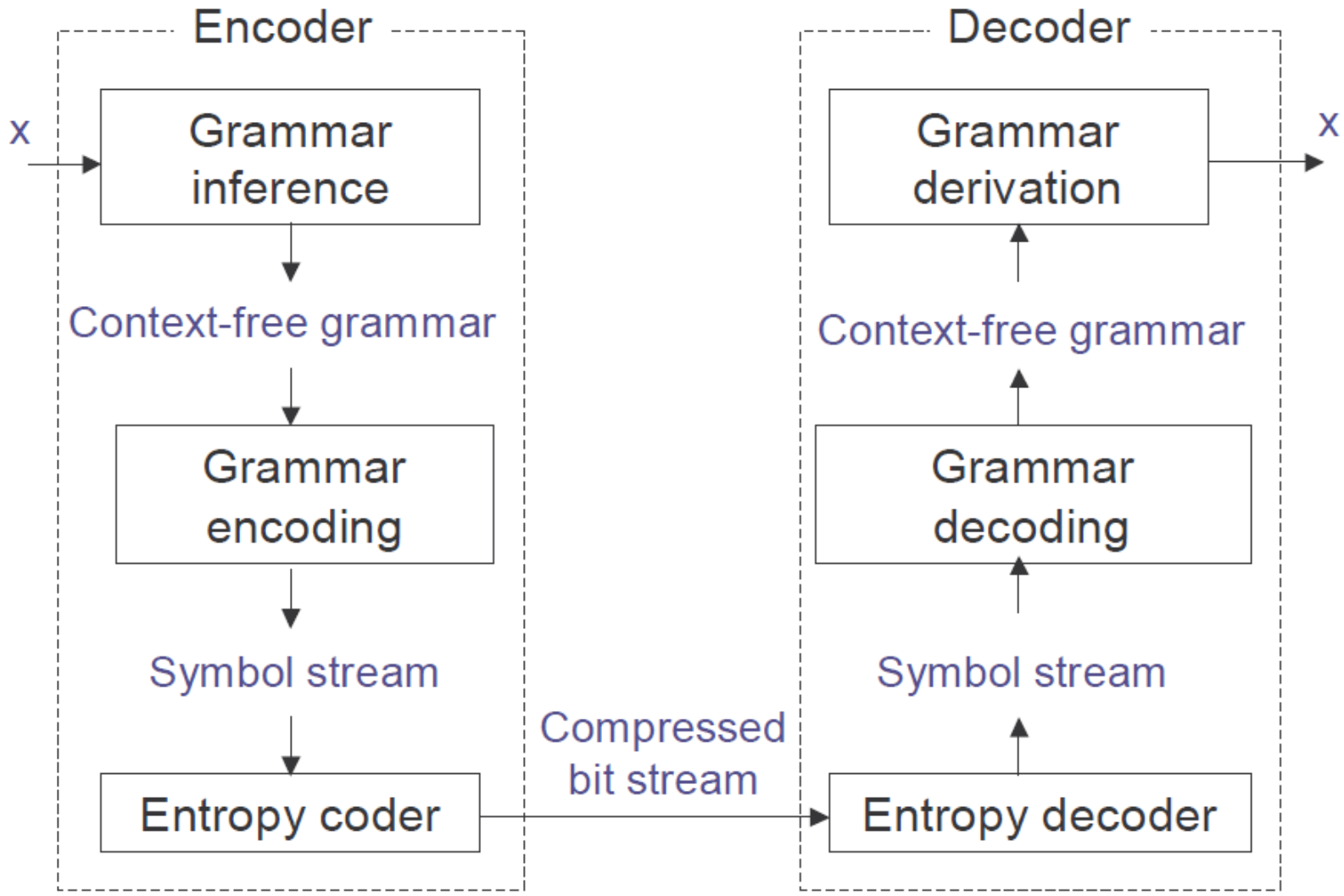
derivation of a * (a + a) + a

S
S+T
T+T
T*F+T
F*F+T
a*F+T
a*(S)+T
a*(S+F)+T
a*(T+F)+T
a*(F+F)+T
a*(a+F)+T
a*(a+a)+T
a*(a+a)+F
a*(a+a)+a

parse tree



Sequitur Coding
Overview of Grammar Compression



Compression Quality

	size	comp	gzip	sequitur	PPMC
bib	111261	3.35	2.51	2.48	2.12
book	768771	3.46	3.35	2.82	2.52
geo	102400	6.08	5.34	4.74	5.01
obj2	246814	4.17	2.63	2.68	2.77
pic	513216	0.97	0.82	0.90	0.98
progc	38611	3.87	2.68	2.83	2.49

= First; = Second; = Third.

Files from the Calgary Corpus
Units in bits per character (8 bits)
comp - based on LZW
gzip - based on LZ77
PPMC - adaptive arithmetic coding with context

Transform Based Methods

Idea of Transform Coding

- Transform is a standard mathematical tool being used in many areas to solve sometimes difficult computation problems in the original form.
- The main idea is to change a group of quantity data such as a vector or a function to another form in which some useful features may occur.
- In the transform coding the input data $x=\{x_0,x_2,\ldots,x_{N-1}\}$ are transformed into coefficients $c_0, c_2,\ldots,c_{N-1}$(real values):

$$c_i = T(x)$$

  - The coefficients have the property that most of them are near zero.
  - Most of the "energy" is compacted into a few coefficients.

Mathematical Properties of Transforms

- **Linear Transformation – The transform is defined by a real nxn matrix $A = (a_{ij})$:**
- **$c = T(x) = Ax$**

$$\begin{bmatrix} a_{00} & \cdots & a_{0,N-1} \\ \vdots & & \vdots \\ a_{N-1,0} & \cdots & a_{N-1,N-1} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} c_0 \\ \vdots \\ c_{N-1} \end{bmatrix}$$

- **Orthogonality: $A^{-1} = A^T$**
  - **The energy of the data equals the energy of the coefficients**

$$\sum_{i=0}^{N-1} c_i^{\,2} = c^T c = (Ax)^T (Ax)$$

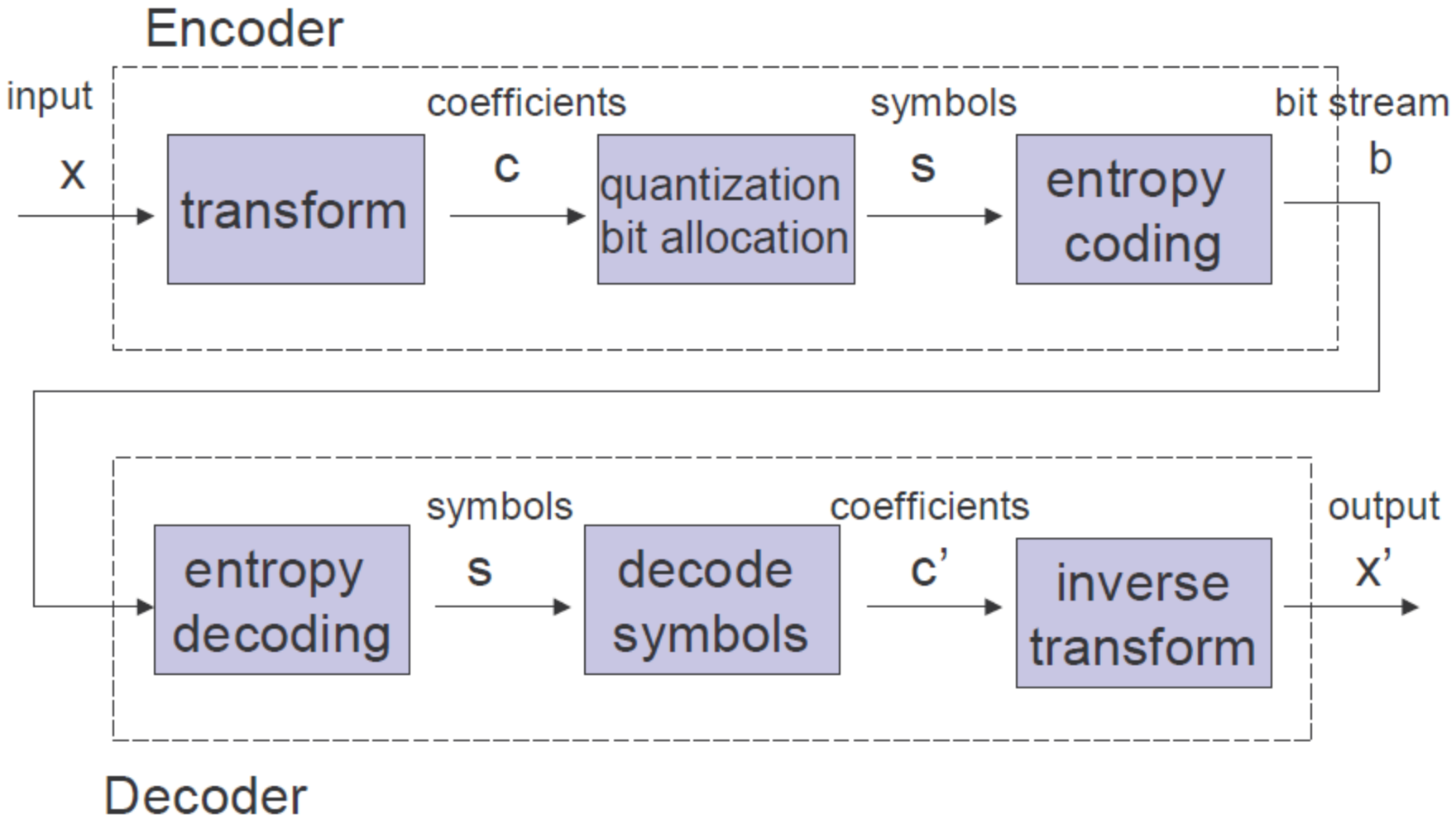
$$= (x^T A^T)(Ax) = x^T (A^T A) x = x^T x = \sum_{i=0}^{N-1} x_i^{\,2}$$

Mathematical Properties of Transforms

- **Orthonormality: $A^{-1} = A^T = A$**
  - **the most important advantage of applying an orthonormal transform is that first, such a transform keeps the *energy of the data* unchanged.**
  - **Secondly, in the transformed vector, the first few elements often concentrate a large proportion of the system energy.**
- **Hence these few elements can be used to give a good approximation of the entire original set of data.**
- **In other words, these few elements only may be sufficient for reconstructing the original set of data and the rest of transformed values can be dropped.**

- **Therefore, an orthonormal transform can be applied to achieve good data compression.**

Transform-Based Compression



Transform Coding

- **The transformation T(.) can be selected from one of the popular transforms like:**
  - **DFT (Discrete Fourier Transform)**
  - **DCT (Discrete Cosine Transform)**
    - **JPEG (Joint Photographic Experts Group)**
    - **MPEG**
    - **MP3 = MPEG 1- Layer 3**
  - **DWT (Discrete Wavelet Transform)**
    - **SPIHT**
    - **EBCOT (JPEG 2000)**
    - **CREW**
    - **EZW**
    - **DjVu**
    - **UWIC (University of Washington Image Coder)**
  - **BWT (Burrows-Wheeler Transform)**

Transform Coding

- A quantization process should be applied to determine the number of bits for each coefficient and reduced it based on importance of each coefficient .

- A good entropy coders should be employed to reduce the number of bit to code each symbol based on its occurrence probability:
  - Arithmetic
  - Huffman
  - Golomb
  - Tunstall

Fourier Transform

- Every periodic function, real or complex, can be represented as the sum of sine and cosine functions.
- It transforms a function between the time and frequency domains.
- If the shape of the function is far from a uniform wave, its Fourier expansion may include an infinite number of frequencies.
- For a continuous function *f(t), the Fourier transform and its* inverse are given by:

$$F(\omega) = \int_{-\infty}^{+\infty} f(t) e^{-j2\pi\omega t} dt = \int_{-\infty}^{+\infty} f(t)[\cos(2\pi\omega t) - j\sin(2\pi\omega t)]dt$$

$$f(t) = \int_{-\infty}^{+\infty} F(\omega) e^{j2\pi\omega t} d\omega = \int_{-\infty}^{+\infty} F(\omega)[\cos(2\pi\omega t) + j\sin(2\pi\omega t)]d\omega$$

Fourier Transform

- **In computer applications, we normally have discrete functions that take just *n (equally* spaced) values.**
- **In such a case the Discrete Fourier Transform (DFT) is:**

$$F(\omega)=\sum_{t=0}^{N-1} f(t)e^{\frac{-j2\pi\omega t}{N}}=\sum_{t=0}^{N-1} f(t)[\cos(\frac{2\pi\omega t}{N})-j\sin(\frac{2\pi\omega t}{N})]\ ,\ \omega=0,1,\ldots,N-1$$

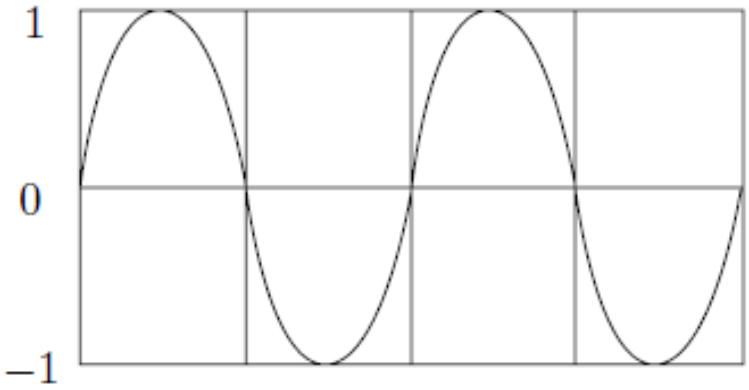
$$f(t)=\frac{1}{N}\sum_{\omega=0}^{N-1} F(\omega)e^{\frac{j2\pi\omega t}{N}}=\frac{1}{N}\sum_{\omega=0}^{N-1} F(\omega)[\cos(\frac{2\pi\omega t}{N})+j\sin(\frac{2\pi\omega t}{N})]\ ,\ t=0,1,\ldots,N-1$$

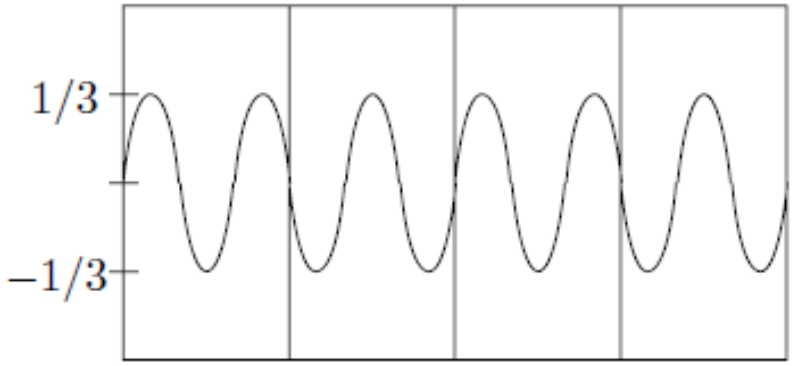
- **For 2D:**

$$F(u,v)=\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)e^{\frac{-j2\pi(ux+vy)}{N}}\ ,\ u,v=0,1,\ldots,N-1$$

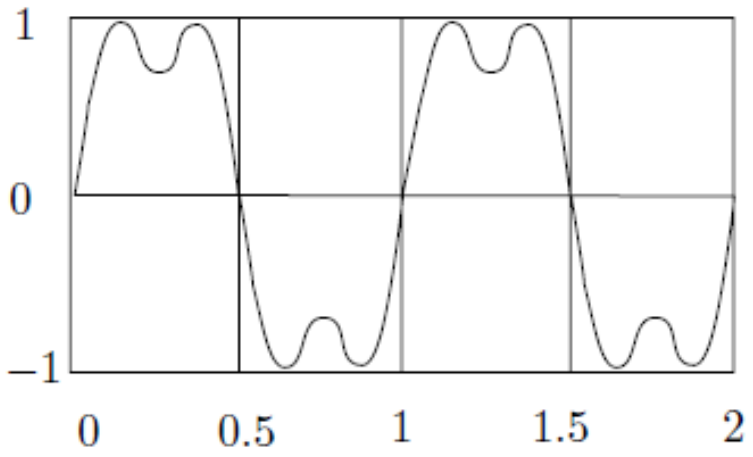
$$f(x,y)=\frac{1}{N^2}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} F(u,v)e^{\frac{j2\pi(ux+vy)}{N}}\ ,\ x,y=0,1,\ldots,N-1$$

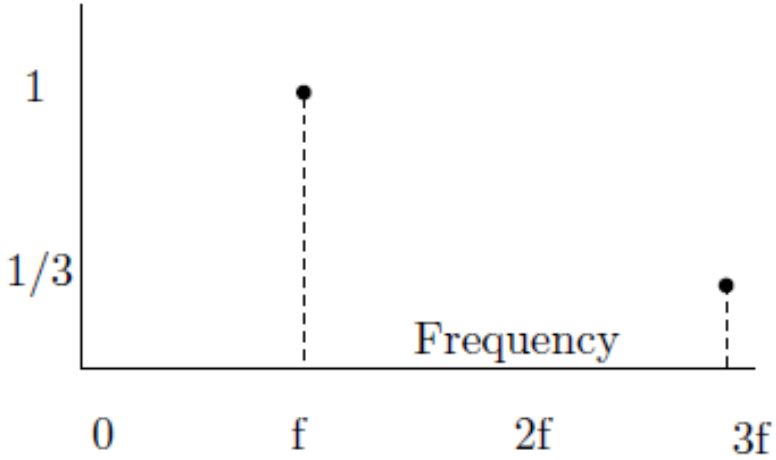
Fourier Transform Example



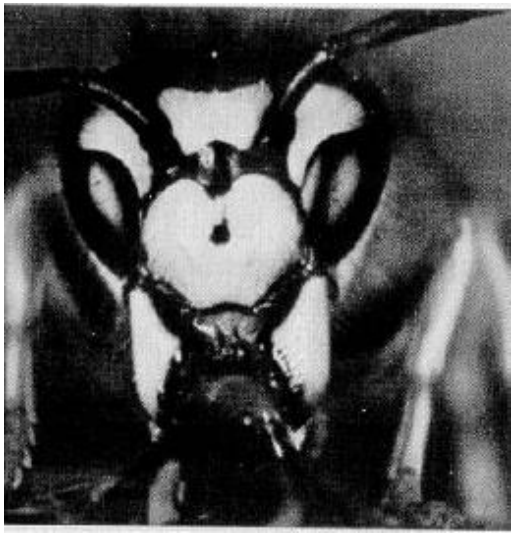


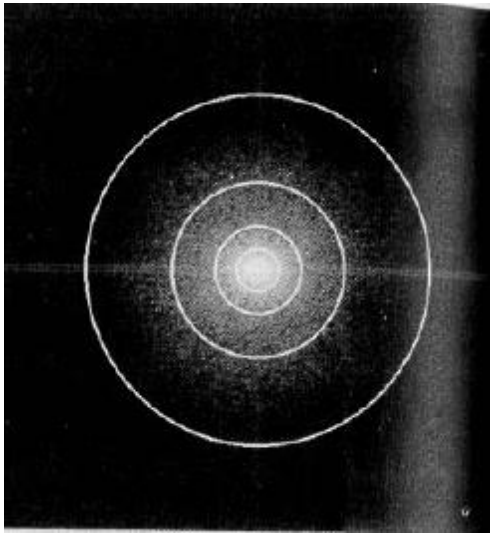
$$g(t) = \sin(2\pi ft) + (1/3)\sin(2\pi(3f)t)$$





Fourier-Based Compression





The magnitude of the FT decreases, as *u*, *v* increase!

$$\hat{f}(x,y) = \frac{1}{N} \sum_{u=0}^{\text{K-1}} \sum_{v=0}^{\text{K-1}} F(u,v) e^{\frac{j2\pi(ux+vy)}{N}}, \quad \text{x,y=0,1,...,N-1}$$

$$\sum_{x,y} (\hat{f}(x,y) - f(x,y))^2 \text{ is very small !!}$$

K << N

Discrete Cosine Transform (DCT)

- **This important transform (DCT for short) has originated by Ahmed et al. in 1974.**
- **The DCT in one dimension is given by:**

$$F(\omega)=\sum_{t=0}^{N-1}\alpha_\omega f(t)\cos\left(\frac{(2t+1)\pi\omega}{2N}\right),\quad \omega=0,1,\ldots,N-1$$

$$f(t)=\sum_{\omega=0}^{N-1}\alpha_\omega F(\omega)\cos\left(\frac{(2t+1)\pi\omega}{2N}\right),\quad t=0,1,\ldots,N-1$$

$$\alpha_\omega=\begin{cases}\sqrt{\frac{1}{N}} & \omega=0\\ \sqrt{\frac{2}{N}} & \omega>0\end{cases}$$

- **The first coefficient F(0) *is called the DC coefficient, and the rest are referred to as the AC coefficients.***
- **Notice that the coefficients are real numbers even if the input data consists of integers.**
- **Similarly, the coefficients may be positive or negative even if the input data consists of nonnegative numbers only.**

Discrete Cosine Transform (DCT)

- **The DCT in two dimensions is given by:**

$$F(u,v) = \sum_{x=0}^{N-1}\sum_{y=0}^{M-1} \alpha_u \alpha_v f(x,y) \cos\left(\frac{(2x+1)u\pi}{2N}\right) \cos\left(\frac{(2y+1)v\pi}{2M}\right) \quad For \quad \begin{matrix} u = 0,1,\ldots,N-1 \\ v = 0,1,\ldots,M-1 \end{matrix}$$

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{M-1} \alpha_u \alpha_v F(u,v) \cos\left(\frac{(2x+1)u\pi}{2N}\right) \cos\left(\frac{(2y+1)v\pi}{2M}\right) \quad For \quad \begin{matrix} x = 0,1,\ldots,N-1 \\ y = 0,1,\ldots,M-1 \end{matrix}$$

$$\alpha_u = \begin{cases} \sqrt{\frac{1}{N}} & u = 0 \\ \sqrt{\frac{2}{N}} & u > 0 \end{cases} \qquad \alpha_v = \begin{cases} \sqrt{\frac{1}{M}} & v = 0 \\ \sqrt{\frac{2}{M}} & v > 0 \end{cases}$$

Discrete Cosine Transform (DCT)

- The important feature of the DCT, the feature that makes it so useful in data compression, is that it takes correlated input data and concentrates its energy in just the first few transform coefficients.

- If the input data consists of correlated quantities, then most of the N transform coefficients produced by the DCT are zeros or small numbers, and only a few are large (normally the first ones).

- The first coefficients contain the important (low-frequency) information and the later coefficients contain the less-important (high-frequency) information.

Discrete Cosine Transform (DCT)

- Compressing data with the DCT is therefore done by quantizing the coefficients.
- The small ones are quantized coarsely (possibly all the way to zero), and the large ones can be quantized finely to the nearest integer.
- After quantization, the coefficients (or variable-length codes assigned to the coefficients) are written on the compressed stream.
- Decompression is done by performing the inverse DCT on the quantized coefficients.

- This results in data items that are not identical to the original ones but are not much different.

DCT Example

- We apply the one dimension DCT on the set of eight correlated data items p = (12, 10, 8, 10, 12, 10, 8, 11) and it results in the eight coefficients:
  - 28.6375, 0.571202, 0.46194, 1.757, 3.18198, −1.72956, 0.191342, −0.308709.
- These can be fed to the IDCT and transformed by it to precisely reconstruct the original data (except for small errors caused by limited machine precision).
- Our goal, however, is to compress the data by quantizing the coefficients. We first quantize them to 28.6, 0.6, 0.5, 1.8, 3.2,−1.8, 0.2,−0.3, and apply the IDCT to get back:
  - 12.0254, 10.0233, 7.96054, 9.93097, 12.0164, 9.99321, 7.94354, 10.9989.

DCT Example

- **We then quantize the coefficients even more, to**
  - ***28, 1, 1, 2, 3, −2, 0, 0***
- ***and apply the IDCT* to get back:**
  - **12*.1883, 10.2315, 7.74931, 9.20863, 11.7876, 9.54549, 7.82865, 10.6557***
- **Finally, we quantize the coefficients to**
  - ***28, 0, 0, 2, 3, −2, 0, 0***
- ***and still get back from the* IDCT the sequence:**
  - **11*.236, 9.62443, 7.66286, 9.57302, 12.3471, 10.0146, 8.05304, 10.6842***
- **where the largest difference between an original value (12) and a reconstructed one (11.236) is 0.764 (or 6.4% of 12).**

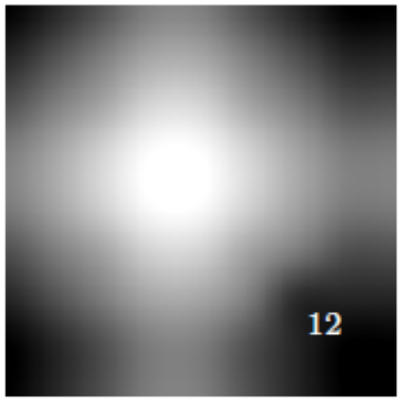
DCT Example

- It seems magical that the eight original data items can be reconstructed to such high precision from just four transform coefficients.

- The explanation, however, relies on the following arguments instead of on magic:
  - 1) The IDCT is given all eight transform coefficients, so it knows the positions, not just the values, of the nonzero coefficients.
  - 2) The first few coefficients (the large ones) contain the important information of the original data items. The small coefficients, the ones that are quantized heavily, contain less important information (in the case of images, they contain the image details).
  - 3) The original data is redundant because of data correlation.

DCT Example

00	10	20	30	30	20	10	00
10	20	30	40	40	30	20	10
20	30	40	50	50	40	30	20
30	40	50	60	60	50	40	30
30	40	50	60	60	50	40	30
20	30	40	50	50	40	30	20
10	20	30	40	40	30	12	10
00	10	20	30	30	20	10	00



A Continuous-Tone Pattern.

239	1.19	−89.76	−0.28	1.00	−1.39	−5.03	−0.79
1.18	−1.39	0.64	0.32	−1.18	1.63	−1.54	0.92
−89.76	0.64	−0.29	−0.15	0.54	−0.75	0.71	−0.43
−0.28	0.32	−0.15	−0.08	0.28	−0.38	0.36	−0.22
1.00	−1.18	0.54	0.28	−1.00	1.39	−1.31	0.79
−1.39	1.63	−0.75	−0.38	1.39	−1.92	1.81	−1.09
−5.03	−1.54	0.71	0.36	−1.31	1.81	−1.71	1.03
−0.79	0.92	−0.43	−0.22	0.79	−1.09	1.03	−0.62

Its DCT Coefficients.

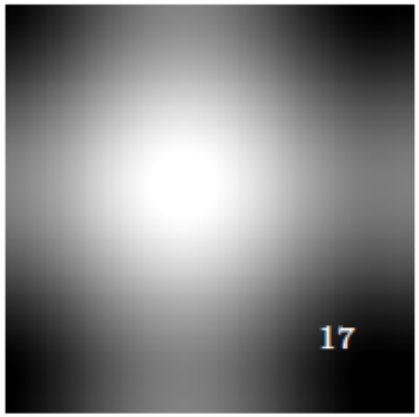
DCT Example

239	1	-90	0	0	0	0	0
0	0	0	0	0	0	0	0
-90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

: Quantized Heavily to Just Four Nonzero Coefficients.

0.65	9.23	21.36	29.91	29.84	21.17	8.94	0.30
9.26	17.85	29.97	38.52	38.45	29.78	17.55	8.91
21.44	30.02	42.15	50.70	50.63	41.95	29.73	21.09
30.05	38.63	50.76	59.31	59.24	50.56	38.34	29.70
30.05	38.63	50.76	59.31	59.24	50.56	38.34	29.70
21.44	30.02	42.15	50.70	50.63	41.95	29.73	21.09
9.26	17.85	29.97	38.52	38.45	29.78	17.55	8.91
0.65	9.23	21.36	29.91	29.84	21.17	8.94	0.30

Results of IDCT.



Differential Encoding

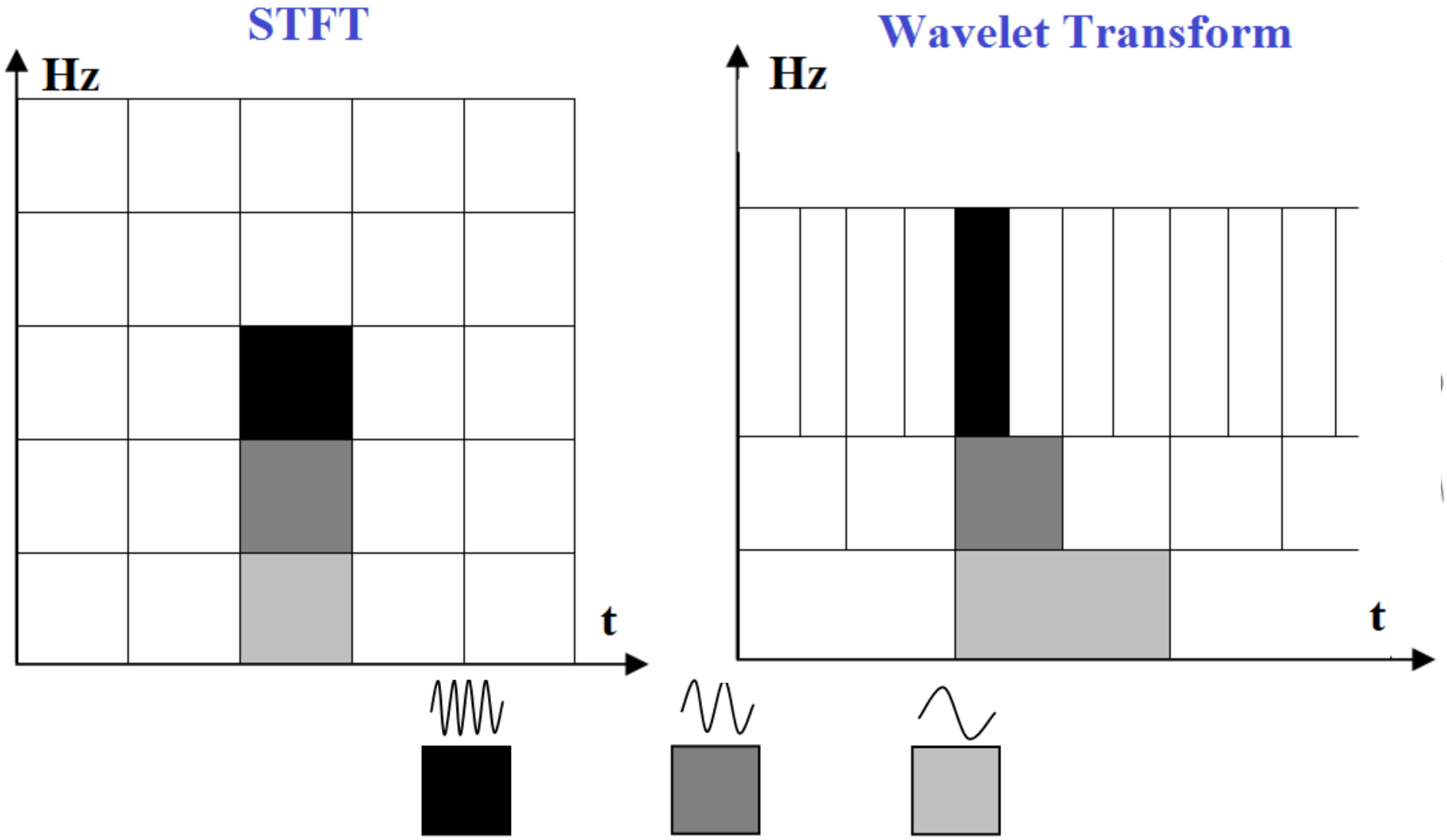
- DC coefficient is the largest in the transformed matrix.
- DC coefficient varies slowly from one block to the next.
- Only the difference in value of the DC coefficients is encoded. Number of bits required to encode is reduced.

- The difference values are encoded in the form (N, V) where field N indicates the number of bits needed to encode the value and the field V indicates the binary form.

Wavelet Transform

- **The Fourier transform is useful and popular, having applications in many areas. It has, however, one drawback:**
  - **It shows the frequency content of a function *f(t), but it* does not specify where (i.e., for what values of *t) the function has low or high frequencies.***
  - **The reason for this is that the basis functions (sine and cosine) used by this transform are infinitely long.**
  - **They pick up the different frequencies of *f(t) regardless of where* they are located.**
- **A better transform should specify the frequency content of *f(t) as a function of t.***
  - **It should produce a two-dimensional function or array of numbers *W(a, b) that describes the frequency content of f(t) for* different values of *t.***

Wavelet Transform



Wavelet Transform

- The wavelet transform is such a method. It has been developed, researched, and applied to many areas of science and engineering since the early 1980s, although its roots go much earlier.
- The main idea is to select a mother wavelet, a function that is nonzero in some small interval, and use it to explore the properties of f(t) in that interval.
- The mother wavelet is then shifted to another interval of t and used again in the same way.
  - Parameter b specifies the translation.
- Different frequency resolutions of f(t) are explored by scaling the mother wavelet with a scale factor a.

Wavelet Transform

- The wavelet transform of a function f(t) involves a mother wavelet ψ(t). The mother wavelet can be any real or complex continuous function that satisfies the following properties:
  - 1. The total area under the curve of the function is zero.
  - 2. The total area of $|\psi(t)|^2$ is finite.
- Property 1 suggests a function that oscillates above and below the t axis. Such a function tends to have a wavy appearance.
- Property 2 implies that the energy of the function is finite, suggesting that the function is localized in some finite interval and is zero, or almost zero, outside this interval (compactly supported).
- These properties justify the name "wavelet."
  - An infinite number of functions satisfy these conditions, and some of them have been researched and are commonly used for wavelet transforms.

