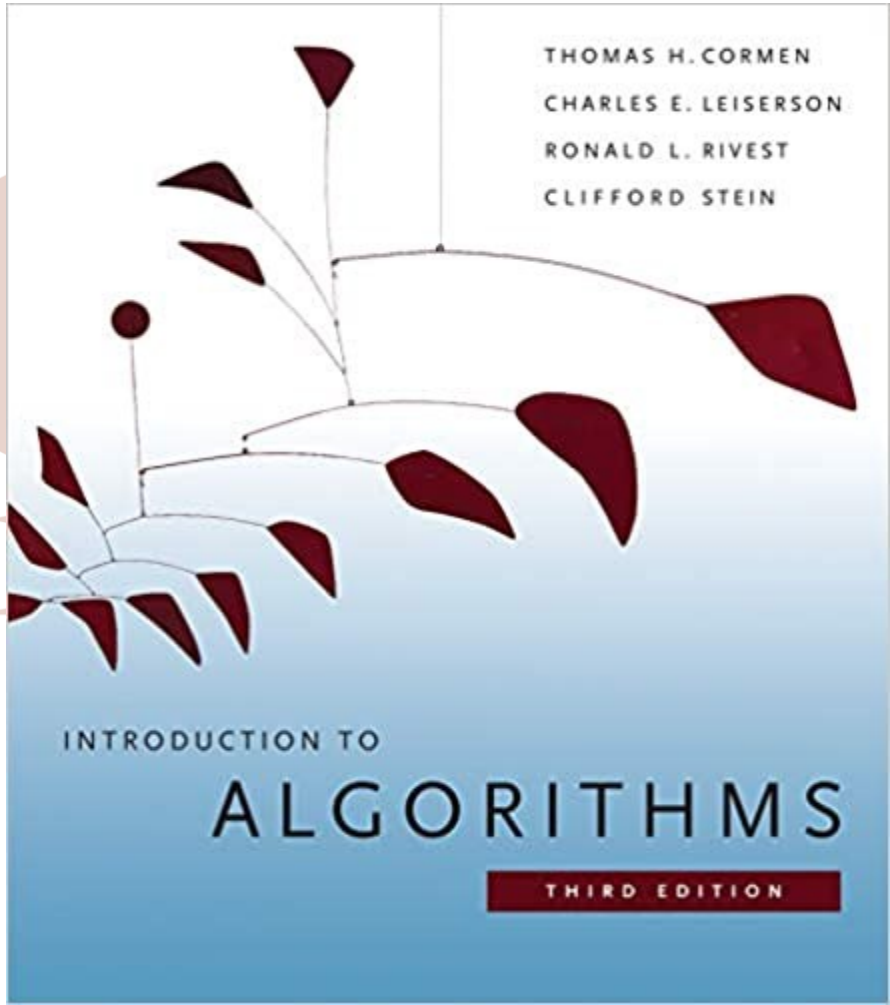
CS 07540 Advanced Design and Analysis of Algorithms

Week 9

- Data Compression
  - Run-Length Encoding
    - Voyager 1 & 2
    - PCX, BMP
  - Huffman Encoding

Uses additional material from *Sedgewick & Wayne, Algorithms 4e*.



Data Compression

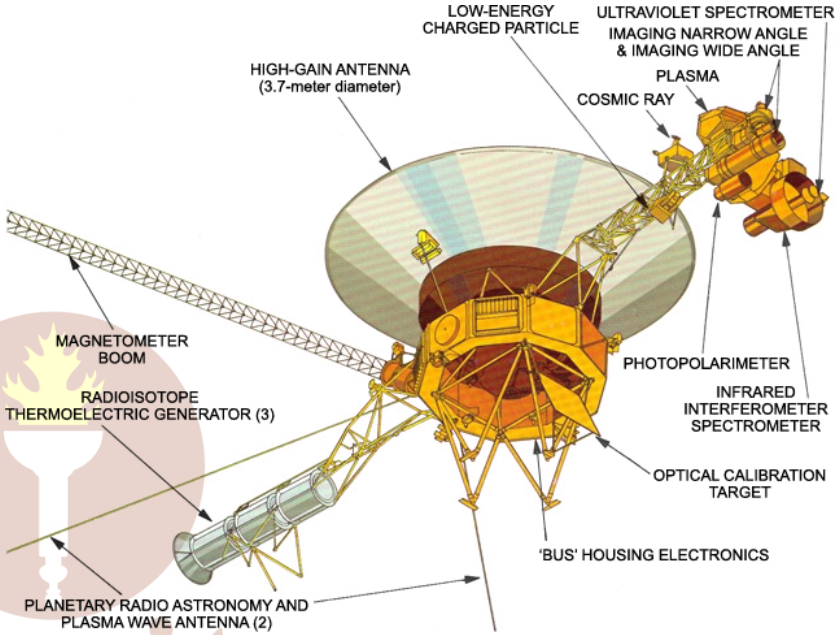
Algorithms designed to represent data efficiently play an important role in computational infrastructure. There are two primary reasons to compress data:

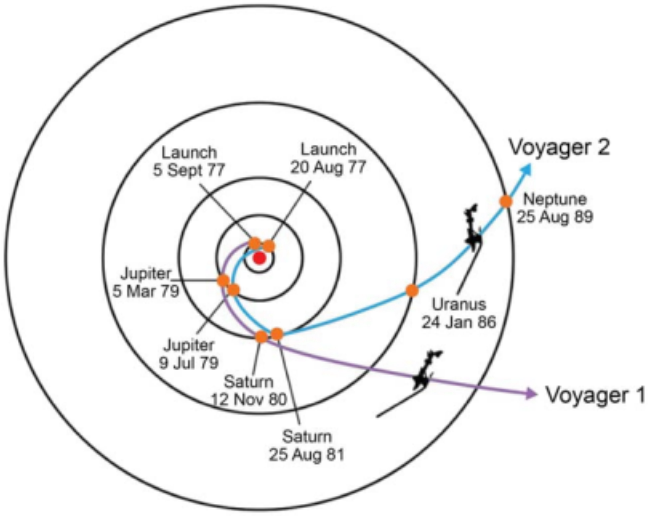
- save storage when saving information,
- save time when communicating information.

While storage prices have generally fallen constantly and transmission speeds in networks have kept going up, savings in storage space and time add up quickly. Energy and bandwidth cost money!

A Historical Example

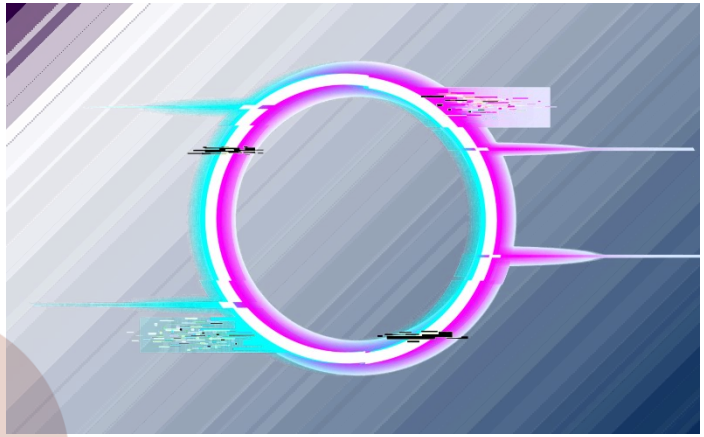
In digital communications, we want to have a high data rate yet also a robust transmission over noisy channels. Note that even fiber optics has noise, such as intersymbol interference and chromatic dispersion! And it only gets worse in space! Voyager 1 was launched on September 5, 1977. It is currently flying at 17 km/s away from earth [at a distance of about] 24×10^9 kilometers (156.3 AU).