Wavelet Transform

- **The CWT of a square integrable function f(t) is defined as:**

$$W(a,b) = \int_{-\infty}^{+\infty} f(t)\psi^{*}_{a,b}(t)dt$$

  - **The transform is a function of the two real parameters a and b.**
  - **The $*$ denotes the complex conjugate.**
- **The translated and scaled version of the mother wavelet($\psi_{a,b}(t)$) is defined as:**

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right)$$

  - **The quantity $1/\sqrt{a}$ is a normalizing factor that ensures that the energy of ψ(t) remains independent of a and b.**
- **Mathematically, the transform is the inner product of the two functions f(t) and $\psi_{a,b}(t)$.**

Wavelet Transform

- **If W(a, b) is the CWT of a function f(t) with a wavelet ψ(t), then the inverse CWT is defined by:**

$$f(t) = \frac{1}{C}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\frac{1}{|a|^2}W(a,b)\psi_{a,b}(t)\,da\,db$$

- **where the quantity C is defined as:** $C = \int_{-\infty}^{+\infty}\frac{|\Psi(\omega)|^2}{|\omega|}\,d\omega$

- **Ψ(ω) the Fourier transform of ψ(t):** $\Psi(\omega) = \int_{-\infty}^{+\infty}\psi(t)e^{-i\omega t}\,dt$
  - **The inverse CWT exists if C is positive and finite.**
  - **Since C is defined by means of Ψ, which itself is defined by means of the wavelet ψ(t), the requirement that C be positive and finite imposes another restriction, called the admissibility condition, on the choice of wavelet.**

Discrete Wavelet Transform

- **Recall that the CWT is the integral of the product $f(t)\psi^*((t-b)/a)$, where a, the scale factor, and b, the time shift, can be any real numbers.**

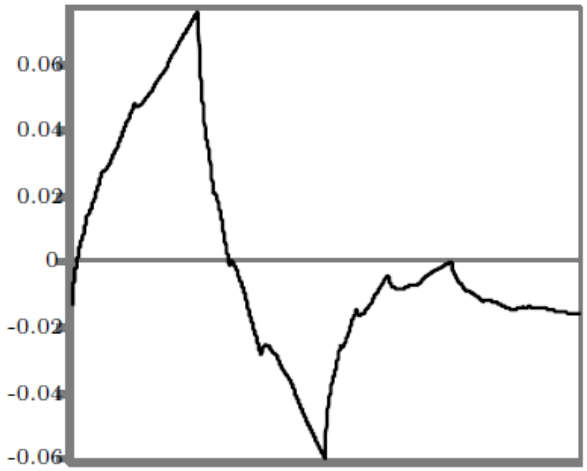
- **The corresponding calculation for the discrete case (the DWT) involves a convolution, but experience shows that the quality of this type of transform depends heavily on two factors,**
  - **the choice of scale factors and time shifts,**
  - **and the choice of wavelet.**

- **In practice, the DWT is computed with scale factors that are negative powers of 2 and time shifts that are nonnegative powers of 2.**

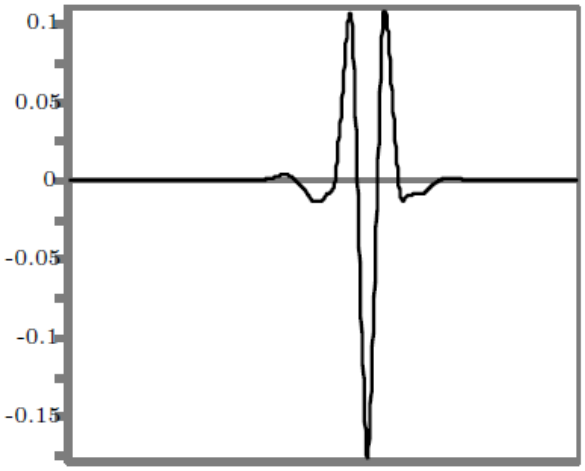
Wavelet Transform

- **Well-known mother wavelet are:**
  - **The Daubechies family of filters maximize the smoothness of the father wavelet (the scaling function) by maximizing the rate of decay of its Fourier transform.**
  - **The Haar wavelet can be considered the Daubechies filter of order 2. It is the oldest filter. It is simple to work with, but it does not produce best results, since it is not continuous.**
  - **The Coifman filter (or "Coiflet") of order p (where p is a positive integer) gives both the mother and father wavelets 2p zero moments.**
  - **Symmetric filters (symmlets) are the most symmetric compactly supported wavelets with a maximum number of zero moments.**
  - **Morlet and Mexican Hat are other mother wavelets used in practice.**

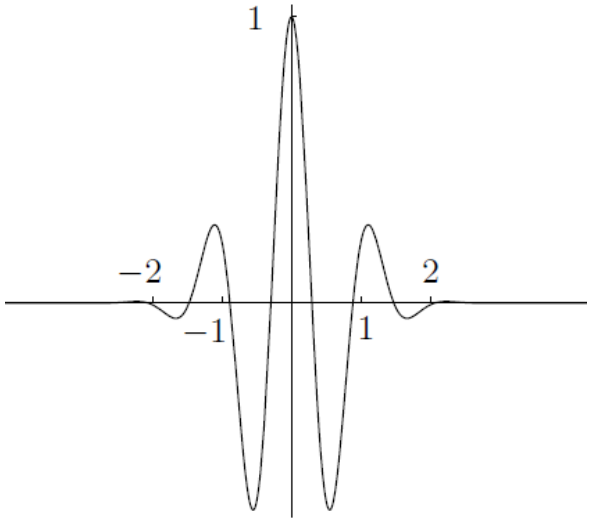
Wavelet Transform



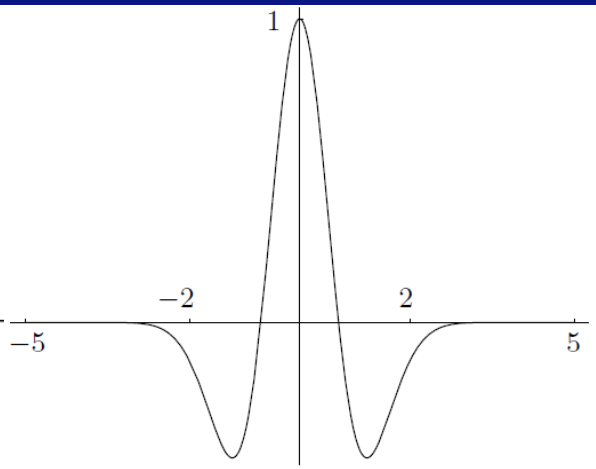
Daubechies 4



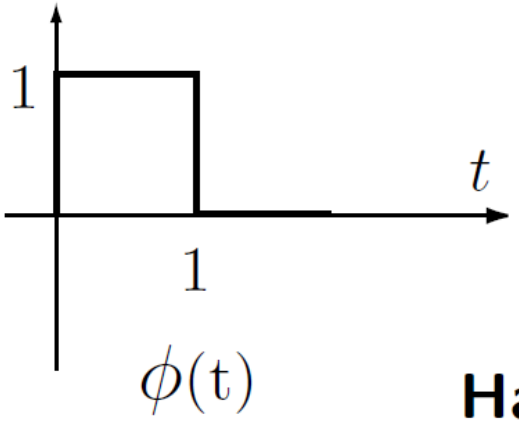
Coifman 18

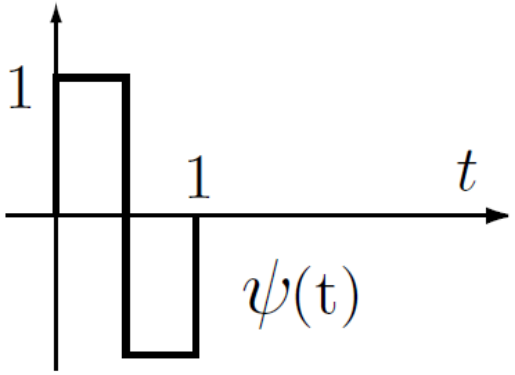


Morlet



Mexican Hat

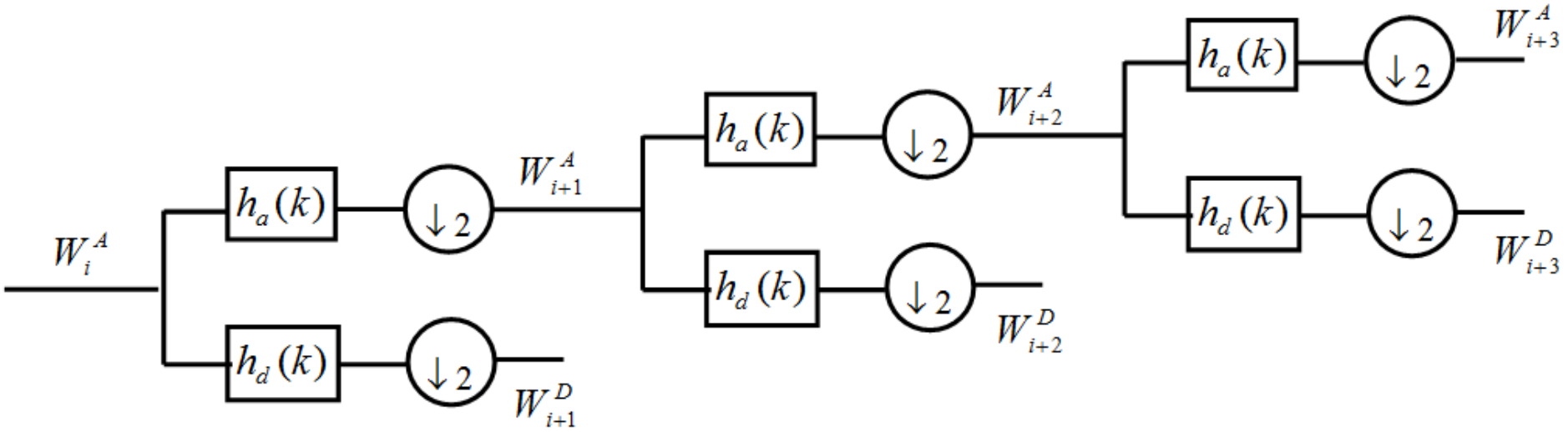




Haar

Wavelet Transform

- **Technically wavelet transforms are special kinds of linear transformations. Easiest to think of them as filters.**
  - **The filters depend only on a constant number of values. (bounded support)**
  - **Preserve energy (norm of the pixels = norm of the coefficients)**
  - **Inverse filters also have bounded support.**

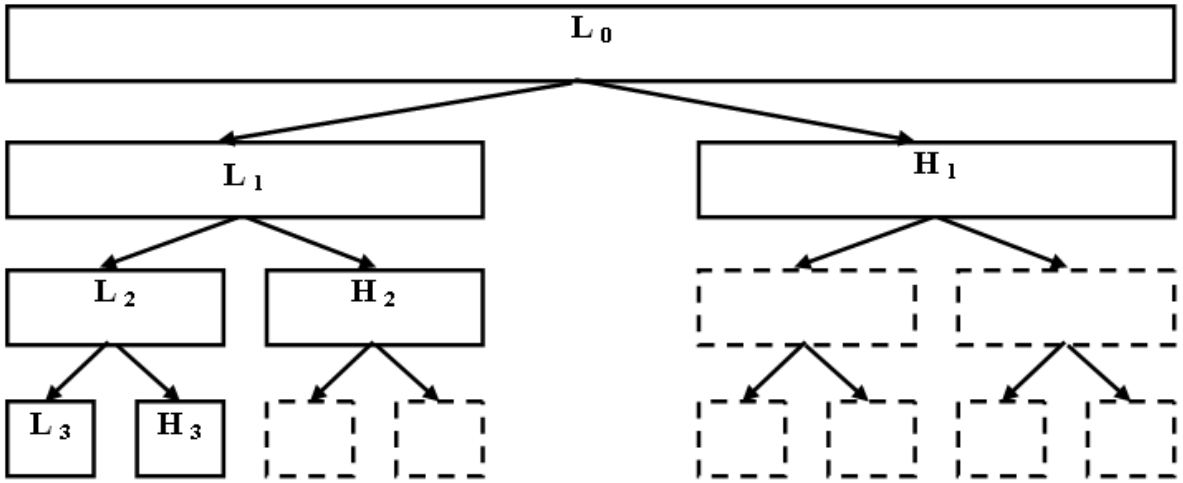


Wavelet Transform

- Traditional DWT is identical to a hierarchical subband system where the subbands are logarithmically spaced in frequency .
- By decomposition of a signal using DWT (we consider 1D signal for simplification), the original signal is transformed into two subbands:
  - low resolution subband (L) and
  - high resolution subband (H).
- The L subband can be further decomposed into two subbands labeled as L2 and H2.
- At each step of the algorithm, only the low resolution part is split into a low-pass and band-pass sets of coefficients.

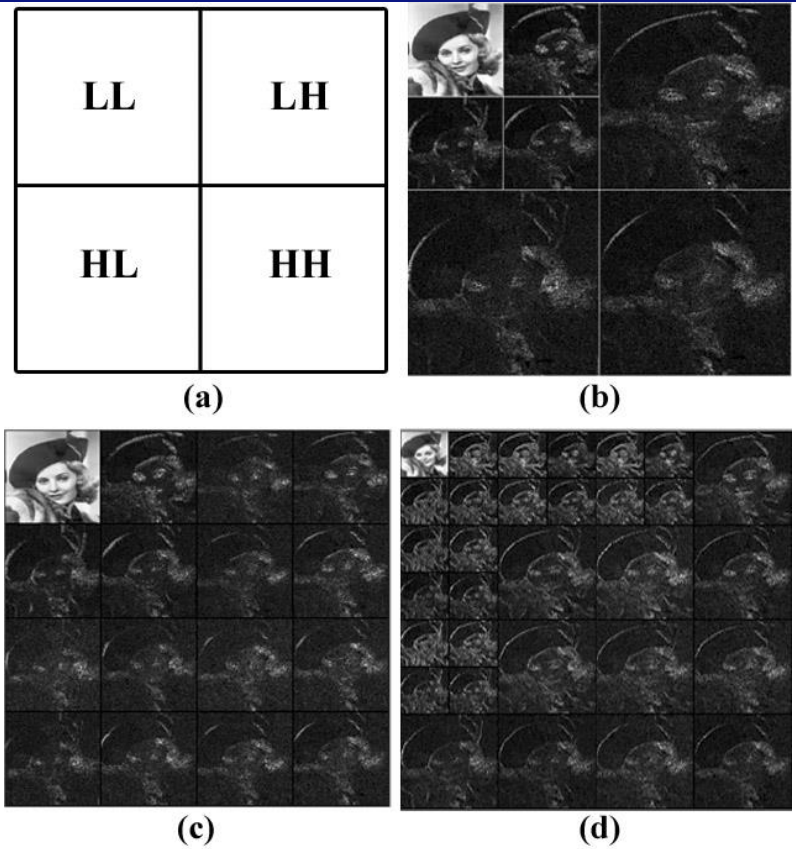
Wavelet Transform

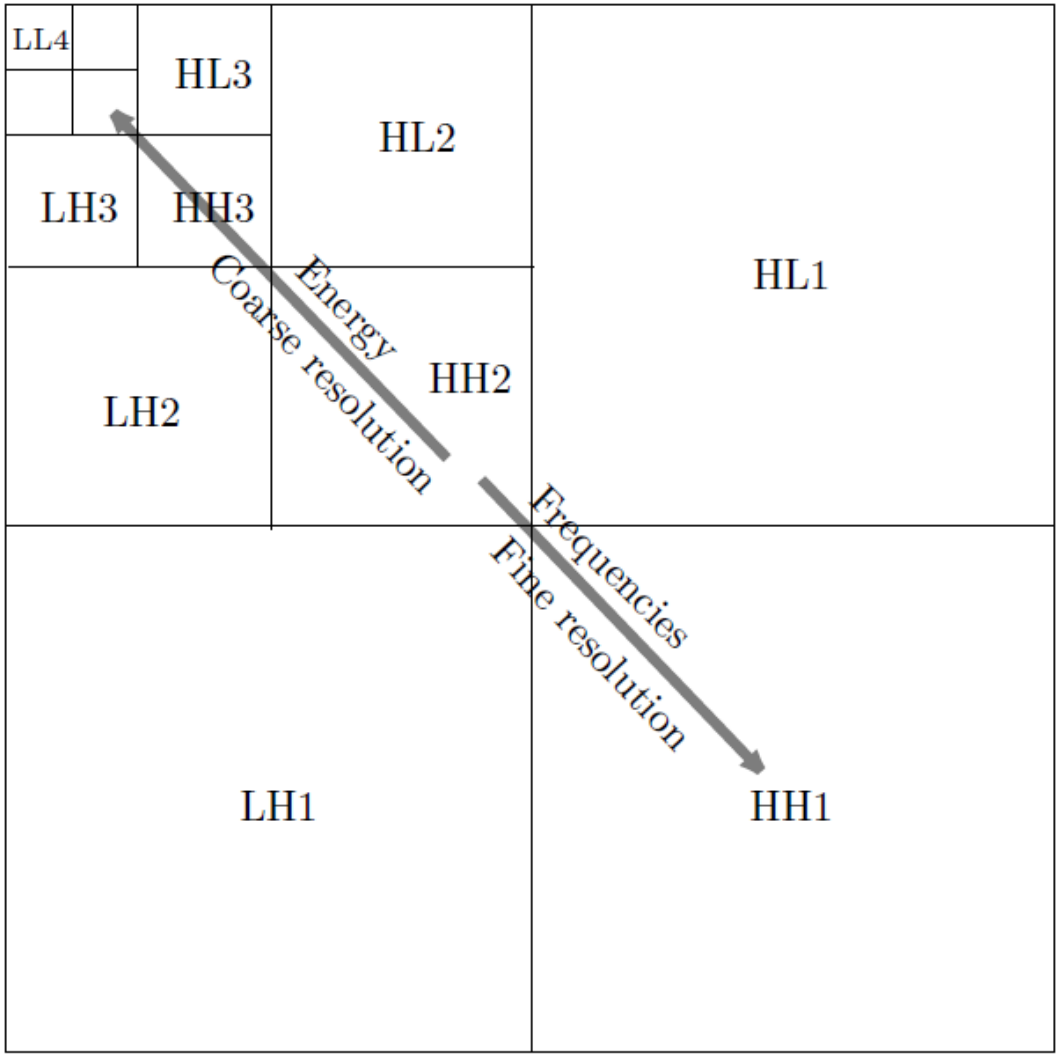
- In traditional DWT, as it is shown in below, the original signal L_0 is transformed into $[L_3 | H_3 | H_2 | H_1]$, which corresponds to a certain choice of the basis representation.



- However, it could be decided at each stage whether to split only the low resolution or the high resolution part.
- Splitting the high resolution part would result a full binary tree of possible basis functions or the wavelet packet transforms.

2D Wavelet Transform

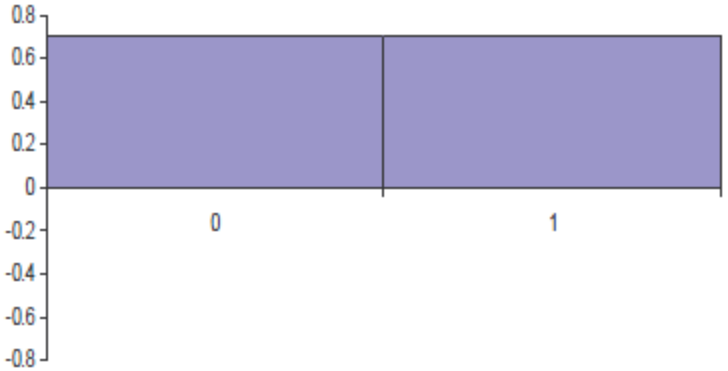


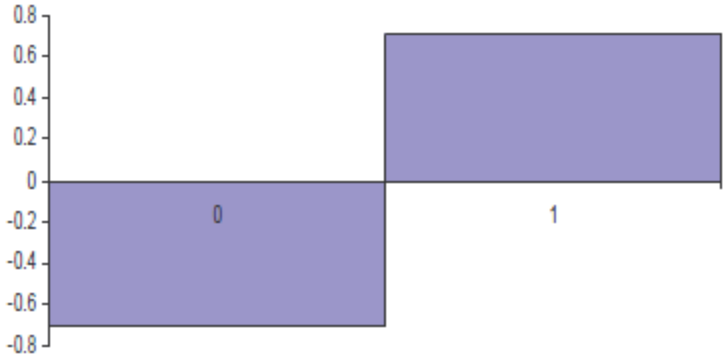


- a) Decomposition Scheme
- b) Traditional 2D DWT
- c) Wavelet Packet
- d) Adaptive Wavelet Packet

Haar Filters

$$\text{low pass} = \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \qquad \text{high pass} = -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}$$





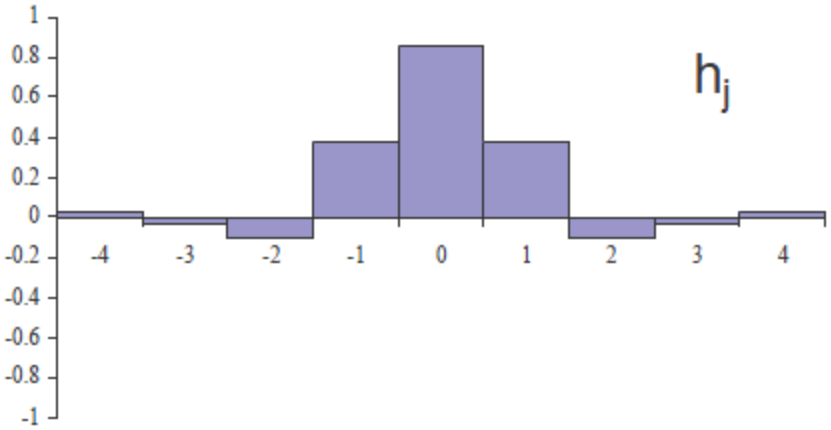
$$\text{low pass} \quad B[i] = \frac{1}{\sqrt{2}} A[2i] + \frac{1}{\sqrt{2}} A[2i+1], \quad 0 \le i < \frac{n}{2}$$

$$\text{high pass} \quad B[n/2+i] = -\frac{1}{\sqrt{2}} A[2i] + \frac{1}{\sqrt{2}} A[2i+1], \quad 0 \le i < \frac{n}{2}$$

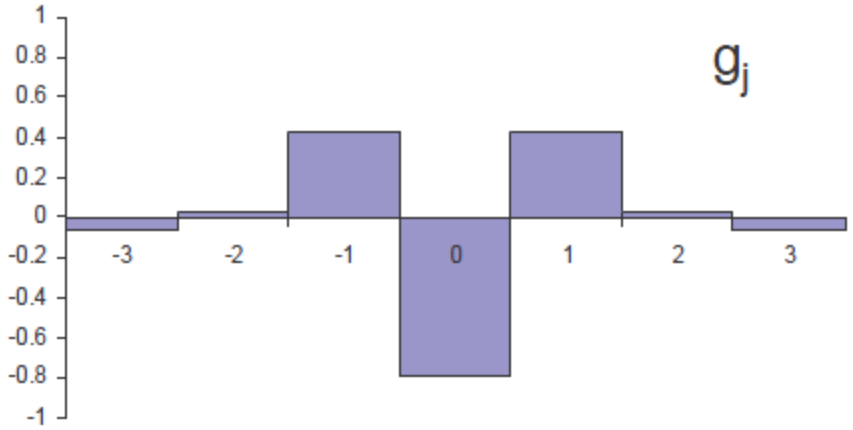
Want the sum of squares of the filter coefficients = 1

Daubechies 9/7 Filters

low pass filter



high pass filter



low pass $$B[i] = \sum_{j=-4}^{4} h_j A[2i + j], \quad 0 \le i < \frac{n}{2}$$

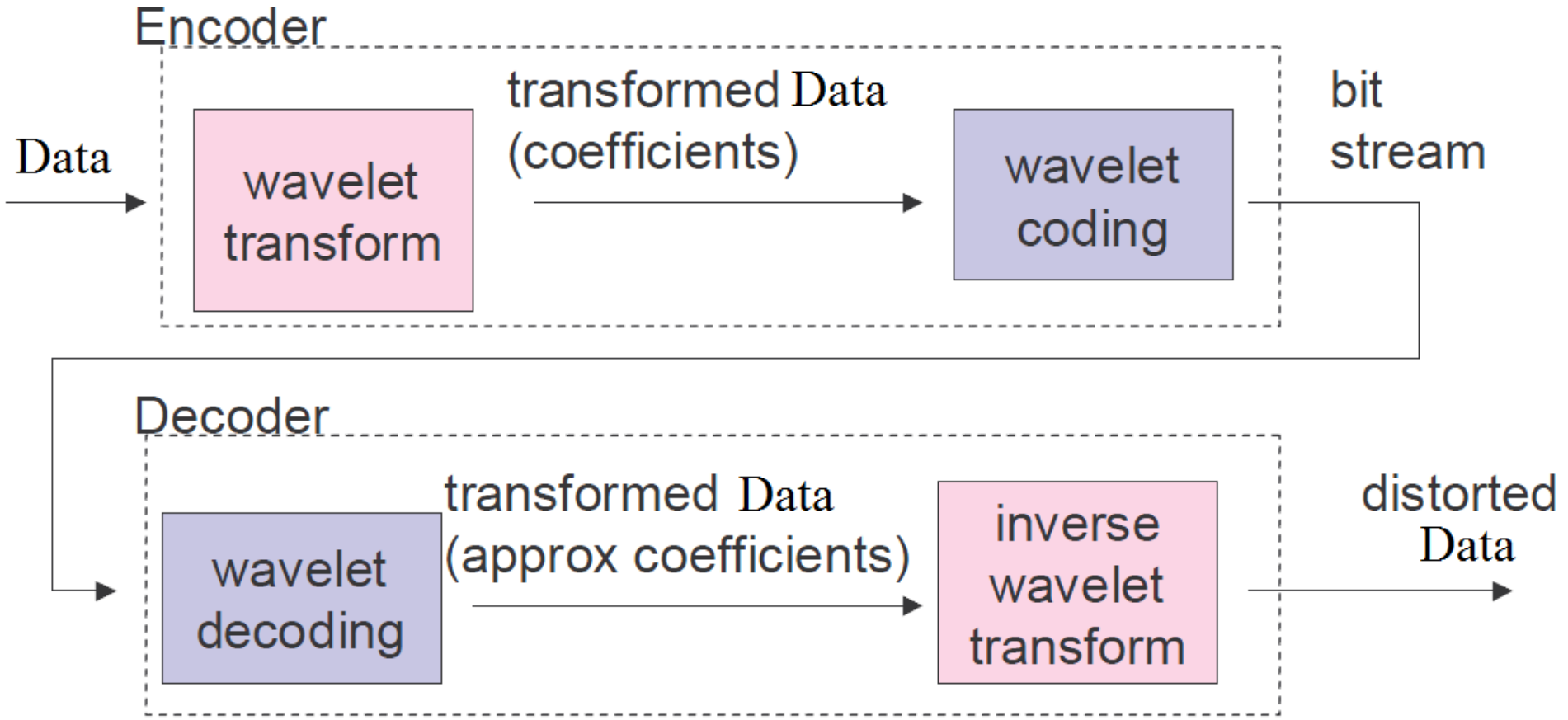
high pass $$B[n/2 + i] = \sum_{j=-3}^{3} g_j A[2i + j], \quad 0 \le i < \frac{n}{2}$$

reflection used near boundaries

Integer Wavelet Transform (IWT)

- The DWT is simple but has an important drawback, namely, it uses noninteger filter coefficients, which is why it produces noninteger transform coefficients.
- There are several ways to modify the basic DWT such that it produces integer transform coefficients or integer wavelet transform (IWT):
  - By quantizing the filter coefficients.
  - By truncating the transform coefficients.
- The transform is reversible, i.e., the data can be fully reconstructed from the (integer) transform coefficients.
- This IWT can be used to compress the data either
  - lossy (by quantizing the transform coefficients) or
  - losslessly (by entropy coding the transform coefficients).

Wavelet Transform Compression



- **A wavelet transform decomposes the data into a low resolution version and details. The details are typically very small so they can be coded in very few bits.**

Wavelet Transform
Bit-plane coding

most of the details are small so they are very dark.

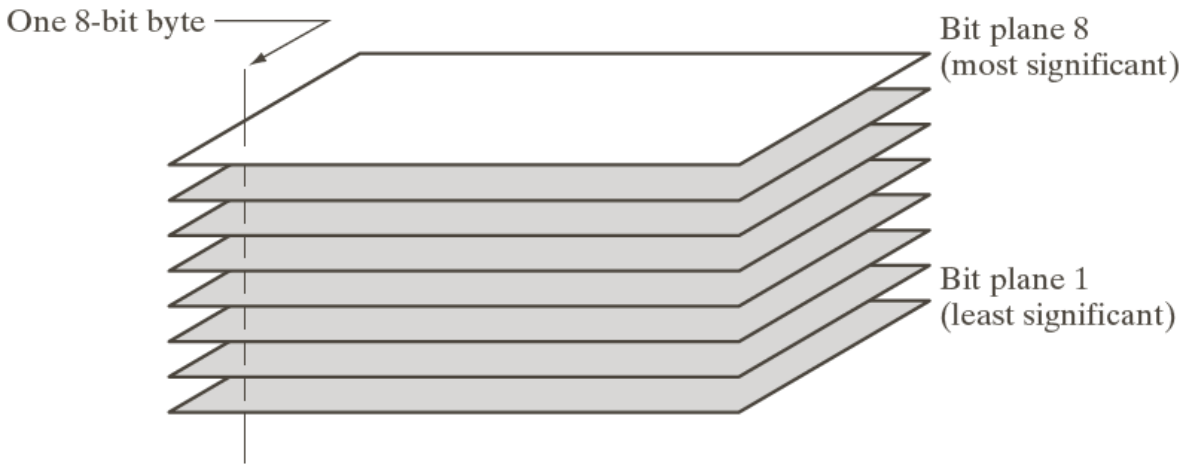
Wavelet-Based Compression

- Normalize the Wavelet coefficients to be between –1 and 1
- Normalized Wavelet coefficients are transmitted in bit-plane order.
  - Transmit one bit-plane at a time
  - In most significant bit planes most coefficients are 0 so they can be coded efficiently.
  - Only some of the bit planes are transmitted. This is where fidelity is lost when compression is gained.
- For each bit-plane
  - Significance pass: Find the newly significant coefficients, transmit their signs.
  - Refinement pass: transmit the bits of the known significant coefficients.

Bit-plane coding (inter data redundancy)

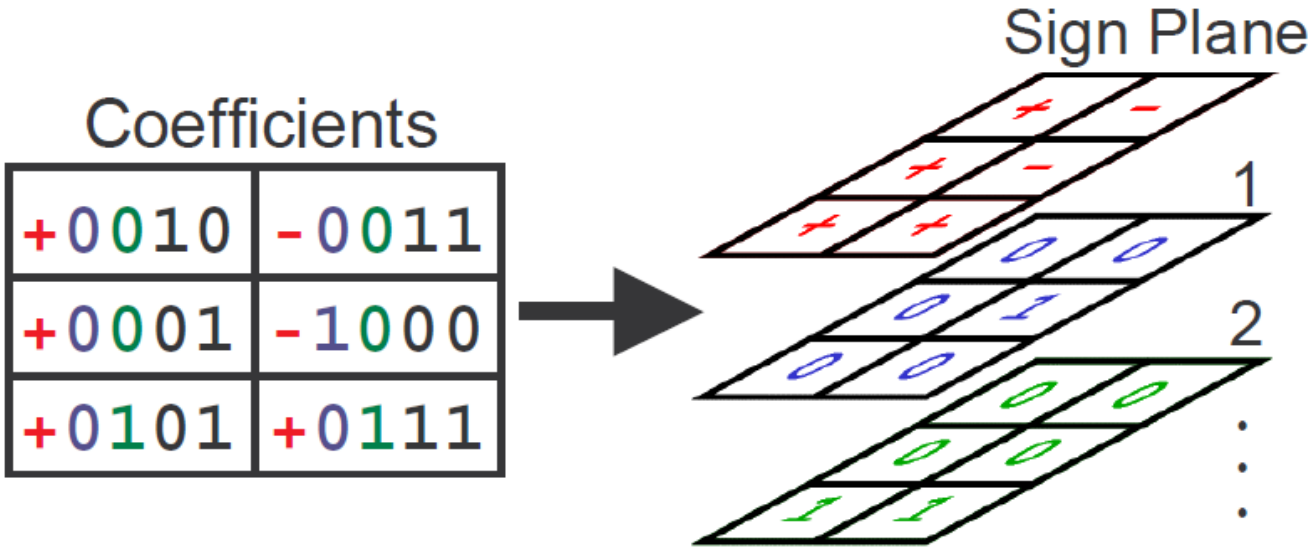
- An effective technique to reduce inter data redundancy is to process each bit plane individually.

(1) Decompose an data into a series of binary data.

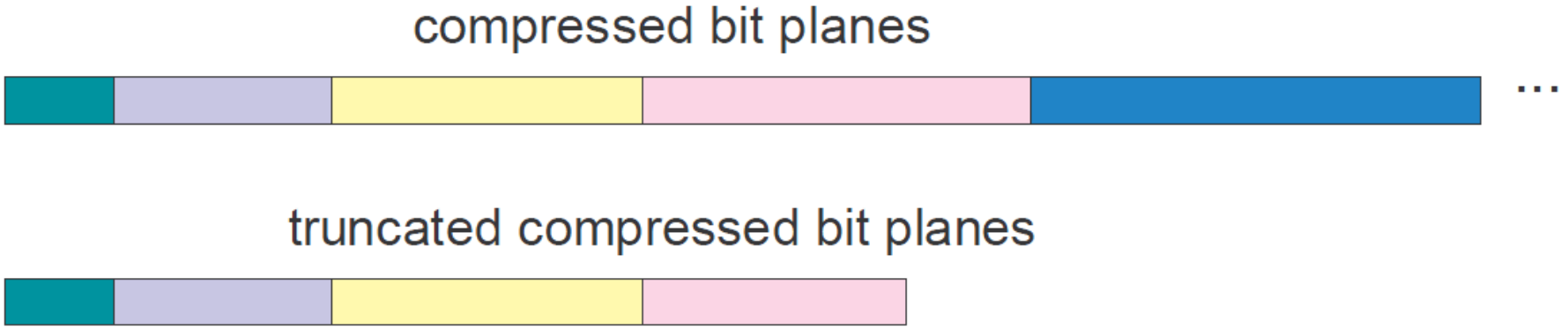


(2) Compress each binary data (e.g., using run-length coding)

Wavelet-Based Compression



- **Natural progressive transmission**



Wavelet Coding Methods

- EZW - Shapiro, 1993
  - Embedded Zerotree coding.
- SPIHT - Said and Pearlman, 1996
  - Set Partitioning in Hierarchical Trees coding. Also uses "zerotrees".
- ECECOW - Wu, 1997
  - Uses arithmetic coding with context.
- EBCOT – Taubman, 2000
  - Uses arithmetic coding with different context.
- JPEG 2000 – new standard based largely on EBCOT
- GTW – Hong, Ladner 2000
  - Uses group testing which is closely related to Golomb codes
- PACW - Ladner, Askew, Barney 2003
  - Like GTW but uses arithmetic coding

Burrows-Wheeler Transform

- The BWT algorithm was introduced by Burrows and Wheeler in 1994.
- The implementation of the method is simple and fast.
- The Burrows-Wheeler transform algorithm is the base of a recent powerful software program for conservative data compression bzip.
- The Burrows-Wheeler (BW) method works in a block mode, where the input stream is read block by block and each block is encoded separately as one string.
  - The method is therefore referred to as **block sorting**.
- The BW method is general purpose, it works well on images, sound, and text, and can achieve very high compression ratios (1 bit per byte or even better).

Burrows-Wheeler Transform

- **The main idea of the BWT method is to start with a string S of n symbols and to scramble them into another string L that satisfies two conditions:**
  - **1) Any region of L will tend to have a concentration of just a few symbols.**
    - **Another way of saying this is, if a symbol a is found at a certain position in L, then other occurrences of a are likely to be found nearby.**
    - **This property means that L can easily and efficiently be compressed with the move-to-front method, perhaps in combination with RLE.**
    - **This also means that the BWT method will work well only if n is large (at least several thousand symbols per string).**
  - **2) It is possible to reconstruct the original string S from L (a little more data may be needed for the reconstruction, in addition to L, but not much).**

Burrows-Wheeler Transform

- The mathematical term for scrambling symbols is permutation, and it is easy to show that a string of n symbols has ***n!*** permutations.
- The BW codec proceeds in the following steps:
  - 1) String L is created, by the encoder, as a permutation of S. Some more information, denoted by I, is also created, to be used later by the decoder in step 3.
  - 2) The encoder compresses L and I and writes the results on the output stream. This step typically starts with RLE, continues with move-to-front coding, and finally applies Huffman coding.
  - 3) The decoder reads the output stream and decodes it by applying the same methods as in 2 above but in reverse order. The result is string L and variable I.
  - 4) Both L and I are used by the decoder to reconstruct the original string S.