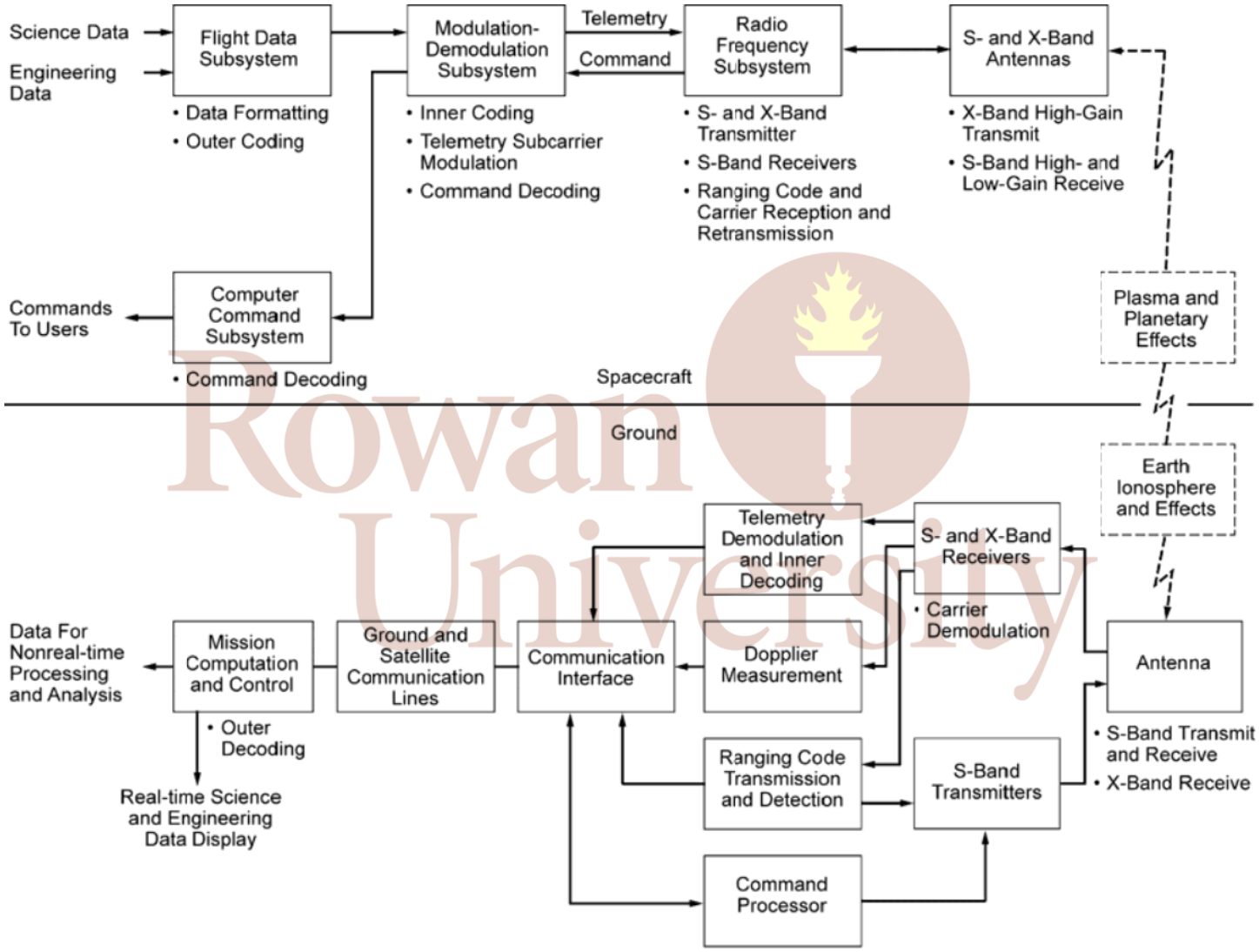


When transmitting data, we can achieve a higher data rate if we

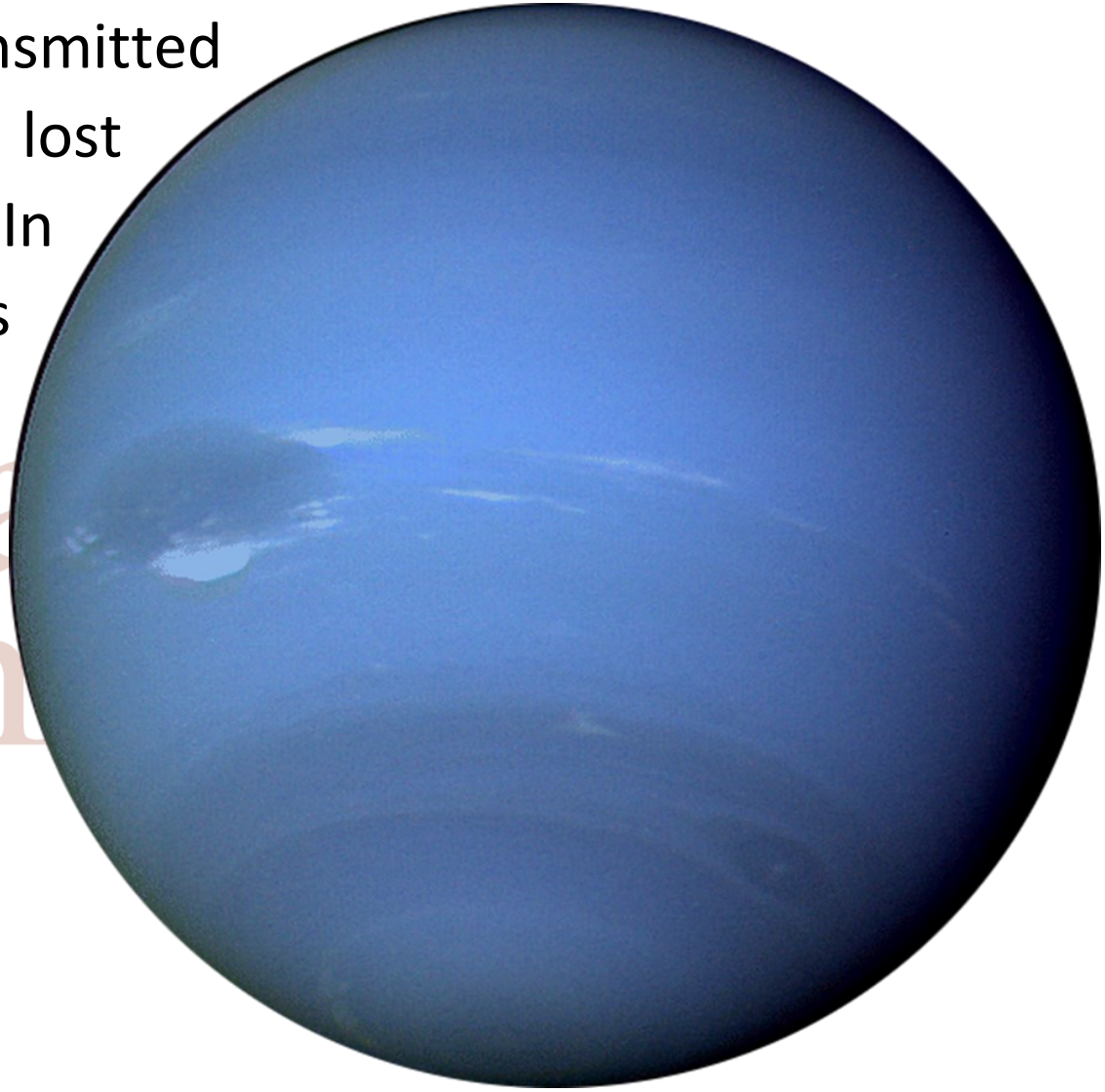
- compress the data, then
- transmit compressed data, and finally
- decompress the data.



The downside is that compression encoded data streams are less robust due to a general vulnerability to transmission errors. Such errors in a compressed data stream may affect control information and present itself with large substreams of false decompressed data. One method of addressing this issue is a layer of error-correcting codes before transmission. Voyager 1 & 2 used Reed-Solomon codes (the same burst error-correcting codes used on compact discs) as “outer codes”.



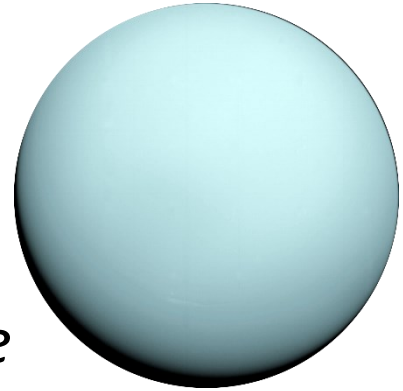
While the ECCs enlarge the transmitted amount of data, retransmitting lost packets is even more expensive. In a spacecraft setting, energy is expensive as well. Hence reducing the size of the transmitted data with compression is important.



Voyager Image Data Compression 📖

"After the Jupiter and Saturn encounters, JPL completed IDC software for Voyager [… and] loaded the software into the backup flight data subsystem (FDS) […] Uncompressed Voyager images contain 800 lines, 800 dots (pixels) per line, and 8 bits per pixel […] much of the data […] is dark space or low-contrast cloud features. By counting only the differences between adjacent pixel gray levels […] image data compression reduced the number of bits for the typical image by 60 percent […]. This reduced the time needed to transmit each complete image from Uranus and Neptune to Earth by the same 60%."

Voyager Error-Correcting Coding 📖



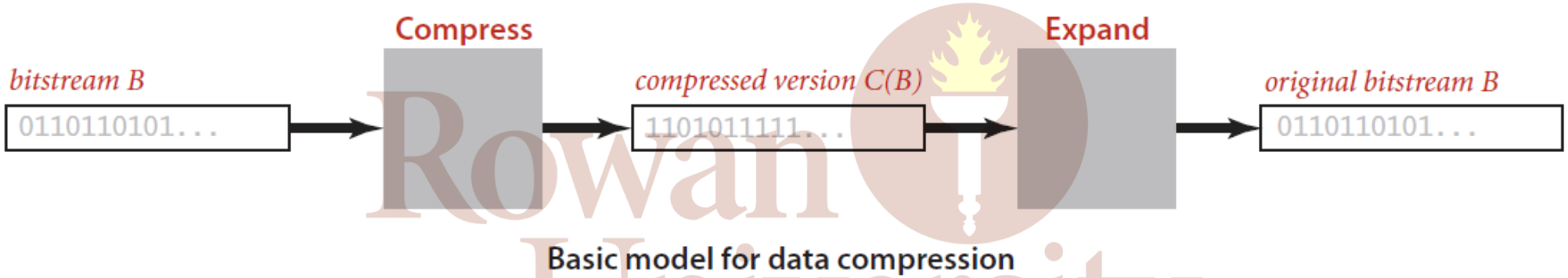
[…] the Voyager telemetry link is subject to noise in the communications channel changing the values of bits transmitted […] coding increases the redundancy of the signal by increasing the number of bits transmitted […]. The Golay encoding algorithm used at Jupiter and Saturn required the transmission of one overhead bit for every information bit transmitted (100 percent overhead). […] The new Reed-Solomon encoding scheme reduced the overhead to about one bit in five (20-percent overhead) and reduced the bit-error rate in the output information from 5×10^{-3} *to* 10^{-6}.

Aside

Encoding is different from encrypting. ***Encoding*** is the conversion of data into another (equivalent) form, usually to make it more robust against transmission errors. Such codes are called ***error correcting codes***. They let us detect and (within reason) correct errors in a data stream.

By contrast, ***encryption*** is the conversion of data into another (equivalent) form which prevents (or makes it harder) for unauthorized middlemen to obtain the original data from intercepted encrypted data. The word encoding is sometimes used as well in this process to describe the conversion part.

We will look at algorithms that compress data lossless and not lossy like mp3 or jpeg. All of the types of data that we process are represented in binary. Hence, we only need think about compressing bitstreams.



Basic model for data compression

Note that ***no algorithm can compress every bitstream to smaller sizes***. Otherwise we could repeatedly compress any data to smaller and smaller sizes, eventually reaching zero. This is an application of the Pigeonhole Principle.

Lossless compression methods must take advantage of known structure in the bitstreams to be compressed. The four methods that we consider exploit, in turn, the following structural characteristics:

- Small alphabets
- Long sequences of identical bits/characters
- Frequently used characters
- Long reused bit/character sequences

Run-Length Encoding

Small Alphabets

An elementary data-compression method is replacing a limited size alphabet with shorter representations.

Genomics Example

ATAGATGCATAGCGCATAGCTAGATGTGCTAGCAT

Using standard ASCII encoding (1 byte), this string is a bitstream of length $8 \times 35 = 280$. Genomes contain only four different characters, so each can be encoded with just 2 bits per character.

```
static final String alphabet = "ACGT";
public static String compress(String DNA)
{
    String compressedDNA = "";
    for (int i = 0; i < DNA.length(); i++) {
        compressedDNA += String.format("%2s",
Integer.toBinaryString(alphabet.indexOf(DNA.charAt(i)))).replace(' ', '0');
    }
    return compressedDNA;
}

public static String expand(String compressedDNA)
{
    String DNA = "";
    for (int i = 0; i < compressedDNA.length(); i+=2) {
        String code = compressedDNA.substring(i, i+2);
        DNA += alphabet.charAt(Integer.parseInt(code, 2));
    }
    return DNA;
}
```

Long Sequences of Identical Bits

The simplest type of redundancy in a bitstream is long runs of repeated bits. For example, consider the following 40-bit string:

$$0000000000000011111110000001111111111$$

This string consists of 15: 0s, then 7: 1s, then 7: 0s, then 11: 1s, so we can encode the bitstring with the numbers 15, 7, 7, and 11. All bitstrings are composed of alternating runs of 0s and 1s (in that order); we just encode the length of the runs. In our example, if we use 4 bits to encode the numbers and start with a run of 0s, we get the 16-bit string

$$1111011101111011 = 0xF77B$$

```java
public static String compressBitstream(String bitstream) {
    StringBuilder compressedBitstream = new StringBuilder();
    int count = 0;
    int pos = 0;
    char bit;
    char lastBit = '0';    // always start with 0s
    while (bitstream.length() > pos) {
        bit = bitstream.charAt(pos);
        if(bit != lastBit) {
            compressedBitstream.append(String.format("%2s", Integer.toHexString(count)).replace(' ', '0'));
            count = 0;
            lastBit = bit;
        }
        else  {
            if (count == 255)
            {
                compressedBitstream.append(String.format("%2s", Integer.toHexString(count)).replace(' ', '0'));
                count = 0;
                compressedBitstream.append(String.format("%2s", Integer.toHexString(count)).replace(' ', '0'));
            }
        }
        count++;
        pos++;
    }
    compressedBitstream.append(String.format("%2s", Integer.toHexString(count)).replace(' ', '0'));
    return compressedBitstream.toString();
}
```