Burrows-Wheeler Encoder

- **The first step is to understand how string L is created from S, and what information needs to be stored in I for later reconstruction.**
  - **Given an input string of n symbols, the encoder constructs n×n matrix where it stores string S in the top row, followed by n−1 copies of S, each circularly shifted (rotated) one symbol to left.**
  - **The matrix is then sorted lexicographically by rows, producing the sorted matrix.**
    - **Notice that every row and every column of each of the two matrices is a permutation of S and thus contains all n symbols of S.**
  - **The permutation L selected by the encoder is the last column of the sorted matrix.**
  - **The only other information needed to reconstruct S from L is the row number of the original string in the sorted matrix.**
    - **This number is stored in I.**

Burrows-Wheeler Example

- **Consider a string S = 'ACCELERATE' of n=10 characters, stored in a one-dimensional array.**
  - **1) Repeating the circular shift n-1 times on the string S to generate the nxn matrix:**

	0	1	2	3	4	5	6	7	8	9
0	A	C	C	E	L	E	R	A	T	E
1	C	C	E	L	E	R	A	T	E	A
2	C	E	L	E	R	A	T	E	A	C
3	E	L	E	R	A	T	E	A	C	C
4	L	E	R	A	T	E	A	C	C	E
5	E	R	A	T	E	A	C	C	E	L
6	R	A	T	E	A	C	C	E	L	E
7	A	T	E	A	C	C	E	L	E	R
8	T	E	A	C	C	E	L	E	R	A
9	E	A	C	C	E	L	E	R	A	T

Burrows-Wheeler Example

- **2) We now sort the rows of the matrix in lexicographic order so the matrix becomes:**

	0	1	2	3	4	5	6	7	8	9
0	A	C	C	E	L	E	R	A	T	E
1	A	T	E	A	C	C	E	L	E	R
2	C	C	E	L	E	R	A	T	E	A
3	C	E	L	E	R	A	T	E	A	C
4	E	A	C	C	E	L	E	R	A	T
5	E	L	E	R	A	T	E	A	C	C
6	E	R	A	T	E	A	C	C	E	L
7	L	E	R	A	T	E	A	C	C	E
8	R	A	T	E	A	C	C	E	L	E
9	T	E	A	C	C	E	L	E	R	A

Burrows-Wheeler Example

- **3) We name the last column L, which is what we need in the BWT for encoding, where S1 indicates the row number of original string S.**

	F								L	
0	A	C	C	E	L	E	R	A	T	E	$\leftarrow s_1$
1	A	T	E	A	C	C	E	L	E	R	
2	C	C	E	L	E	R	A	T	E	A	
3	C	E	L	E	R	A	T	E	A	C	
4	E	A	C	C	E	L	E	R	A	T	
5	E	L	E	R	A	T	E	A	C	C	
6	E	R	A	T	E	A	C	C	E	L	
7	L	E	R	A	T	E	A	C	C	E	
8	R	A	T	E	A	C	C	E	L	E	
9	T	E	A	C	C	E	L	E	R	A	

Burrows-Wheeler Example

- L can be stored in a one-dimensional array, and can be written as:

0	1	2	3	4	5	6	7	8	9
E	R	A	C	T	C	L	E	E	A

- Similarly, if we name the first column of the matrix F, then F can be derived from L by just sorting L. In other words, we should get F if we sort L. This is the sorted L, or F:

0	1	2	3	4	5	6	7	8	9
A	A	C	C	E	E	E	L	R	T

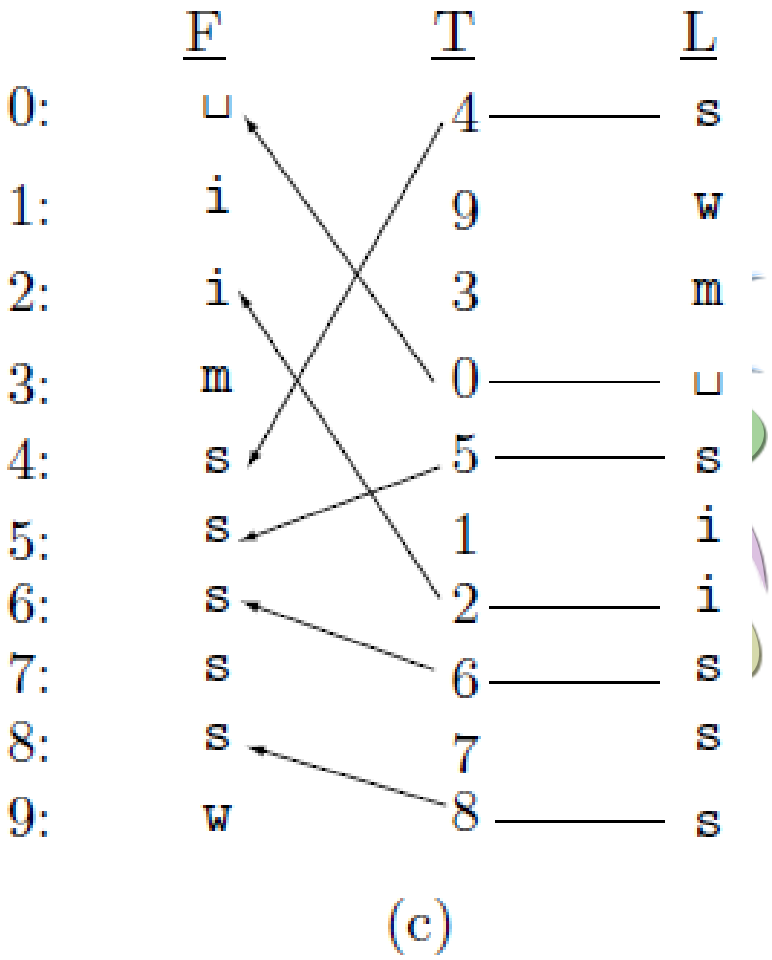
- The position of the original string S (i.e. 1) is also transmitted in BWT to the decoder for decoding purposes.

Burrows-Wheeler Decoder

- The decoder reads a compressed stream, decompresses it using Huffman and move to-front (and perhaps also RLE), and then reconstructs string S from the decompressed L in three steps:
  - 1. The first column of the sorted matrix is constructed from L.
    - This is a straightforward process, since F and L contain the same symbols (both are permutations of S) and F is sorted. The decoder simply sorts string L to obtain F.
  - 2. While sorting L, the decoder prepares an auxiliary array T that shows the relations between elements of L and F.
    - The first element of T is implying that the first symbol of L is located in what position of F. and so on.
  - 3. String F is no longer needed. The decoder uses L, I, and T to reconstruct S according to
    - S[n-1-i] $\leftarrow L[T_i[I]]$, *for* $i = 0, 1, \ldots, n-1$,
    - where $T_0[I] = I$, *and* $T_{i+1}[I] = T[T_i[I]]$.

Burrows-Wheeler Example

swiss␣miss	␣missswiss	
wiss␣misss	iss␣misssw	
iss␣misssw	issswiss␣m	
ss␣missswi	missswiss␣	
s␣missswis	s␣missswis	
␣missswiss	ss␣missswi	
missswiss␣	ssswiss␣mi	
issswiss␣m	sswiss␣mis	
ssswiss␣mi	swiss␣miss	
sswiss␣mis	wiss␣misss	
(a)	(b)	



(c)

Burrows-Wheeler Example

- I=8
- S[n-1-i] ← L[T_i[I]], for i = 0, 1, . . . , n − 1,
- where T_0[I] = I, and T_{i+1}[I] = T[T_i[I]].

- S[10-1-0]= L[T_0[I]= L[T_0[8]=L[8]=s
- S[10-1-1]= L[T_1[I]= L[T[T_0[8]]= L[7]=s
- S[10-1-2]= L[T_2[I]= L[T[T_1[8]]= L[6]=i
- S[10-1-3]= L[T_3[I]= L[T[T_2[8]]= L[2]=m
- S[10-1-4]= L[T_4[I]= L[T[T_3[8]]= L[3]=_
- S[10-1-5]= L[T_5[I]= L[T[T_4[8]]= L[0]=s
- S[10-1-6]= L[T_6[I]= L[T[T_5[8]]= L[4]=s
- S[10-1-7]= L[T_7[I]= L[T[T_6[8]]= L[5]=i
- S[10-1-8]= L[T_8[I]= L[T[T_7[8]]= L[1]=w
- S[10-1-9]= L[T_9[I]= L[T[T_8[8]]= L[9]=s

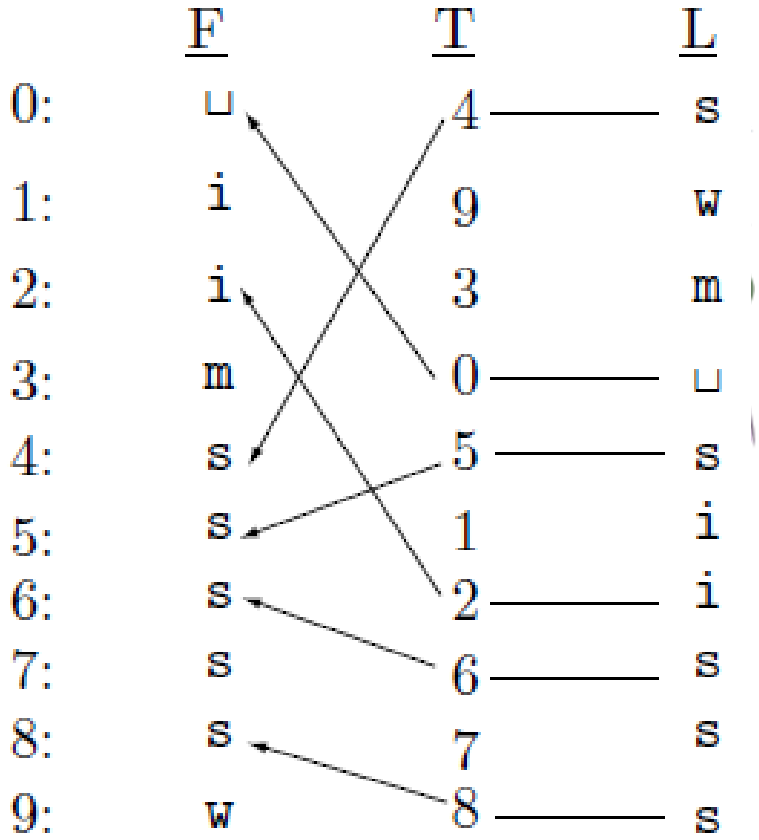


Image Compression

Image representation

- A digital image is a rectangular array of dots, or picture elements, arranged in m rows and n columns.
- The expression m×n is called the resolution of the image, and the dots are called pixels (except in the cases of fax images and video compression, where they are referred to as pels).
- The term "resolution" is sometimes also used to indicate the number of pixels per unit length of the image.
  - Thus, dpi stands for dots per inch.

Different Type of Images

- **Continuous-tone image:**
  - This type of image is a relatively natural image which may contain areas with colours that seem to vary continuously as the eye moves along the picture area.
- **Discrete-tone image(graphical image or synthetic):**
  - This type of image is a relatively artificial image. There is usually no noise and blurring as there is in a natural image. Adjacent pixels in a discrete-tone image are often either identical or vary significantly in value.
- **Cartoon-like image:**
  - This type of image may consist of uniform colour areas but adjacent areas may have very different colours. The uniform colour areas in cartoon-like images are regarded as good features for compression.

Different Type of Images

- **1. Bi-level (or monochromatic) image:**
  - The pixels of this image have one of two values, normally referred to as black and white.
- **2. Grayscale image:**
  - The pixels in such an image can have one of the n values (0 through n−1).
  - The value of n is normally compatible with a byte size; i.e., it is 4, 8, 12, 16, 24, or some other convenient multiple of 4 or of 8. Most common grayscale level is 256.
- **3. Colour image:**
  - Colour images are captured by (colour) scanners, using multiple intensity levels and filtering to capture the brightness levels for each of the primary colours, R, G and B (Red, Green, Blue).
  - Some computer systems use 16 bits to hold colour values. Most common colour images are of a 24 bit colour depth.

Goal of Image Compression

- **Digital images require huge amounts of space for storage and large bandwidths for transmission.**
  - **A 640 x 480 color image requires close to 1MB of space.**

- **The goal of image compression is to reduce the amount of data required to represent a digital image.**
  - **Reduce storage requirements and increase transmission rates.**

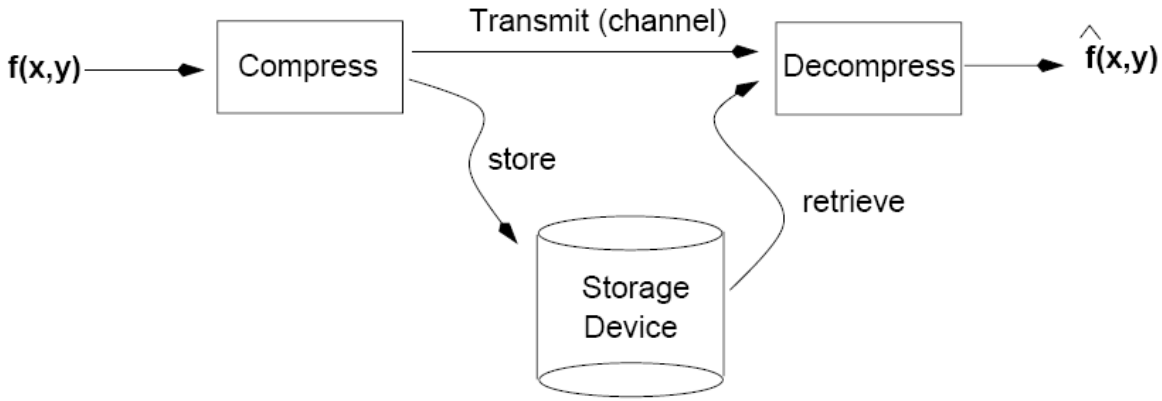
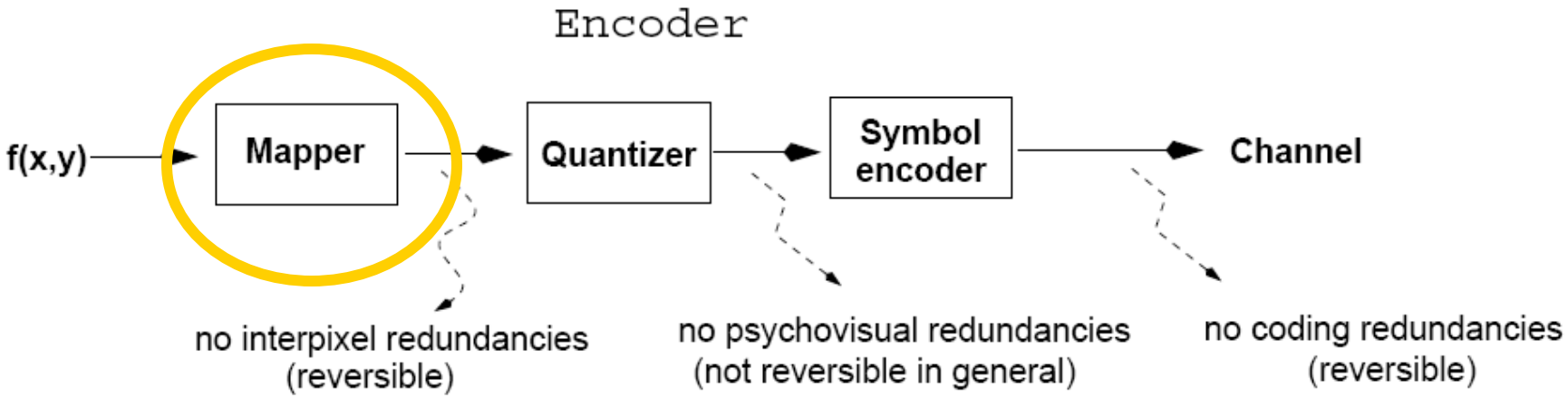


Image Compression

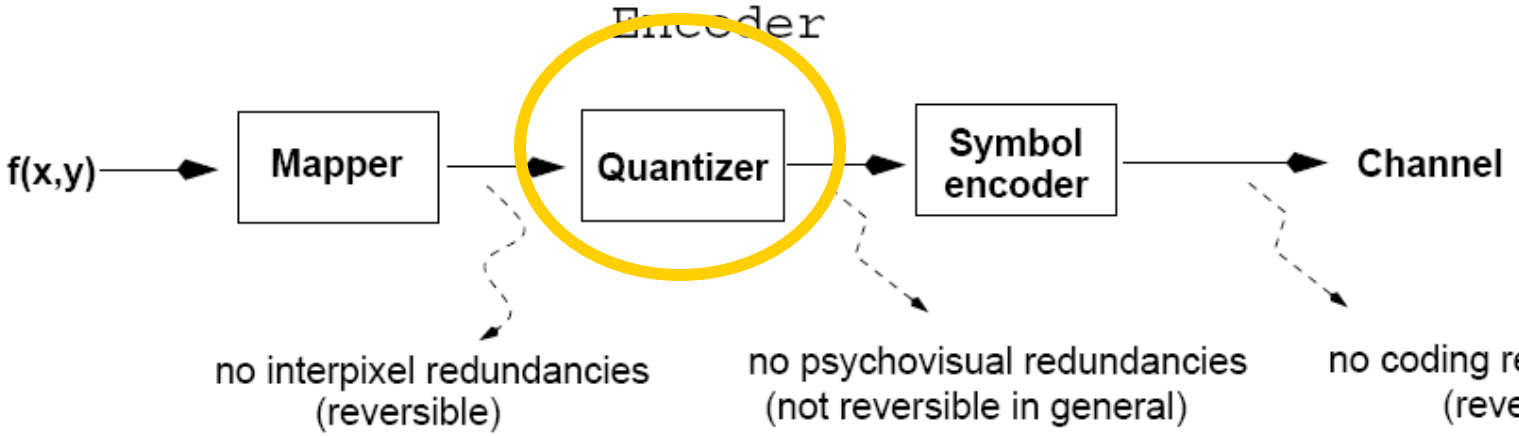
- Images are two or three dimensional array of pixel values that have spatial and temporal redundancy.
- Human eye is less sensitive to chrominance signal than to luminance signal (U and V in YUV color space or Cb and Cr in YCbCr color space can be coarsely coded).
- Human eye is less sensitive to the higher spatial frequency components.
- Human eye is less sensitive to quantizing distortion at high luminance levels.

Image Compression Model



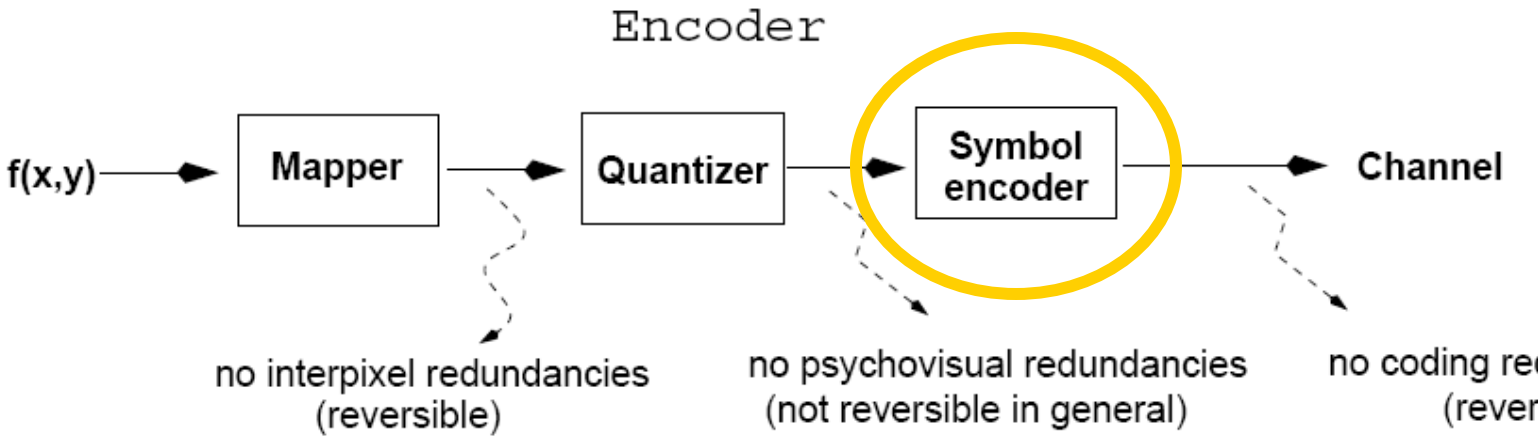
- **Mapper: transforms input data, (using DCT, DWT, KLT,…) in a way that facilitates reduction of inter pixel redundancies.**

Image Compression Model



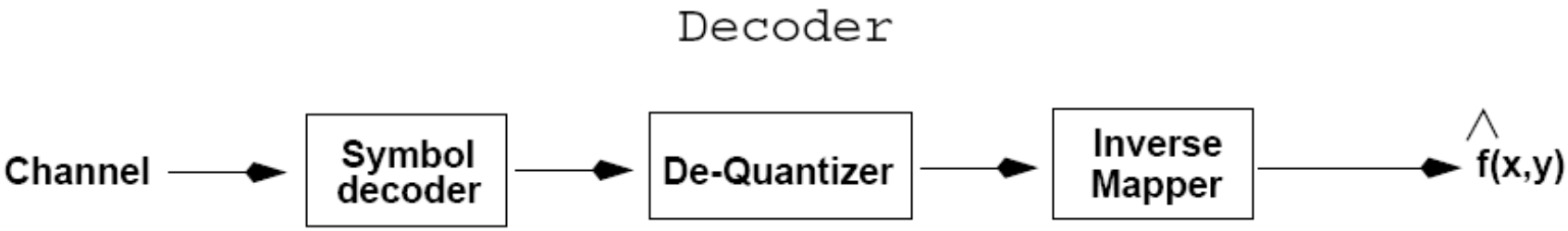
- **Quantizer: reduces the accuracy of the mapper's output in accordance with some pre-established fidelity criteria.**

Image Compression Model



- **Symbol encoder: assigns the shortest code to the most frequently occurring output values.**

Image Compression Models



- **Inverse operations are performed.**
- **But quantization is irreversible in general.**

Image Compression Methods

- The most popular image compression methods are:
  - DCT (Discrete Cosine Transform)
    - JPEG (Joint Photographic Experts Group)
  - DWT (Discrete Wavelet Transform)
    - SPIHT
    - EBCOT (JPEG 2000)
    - PACW
    - WSQ
  - JBIG (Joint Bi-Level Image processing Group)
  - DPCM

JPEG Encoder

- **JPEG (Joint Photographic Experts Group) is a sophisticated lossy/lossless compression method for color or grayscale still images (not videos) at 1987.**
  - **It does not handle bi-level (black and white) images very well.**
  - **It also works best on continuous-tone images, where adjacent pixels have similar colors.**
- **An important feature of JPEG is its use of many parameters, allowing the user to adjust the amount of the data lost (and thus also the compression ratio) over a very wide range.**
- **Often, the eye cannot see any image degradation even at compression factors of 10 or 20.**

JPEG Encoder

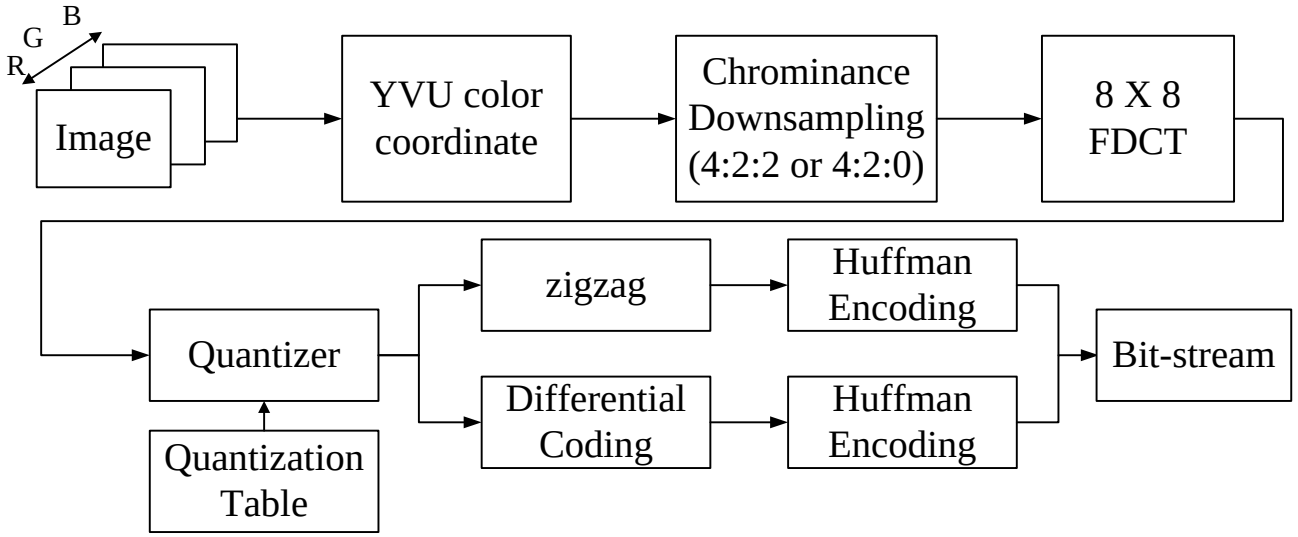
- **The main goals of JPEG compression are the following:**
  - **1. High compression ratios, especially in cases where image quality is judged as very good to excellent.**
  - **2. The use of many parameters, allowing knowledgeable users to experiment and achieve the desired compression/quality trade-off.**
  - **3. Obtaining good results with any kind of continuous-tone image, regardless of image dimensions, color spaces, pixel aspect ratios, or other image features.**
  - **4. A sophisticated, but not too complex compression method, allowing software and hardware implementations on many platforms.**
  - **5. Several modes of operation.**

JPEG Encoder

- **JPEG encoder can be employed in two operating modes:**
  - **lossy (also called baseline):**
    - **1) A sequential mode where each image component (color) is compressed in a single left-to-right, top-to-bottom scan;**
    - **2) A progressive mode where the image is compressed in multiple blocks (known as "scans") to be viewed from coarse to fine detail;**
    - **3) A hierarchical mode where the image is compressed at multiple resolutions allowing lower-resolution blocks to be viewed without first having to decompress the following higher-resolution blocks.**
  - **lossless (typically produces compression ratios of around 0.5): It is important in cases where the user decides that no pixels should be lost (the trade-off is low compression ratio compared to the lossy modes);**

JPEG Encoder's Main Steps

- 1) Color images transformation
- 2) Color images downsamplling
- 3) Image/block preparation
- 4) DCT computation
- 5) Quantization
- 6) Entropy coding
- 7) Frame building



JPEG Encoder's Main Steps
Color images transformation

- **1) Color images are transformed from RGB into a luminance/ chrominance color space.**
  - **The eye is sensitive to small changes in luminance but not in chrominance, so the chrominance part can later lose much data, and thus be highly compressed, without visually impairing the overall image quality much. Therefore, the values in the RGB color space are transformed to YUV or YCbCr color space:**

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.334 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix}$$

    - **Y is the components of luminance**
    - **Cb and Cr are the components of chrominance**
  - **This step is optional but important because the remainder of the algorithm works on each color component separately.**

JPEG Encoder's Main Steps
Color images downsamplling

- **2) Color images are downsampled by creating low-resolution pixels from the original ones.**
  - **This step is used only when hierarchical compression is selected; it is always skipped for grayscale images.**
  - **The downsampling is only done for chrominance parts and done at a ratio of 2:1 both horizontally and vertically (so called 2h2v or 4:1:1 sampling) or at ratios of 2:1 horizontally and 1:1 vertically (2h1v or 4:2:2 sampling).**
  - **Since this is done on two of the three color components, 2h2v reduces the image to $1/3 + (2/3) \times (1/4) = 1/2$ its original size, while 2h1v reduces it to $1/3 + (2/3) \times (1/2) = 2/3$ its original size.**
  - **Since the luminance component is not touched, there is no noticeable loss of image quality.**

JPEG Encoder's Main Steps
Image/block preparation

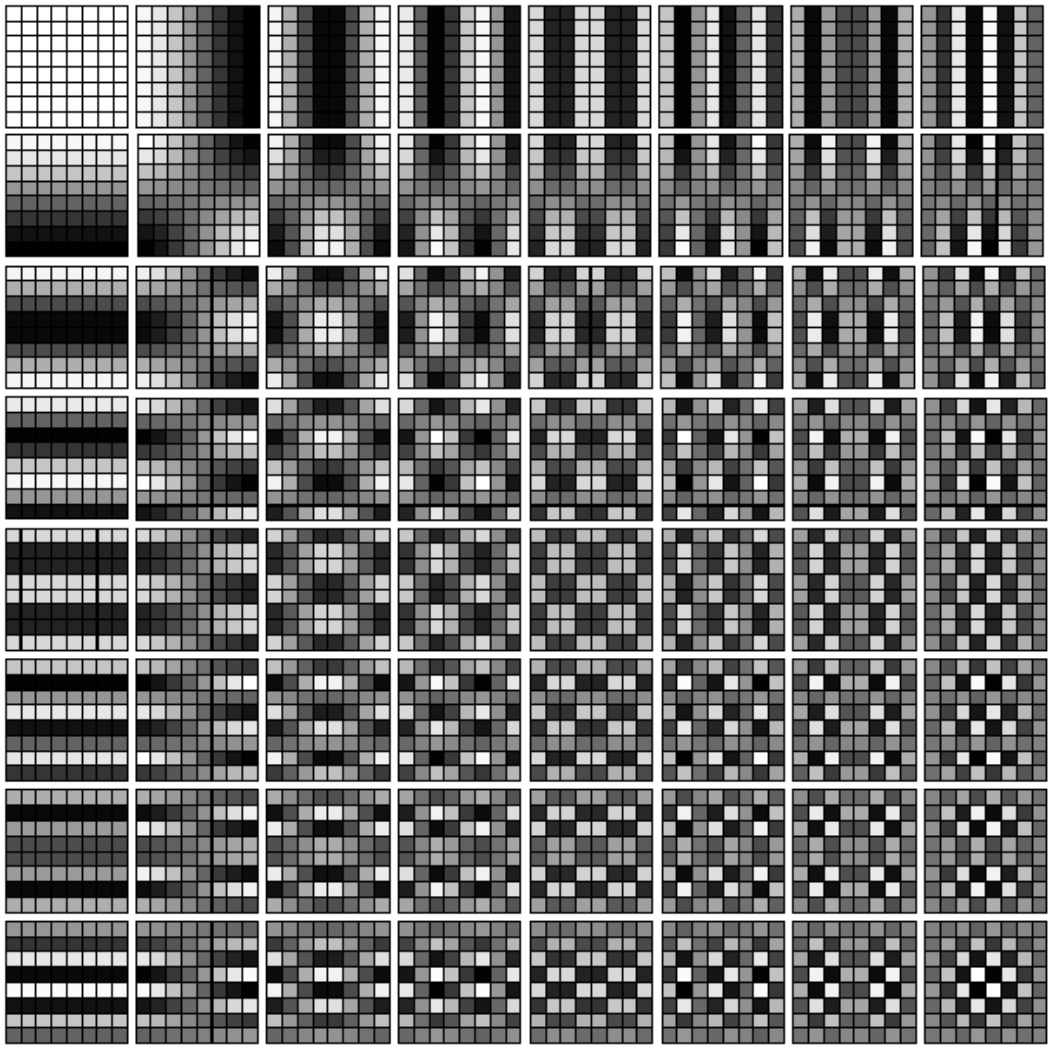
- **3) The pixels of each color component are organized in groups of 8×8 pixels called data units or Blocks, and each block is compressed separately.**
  - **If the number of image rows or columns is not a multiple of 8, the bottom row and the rightmost column are duplicated as many times as necessary.**
  - **In the non-interleaved mode, the encoder handles all the blocks of the first image component, then the data units of the second component, and finally those of the third component.**
  - **In the interleaved mode the encoder processes the three top-left blocks of the three image components, then the three data units to their right, and so on.**
  - **The fact that each blocks is compressed separately is one of the downsides of JPEG. If the user asks for maximum compression, the decompressed image may exhibit blocking artifacts due to differences between blocks.**

JPEG Encoder's Main Steps
DCT computation

- **4) The discrete cosine transform (DCT) is then applied to each block to create an 8×8 map of frequency components .**
  - **All values are shifted to the range of -128 to + 127 before computing DCT.**
  - **They represent the average pixel value and successive higher-frequency changes within the group.**
  - **This prepares the image data for the crucial step of losing information.**
  - **Since DCT involves the transcendental function cosine, it must involve some loss of information due to the limited precision of computer arithmetic.**
  - **This means that even without the main lossy step (step 5 below), there will be some loss of image quality, but it is normally small.**

JPEG Encoder's Main Steps
DCT computation

- The DCT interpreted as a set of basis images.
  - In the case n=8 we create 64 small basis images of 8 × 8 pixels.
- The DCT separates the frequencies in the block and represents it as a linear combination (or a weighted sum) of the basis images.
- The weights are the DCT coefficients.
  - Any block B of 8×8 pixels can be expressed as a linear combination of the basis images, and the 64 weights of this linear combination are the DCT coefficients of B.



JPEG Encoder's Main Steps Quantization

- **5) Each of the 64 frequency components in a block (C) is divided by a separate number called its quantization coefficient (Q), and then rounded to an integer:**

$$C_q(u,v) = Round[\frac{C(u,v)}{Q(u,v)}]$$

  - **This is where information is irretrievably lost.**
  - **Large QCs cause more loss, so the high frequency components typically have larger QCs.**
  - **Each of the 64 QCs is a JPEG parameter and can, in principle, be specified by the user.**
  - **In practice, most JPEG implementations use the QC tables recommended by the JPEG standard for the luminance and chrominance image components.**

JPEG Encoder's Main Steps
Quantization Table

- **Quantization Table may be produced as below:**

```
for i=0 to n;
  for j=0 to n;
    Q[i,j]= 1 + (1+i+j)*quality;
  end j;
end i;
```

(quality = 2)

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

$1 \le quality \le 25$

(best - low compression) (worst - high compression)

JPEG Encoder's Main Steps
Quantization Example

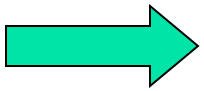
(c) Block after FDCT Eqn. (5)

185	-17	14	-8	23	-9	-13	-18
20	-34	26	-9	-10	10	13	6
-10	-23	-1	6	-18	3	-20	0
-8	-5	14	-14	-8	-2	-3	8
-3	9	7	1	-11	17	18	15
3	-2	-18	8	8	-3	0	-6
8	0	-2	3	-1	-7	-1	-1
0	-7	-2	1	1	4	-6	0

(d) Quantization table (quality = 2)

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Quantization



(e) Block after quantization Eqn. (6)

61	-3	2	0	2	0	0	-1
4	-4	2	0	0	0	0	0
-1	-2	0	0	-1	0	-1	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

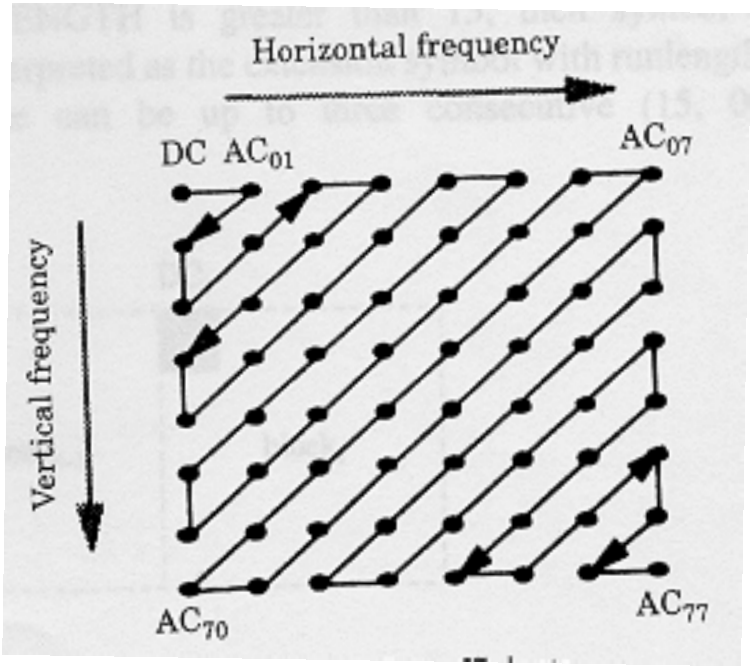
JPEG Encoder's Main Steps
Entropy coding

- 6) The 64 quantized frequency coefficients (which are now integers) of each data unit are encoded using a combination of RLE and Huffman coding.
  - After quantization, most of the high frequency coefficients(lower right corner) are zero.
  - To exploit the number of zeros, a zig-zag scan of the matrix is used.
  - Zig-zag scan allows all the DC coefficients and lower frequency AC coefficients to be scanned first.
  - DC are encoded using differential encoding and AC coefficients are encoded using run-length encoding.
  - Huffman coding is used to encode both after that.
  - An arithmetic coding variant known as the QM coder can optionally be used instead of Huffman coding.

JPEG Encoder's Main Steps
Entropy coding (Zig-zag Example)

(e) Block after quantization Eqn. (6)

61	-3	2	0	2	0	0	-1
4	-4	2	0	0	0	0	0
-1	-2	0	0	-1	0	-1	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0



(f) Zig-zag sequence

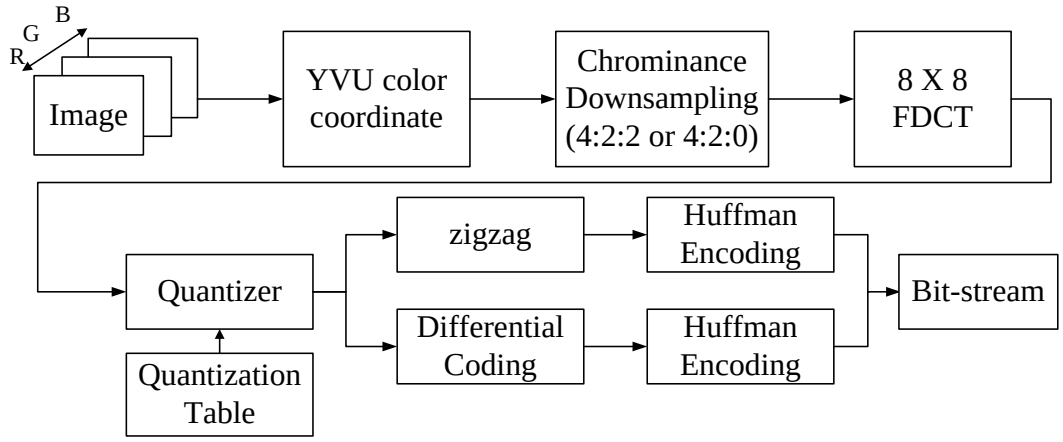
61,-3,4,-1,-4,2,0,2,-2,0,0,0,0,0,2,0,0,0,1,0,0,0,0,0,0,-1,0,0,-1,0,0,
0,0,-1,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

JPEG Encoder's Main Steps
Frame building

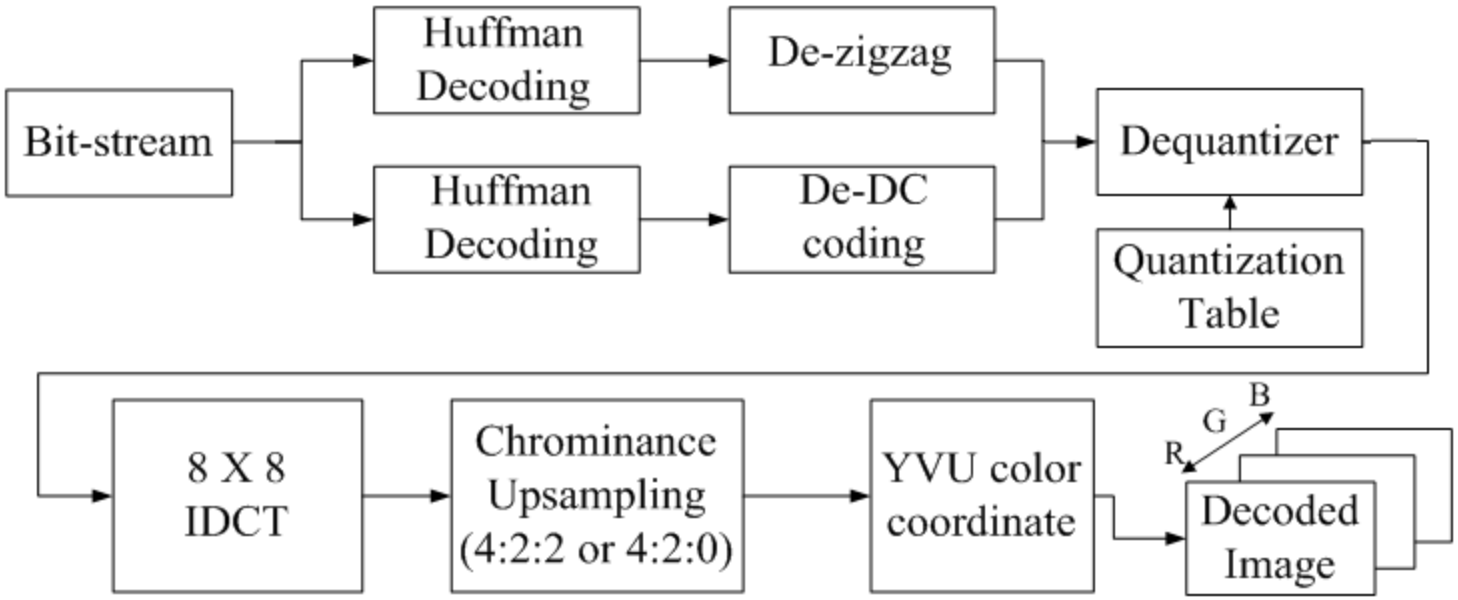
- **7) The last step adds headers and all the required JPEG parameters, and outputs the result. The compressed file may be in one of three formats:**
  - **1) the interchange format, in which the file contains the compressed image and all the tables needed by the decoder (mostly quantization tables and tables of Huffman codes),**
  - **2) the abbreviated format for compressed image data, where the file contains the compressed image and may contain no tables (or just a few tables)**
    - **In cases where the same encoder/decoder pair is used, and they have the same tables built in.**
  - **3) the abbreviated format for table-specification data, where the file contains just tables, and no compressed image.**
    - **In cases where many images have been compressed previously and need to be decompressed, they are sent to a decoder preceded by one file with table-specification data.**

JPEG Encoder/Decoder

Encoder



Decoder



JPEG Compression
Example 1: homogeneous block

An 8×8 block from the Y image of 'Lena'

200	202	189	188	189	175	175	175
200	203	198	188	189	182	178	175
203	200	200	195	200	187	185	175
200	200	200	200	197	187	187	187
200	205	200	200	195	188	187	175
200	200	200	200	200	190	187	175
205	200	199	200	191	187	187	175
210	200	200	200	188	185	187	186

$f(i,j)$

515	65	-12	4	1	2	-8	5
-16	3	2	0	0	-11	-2	3
-12	6	11	-1	3	0	1	-2
-8	3	-4	2	-2	-3	-5	-2
0	-2	7	-5	4	0	-1	-4
0	-3	-1	0	4	1	-1	0
3	-2	-3	3	3	-1	-1	3
-2	5	-2	4	-2	2	-3	0

$F(u,v)$

Fig. 9.2: JPEG compression for a smooth image block.

JPEG Compression
Example 1: homogeneous block

Quantized

$$\hat{F}(u,v) = \begin{bmatrix} 32 & 6 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

De-quantized

$$\tilde{F}(u,v) = \begin{bmatrix} 512 & 66 & -10 & 0 & 0 & 0 & 0 & 0 \\ -12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & 0 & 16 & 0 & 0 & 0 & 0 & 0 \\ -14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

JPEG Compression
Example 1: homogeneous block

Reconstructed

199	196	191	186	182	178	177	176
201	199	196	192	188	183	180	178
203	203	202	200	195	189	183	180
202	203	204	203	198	191	183	179
200	201	202	201	196	189	182	177
200	200	199	197	192	186	181	177
204	202	199	195	190	186	183	181
207	204	200	194	190	187	185	184

$\tilde{f}(i,j)$

Error

1	6	-2	2	7	-3	-2	-1
-1	4	2	-4	1	-1	-2	-3
0	-3	-2	-5	5	-2	2	-5
-2	-3	-4	-3	-1	-4	4	8
0	4	-2	-1	-1	-1	5	-2
0	0	1	3	8	4	6	-2
1	-2	0	5	1	1	4	-6
3	-4	0	6	-2	-2	2	2

$\epsilon(i,j) = f(i,j) - \tilde{f}(i,j)$

JPEG Compression
Example 2: Less homogeneous block

Another 8×8 block from the Y image of 'Lena'

70	70	100	70	87	87	150	187
85	100	96	79	87	154	87	113
100	85	116	79	70	87	86	196
136	69	87	200	79	71	117	96
161	70	87	200	103	71	96	113
161	123	147	133	113	113	85	161
146	147	175	100	103	103	163	187
156	146	189	70	113	161	163	197

$f(i,j)$

-80	-40	89	-73	44	32	53	-3
-135	-59	-26	6	14	-3	-13	-28
47	-76	66	-3	-108	-78	33	59
-2	10	-18	0	33	11	-21	1
-1	-9	-22	8	32	65	-36	-1
5	-20	28	-46	3	24	-30	24
6	-20	37	-28	12	-35	33	17
-5	-23	33	-30	17	-5	-4	20

$F(u,v)$

JPEG Compression
Example 2: Less homogeneous block

Quantized

$$\hat{F}(u,v) = \begin{bmatrix} -5 & -4 & 9 & -5 & 2 & 1 & 1 & 0 \\ -11 & -5 & -2 & 0 & 1 & 0 & 0 & -1 \\ 3 & -6 & 4 & 0 & -3 & -1 & 0 & 1 \\ 0 & 1 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

De-quantized

$$\tilde{F}(u,v) = \begin{bmatrix} -80 & -44 & 90 & -80 & 48 & 40 & 51 & 0 \\ -132 & -60 & -28 & 0 & 26 & 0 & 0 & -55 \\ 42 & -78 & 64 & 0 & -120 & -57 & 0 & 56 \\ 0 & 17 & -22 & 0 & 51 & 0 & 0 & 0 \\ 0 & 0 & -37 & 0 & 0 & 109 & 0 & 0 \\ 0 & -35 & 55 & -64 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

JPEG Compression
Example 2: Less homogeneous block

Reconstructed – spatial

70	60	106	94	62	103	146	176
85	101	85	75	102	127	93	144
98	99	92	102	74	98	89	167
132	53	111	180	55	70	106	145
173	57	114	207	111	89	84	90
164	123	131	135	133	92	85	162
141	159	169	73	106	101	149	224
150	141	195	79	107	147	210	153

$\tilde{f}(i,j)$

Error

0	10	-6	-24	25	-16	4	11
0	-1	11	4	-15	27	-6	-31
2	-14	24	-23	-4	-11	-3	29
4	16	-24	20	24	1	11	-49
-12	13	-27	-7	-8	-18	12	23
-3	0	16	-2	-20	21	0	-1
5	-12	6	27	-3	2	14	-37
6	5	-6	-9	6	14	-47	44

$\epsilon(i,j) = f(i,j) - \tilde{f}(i,j)$

JPEG Compression
Effect of "Quality"

Razi University

10 (8k bytes)

50 (21k bytes)

90 (58k bytes)

worst quality,

highest compression

best quality,

lowest compression

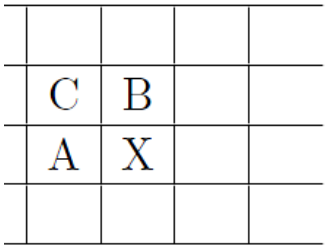
JPEG Compression
Effect of "Quality"

Table 6. Results of JPEG Compression for Grayscale Image 'Lisa' (320 ×240 pixels)

Quality factors	Original number of bits	Compressed number of bits	Compression ratio (Cr)	Bits/pixel (Nb)	RMS error	Execution times [ms]	
						Sun Sparc 10/41	Sun IPC
1	512,000	48,021	10·66	0·75	2·25	0·59	6·31
2	512,000	30,490	16·79	0·48	2·75	0·59	6·22
4	512,000	20,264	25·27	0·32	3·43	0·58	6·39
8	512,000	14,162	36·14	0·22	4·24	0·59	6·44
15	512,000	10,479	48·85	0·16	5.36	0·58	6·45
25	512,000	9,034	56·64	0·14	6·40	0·58	6·32
DC only	512,000	7,688	66·60	0·12	7·92	0·57	6·25

Lossless JPEG Compression

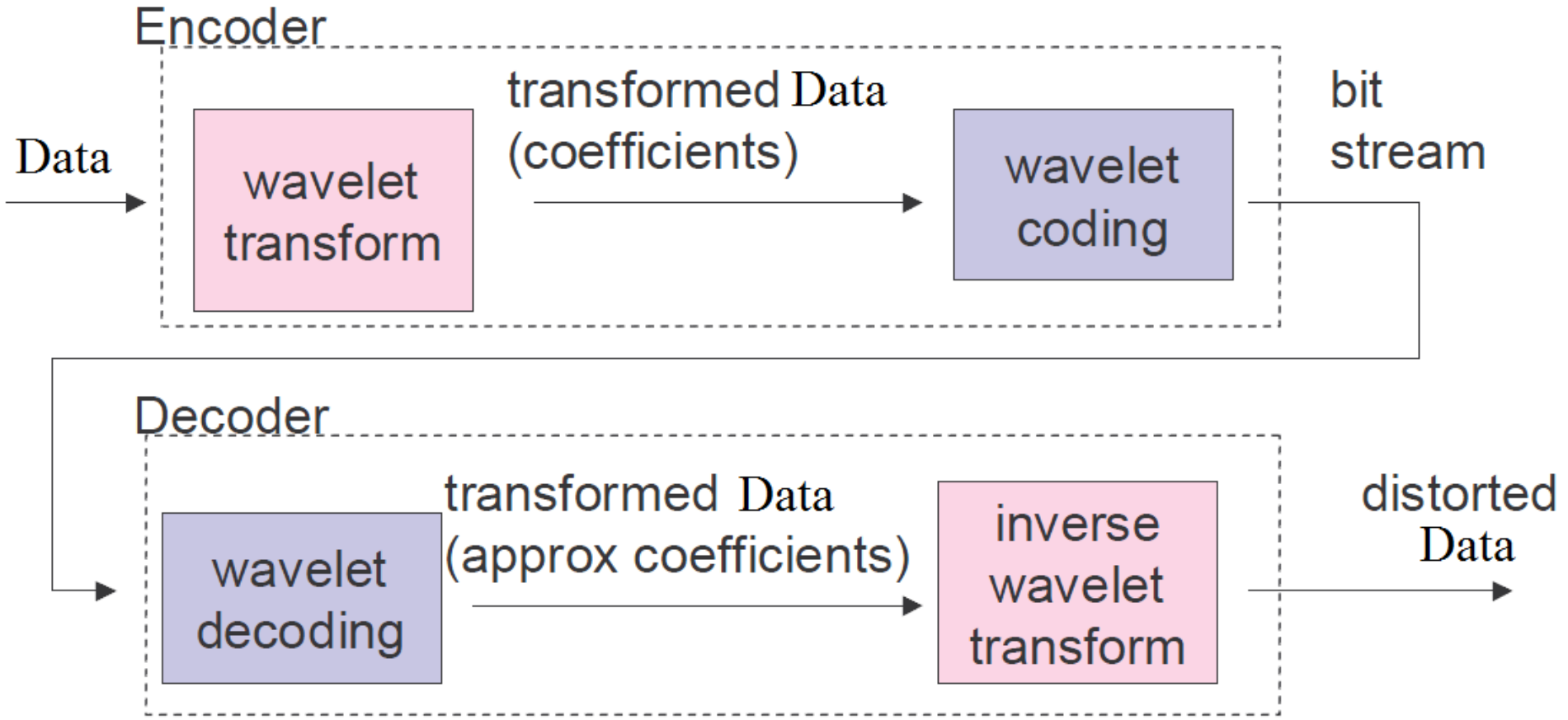
- **The lossless mode of JPEG uses differencing to reduce the values of pixels before they are compressed.**
  - **This particular form of differencing is called *predicting*.**
- **The values of some near neighbors of a pixel are subtracted from the pixel to get a small number, which is then compressed further using Huffman or arithmetic coding.**
  - **Predictor 0 is used only in the hierarchical mode of JPEG.**
  - **Predictors 1, 2, and 3 are called one-dimensional.**
  - **Predictors 4, 5, 6, and 7 are two-dimensional.**



Selection value	Prediction
0	no prediction
1	A
2	B
3	C
4	A + B − C
5	A + ((B − C)/2)
6	B + ((A − C)/2)
7	(A + B)/2

Pixel Prediction in the Lossless Mode.

Wavelet Based Compression



- **A wavelet transform decomposes the data into a low resolution version and details. The details are typically very small so they can be coded in very few bits.**

Wavelet Based Compression
SPIHT

- It seems that different mother wavelets produce different compression results depending on the image type, but it is currently not clear what mother wavelet is the best for any given image type.
  - Regardless of the particular wavelet used, the image is decomposed into subbands, such that:
    - lower subbands correspond to higher image frequencies
    - higher subbands correspond to lower image frequencies
  - This is why we can expect the detail coefficients to get smaller as we move from high to low levels. Also, there are spatial similarities among the subbands. An image part, such as an edge, occupies the same spatial position in each subband.
- These features of the wavelet decomposition are exploited by the SPIHT (Set Partitioning In Hierarchical Trees) method.

Wavelet Based Compression
SPIHT

- **SPIHT was designed for optimal progressive transmission, as well as for compression.**
  - **Invented by Shapiro, 1993, and refined by Said and Pearlman, 1996.**
- **One of the important features of SPIHT (perhaps a unique feature) is that at any point during the decoding of an image, the quality of the displayed image is the best that can be achieved for the number of bits input by the decoder up to that moment.**
- **Another important SPIHT feature is its use of embedded coding.**
  - **This feature is defined as follows: If an encoder produces two files, a large one of size *M* and a small one of size m, then the smaller file is identical to the first *m* bits of the larger file.**

Wavelet Based Compression
SPIHT

- The main aim in progressive transmission is to transmit the most important image information first.
- Another principle is based on the observation that the most significant bits of a binary integer whose value is close to maximum tend to be ones. This suggests that the most significant bits contain the most important image information, and that they should be sent to the decoder first (or written first on the compressed stream).
- The progressive transmission method used by SPIHT incorporates these two principles:
  - sorts the coefficients
  - transmits their most significant bits first

Wavelet Based Compression
SPIHT Algorithm

- **Step 1:** Perform suitable wavelet transform on input image, and represent the resulting coefficients $C_{i,j}$ with fixed signed-magnitude numbers.
- **Step 2:** Sorting pass: Transmit the number L of coefficients $C_{i,j}$ that satisfy $2^n \le |C_{i,j}| < 2^{n+1}$.
  - Follow with the L pairs of coordinates and the L sign bits of those coefficients.
- **Step 3:** Refinement pass: Transmit the n^{th} most significant bit of all coefficients satisfying $|C_{i,j}| \ge 2^{n+1}$.
  - These are the coefficients that were selected in previous sorting passes (not including the immediately preceding sorting pass).
- **Step 4:** Iterate: Decrement n by 1. If more iterations are needed (or desired), go back to Step 2.

Wavelet Based Compression
SPIHT

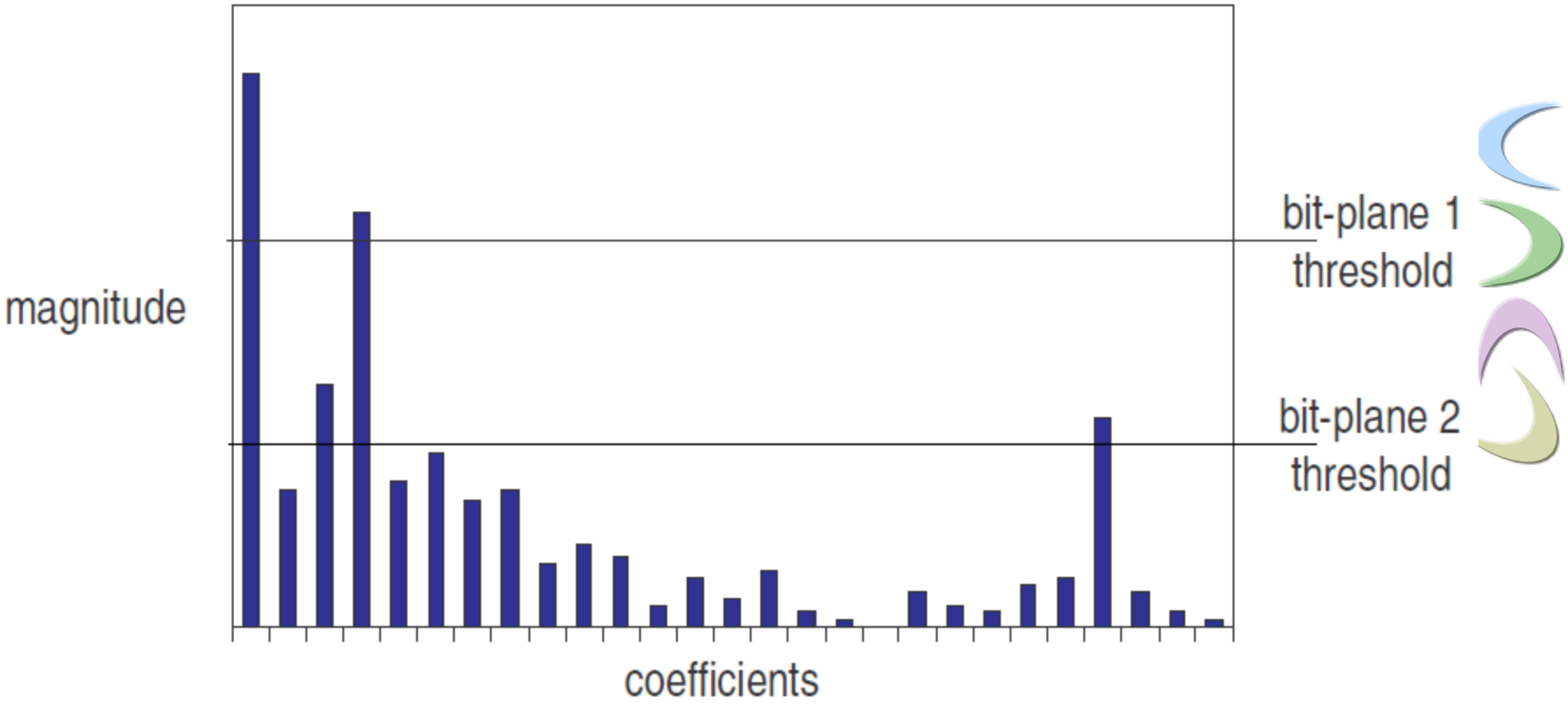
- **The method as described so far is simple, since we have assumed that the coefficients had been sorted before the loop started.**
  - **In practice, the image may have 1K*×1K pixels* or more; there may be more than a million coefficients, so sorting all of them is too slow.**
- **Instead of sorting the coefficients, SPIHT uses the fact that sorting is done by comparing two elements at a time, and each comparison results in a simple yes/no result.**
  - **If both encoder and decoder use the same sorting algorithm, the encoder can simply send the sequence of yes/no results, and the decoder can use thom to perform same sorting.**
  - **This is true not just for sorting but for any algorithm based on comparisons or on any type of branching.**

Wavelet Based Compression
SPIHT

- The actual algorithm used by SPIHT is based on the realization that there is really no need to sort all the coefficients. The main task of the sorting pass in each iteration is to select those coefficients that satisfy $2^n \le |C_{i,j}| < 2^{n+1}$. This task is divided into two parts.
  - For a given value of n, if a coefficient $C_{i,j}$ satisfies $|C_{i,j}| \ge 2^n$, then we say that it is significant;
  - otherwise, it is called insignificant.
- In the first iteration, relatively few coefficients will be significant, but their number increases from iteration to iteration, because n keeps getting decremented.
  - The sorting pass has to determine which of the significant coefficients satisfies $|C_{i,j}| < 2^{n+1}$ and transmit their coordinates to the decoder. This is an important part of SPIHT algorithm.

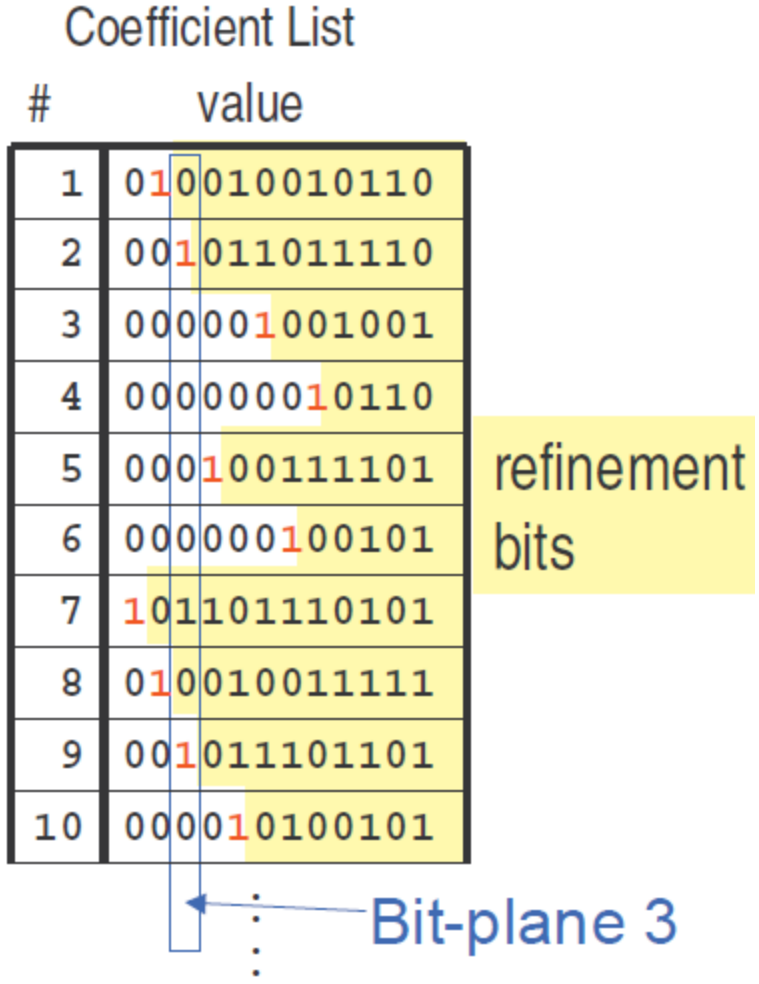
Wavelet Based Compression
SPIHT

Significant Coefficients



Wavelet Based Compression
SPIHT

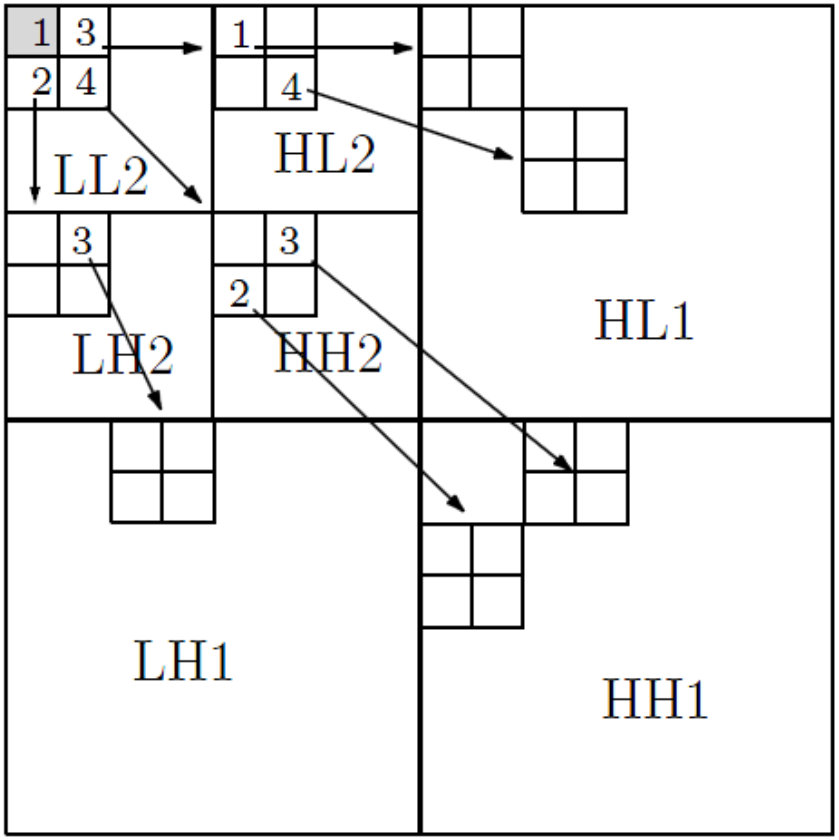
- **Significance & Refinement Passes:**
  - **Code a bit-plane in two passes**
    - **Significance pass**
      - **codes previously insignificant coefficients**
      - **codes sign bit**
    - **Refinement pass**
      - **refines values for previously significant coefficients**
  - **Main idea:**
    - **Significance-pass bits likely to be 0;**
    - **Refinement-pass bit are not**

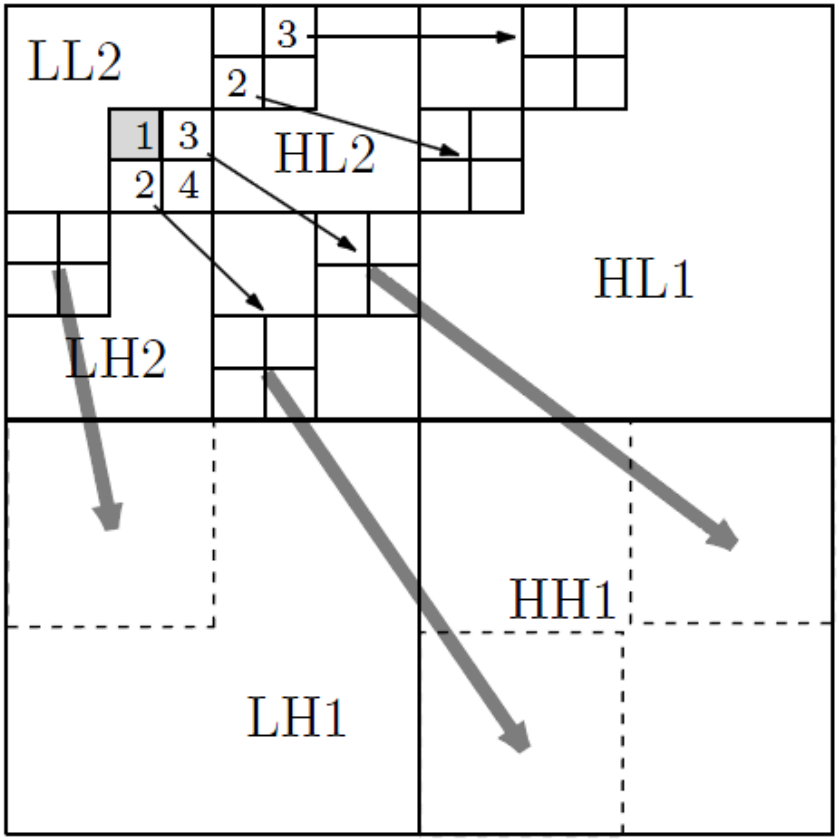


Wavelet Based Compression
SPIHT

- The encoder partitions all the coefficients into a number of sets T_k and performs the significance test.
  - The result may be either "no" (all the coefficients in T_k are insignificant, so T_k itself is considered insignificant)
  - or "yes" (some coefficients in T_k are significant, so T_k itself is significant). This result is transmitted to the decoder.
- If the result is "yes," then T_k is partitioned by both encoder and decoder, using the same rule, into subsets and the same significance test is performed on all the subsets.
- This partitioning is repeated until all the significant sets are reduced to size 1 (i.e., they contain one coefficient each, and that coefficient is significant).

Wavelet Based Compression
SPIHT





- **If a bit plane value in a low resolution subband is insignificant then it is likely that the corresponding values in higher subbands are also insignificant in the same bit plane.**

Wavelet Based Compression
SPIHT

- **It is important to have the encoder and decoder test sets for significance in the same way. The coding algorithm therefore uses three lists called:**
  - ***list of significant pixels (LSP)***
  - ***list of insignificant pixels (LIP)***
    - **The LIP contains coordinates of coefficients that were insignificant in the previous sorting pass and in the current pass they are tested, and those that test significant are moved to the LSP.**
  - ***list of insignificant sets (LIS)***
    - **Sets in the LIS are tested in sequential order, and when a set is found to be significant, it is removed from the LIS and is partitioned. The new subsets with more than one coefficient are placed back in the LIS, to be tested later, and the subsets with one element are tested and appended to the LIP or the LSP.**
- **The refinement pass transmits the *nth most* significant bit of the entries in the LSP.**

Wavelet Based Compression
SPIHT

1. Set the threshold. Set LIP to all root nodes coefficients. Set LIS to all trees (assign type D to them). Set LSP to an empty set.
2. Sorting pass: Check the significance of all coefficients in LIP:
   - 2.1 If significant, output 1, output a sign bit, and move the coefficient to the LSP.
   - 2.2 If not significant, output 0.
3. Check the significance of all trees in the LIS according to the type of tree:
   - 3.1 For a tree of type D:
     - 3.1.1 If it is significant, output 1, and code its children:
       - 3.1.1.1 If a child is significant, output 1, then a sign bit, add it to the LSP
       - 3.1.1.2 If a child is insignificant, output 0 and add the child to the end of LIP.
       - 3.1.1.3 If the children have descendants, move the tree to the end of LIS as type L, otherwise remove it from LIS.
     - 3.1.2 If it is insignificant, output 0.
   - 3.2 For a tree of type L:
     - 3.2.1 If it is significant, output 1, add each of the children to the end of LIS as an entry of type D and remove the parent tree from the LIS.
     - 3.2.2 If it is insignificant, output 0.
4. Loop: Decrement the threshold and go to step 2 if needed.