PCX 🕮

PCX image data are compressed using run-length encoding. The two most-significant bits of a byte are used to determine whether the given data represent a single pixel of a given palette index or color value, or an RLE pair representing a series of several pixels of a single value:

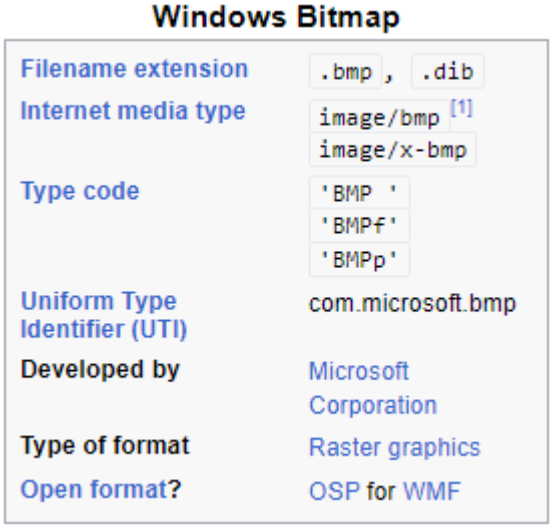
1. if both bits are 1, the byte is interpreted as the run length. This leaves 6 bits for the actual run length value (0 – 63)
2. in any other case, the byte is interpreted as a single pixel value.

PCX

Filename extension	`.pcx`
Internet media type	`image/vnd.zbrush.pcx`, `image/x-pcx` (deprecated)[1]
Developed by	ZSoft Corporation
Initial release	1985; 38 years ago
Latest release	5 1991; 32 years ago
Type of format	Lossless bitmap image format

BMP 8-Bit RLE 🕮

Microsoft BI_RLE8 is an 8-bit run-length bitmap encoding using marker/escape **bytes**. Most RLE schemes work similarly.



Windows Bitmap

Filename extension	`.bmp`, `.dib`
Internet media type	`image/bmp` [1] `image/x-bmp`

Type code	`'BMP '`
	`'BMPf'`
	`'BMPp'`

Uniform Type Identifier (UTI)	com.microsoft.bmp
Developed by	Microsoft Corporation
Type of format	Raster graphics
Open format?	OSP for WMF

- Encoding is either with
  - pairs (**`0x00`** < **`Count`** <= **`0xFF`**), **`Value`**
  - or a value **`0x00`** followed by
    - **`0x00`**, **`0x01`**, or **`0x02`** (escape value)
    - a **`Count`** > **`0x03`** of “absolute” pixel values (uncompressed)

BI_RLE8 is only defined for 8-bit bitmaps.

Advantages of Run-Length Encoding

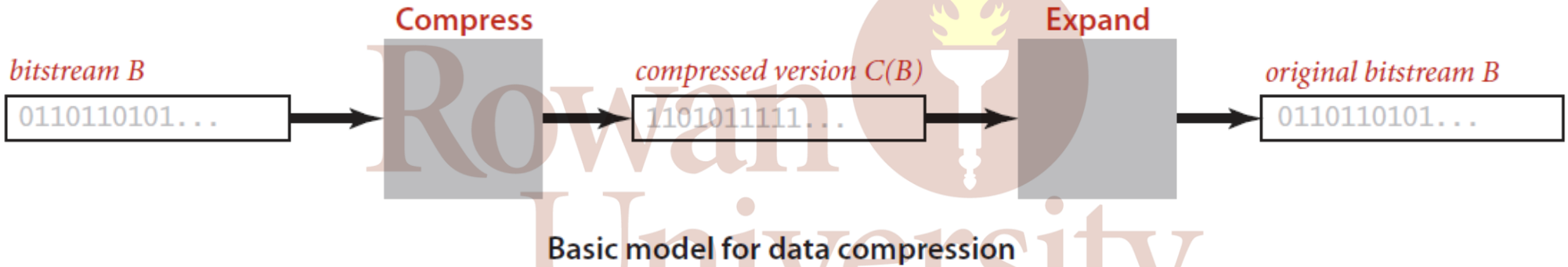
- Simple and fast compression method

Drawbacks of Run-Length Encoding

- The original data is not instantly accessible.
- Need to reserve memory for decoded data, so metadata should be provided with the compressed data.