Comparison DCT and Wavelet based compression methods

32:1 compression ratio
0.25 bits/pixel (8 bits)

ORIGINAL

SPIHT

JPEG

Wavelet Based Compression
EBCOT (JPEG 2000)

- **Embedded Block Coding with Optimized Truncation (EBCOT)**
  - **Taubman – journal paper 2000**
  - **Algorithm goes back to 1998 or maybe earlier**
  - **Basis of JPEG 2000**
- **Embedded**
  - **Prefixes of the encoded bit stream are legal encodings at lower fidelity, like SPIHT and GTW**
- **Block coding**
  - **Entropy coding of blocks of bit planes, not block transform coding like JPEG.**

Wavelet Based Compression EBCOT (JPEG 2000)

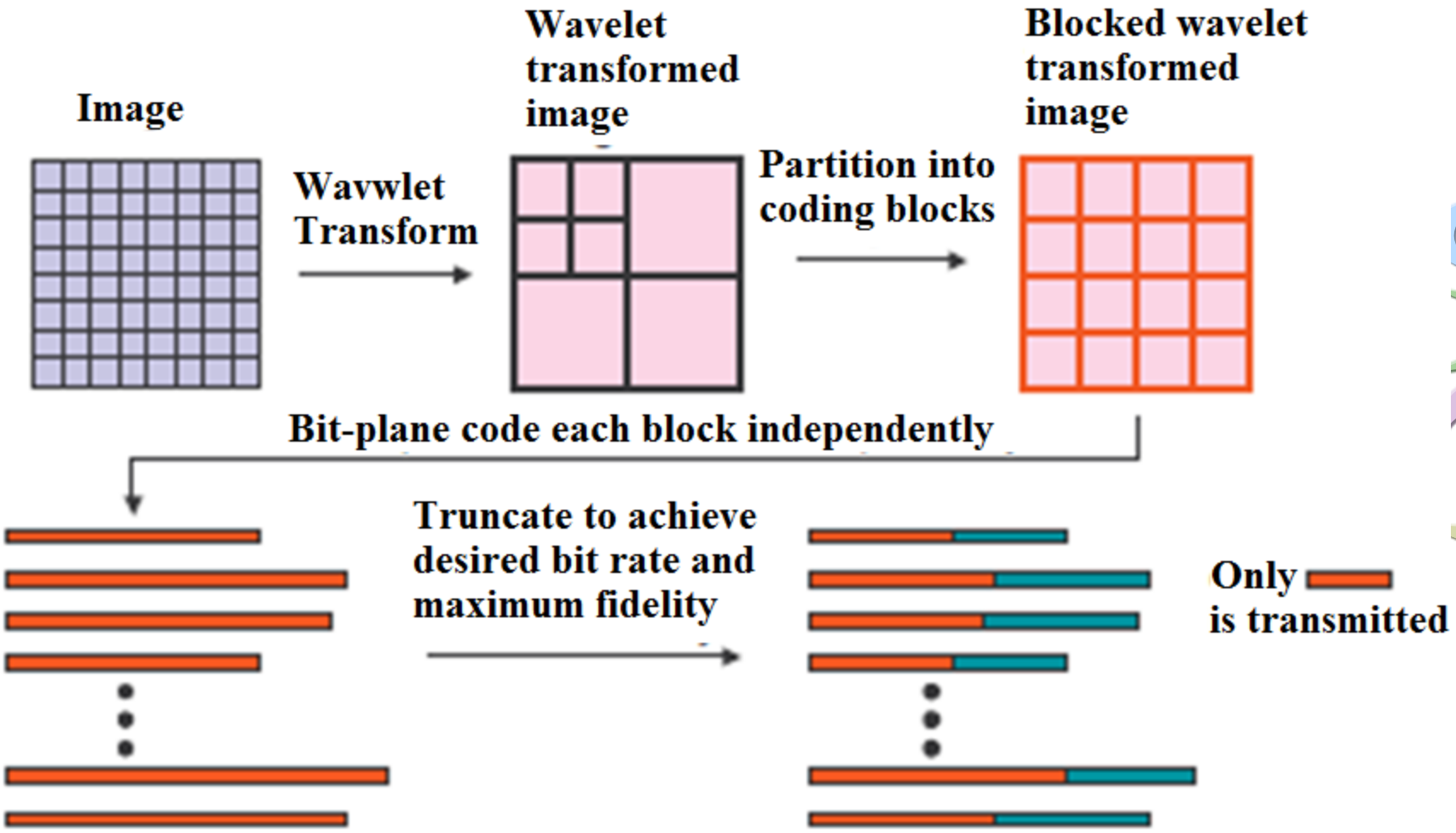
- **SNR scalability (Signal to Noise Ratio)**
  - **SNR is calculated using original (R)and obtained Image (C) from compressed version as:**

$$SNR = \sqrt{\frac{1}{n}\sum_{i=1}^{n} R_i^2} \Big/ \sqrt{\frac{1}{n}\sum_{i=1}^{n} (R_i - C_i)^2}$$

  - **Embedded code - The compressed bit stream can be truncated to yield a smaller compressed image at lower fidelity.**
  - **Layered code – The bit stream can be partitioned into a base layer and enhancement layers. Each enhancement layer improves the fidelity of the image.**
- **Resolution scalability**
  - **The lowest subband can be transmitted first yielding a smaller image at high fidelity.**
  - **Successive subbands can be transmitted to yield larger and larger images.**

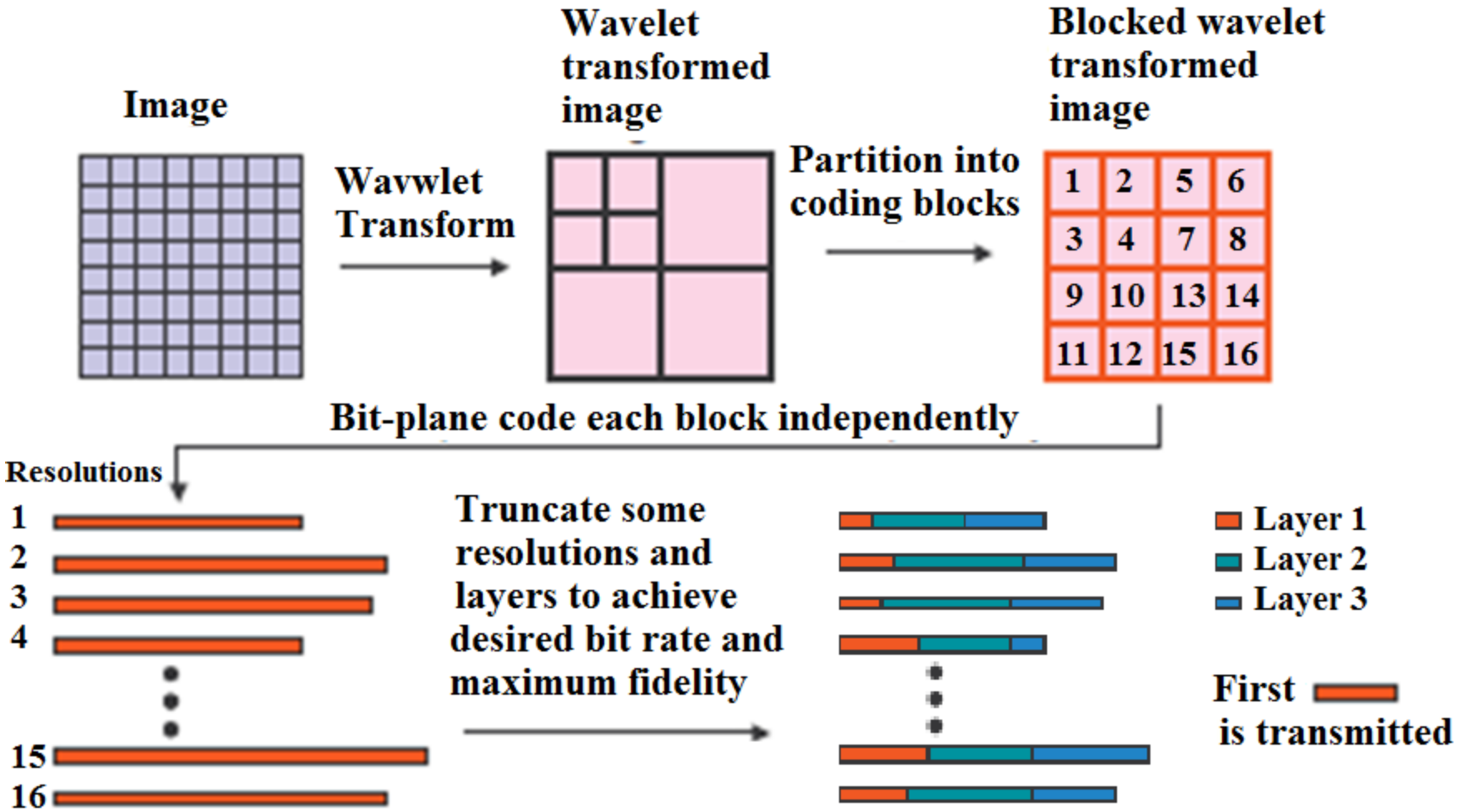
Wavelet Based Compression
EBCOT (JPEG 2000)

Block Diagram of Encoder:



Wavelet Based Compression
EBCOT (JPEG 2000)

Applying Resolution Ordering and Layering to Achieve Better Results

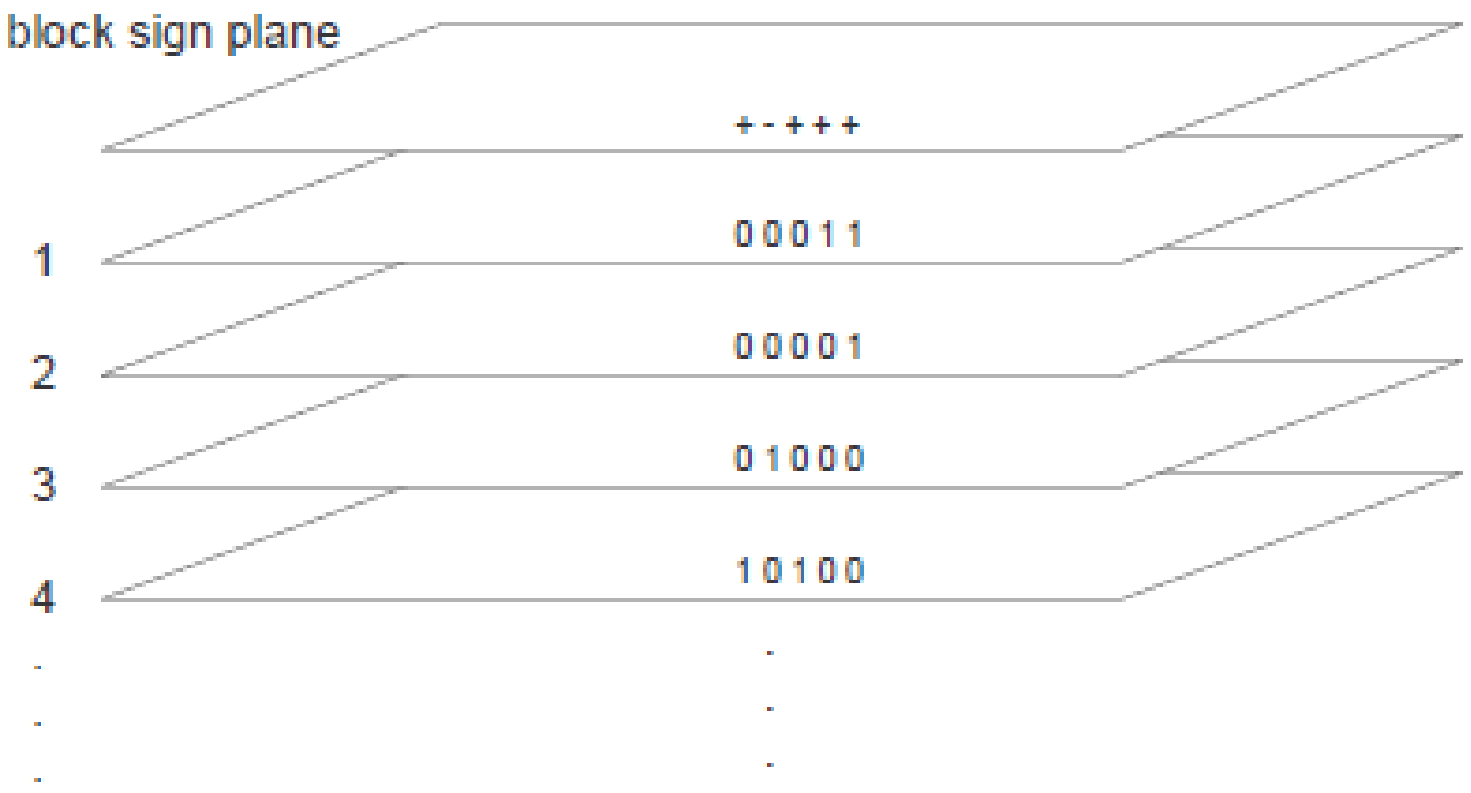


Wavelet Based Compression
EBCOT (JPEG 2000)

- **Block Coding:**
  - Assume we are in block k, and c(i,j) is a coefficient in block k.
  - Divide c(i,j) into its sign s(i,j) and its magnitude m(i,j) .
  - Quantize the magnitude m to: $v(i,j) = \lfloor m(i,j)/q_k + 0.5 \rfloor$ where q_k is the quantization step for block k.
  - Example: c(i,j) = -10, q_k= 3.
    - s(i,j) = 1
    - v(i,j) = floor(10/3 + 0.5) = 3

Wavelet Based Compression
EBCOT (JPEG 2000)

- **Bit Planes of Normalized Quantized Coefficients**
  - **Quantized coefficients are normalized between –1 and 1**

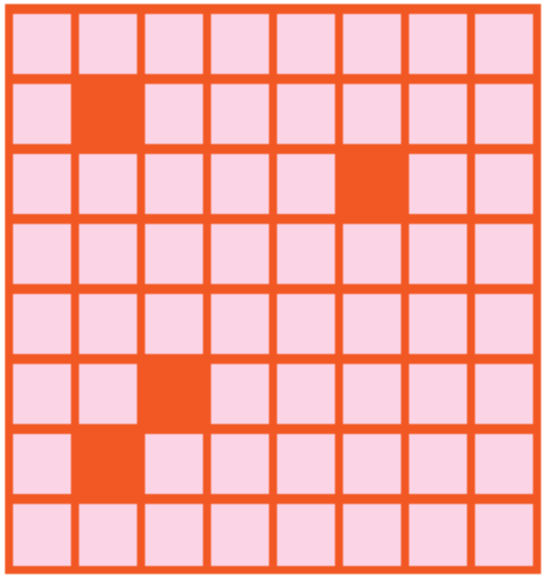


Wavelet Based Compression
EBCOT (JPEG 2000)

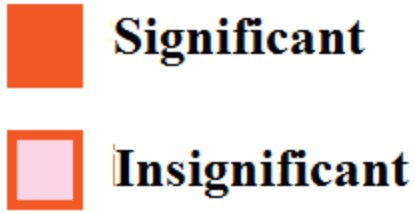
- **Bit-Plane Coding of Blocks**
  - **Sub-block significance coding (like group testing)**
    - **Some sub-blocks are declared insignificant**
    - **Significant sub-blocks must be coded**
  - **Contexts are defined based on the previous bit-plane significance.**
    - **Zero coding (ZC) – 9 contexts**
    - **Run length coding (RLC) – 1 context**
    - **Sign coding (SC) – 5 contexts**
    - **Magnitude refinement coding (MR) – 3 contexts**
  - **Block coded in raster order using arithmetic coding**

Wavelet Based Compression
EBCOT (JPEG 2000)

- **Sub-Block Significance Coding**
  - **Quad-tree organized group testing.**
  - **Block divided into 16x16 sub-blocks.**
  - **Identify in few bits the sub-blocks that are significant.**

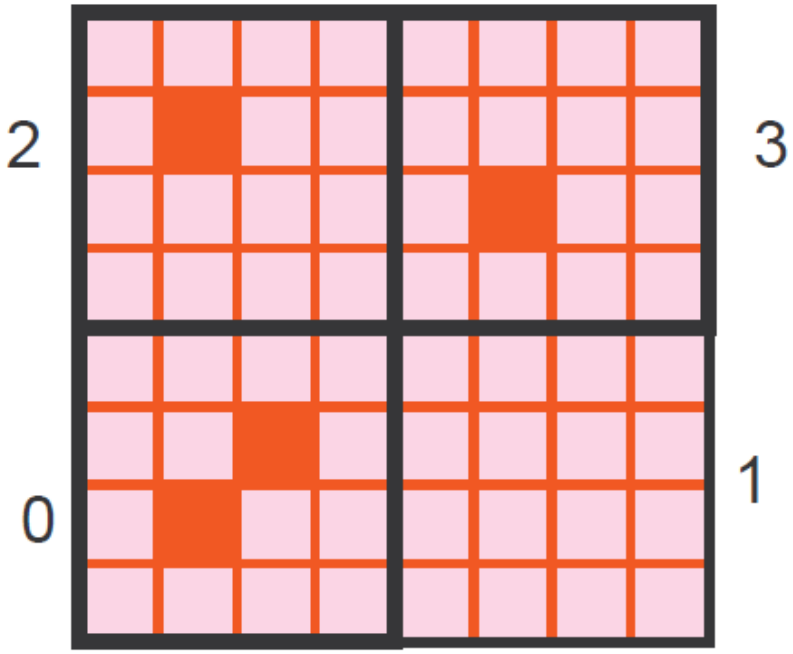


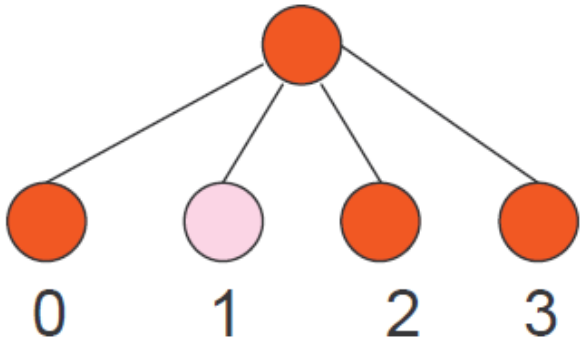
Block



Wavelet Based Compression
EBCOT (JPEG 2000)

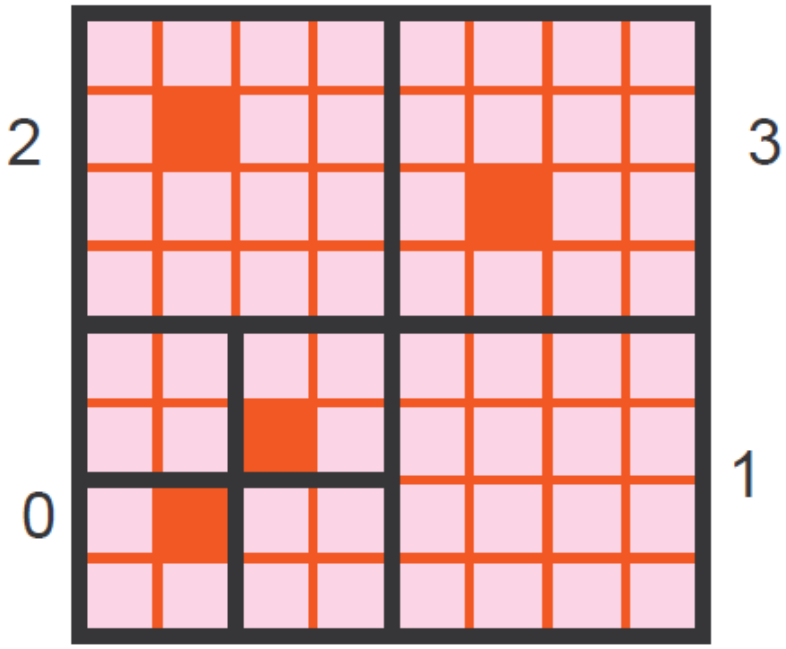
Quad-Tree Subdivision

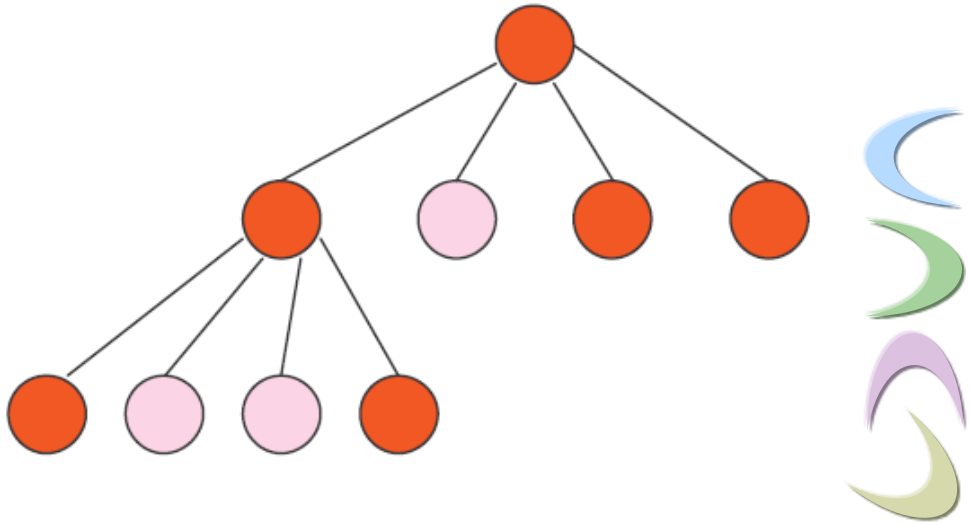




Wavelet Based Compression
EBCOT (JPEG 2000)

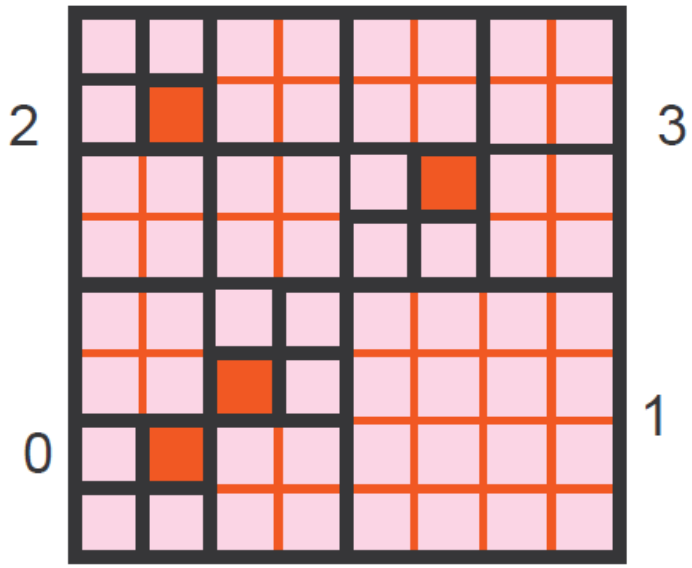
Quad-Tree Subdivision

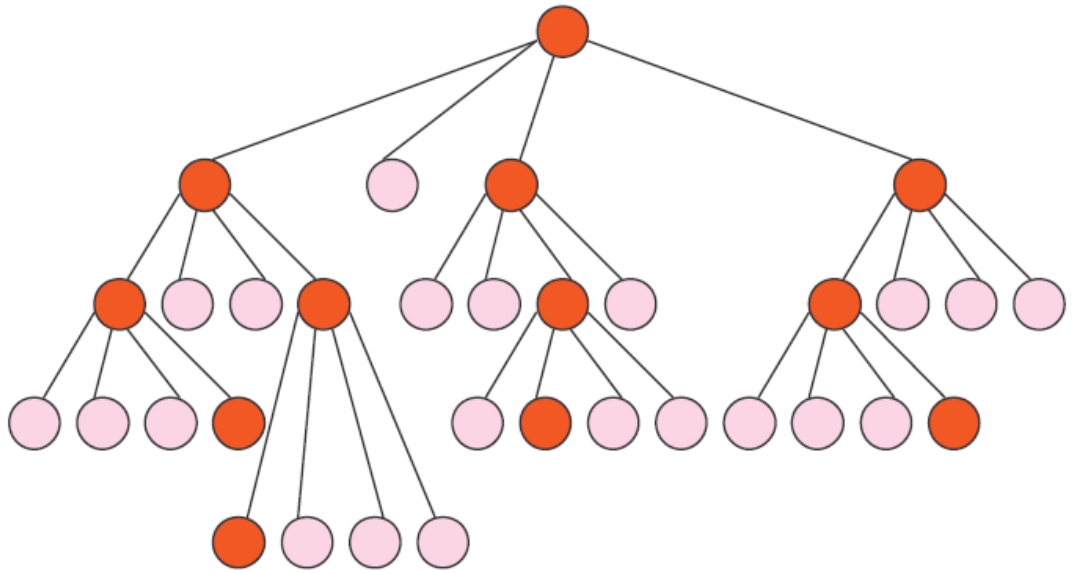




Wavelet Based Compression
EBCOT (JPEG 2000)

Quad-Tree Subdivision



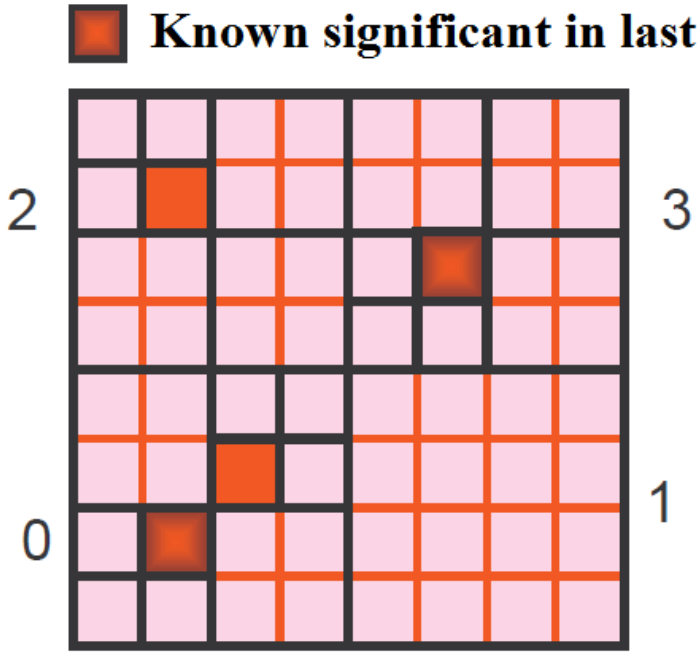


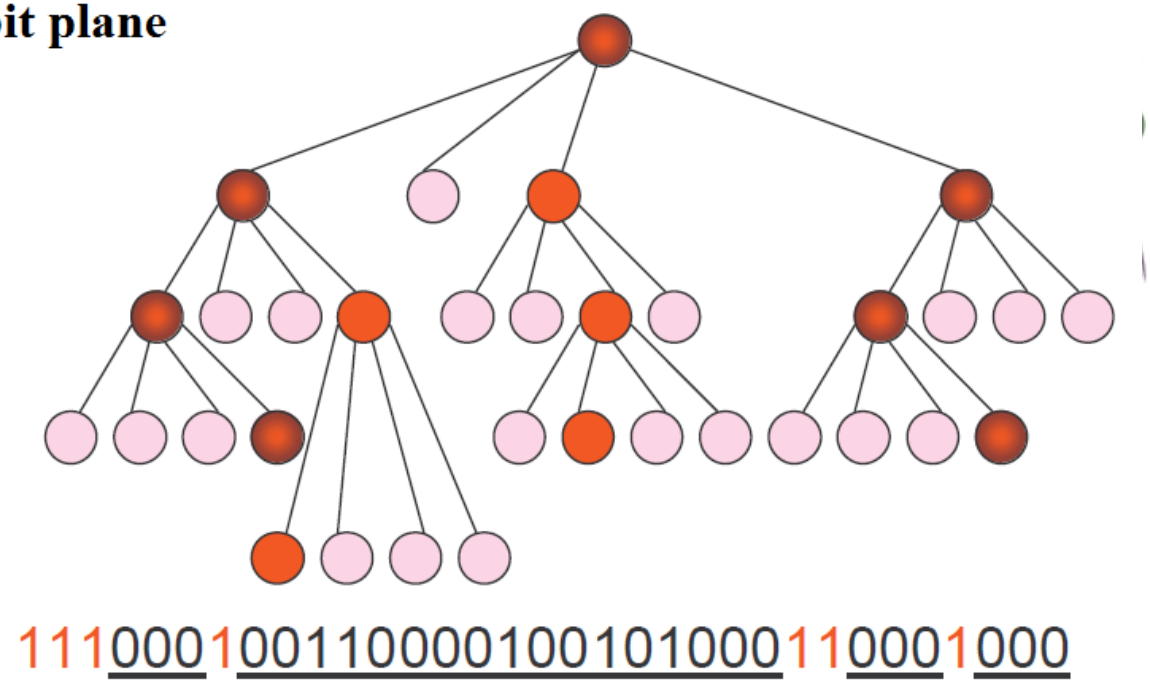
1110001001100001001010001100010000

Depth-first code = 1 for significant
0 for insignificant

Wavelet Based Compression
EBCOT (JPEG 2000)

- **Skip symbols that are already known:**
  - **1. nodes significant in previous bit plane**
  - **2. last child of significant parent of other children are insignificant**

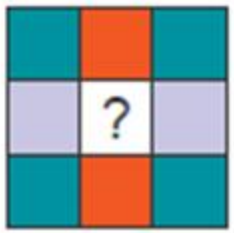


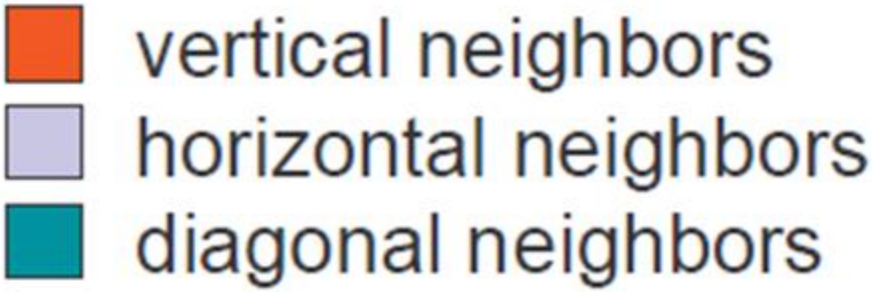


Wavelet Based Compression
EBCOT (JPEG 2000)

- **ZC – Zero Coding : The eight near neighbors of the current wavelet coefficient X are used to determine the context used in encoding each bit of X.**
  - **One of nine contexts is selected and is used to estimate the probability of the bit being encoded.**
    - **Context 0 is selected when coefficient X has no significant near neighbors.**
    - **Context 8 is selected when all eight neighbors of X are significant.**

D_0	V_0	D_1
H_0	X	H_1
D_2	V_1	D_3





Wavelet Based Compression
EBCOT (JPEG 2000)

- **ZC – Zero Coding**
  - **LH is transposed so that it can be treated the same as HL. (LH)T has similar characteristics to HL.**
  - **Each coefficient has its neighbors in the same subband**

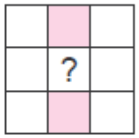
LL and LH Subbands (vertical highpass)			HL Subband (horizontal highpass)			HH Subband (diagonal highpass)		Context
$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum H_i$	$\sum V_i$	$\sum D_i$	$\sum (H_i+V_i)$	$\sum D_i$	
2				2			≥ 3	8
1	≥ 1		≥ 1	1		≥ 1	2	7
1	0	≥ 1	0	1	≥ 1	0	2	6
1	0	0	0	1	0	≥ 2	1	5
0	2		2	0		1	1	4
0	1		1	0		0	1	3
0	0	≥ 2	0	0	≥ 2	≥ 2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

Wavelet Based Compression
EBCOT (JPEG 2000)

Examples

significant in previous bit-plane

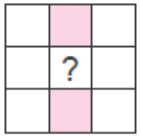
HL



h = 0
v = 2
d = 0

Context 4

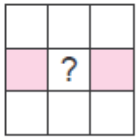
HH



h = 0
v = 2
d = 0

Context 2

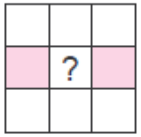
HL



h = 2
v = 0
d = 0

Context 8

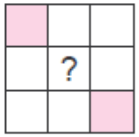
HH



h = 2
v = 0
d = 0

Context 2

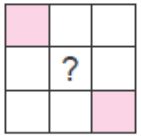
HL



h = 0
v = 0
d = 2

Context 2

HH

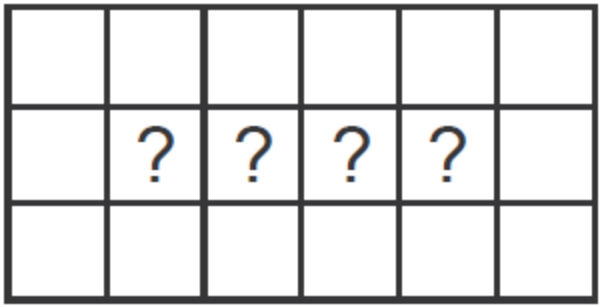


h = 0
v = 0
d = 2

Context 6

Wavelet Based Compression
EBCOT (JPEG 2000)

- **RLC – Run Length Coding:**
  - **Looks for runs of 4 that are likely to be insignificant**



  - **If all insignificant then code as a single symbol**
  - **Main purpose – to lighten the load on the arithmetic coder.**

Wavelet Based Compression
EBCOT (JPEG 2000)

- **Bit Allocation**
  - **How do we truncate the encoded blocks to achieve a desired bit rate and get maximum fidelity.**
- **Basic Set Up:**
  - **Encoded block k can be truncated to n_k bits.**
  - **Total Bit Rate is:**

$$\sum_k n_k$$

  - **Distortion attributable to block k is**

$$D_k^{n_k} = W_k^2 \sum_{(i,j)\in B_k} (C^{n_k}(i,j) - C(i,j)^2$$

where wk is the "weight" of the basis vectors for block k and $C^{nk}(i,j)$ is the recovered coefficients from n_k bits of block k.