Huffman Trees and Codes

We will look at an algorithm that compresses data lossless (unlike mp3 or jpeg). All types of data we process are represented in binary. Hence, we only need to think about compressing bitstreams.



Basic model for data compression

No algorithm can compress every bitstream to smaller sizes. Otherwise, we could repeatedly compress any data to smaller and smaller sizes, eventually reaching zero. (This is an application of the Pigeonhole Principle.)

Huffman encoding is a data-compression technique that can save a substantial amount of space in natural language files (and many other kinds of files). The idea is to abandon the way in which text files are usually stored: instead of using the usual 7 or 8 bits for each character, we use fewer bits for characters that appear more often than for those that appear rarely.

Huffman's greedy algorithm uses a table giving the frequency of each character to build up an optimal way of representing (**encoding**) each character as a binary string.

Lossless compression methods must take advantage of known structure in the bitstreams to be compressed. The four methods that we consider exploit, in turn, the following structural characteristics:

- Small alphabets
- Long sequences of identical bits/characters
- Frequently used characters
- Long reused bit/character sequences

Variable-length prefix-free codes

A code associates each character with a bitstring: a key-value symbol table with characters as keys and bitstrings as values. We may assign each symbol its unique binary string (code, encoding, or ***codeword***). Codes can be ***fixed-length***, that is, every symbol has a codeword of the same fixed length. An example for text symbols is the ASCII character encoding.

A ***variable-length*** code is such that symbols may have varying length. A problem that immediately arises is how to distinguish the different codewords of variable length if they are not marked (by metadata, such as length, which adds overhead).

A key insight is that ***delimiters are not needed if no symbol code is the prefix of another***.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters `a`–`f`, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

All ***prefix-free codes*** (sometimes confusingly called prefix codes) are uniquely decodable (without needing any delimiters). Note that fixed-length codes (also called block codes) such as ASCII are also prefix-free.

Creating Prefix-Free Codes

We associate the alphabet's symbols with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1. The codeword of a symbol can then be obtained by recording the labels on the simple path from the root to the symbol's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