Wavelet Based Compression
EBCOT (JPEG 2000)

- **Bit Allocation as an Optimization Problem:**
  - **Input: Given m embedded codes and a bit rate target R**
  - **Output: Find truncation values n_k, 0<k<m,**

    such that $D = \sum_k D_k^{n_k}$ **is minimized and** $\sum_k n_k \le R$

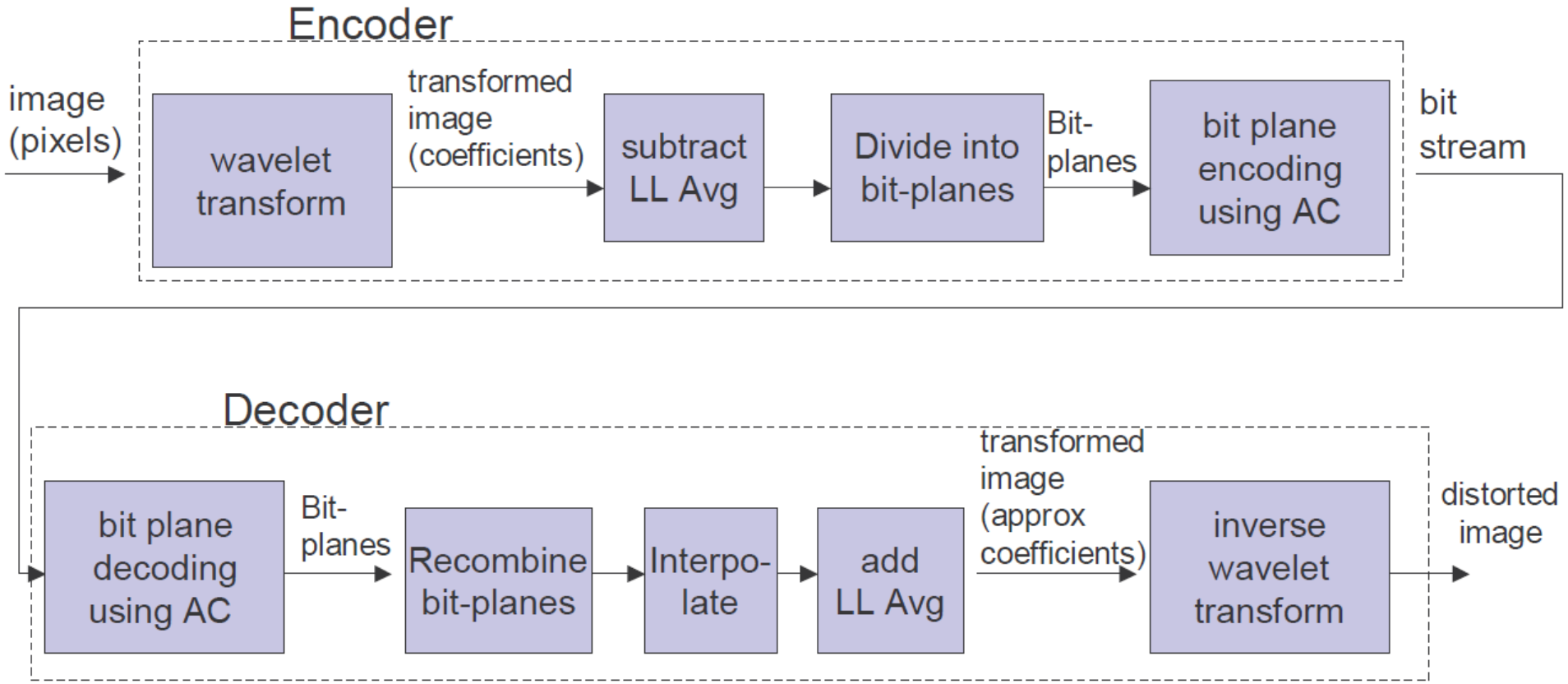
  - **It is an NP-hard problem generally**
  - **There are fast approximate algorithms that**
  - **work well in practice**
    - **Lagrange multiplier method**
    - **Multiple choice knapsack method**

Wavelet Based Compression
PACW

- **Implemented by undergraduates Amanda Askew and Dane Barney in Summer 2003.**

- **A simple image coder based on**
  - **Bit-plane coding**
    - **Significance pass**
    - **Refinement pass**
  - **Arithmetic coding**
  - **Careful selection of contexts based on statistical studies**

Wavelet Based Compression
PACW



Wavelet Based Compression
Arithmetic Coding in PACW

- Performed on each individual bit plane.
  - Alphabet is S={0,1}
  - Signs are coded as needed
- Uses integer implementation with 32-bit integers. (Initialize L = 0, R = 2^{32} -1)
- Uses scaling and adaptation.
- Uses contexts based on statistical studies.

Wavelet Based Compression
PACW

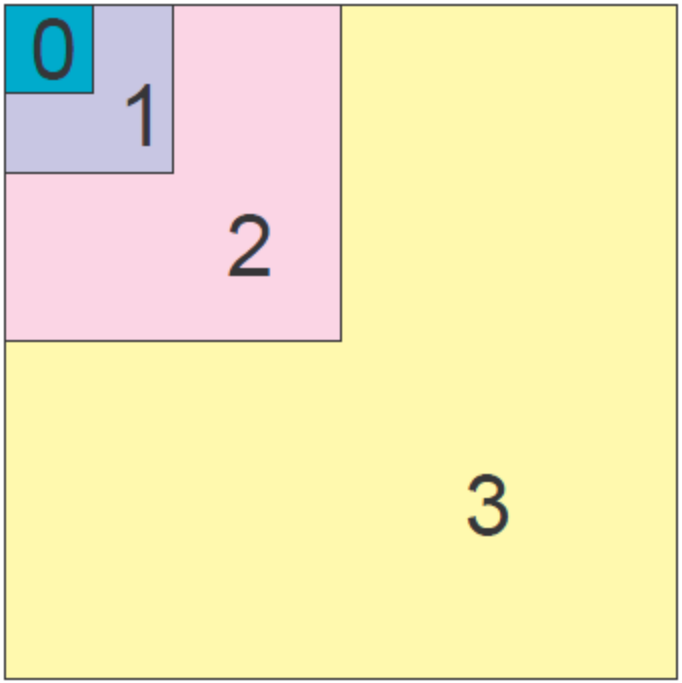
- **Encoding the Bit-Planes:**
  - **Code most significant bit-planes first**
  - **Significance pass for a bit-plane**
    - **First code those coefficients that were insignificant in the previous bit-plane.**
    - **Code these in a priority order.**
    - **If a coefficient becomes significant then code its sign.**
  - **Refinement pass for a bit-plane**
    - **Code the refinement bit for each coefficient that is significant in a previous bit-plane**

Wavelet Based Compression PACW Contexts (per bit plane)

- **Significance pass contexts:**
  - **Contexts based on**
    - **Subband level**
    - **Number of significant neighbors**
  - **Sign context**
- **Refinement contexts**
  - **1st refinement bit is always 1 so no context needed**
  - **2nd refinement bit has a context**
  - **All other refinement bits have a context**
- **Context Principles**
  - **Bits in a given context have a probability distribution**
  - **Bits in different contexts have different probability distributions**

Wavelet Based Compression
PACW

- **Subband Level**
  - **Image is divided into subbands until LL band (subband level 0) is less than 16x16**
  - **Barbara image has 7 subband levels**

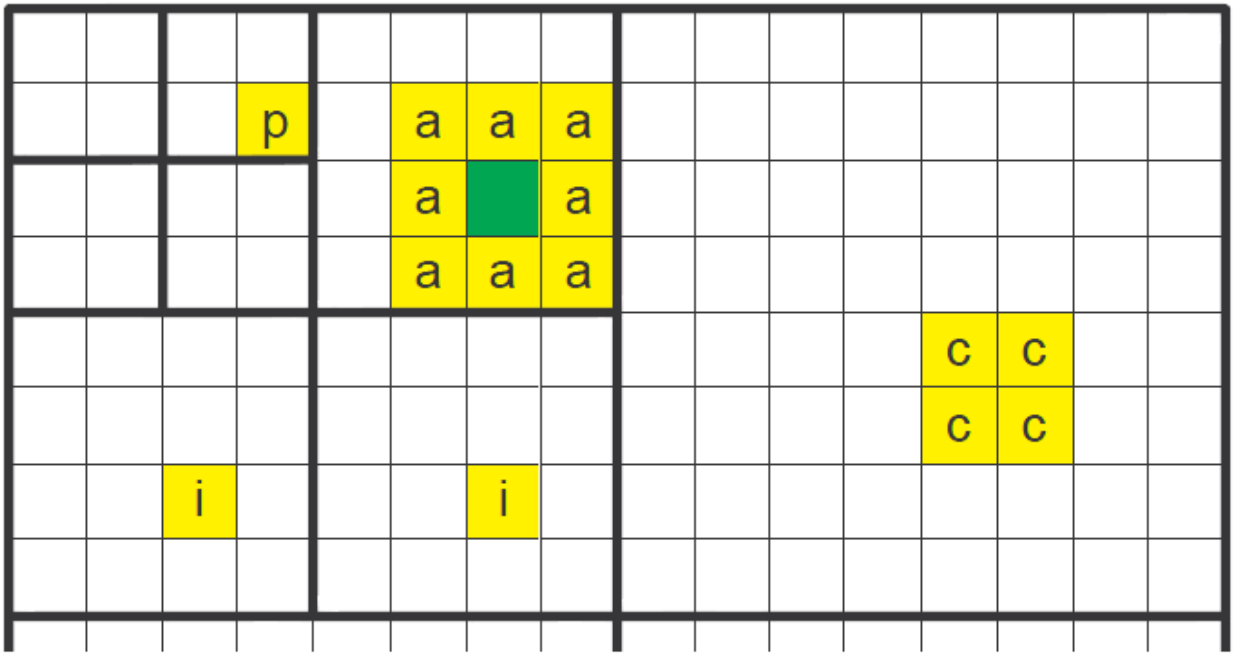


Wavelet Based Compression
PACW

- **Significant Neighbor Metric**
  - **Count the number of significant neighbors**
    - **children count for at most 1**
    - **0,1,2,3+**

Neighbors of ■ :

- p parent
- a spatially adjacent
- i spatially identical
- c child



Wavelet Based Compression
PACW: Context Details

- **Significance pass contexts per bit-plane:**
  - **Max neighbors* num subband levels contexts**
  - **For Barbara: contexts for sig neighbor counts of 0 - 3 and subband levels of 0-6 = 4*7 = 28 contexts**
  - **Index of a context.**
    - **Max neighbors * subband level + num sig neighbors**
    - **Example no. sig neighbors=2, subband level=3, index=4*3+2=14**
- **Sign context**
  - **1 contexts**
- **Two Refinement contexts**
  - **1st refinement bit is always 1 not transmitted**
  - **2nd refinement bit has a context**
  - **all other refinement bits have a context**

Wavelet Based Compression
PACW

- **Statistics for Subband Levels**

Barbara (8bpp)

Subband Level	# significant	# insignificant	% significant
0	144	364	28.3%
1	272	1048	20.6%
2	848	4592	15.6%
3	3134	23568	11.7%
4	12268	113886	9.7%
5	48282	504633	8.7%
6	190003	2226904	7.8%

Wavelet Based Compression
PACW

- **Number of Significant Neighbors**

Barbara (8bpp)

Significant neighbors	# significant	# insignificant	% significant
0	4849	2252468	.2%
1	13319	210695	5.9%
2	22276	104252	17.6%
3	30206	78899	27.7%
4	33244	55841	37.3%
5	27354	39189	41.1%
6	36482	44225	45.2%
7	87566	91760	48.8%

Wavelet Based Compression
PACW

- **Refinement Bit Context Statistics**

Barbara (8bpp)

	0's	1's	% 0's
2nd Refinement Bits	146,293	100,521	59.3%
Other Refinement Bits	475,941	433,982	53.3%
Sign Bits	128,145	130,100	49.6%

- **Barbara at 2bpp: 2nd Refinement bit % 0's = 65.8%**

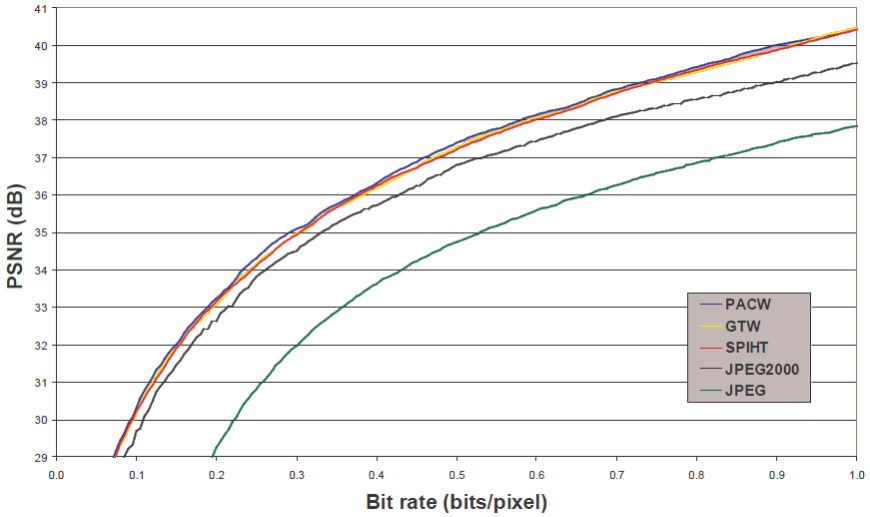
Wavelet Based Compression
PACW

Original Barbara Image

.5 bpp (PSNR = 31.68)

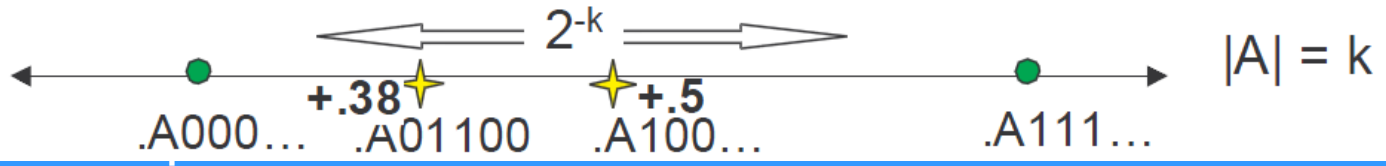
.25 bpp (PSNR = 27.75)

.1 bpp (PSNR = 24.53)



Wavelet Based Compression
PACW Decoding

- **Emulate the encoder to find the bit planes.**
  - **The decoder know which bit-plane is being decoded**
  - **Whether it is the significant or refinement pass**
  - **Which coefficient is being decoded.**
- **Interpolate to estimate the coefficients.**
  - **Coefficients are decoded to a certain number of bit planes**
    - **.101110XXXXX What should X's be?**
    - **.101110000… < .101110XXXXX < .101110111…**
    - **.101110100000 is half-way**
  - **Handled the same as SPIHT and GTW**
    - **if coefficient is still insignificant, do no interpolation**
    - **if newly significant, add on .38 to scale**
    - **if significant, add on .5 to scale**

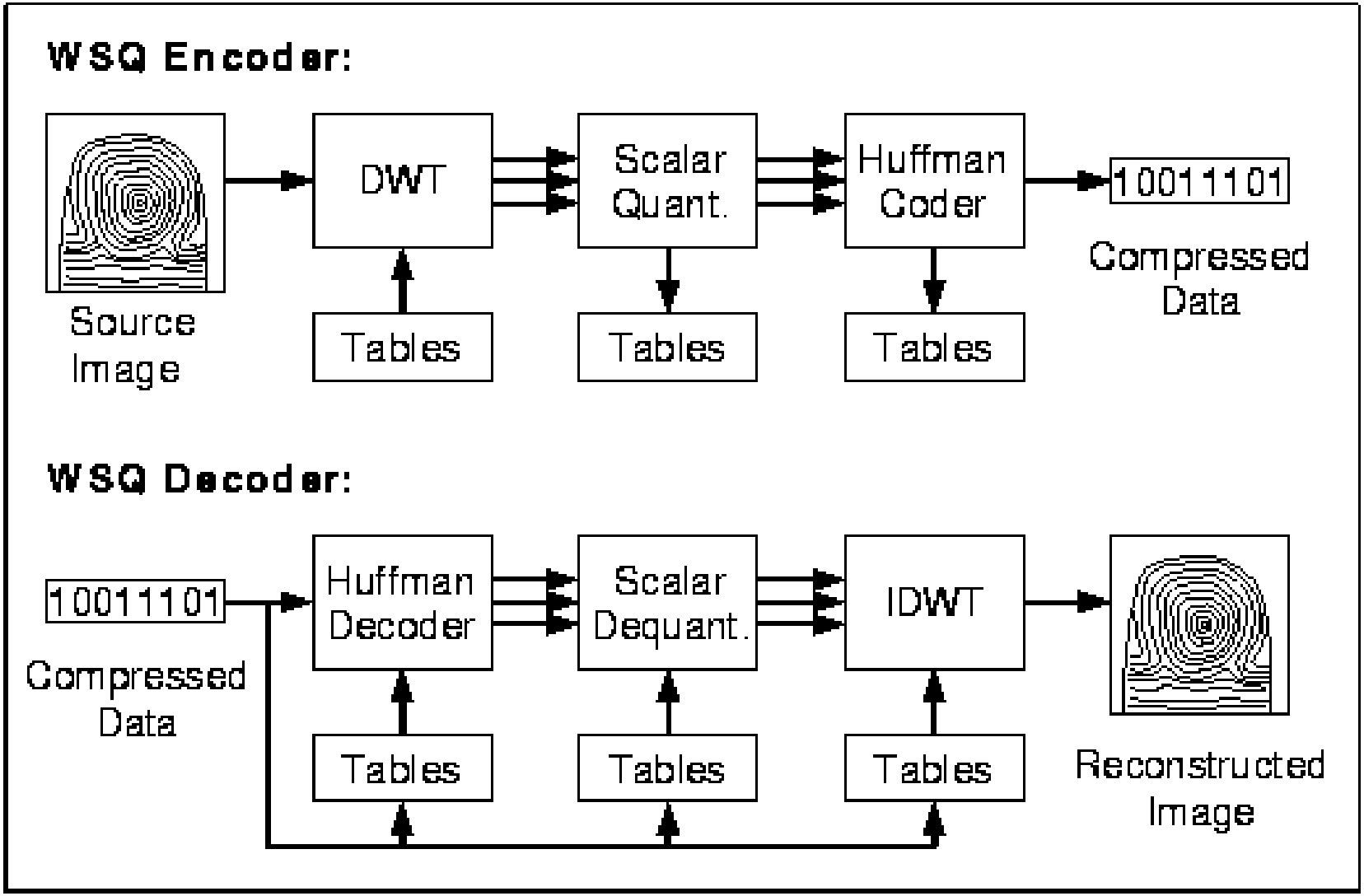


Wavelet/Scalar Quantization (WSQ): Fingerprint Compression

- **The standard employs a discrete wavelet transform-based algorithm (Wavelet/Scalar Quantization or WSQ).**

- **An image coding standard for digitized fingerprints, developed and maintained by:**
  - **FBI**
    - **FBI is digitizing fingerprints at 500 dots per inch with 8 bits of grayscale resolution.**
    - **A single fingerprint card turns into about 10 MB of data!**
  - **Los Alamos National Lab (LANL)**
  - **National Institute for Standards and Technology (NIST).**

WSQ Algorithm



Joint Bi-Level Image processing Group (JBIG) Coding

- No compression method can efficiently compress every type of data.
- This is why new special-purpose methods are being developed all the time.
  - JBIG is an example of a special-purpose method.
- It has been developed specifically for progressive compression of bi-level images.
  - Such images are common in applications where drawings (technical or artistic), with or without text, need to be saved in a database and retrieved.
  - It is customary to use the terms "foreground" and "background" instead of black and white, respectively.

Joint Bi-Level Image processing Group (JBIG) Coding

- One advantage of the JBIG method is its ability to generate low-resolution versions (layers) of the image in the compressed stream. The decoder decompresses these layers progressively, from the lowest to the highest resolution.
  - We first look at the order of the layers. The encoder is given the entire image (the highest-resolution layer), so it is natural for it to construct the layers from high to low resolution and write them on the compressed file in this order.
  - The decoder, on the other hand, has to start by decompressing and displaying the lowest-resolution layer, so it is easiest for it to read this layer first.
  - As a result, either the encoder or the decoder (based on importance of speed in each of them) should use buffering to reverse the order of the layers.

Joint Bi-Level Image processing Group (JBIG) Encoding

- **An important feature of the definition of JBIG is that the operation of the encoder is not defined in detail. The JBIG standard discusses the details of the decoder and the format of the compressed file.**
  - **It is implied that any encoder that generates a JBIG file is a valid JBIG encoder.**
    - **One feature of arithmetic coding is that it is easy to separate the statistical model (the table with frequencies and probabilities) from the encoding and decoding operations.**
    - **It is easy to encode, for example, the first half of a data stream using one model and the second half using another model. This is called *multiple arithmetic coding*, and it is especially useful in encoding images, since it takes advantage of any local structures and interrelationships that might exist in the image.**
  - **JBIG uses multiple arithmetic coding with many models, each a two-entry table that gives the probabilities of a white and a black pixel.**

Joint Bi-Level Image processing Group (JBIG) Decoding

- The decoder starts by initializing the page buffer to a certain value, 0 or 1, according to a code read from the compressed file.
- It then inputs the rest of the file segment by segment and executes each segment by a different procedure.
- There are seven main procedures:
- 1. The procedure to decode segment headers:
  - Each segment starts with a header that includes, among other data and parameters, the segment's type, the destination of the decoded output from the segment, and which other segments have to be used in decoding this segment.

Joint Bi-Level Image processing Group (JBIG) Decoding

- **2) A procedure to decode a generic region:**
  - **This is invoked when the decoder finds a segment describing such a region. The segment is compressed with either arithmetic coding or MMR, and the procedure decompresses it (pixel by pixel in the former case and runs of pixels in the latter).**
  - **In the case of arithmetic coding, previously decoded pixels are used to form a prediction context. Once a pixel is decoded, the procedure does not simply store it in the page buffer, but combines it with the pixel already in the page buffer according to a logical operation (AND, OR, XOR, or XNOR) specified in the segment.**

- **3) A procedure to decode a generic refinement region:**
  - **This is similar to the above except that it modifies an auxiliary buffer instead of the page buffer.**

Joint Bi-Level Image processing Group (JBIG) Decoding

- **4) A procedure to decode a symbol dictionary:**
  - **This is invoked when the decoder finds a segment containing such a dictionary. The dictionary is decompressed and is stored as a list of symbols. Each symbol is a bitmap that is either explicitly specified in the dictionary or is specified as a refinement (i.e., a modification) of a known symbol:**
    - **a preceding symbol from this dictionary**
    - **a symbol from another existing dictionary**
    - **or is specified as a logical combination of several known symbols.**

- **5) A procedure to decode a halftone dictionary:**
  - **This is invoked when the decoder finds a segment containing such a dictionary. The dictionary is decompressed and is stored as a list of halftone patterns (fixed-size bitmaps).**

Joint Bi-Level Image processing Group (JBIG) Decoding

- **6) A procedure to decode a symbol region:**
  - This is invoked when the decoder finds a segment describing such a region. The segment is decompressed and it yields a triplet for each symbol:
    - The coordinates of the symbol relative to the preceding symbol
    - A pointer (index) to the symbol in the symbol dictionary.
    - Since the decoder may keep several symbol dictionaries at any time, the segment should indicate which dictionary is to be used.
  - The symbol's bitmap is brought from the dictionary, and the pixels are combined with the pixels in the page buffer according to the logical operation specified by the segment.
- **7) A procedure to decode a halftone region:**
  - This is invoked when the decoder finds a segment describing such a region. The segment is decompressed into a set of pointers (indexes) to the patterns in the halftone dictionary.

Joint Bi-Level Image processing Group (JBIG) Coding

- Even though JBIG was designed for bi-level images, where each pixel is one bit, it is possible to apply it to grayscale images by separating the bitplanes and compressing each individually, as if it were a bi-level image.

- Also to apply the JBIG on text regions, normally arranged in rows. The encoder treats each symbol as a rectangular bitmap, which it places in a dictionary.
  - The dictionary is compressed by either arithmetic coding and written, as a segment, on the compressed file.
  - Similarly, the text region itself is encoded and written on the compressed file as another segment.

Differential Pulse Code Modulation (DPCM) Coding (inter data redundancy)

- A predictive coding approach.
- Each data value (except at the boundaries) is predicted based on its neighbors (e.g., linear combination) to get a predicted data.
- The difference between the original and predicted data yields a differential or residual data.
  - i.e., has much less dynamic range of pixel values.
- The differential data is encoded using Huffman coding.

Video Compression

Video Compression
Human Perception of Video

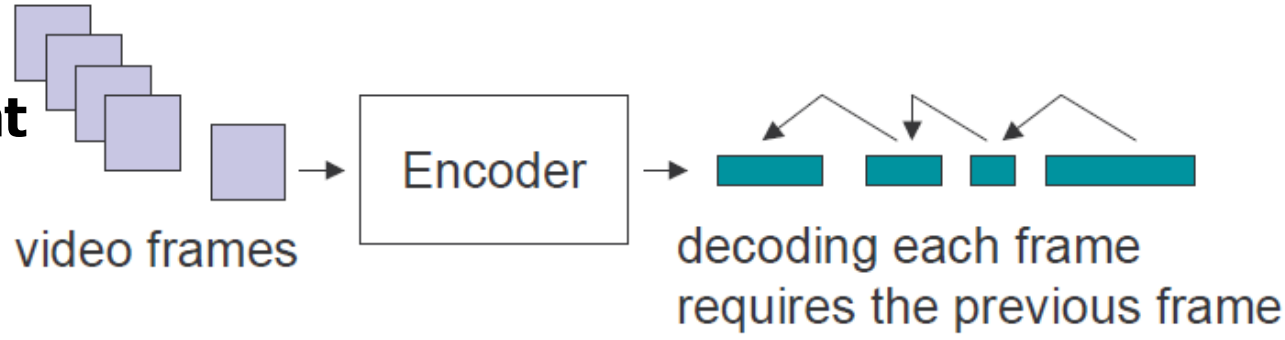
- Sound recording and the movie camera were among the greatest inventions of Thomas Edison. They were later united in video recordings.
- Digital video is a sequence of images, called frames, displayed at a certain *frame rate*.
  - 30 frames per second (fps) seems to allow the visual system to integrate the discrete frames into continuous perception.
- If distorted, nearby frames in the same scene should have only small details wrong.
  - A difference in average intensity is noticeable
- Compression choice when reducing bit rate
  - skipped frames cause stop action
  - lower fidelity frames may be better

Video Compression
Applications of Digital Video

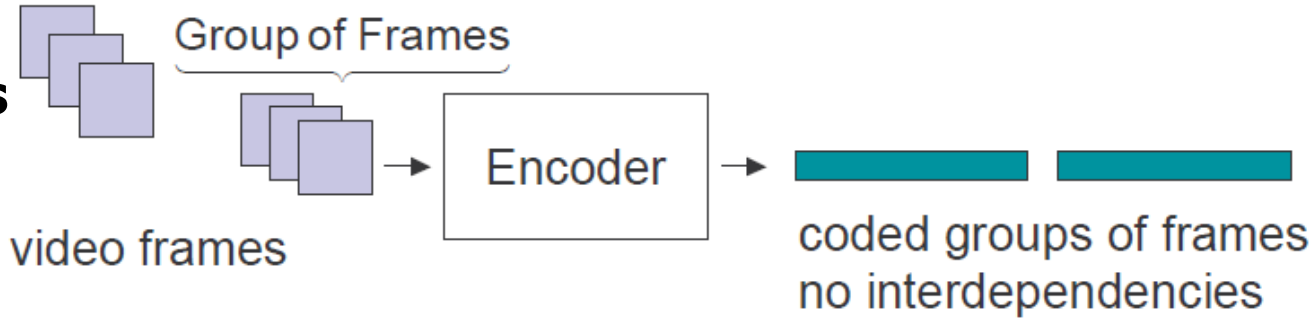
- **Teleconference or video phone**
  - **Very low delay (1/10 second is a standard).**
- **Live Broadcast Video**
  - **Modest delay is tolerable (seconds is normal)**
  - **Error tolerance is needed.**
- **Video-in-a-can (DVD, Video-on-Demand)**
  - **Random access to compressed data is desired**
  - **Encoding can take a lot of time.**
  - **Decoding must always be at least the frame rate.**

Video Compression
Coding Techniques

- **Frame-by-frame coding with prediction**
  - **Very low bit rates**
  - **low delay**
  - **Not error resilient**

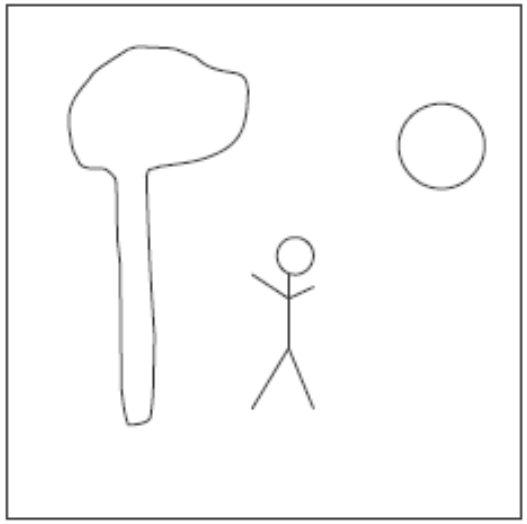


- **Group-of-frames coding**
  - **Higher bit rates – within a group prediction is used**
  - **Error resilient**
  - **Random Access**
  - **Higher delay**

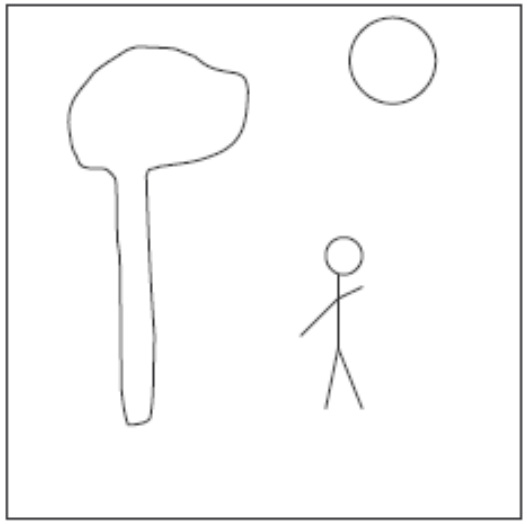


Video Compression
Motion Compensation

- **High Compression Ratios Possible**
  - **Nearby frames are highly correlated. Use the previous frame to predict the current one.**
  - **Need to take advantage of the fact that usually objects move very little in 1/30 th of a second.**
    - **Video coders use motion compensation as part of prediction.**



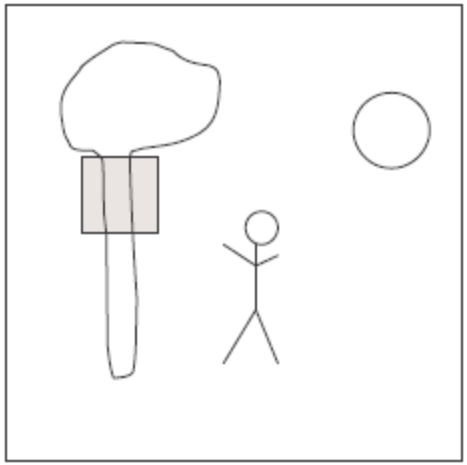
Previous Frame

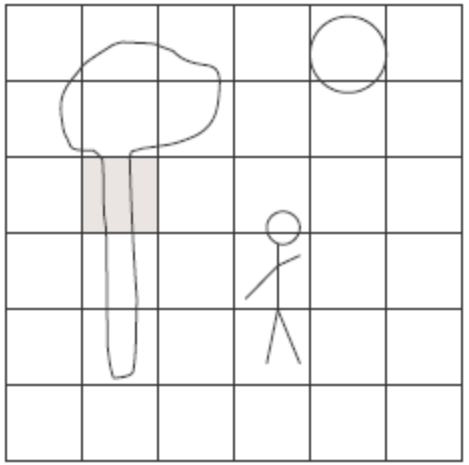


Frame

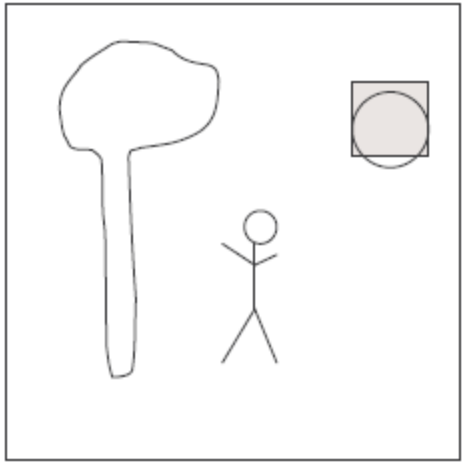
Video Compression
Motion Compensation

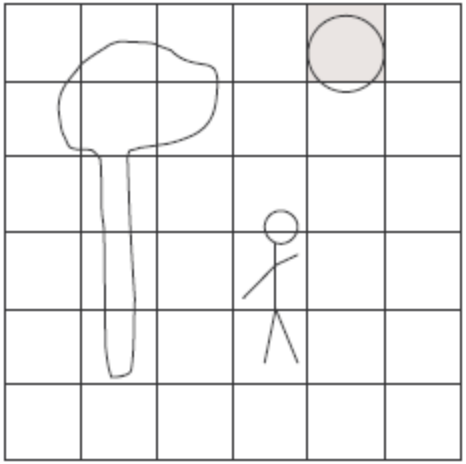
- For each **motion compensation block**
  - Find the block in the previous decoded frame that gives the least distortion.
  - If the distortion is too high then code the block independently. (intra block)
  - Otherwise code the difference (inter block)





motion vector = (0,0)





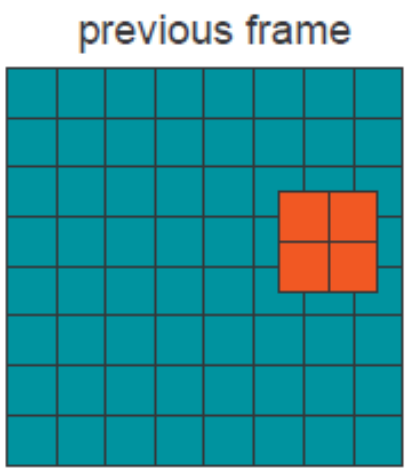
motion vector = (20,5)
20 down and 5 to right

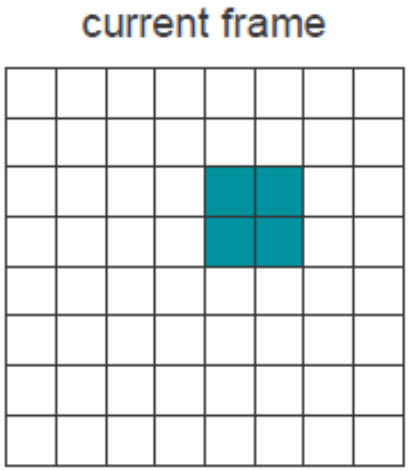
Video Compression
Motion Compensation

- Distortion measured in squared error or absolute error
  - Absolute error is quicker to calculate
- Block size
  - Too small then too many motion vectors
  - Too large then there may be no good match
- Searching range to find best block
  - Too large a search range is time consuming
  - Too small then may be better matches
  - Prediction can help.
- Prediction resolution
  - Full pixel, half-pixel, quarter-pixel resolution
  - Higher resolution takes longer, but better prediction results

Video Compression
Fractional Motion Compensation

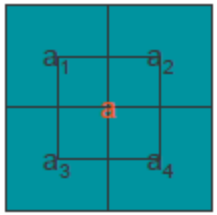
- **Fractional motion compensation is achieved by linear interpolation.**
- **Half or quarter pixel motion compensation may achieve better predictions.**
  - **Calculate an interpolated pixel as the average of overlapping pixels.**
  - **Better interpolation methods exist.**





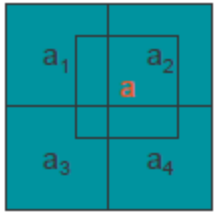
Half pixel motion compensation

Half



$$a = \frac{a_1}{4} + \frac{a_2}{4} + \frac{a_3}{4} + \frac{a_4}{4}$$

Quarter



$$a = \frac{3a_1}{16} + \frac{9a_2}{16} + \frac{a_3}{16} + \frac{3a_4}{16}$$

Video Compression
Standardization Organizations

- **Two organizations have historically dominated general-purpose video compression standardization:**
  - **ITU-T Video Coding Experts Group (VCEG) International Telecommunications Union – Telecommunications Standardization Sector (ITU-T, a United Nations Organization, formerly CCITT), Study Group 16, Question 6**
  - **ISO/IEC Moving Picture Experts Group (MPEG) International Standardization Organization and International Electrotechnical Commission, Joint Technical Committee Number 1, Subcommittee 29, Working Group 11.**

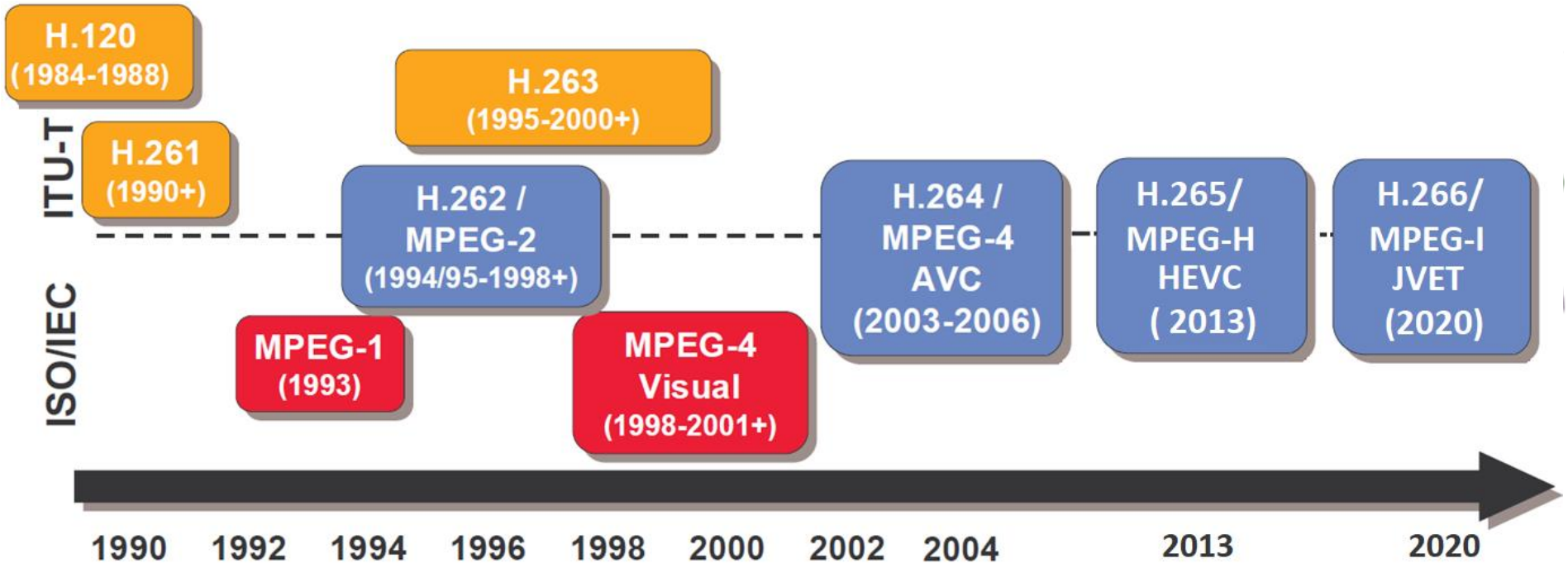
- **Recently, the Society for Motion Picture and Television Engineers (SMPTE) has also entered with "VC-1", based on Microsoft's WMV 9 but this talk covers only the ITU and ISO/IEC work.**

Video Compression

- **ISO/IEC (MPEG) Methods**
- **MPEG-1**
  - Suitable for non-interactive applications
- **MPEG-2/H.262**
  - Developed jointly by ITU-T and ISO/IEC
  - Application independent standard
  - Now in very wide use for DVD and DTV
- **MPEG-4**
  - Multimedia applications
  - Model based coding

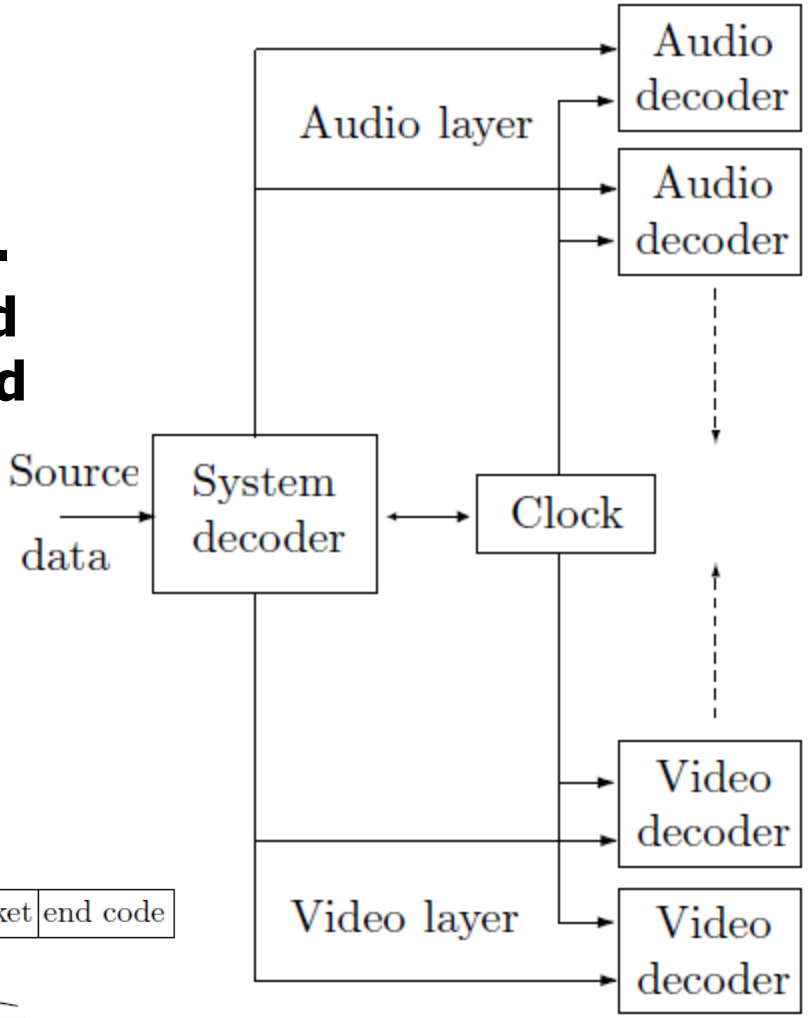
- **ITU-T (VCEG) Methods**
- **H.120**
  - Motion compensation and background prediction
- **H.261**
  - Frame-by-frame encoder
- **H.263**
  - More error resilience
- **H.264/MPEG-4 AVC**
  - Quarter pixel motion compensation
  - Variable size motion blocks
  - Multiple frame prediction

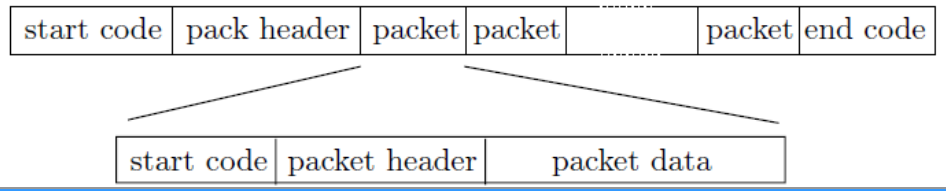
Video Compression
Chronology of International Video Standards



Video Compression

- **The MPEG decoder has three main parts, called layers:**
  - **The video layer (Layer 1):** to decode the packet of input frames.
  - **The system layer (Layer 2):** to read and interpret the various codes and headers in the source data, and routes the packets to either the audio or the video layers to be buffered and later decoded.
  - **The audio layer (Layer 3 or MP3):** to decode the packet of audio signal.





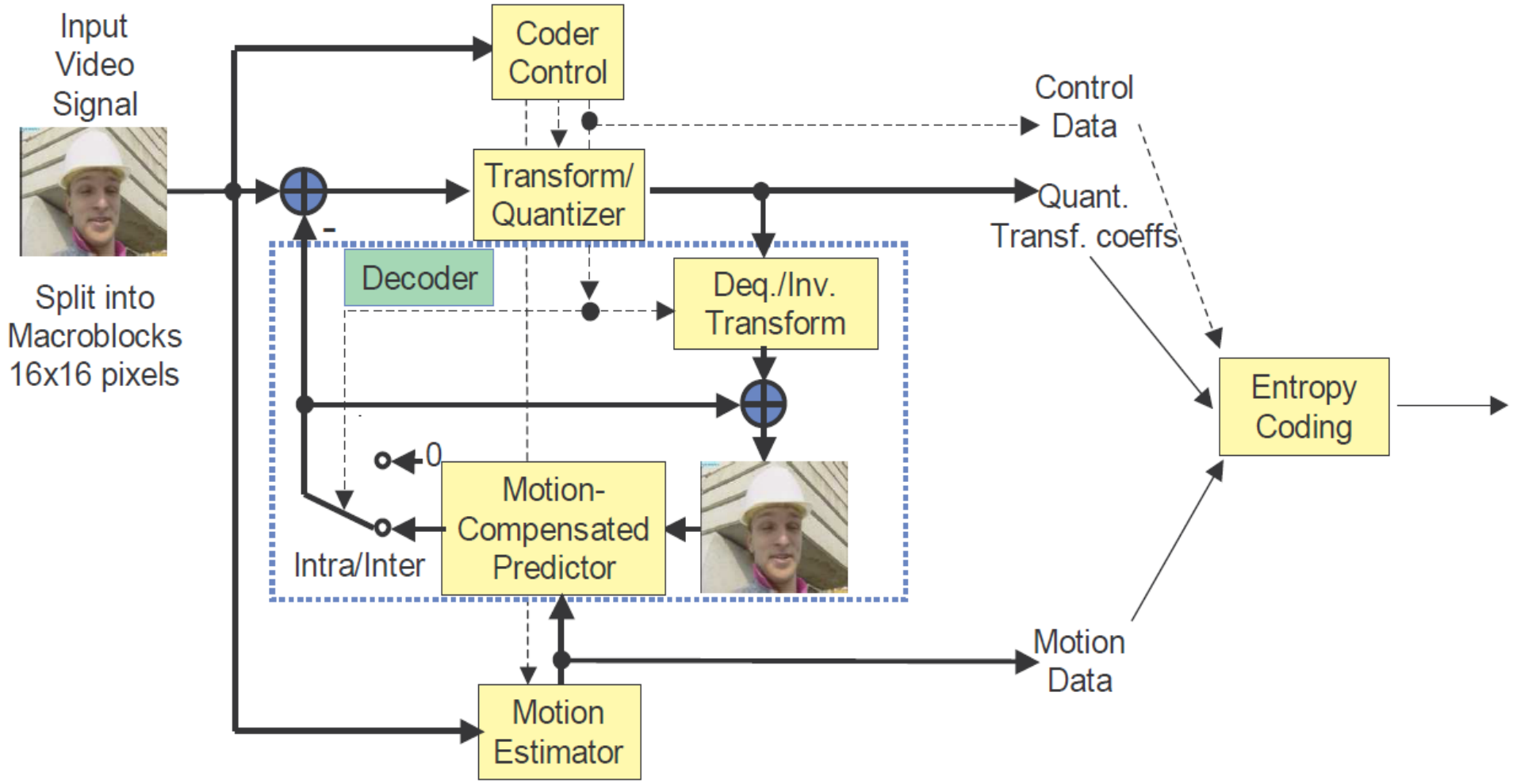
Video Compression
H.120 Digital Video Coding Standard

- **ITU-T (ex-CCITT) Rec. H.120: The first digital video coding standard (1984)**
  - **v1 (1984) had conditional replenishment, DPCM, scalar quantization, variable-length coding.**
  - **v2 (1988) added motion compensation and background prediction.**
  - **Basic "intra" image representation: Discrete Cosine Transform (DCT as in JPEG)**
    - **Analyze 8x8 blocks of image according to DCT frequency content (images tend to be smooth).**
    - **Find magnitude of each discrete frequency within the block.**
    - **Round off ("quantize") the amounts to scaled integer values.**
    - **Send integer approximations to decoder using "Huffman" codes.**
  - **Used Inter frame Motion Prediction.**
  - **Few units made, essentially not in use today.**

Video Compression
H.261

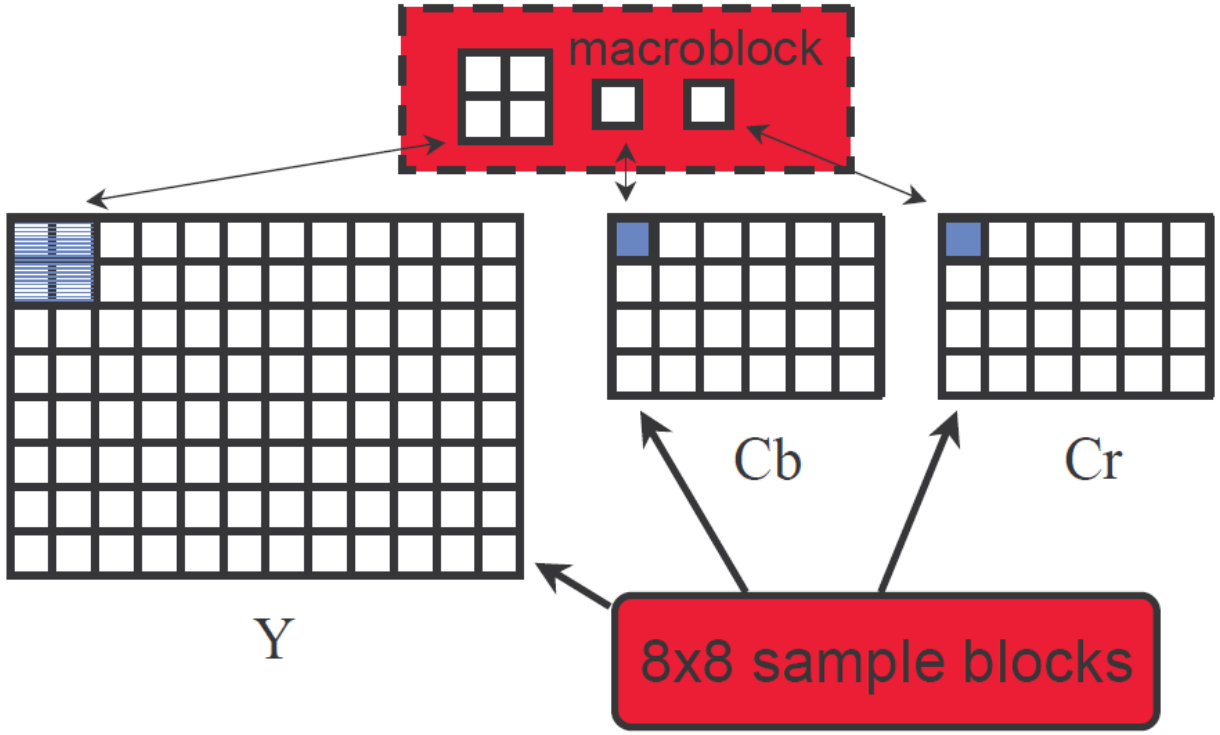
- H.261 is the Basis of Modern Video Compression
- First design (late '90) embodying typical structure dominating today:
  - 16x16 macroblock motion compensation,
  - 8x8 DCT,
  - scalar quantization,
  - zig-zag scan,
  - run-length, and
  - Arithmatic variable-length coding
- v2 (early '93) added a backward-compatible high-resolution graphics trick mode.
- Still in use, although mostly overtaken by H.263.

Video Compression
Structure of H.261

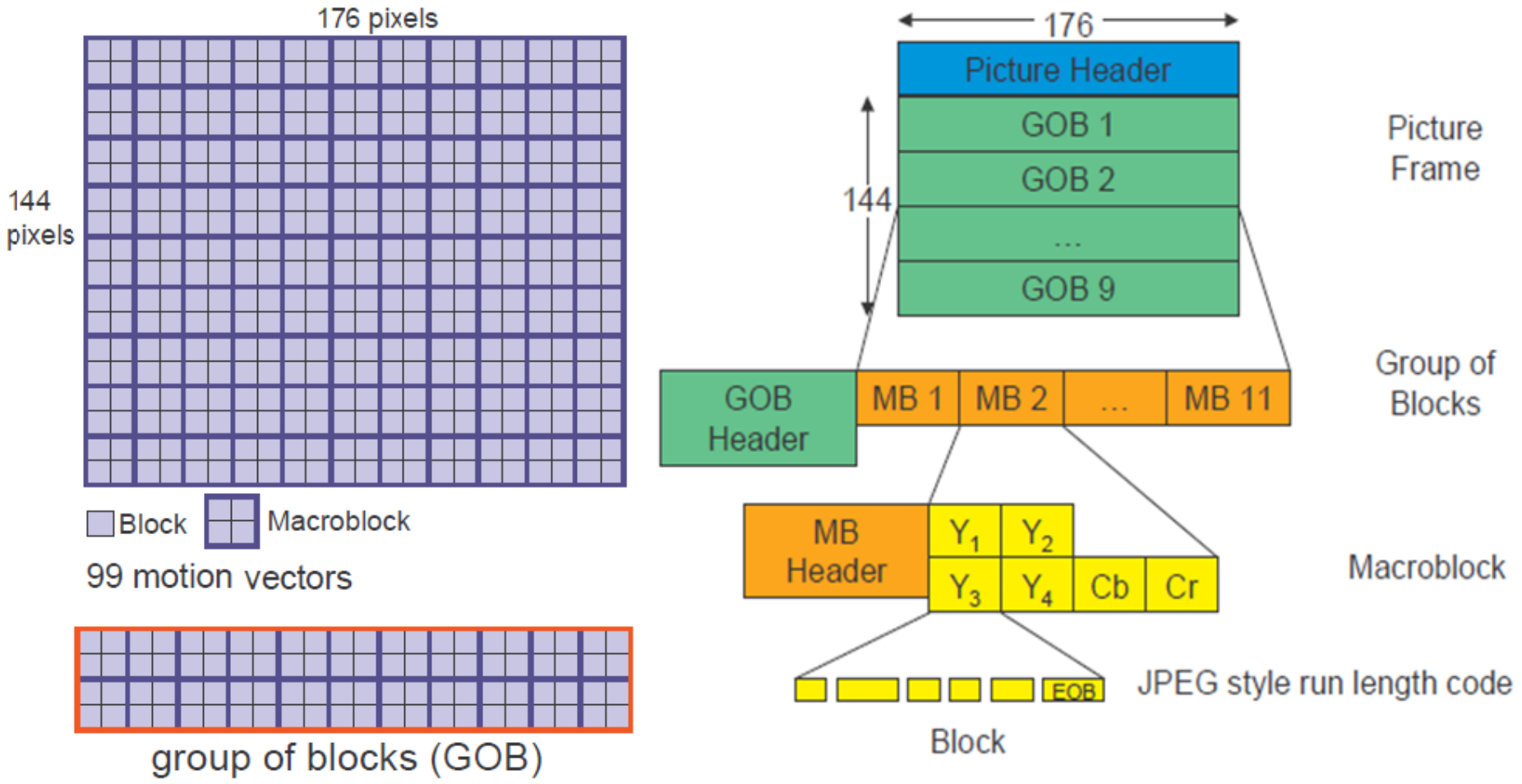


Video Compression
H. 261 Macroblock

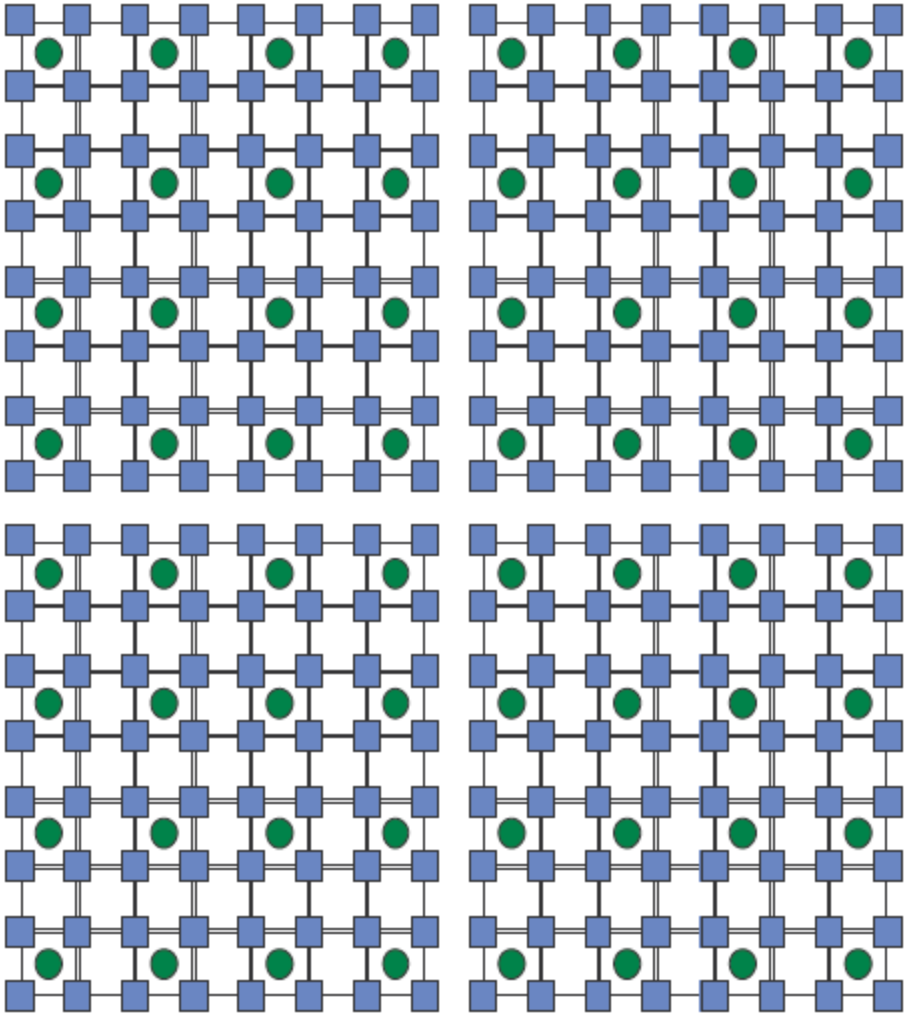
- **The luma and chroma planes are divided into blocks.**
- **Luma blocks are associated with Cb and Cr blocks to create a macroblock.**



Video Compression
H. 261 Macroblock



Video Compression
H. 261 Macroblock



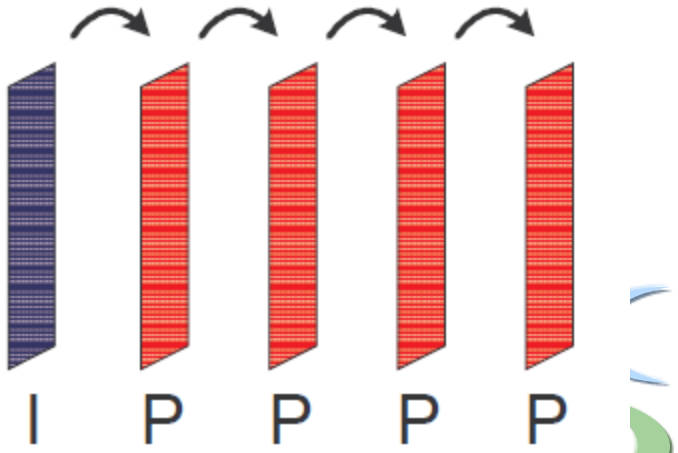
= luma sample

= chroma sample

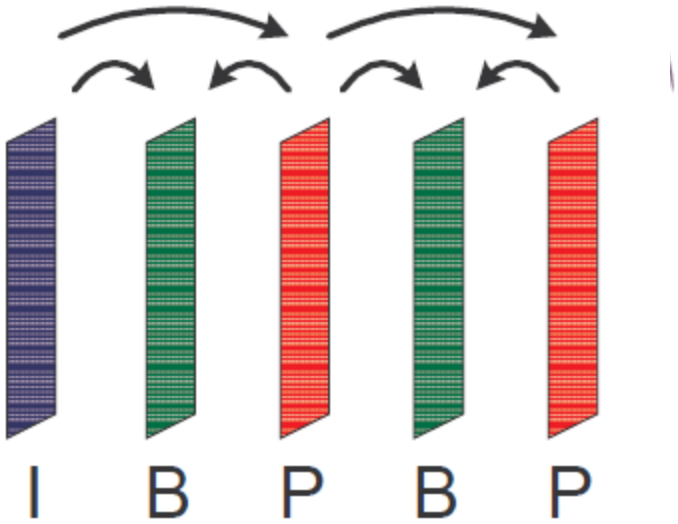
- **Intra/Inter Decisions:**
  - **16x16 macroblock**
  - **DCT of 8x8 blocks**
- **H.261:**
  - **16x16 1-pel motion**
- **H.263:**
  - **16x16 1/2-pel motion**
  - **or (AP mode)**
  - **8x8 1/2-pel motion with overlapping**

Video Compression
H. 261 Motion Predictor

- **P-Pictures Predictive Coding**
  - **P-frames predicted by I-frames or other P-frames existing before it.**
  - **An *I frame is decoded independently of any other frame.***



- **B-Pictures Predictive Coding**
  - **P-frames predicted by I-frames or other P-frames existing before it.**
  - **B-frames predicted by I-frame and P-frame**



Video Compression
MPEG-1

- **MPEG-1 Formally developed by ISO/IEC in 1993.**
  - **Superior quality to H.261 when operated a higher bit rates**
- **Technical features inherited from H.261**
  - **16x16 macroblocks**
  - **16x16 motion compensation, 8x8 DCT,**
  - **scalar quantization,**
  - **zig-zag scan, run-length,**
  - **variable-length coding**
- **Technical features added:**
  - **Bi-directional motion prediction**
  - **Half-pixel motion**
  - **Slice-structured coding**
  - **DC-only "D" pictures**
  - **Quantization weighting matrices**

Video Compression
MPEG-2/H.262

- **Formally developed ('94) jointly by ITU-T and ISO/IEC**
  - **Now in very wide use for DVD and DTV**
- **Primary new technical features:**
  - **Support for interlaced-scan pictures**
  - **Increased DC quantization precision**
  - **Various forms of scalability (SNR, Spatial, breakpoint)**
  - **I-picture concealment motion vectors**
- **Essentially the same as MPEG-1 for progressive-scan pictures, and MPEG-1 forward compatibility required.**
- **Essentially fixed frame rate**

Video Compression
H.263

- **H.263 (v1: 1995) is the next generation of video coding performance, developed by ITU-T**
  - **the current premier ITU-T video standard (has overtaken H.261 as dominant videoconferencing codec)**
- **Baseline Algorithm Features**
  - **Half-pixel motion compensation (also in MPEG-1)**
  - **3-D variable length coding of DCT coefficients**
  - **Median motion vector prediction**
  - **Increased motion vector range with picture extrapolation**
  - **Variable-size, overlapped motion with picture extrapolation**
  - **PB-frames (bi-directional prediction)**
  - **Arithmetic entropy coding**
  - **Continuous-presence multipoint / video mux**
- **A somewhat tangled relationship with MPEG-4**

Video Compression
H.263++

- Annex U: Fidelity enhancement by macroblock and block-level reference picture selection
  - a significant improvement in picture quality
- Annex V: Packet Loss & Error Resilience using data partitioning with reversible VLCs (roughly similar to MPEG-4 data partitioning, but improved by using reversible coding of motion vectors rather than coefficients)
- Annex W:Additional Supplemental Enhancement Information
  - IDCT Mismatch Elimination (specific fixed-point fast IDCT)
  - Arbitrary binary user data
  - Text messages (arbitrary, copyright, caption, video description)
  - Error Resilience:
    - Picture header repetition (current, previous, next+TR, next-TR)
    - Spare reference pictures for error concealment
  - Interlaced field indications (top & bottom)

Video Compression
MPEG-4

- MPEG-4 part 2 (v1: early 1999), formally ISO/IEC.
- Contains the H.263 baseline design:
  - coding efficiency enhancements (esp. at low rates)
- Adds many creative new extra features:
  - more coding efficiency enhancements
  - error resilience / packet loss enhancements
  - segmented coding of shapes
  - zero-tree wavelet coding of still textures
  - coding of synthetic and semi-synthetic content,
  - Motion vectors over picture boundaries
  - Variable block-size motion compensation
  - Intra DCT coefficient prediction
  - Handling of four streams in most levels
  - v2 (early 2000) & v3 (early 2001) & …

Video Compression
H.264/MPEG-4 AVC (Advanced Video Coding)

- **Primary technical objectives:**
  - Significant improvement in coding efficiency
  - High loss/error robustness
  - "Network Friendliness" (carry it well on MPEG-2 or RTP or H.32x or in MPEG-4 file format or MPEG-4 systems or …)
  - Low latency capability (better quality for higher latency)
  - Exact match decoding.

- **Additional version 2 objectives (in FRExt):**
  - Professional applications (more than 8 bits per sample, 4:4:4 color sampling, etc.)
  - Higher-quality high-resolution video
  - Alpha plane support (a degree of "object" functionality)

Video Compression
H.264/MPEG-4 AVC Profiles

- **Three profiles exist in version 1:**
  - **Baseline,**
  - **Main, and**
  - **Extended**

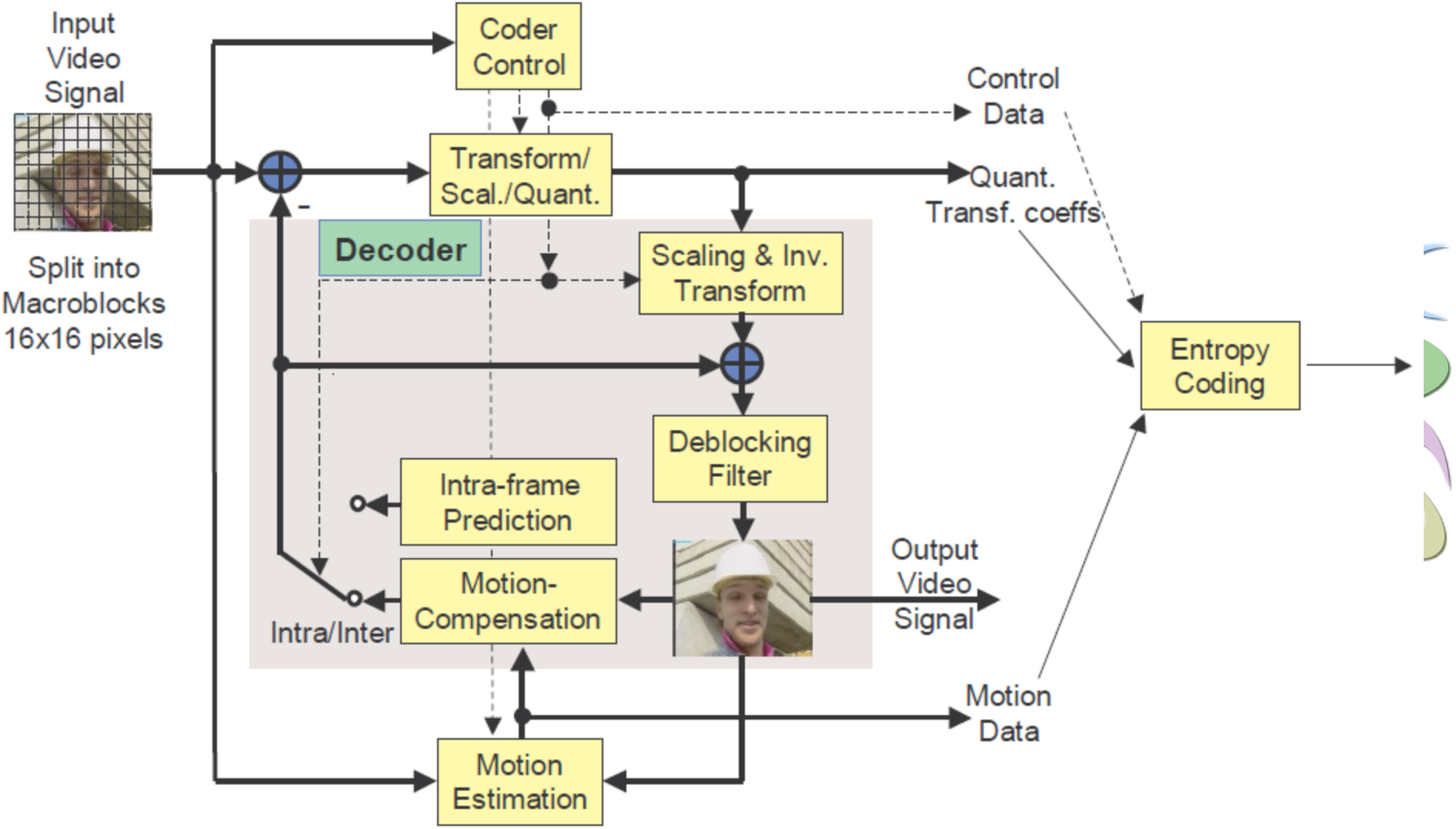
- **Baseline (esp. Videoconferencing & Wireless)**
  - **I and P progressive-scan picture coding (not B)**
  - **In-loop deblocking filter**
  - **1/4-sample motion compensation**
  - **Tree-structured motion segmentation down to 4x4 block size**
  - **VLC-based entropy coding**
  - **Some enhanced error resilience features**
    - **Flexible macroblock ordering/arbitrary slice ordering**
    - **Redundant slices**

Video Compression
H.264/MPEG-4 AVC Profiles

- **Main Profile (esp. Broadcast)**
  - All Baseline features except enhanced error resilience features
  - Interlaced video handling
  - Generalized B pictures
  - Adaptive weighting for B and P picture prediction
  - CABAC (arithmetic entropy coding)

- **Extended Profile (esp. Streaming)**
  - All Baseline features
  - Interlaced video handling
  - Generalized B pictures
  - Adaptive weighting for B and P picture prediction
  - More error resilience: Data partitioning
  - SP/SI switching pictures

Video Compression
H.264/MPEG-4 AVC Structure



Video Compression
H.265/MPEG-H HEVC Profiles

- **High Efficiency Video Coding (HEVC), also known as H.265 and MPEG-H Part 2.**
  - **In comparison to AVC, HEVC offers from 25% to 50% better data compression at the same level of video quality, or substantially improved video quality at the same bit rate.**
- **The primary changes for HEVC include**
  - **used integer DCT and DST transforms with varied block sizes between 4x4 and 32x32**
  - **the expansion of the pattern comparison and difference-coding areas from 16×16 pixel to sizes up to 64×64,**
  - **improved variable-block-size segmentation,**
  - **improved "intra" prediction within the same picture,**
  - **improved motion vector prediction and motion region merging,**
  - **improved motion compensation filtering, and an additional filtering step called sample-adaptive offset filtering.**

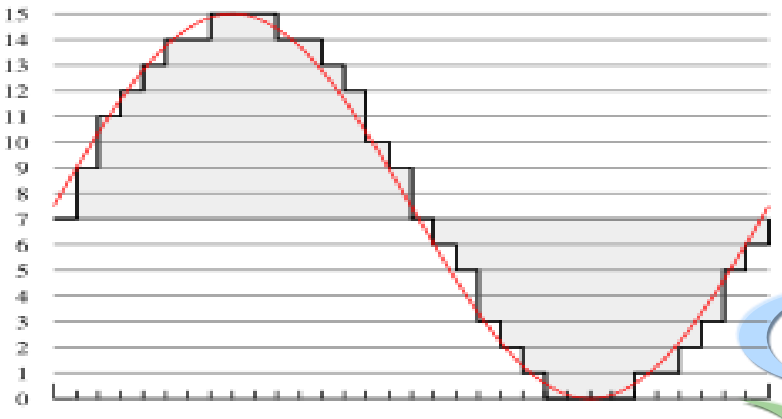
Video Compression
H.266/MPEG-I JVET Profiles

- **Versatile Video Coding (VVC), also known as H.266, MPEG-I Part 3 and Future Video Coding (FVC), is a video compression standard finalized on 6 July 2020, by the Joint Video Experts Team (JVET), and should support:**
  - **resolutions from 4K to 16K as well as 360° videos.YCbCr 4:4:4, 4:2:2 and 4:2:0 with 10 to 16 bits per component,**
  - **BT.2100 wide color gamut and high dynamic range (HDR) of more than 16 stops (with peak brightness of 1000, 4000 and 10000 nits),**
  - **auxiliary channels (for depth, transparency, etc.),**
  - **variable and fractional frame rates from 0 to 120 Hz,**
  - **scalable video coding for temporal (frame rate), spatial (resolution), SNR, color gamut and dynamic range differences, stereo/multiview coding, panoramic formats, and still picture coding.**

Audio Compression

Audio signal – overview

- **Sampling the analog signal**
  - Sample at some fixed rate
  - Each sample is an arbitrary real number
  - Sampling rate (# of samples per second)



4 bit representation (values 0-15)

- **Quantizing each sample**
  - Round each sample to one of a finite number of values
  - Represent each sample in a fixed number of bits
  - Bit rate (# of bits per second).
    - Typically, uncompressed stereo 16-bit 44.1KHz signal has a 1.4MBps bit rate

- **Number of channels (mono / stereo / multichannel).**

Audio Compression

- **Audio data requires too much bandwidth**
  - **Speech: 64 kbps is too high for a dial-up modem user**
  - **Stereo music: 1.411 Mbps exceeds most access rates**
- **Use compression to reduce the size**
  - **Remove redundancy**
  - **Remove details that human tend not to perceive**
- **Audio Compression Methods:**
  - **Simple Audio Compression:**
    - **Prediction based methods**
  - **Psychoacoustic Model**
  - **MPEG Audio Compression**
    - **Layer I and II**
    - **MP3 (MPEG Layer III)**

Simple Audio Compression Methods

- **Silence Compression**
  - Detect the "silence‟ part in the audio signal
  - Code “silence” part using run-length coding method.
- **Adaptive Differential Pulse Code Modulation (ADPCM)**
  - The range of the differences in amplitude between successive samples of the audio waveform is less than the range of the actual sample amplitudes. Therefore, It encode the difference between two or more consecutive signals.
  - Adaptive quantization for the difference signal is used to save the bandwidth or improve the quality.
  - Even higher levels of compression -but at higher levels of complexity- can be obtained by also making the predictor coefficients adaptive. It predicts where the waveform is headed. (G.722-G.726 Standards)

Simple Audio Compression Methods

- **Linear Predictive Coding (LPC)**
  - LPC extract salient features of speech directly from the waveform, rather than transforming to the frequency domain.
  - Uses a time-varying model of vocal tract sound generated from a given audio signal.
  - Transmits only a set of parameters modeling the shape and excitation of the vocal tract, not actual signals or differences.
    - Fits signal to speech model, then transmits the parameters of the model.
  - The decoder (or synthesizer) regenerate the speech using the received model and quantized samples.
  - It sounds like a computer talking, 2.4 kbits/sec.
  - About "Linear": The speech signal generated by the output vocal tract model is calculated as a function of the current speech output plus a second term linear in previous model coefficients.