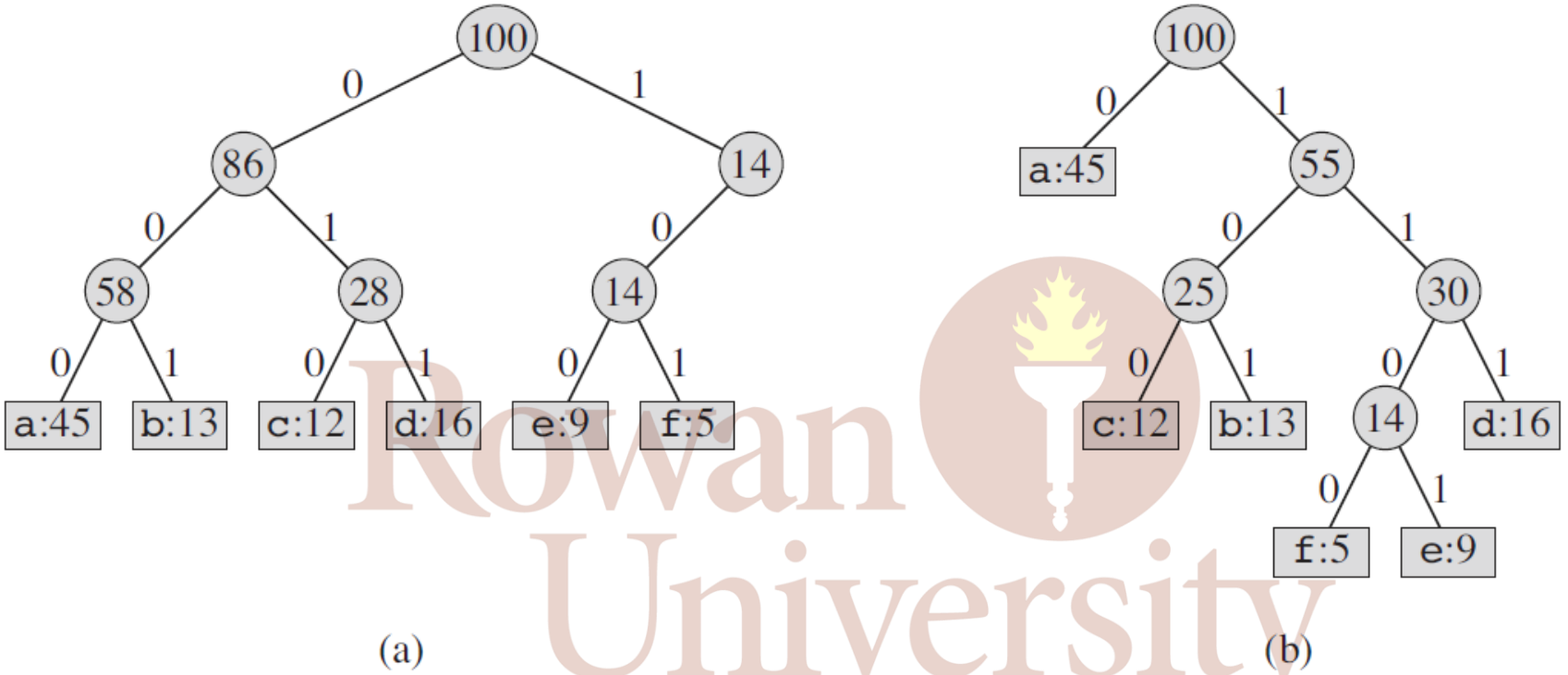


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code `a` = 000, . . . , `f` = 101. **(b)** The tree corresponding to the optimal prefix code `a` = 0, `b` = 101, . . . , `f` = 1100.

Huffman's Algorithm

1. Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's weight.
2. Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       x = EXTRACT-MIN(Q)
6       y = EXTRACT-MIN(Q)
7       z.left = x
8       z.right = y
9       z.freq = x.freq + y.freq
10      INSERT(Q, z)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Example: Consider the five-symbol alphabet $\{A, B, C, D, \_\}$ with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

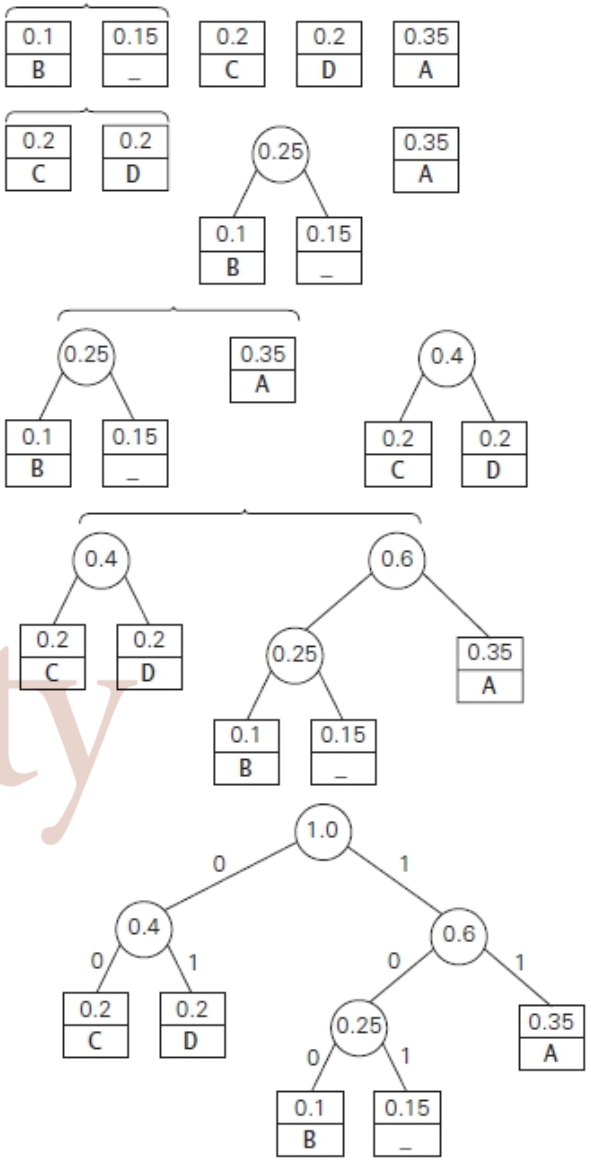
Construct the Huffman tree.

Example: (continued)

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Calculate the expected (average) codeword length using the given occurrence frequencies and actual codeword length.

Calculate the compression ratio when compared to a fixed length code with 3 bits (which are necessary to encode five symbols).



Example: (continued) Decode 1110010011 with the constructed table.