Simple Audio Compression Methods

- **Code Excited Linear Predictor (CELP) is a more complex family of coders that attempts to mitigate the lack of quality of the simple LPC model.**

- **CELP uses a more complex description of the excitation:**
  - **An entire set (a codebook) of excitation vectors is matched to the actual speech, and the index of the best match is sent to the receiver.**
  - **Performs LPC, but also transmits error term to improve the quality of regenerated audio.**
  - **The complexity increases the bit-rate to 4,800-9,600 bps.**
  - **The resulting speech is perceived as being more similar and continuous.**
  - **Quality achieved this way is sufficient for audio conferencing**

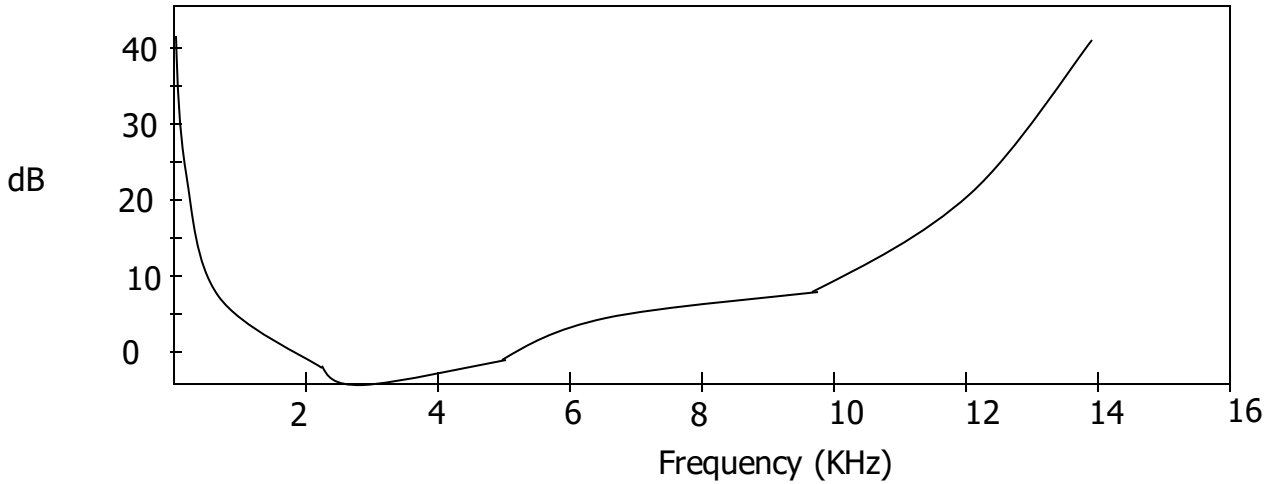
Audio Compression

- **Psychoacoustic Model**
  - **Psychoacoustics – study of how sounds are perceived by humans.**
  - **Uses perceptual coding**
    - **eliminate information from audio signal that is inaudible to the ear.**
  - **Detects conditions under which different audio signal components mask each other:**
    - **Threshold cut-off**
    - **Spectral (Frequency / Simultaneous) Masking**
    - **Temporal Masking**

- **Threshold cut-off and spectral masking occur in frequency domain, temporal masking occurs in time domain.**

Threshold of hearing

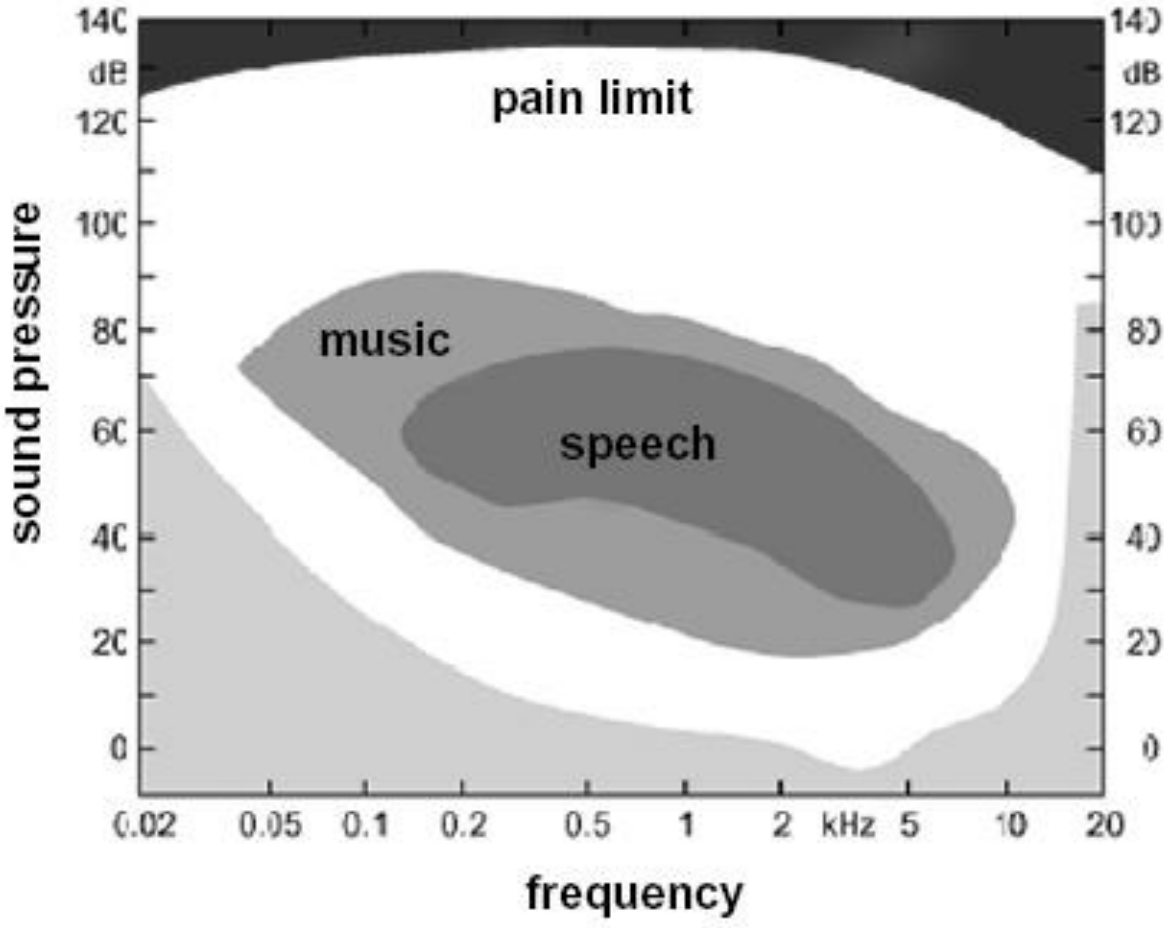
Experiment: Put a person in a quiet room. Raise level of 1 kHz tone until just barely audible. Vary the frequency and plot



- **The ear is most sensitive to frequencies between 1 and 5 kHz, where we can actually hear signals below 0 dB.**
- **Two tones of equal power and different frequencies will not be equally loud.**
- **Sensitivity decreases at low and high frequencies.**

Audio perception

- **Human perception of audio signal**



Audio Compression
Psychoacoustic Model

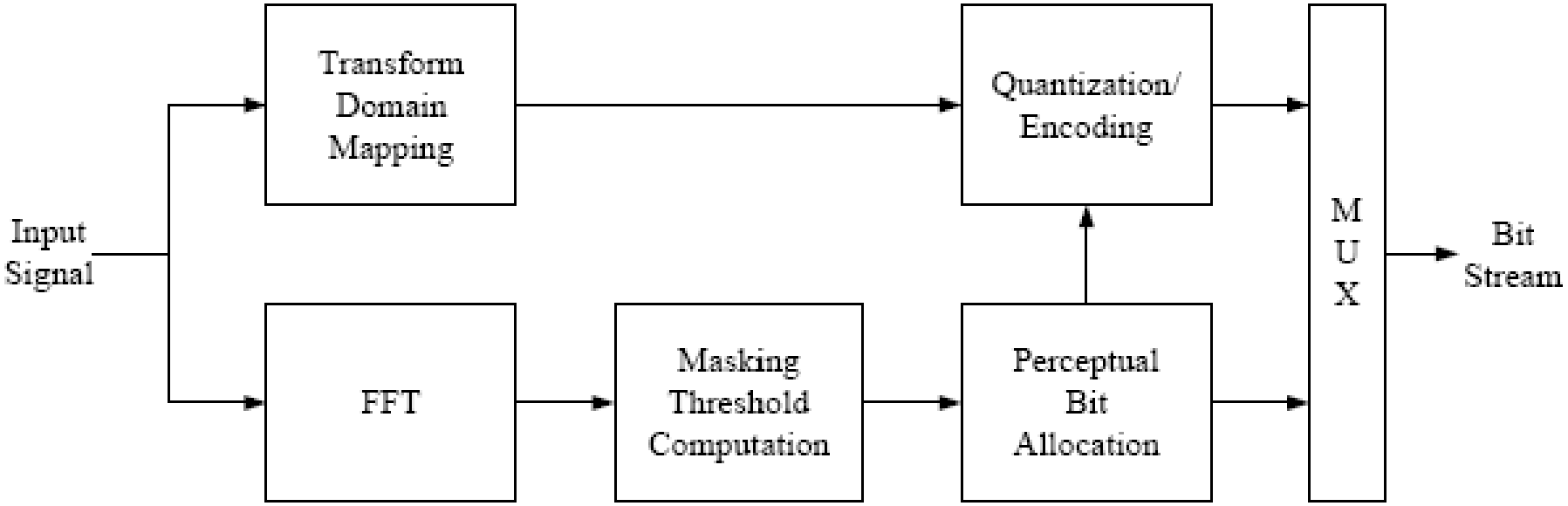


Fig. 1.1 Basic structure of a perceptual audio coder.

- **FFT is used to compute the masking thresholds**

Audio Compression Psychoacoustic Model

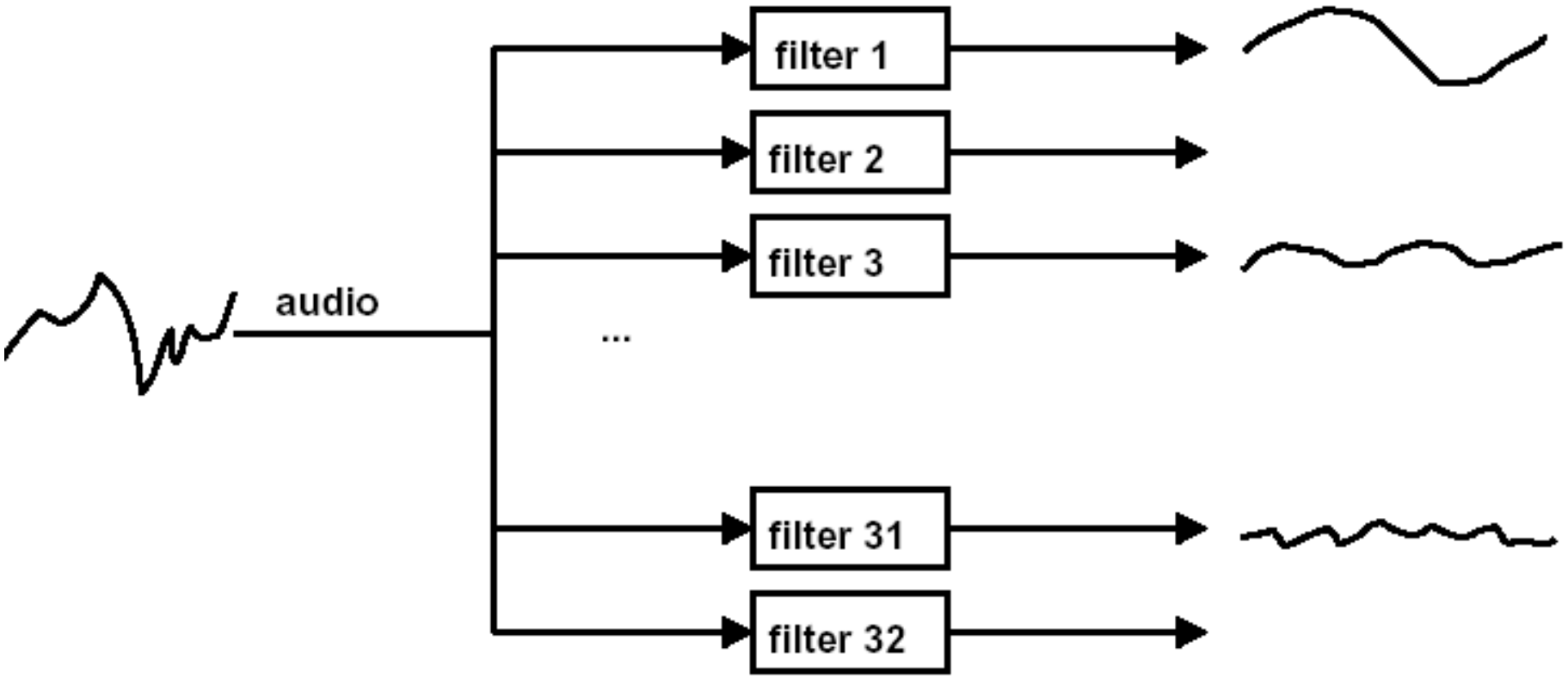
- **Threshold cut-off**
  - Hearing threshold level – a function of frequency
  - Any frequency components below the threshold will not be perceived by human ear
- **Spectral Masking**
  - A frequency component can be partly or fully masked by another component that is close to it in frequency
  - This shifts the hearing threshold
- **Temporal Masking**
  - A quieter sound can be masked by a louder sound if they are temporally close
  - Sounds that occur both (shortly) before and after volume increase can be masked

Audio Compression
Spectral Analysis

- **Spectral Analysis is done to:**
  - **Derive masking thresholds to determine which signal components can be eliminated**
  - **Generate a representation of the signal to which masking thresholds can be applied**
- **Spectral Analysis is done through transforms or filter banks:**
  - **Fast Fourier Transform (FFT)**
  - **Discrete Cosine Transform (DCT) - similar to FFT but uses cosine values only**
  - **Modified Discrete Cosine Transform (MDCT) [used by MPEG-1 Layer-III, MPEG-2 AAC, Dolby AC-3] – overlapped and windowed version of DCT**
  - **Filter Banks**
    - **Poly-phase and wavelet banks are most popular filter structures**

Audio Compression
Filter Bank Structures

- **The audio signal passes through 32 filters with different frequency**



Audio Compression
Filter Bank Structures

- **Polyphase Filter Bank**
  - Signal is separated into subbands, the widths of which are equal over the entire frequency range
  - The resulting subband signals are downsampled to create shorter signals (which are later reconstructed during decoding process) .

- **Wavelet Filter Bank**
  - Unlike polyphase filter, the widths of the subbands are not evenly spaced (narrower for higher frequencies)
  - This allows for better time resolution (ex. short attacks), but at expense of frequency resolution

More Facts

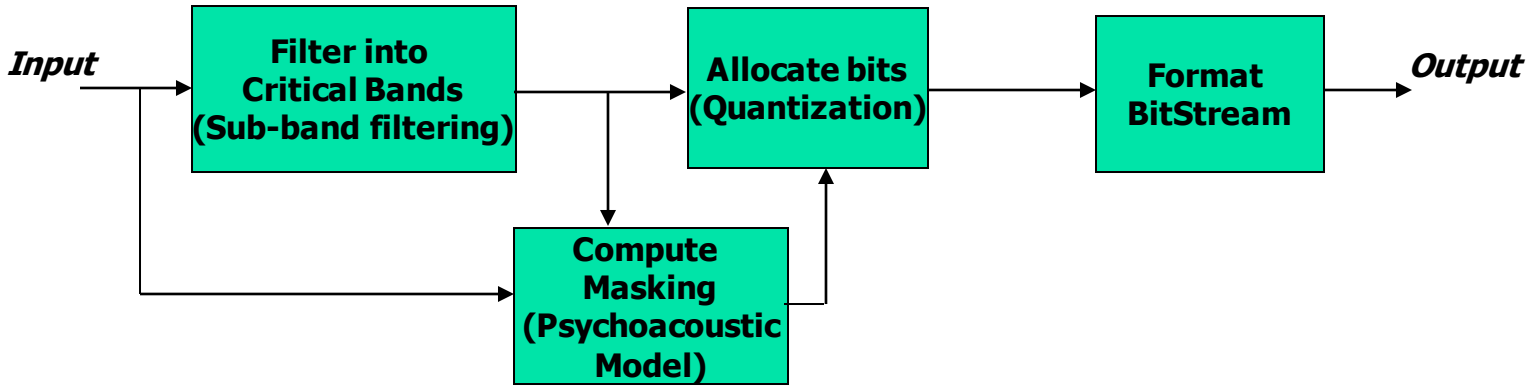
- MPEG-1: Bitrate of 1.5 Mbits/sec for audio and video About 1.2 Mbits/sec for video, 0.3 Mbits/sec for audio
  - (Uncompressed CD audio is 44,100 samples/sec * 16 bits/sample * 2 channels > 1.4 Mbits/sec)
- Compression factor ranging from 2.7 to 24.
- With Compression rate 6:1 (16 bits stereo sampled at 48 KHz is reduced to 256 kbits/sec)
  - Under optimal listening conditions, expert listeners could not distinguish between coded and original audio clips.
- Supports one or two audio channels in one of the four modes:
  1. Monophonic -- single audio channel
  2. Dual-monophonic -- two independent channels, e.g., English and French
  3. Stereo -- for stereo channels that share bits, but not using Joint-stereo coding
  4. Joint-stereo -- takes advantage of the correlations between stereo channels

MPEG Audio Compression

- The two most common advanced (beyond simple ADPCM) techniques for audio coding are:
  - Sub-Band Coding (SBC) based
  - Adaptive Transform Coding based

- MPEG audio coding is comprised of three independent layers. Each layer is a self-contained SBC coder with its own time-frequency mapping, psychoacoustic model, and quantizer.
  - Layer I: Uses sub-band coding
  - Layer II: Uses sub-band coding (longer frames, more compression)
  - Layer III: Uses both sub-band coding and transform coding.

MPEG Audio Compression



1. Use convolution filters to divide the audio signal (e.g., 48 kHz sound) into 32 frequency sub-bands (sub-band filtering).
2. Determine amount of masking for each band caused by nearby band using the psychoacoustic model .
3. If the power in a band is below the masking threshold, don't encode it.
4. Otherwise, determine number of bits needed to represent the coefficient such that, the noise introduced by quantization is below the masking effect (Recall that one fewer bit of quantization introduces about 6 dB of noise).
5. Format bitstream.

MP3 schematic

- **Input: 16 bit at 44kHz sampling is 768kbit/s**

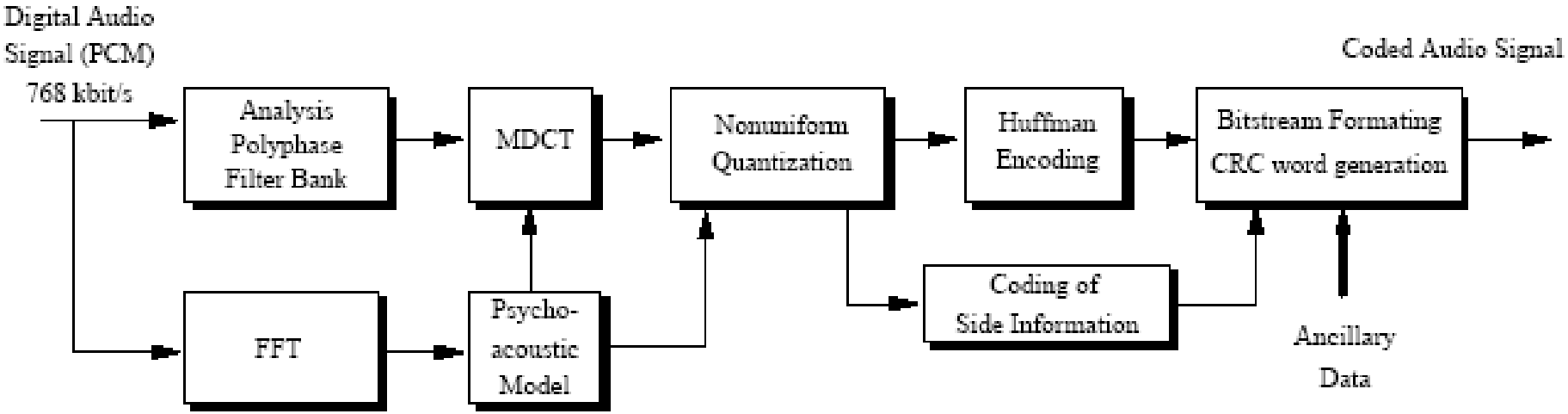


Figure 2.1: *Block diagram of MPEG/Audio encoder.*

MP3 schematic

- **Filter bank: band pass filter into 32 sub-bands each centered at a different frequency**
- **MDCT: Modified Discrete Cosine Transform– each sub-band is divided into time windows.**

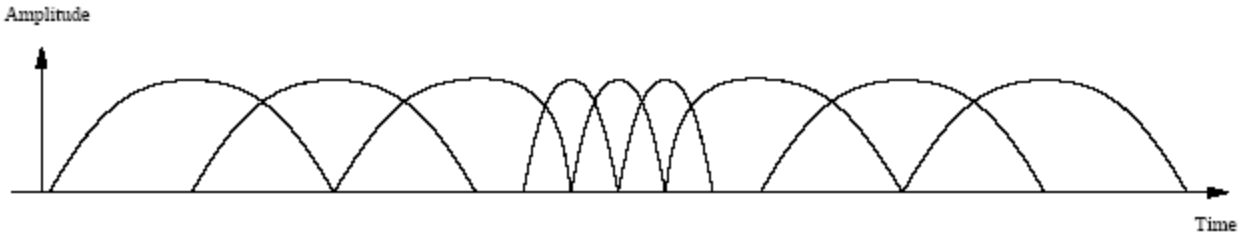


Figure 2.3: *Illustration of a typical sequence of windows to be applied to a subband.*

- **Windows overlap to get rid of a problem called aliasing (high frequencies are confused with low ones). Overlap needed for MDCT.**

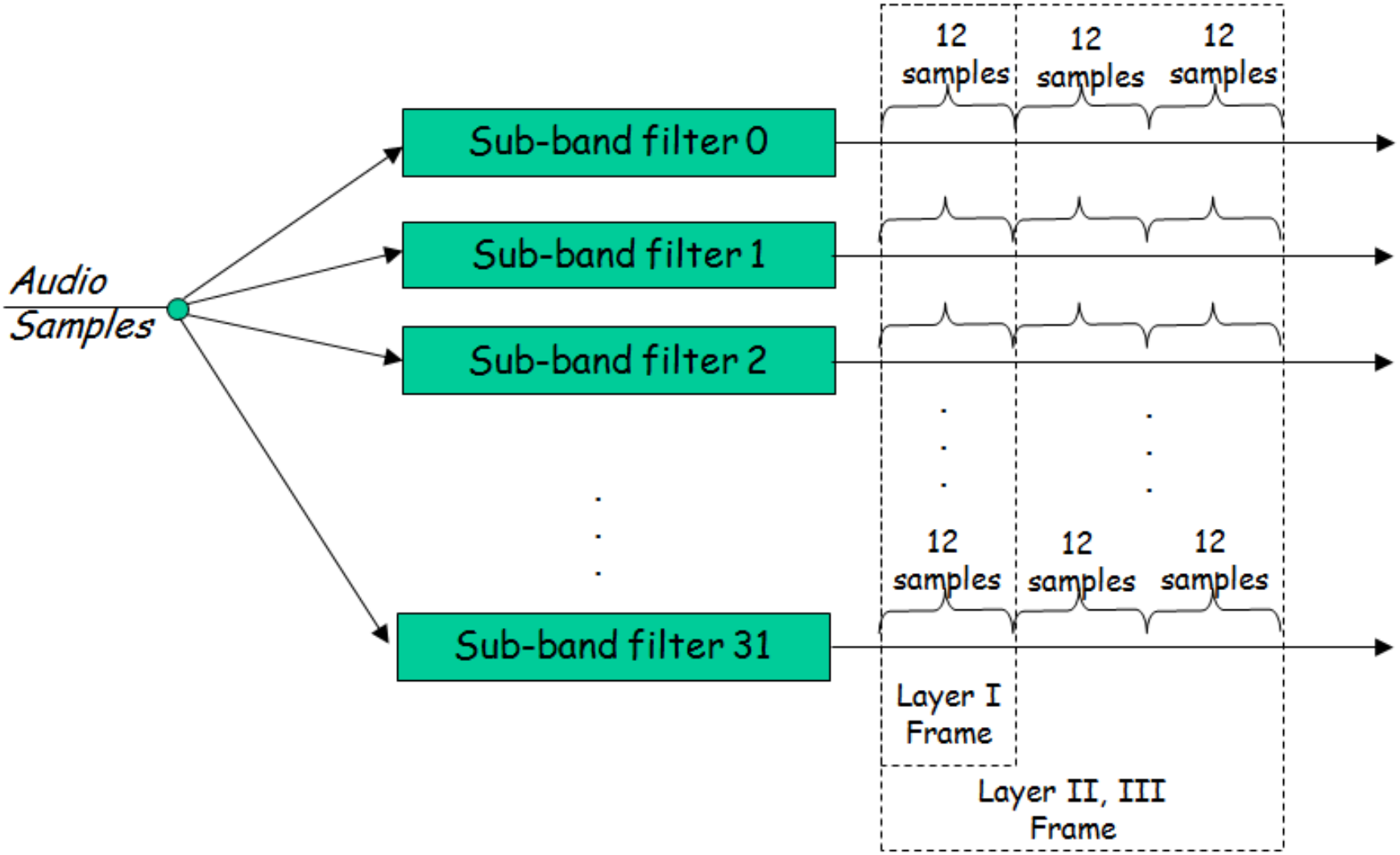
Masking and Quantization

- Performing the sub-band filtering step on the input results in the following values (for demonstration, we are only looking at the first 16 of the 32 bands):

Band	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Level	0	8	12	10	6	2	10	60	35	20	15	2	3	5	3	1

- The 60dB level of the 8th band gives a masking of 12 dB in the 7th band, 15dB in the 9th. (according to the Psychoacoustic model)
- The level in 7th band is 10 dB (< 12 dB), so ignore it.
- The level in 9th band is 35 dB (> 15 dB), so send it.
  - We only send the amount above the masking level .
  - Therefore, instead of using 6 bits to encode it, we can use 4 bits -- a saving of 2 bits (= 12 dB).
    - "determine number of bits needed to represent the coefficient such that, the noise introduced by quantization is below the masking effect" [noise introduced = 12bB; masking = 15 dB]

MPEG Coding Specifics



MPEG Coding Specifics

- **MPEG Layer I**
  - **Filter is applied one frame (12x32 = 384 samples) at a time. At 48 kHz, each frame carries 8ms of sound.**
  - **Uses a 512-point FFT to get detailed spectral information about the signal. (sub-band filter). Uses equal frequency spread per band.**
  - **Psychoacoustic model only uses frequency masking.**
  - **Typical applications: Digital recording on tapes, hard disks, or magneto-optical disks, which can tolerate the high bit rate.**
  - **Highest quality is achieved with a bit rate of 384k bps.**

MPEG Coding Specifics

- **MPEG Layer II**
  - **Use three frames in filter (before, current, next, a total of 1152 samples). At 48 kHz, each frame carries 24 ms of sound.**
  - **Models a little bit of the temporal masking.**
  - **Uses a 1024-point FFT for greater frequency resolution. Uses equal frequency spread per band.**
  - **Highest quality is achieved with a bit rate of 256k bps.**
  - **Typical applications: Audio Broadcasting, Television, Consumer and Professional Recording, and Multimedia.**

MPEG Coding Specifics

- **MPEG Layer III**
  - **Better critical band filter is used**
  - **Uses non-equal frequency bands**
  - **Psychoacoustic model includes temporal masking effects, takes into account stereo redundancy, and uses Huffman coder.**
- **Stereo Redundancy Coding:**
  - **Intensity stereo coding -- at upper-frequency sub-bands, encode summed signals instead of independent signals from left and right channels.**
  - **Middle/Side (MS) stereo coding -- encode middle (sum of left and right) and side (difference of left and right) channels.**

Effectiveness of MPEG Audio

Table 4.2 Summary of MPEG layer1,2 and 3 perceptual encoders

Layer	Apllication	Compressed bit rate	Quality	Example input-to-output delay
1	Digital audio cassette	32-448kbps	Hi-fi quality at 192 kbps per channel	20ms
2	Digital audio and digital video broadcasting	32-192kbps	Near CD-quality at 128 kbps per channel	40ms
3	CD-quality	64kbps	CD-quality of 64kbps per channel	60ms

Effectiveness of MPEG Audio

Layer	Target bit-rate	Ratio	Quality* at 64 kbps	Quality at 128 kbps
Layer I	192 kbps	4:1	--	--
Layer II	128 kbps	6:1	2.1 to 2.6	4+
Layer III	64 kbps	12:1	3.6 to 3.8	4+

Quality factor:

- **5 – perfect**
- **4 - just noticeable**
- **3 - slightly annoying**
- **2 – annoying**
- **1 - very annoying**

Appendix

Codec and Container

- A codec is a portmanteau of *coder-decoder* and is a device or computer program which encodes (compress) or decodes (decompress) a digital data stream or signal.
  - The most widely used standard codecs such as JPEG for images, H.26x and MPEG for video, and MP3 and AAC for audio.

- A container format (informally, sometimes called a wrapper) belongs to a class of computer files that exist to allow multiple data streams to be embedded into a single file, usually along with metadata for identifying and further detailing those streams.
  - Notable examples of container formats include archive files (such as the ZIP format) and formats used for multimedia playback (such as Matroska (MKV), MP4, and AVI).

Codec and Container
Multimedia container formats

- **The container file is used to identify and interleave different data types.**
  - **Simpler container formats can contain one type of data format,**
  - **While more advanced container formats can support multiple audio and video streams, subtitles, chapter-information, and meta-data (tags) along with the synchronization information needed to play back the various streams together.**
- **Some containers are exclusive to audio:**
  - **AIFF (IFF file format, widely used on Mac OS platform)**
  - **WAV (RIFF file format, widely used on Windows platform)**
  - **XMF (Extensible Music Format)**
- **Other containers are exclusive to still images:**
  - **FITS (Flexible Image Transport System) still images, raw data, and associated metadata.**
  - **TIFF (Tagged Image File Format) still images and associated metadata.**

Codec and Container
Multimedia container formats

- **The most popular and flexible multi-media containers can hold many types of audio and video, as well as other media:**
  - **3GP (used by many mobile phones; based on the ISO base media file format)**
  - **ASF (container for Microsoft WMA and WMV, which today usually do not use a container)**
  - **AVI (the standard Microsoft Windows container, also based on RIFF)**
  - **DVR-MS ("Microsoft Digital Video Recording", proprietary video container format developed by Microsoft based on ASF)**
  - **Flash Video (FLV, F4V) (container for video and audio from Adobe Systems)**
  - **IFF (first platform-independent container format)**
  - **Matroska (MKV) (not limited to any coding format, as it can hold virtually anything; it is an open standard container format)**
  - **MJ2 - Motion JPEG 2000 file format, based on the ISO base media file format which is defined in MPEG-4 Part 12 and JPEG 2000 Part 12**

Codec and Container
Multimedia container formats

- QuickTime File Format (standard QuickTime video container from Apple Inc.)
- MPEG program stream (standard container for MPEG-1 and MPEG-2 elementary streams on reasonably reliable media such as disks; used also on DVD-Video discs)
- MPEG-2 transport stream (a.k.a. MPEG-TS) (standard container for digital broadcasting and for transportation over unreliable media; used also on Blu-ray Disc video; typically contains multiple video and audio streams, and an electronic program guide)
- MP4 (standard audio and video container for the MPEG-4 multimedia portfolio, based on the ISO base media file format defined in MPEG-4 Part 12 and JPEG 2000 Part 12) which in turn was based on the QuickTime file format.
- Ogg (standard container for Xiph.org audio formats Vorbis and Opus and video format Theora)
- RM (RealMedia; standard container for RealVideo and RealAudio)
- There are many other container formats, such as NUT, MXF, GXF, ratDVD, SVI, VOB and DivX Media Format