Data Compression

Objectives

After studying this chapter, the student should be able to:

- **Distinguish between lossless and lossy compression.**

- **Describe run-length encoding and how it achieves compression.**

- **Describe Huffman coding and how it achieves compression.**

- **Describe Lempel Ziv encoding and the role of the dictionary in encoding and decoding.**

- **Describe the main idea behind the JPEG standard for compressing still images.**

Data compression implies sending or storing a smaller number of bits. Although many methods are used for this purpose, in general these methods can be divided into two broad categories: **lossless** and **lossy** methods.

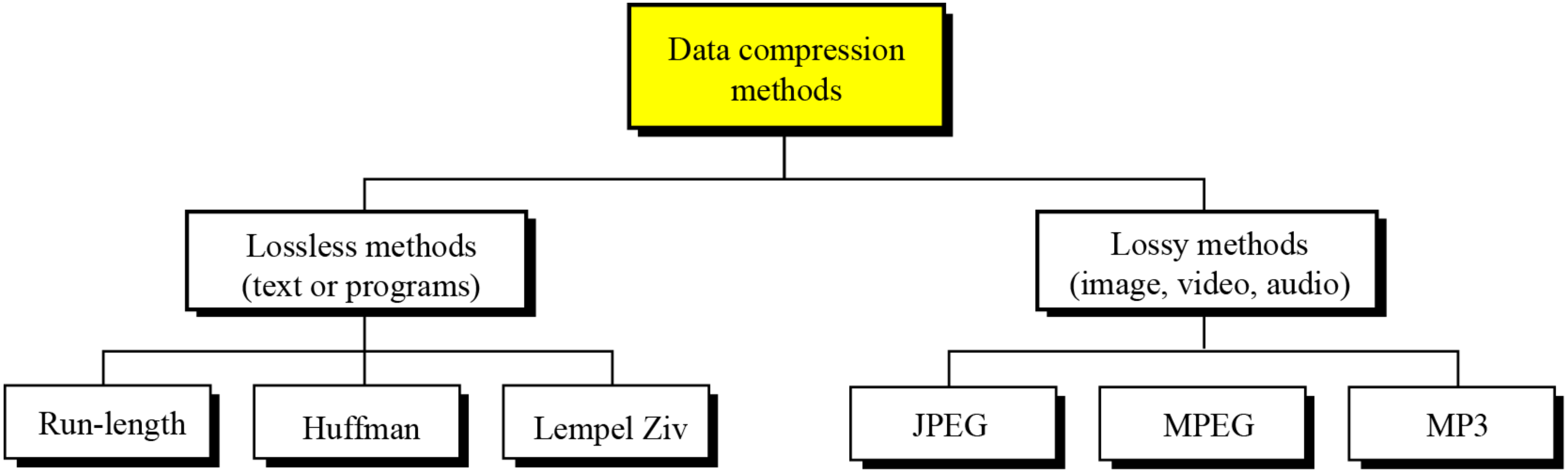

Data compression methods
Lossless methods (text or programs)
Lossy methods (image, video, audio)
Run-length
Huffman
Lempel Ziv
JPEG
MPEG
MP3

Figure 1 Data compression methods

LOSSLESS COMPRESSION

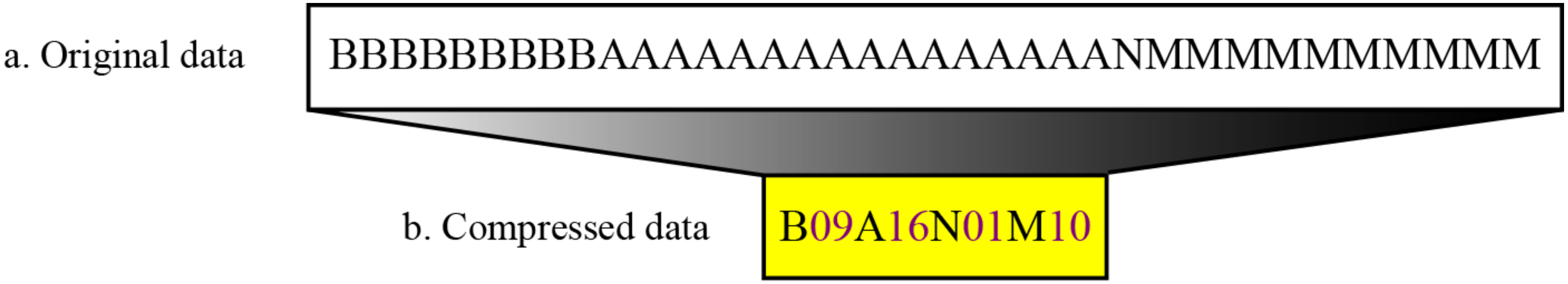
In **lossless** data compression, the integrity of the data is preserved. The original data and the data after compression and decompression are exactly the same because, in these methods, the compression and decompression algorithms are exact inverses of each other: no part of the data is lost in the process. Redundant data is removed in compression and added during decompression. Lossless compression methods are normally used when we cannot afford to lose any data.

Run-length encoding

Run-length encoding is probably the simplest method of compression. It can be used to compress data made of any combination of symbols. It does not need to know the frequency of occurrence of symbols and can be very efficient if data is represented as 0s and 1s.

The general idea behind this method is to replace consecutive repeating occurrences of a symbol by one occurrence of the symbol followed by the number of occurrences.

The method can be even more efficient if the data uses only two symbols (for example 0 and 1) in its bit pattern and one symbol is more frequent than the other.



a. Original data

BBBBBBBBBAAAAAAAAAAAAAAAANMMMMMMMMMM

b. Compressed data

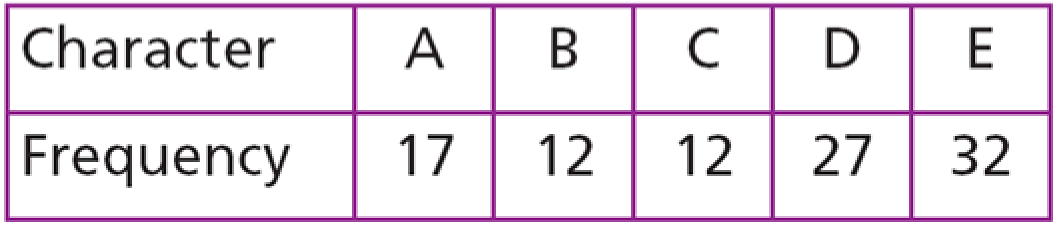
B09A16N01M10

Figure 2 **Run-length encoding example**

Huffman coding

Huffman coding assigns shorter codes to symbols that occur more frequently and longer codes to those that occur less frequently. For example, imagine we have a text file that uses only five characters (A, B, C, D, E). Before we can assign bit patterns to each character, we assign each character a weight based on its frequency of use. In this example, assume that the frequency of the characters is as shown in Table 1.

Table 1 Frequency of characters



Character	A	B	C	D	E
Frequency	17	12	12	27	32

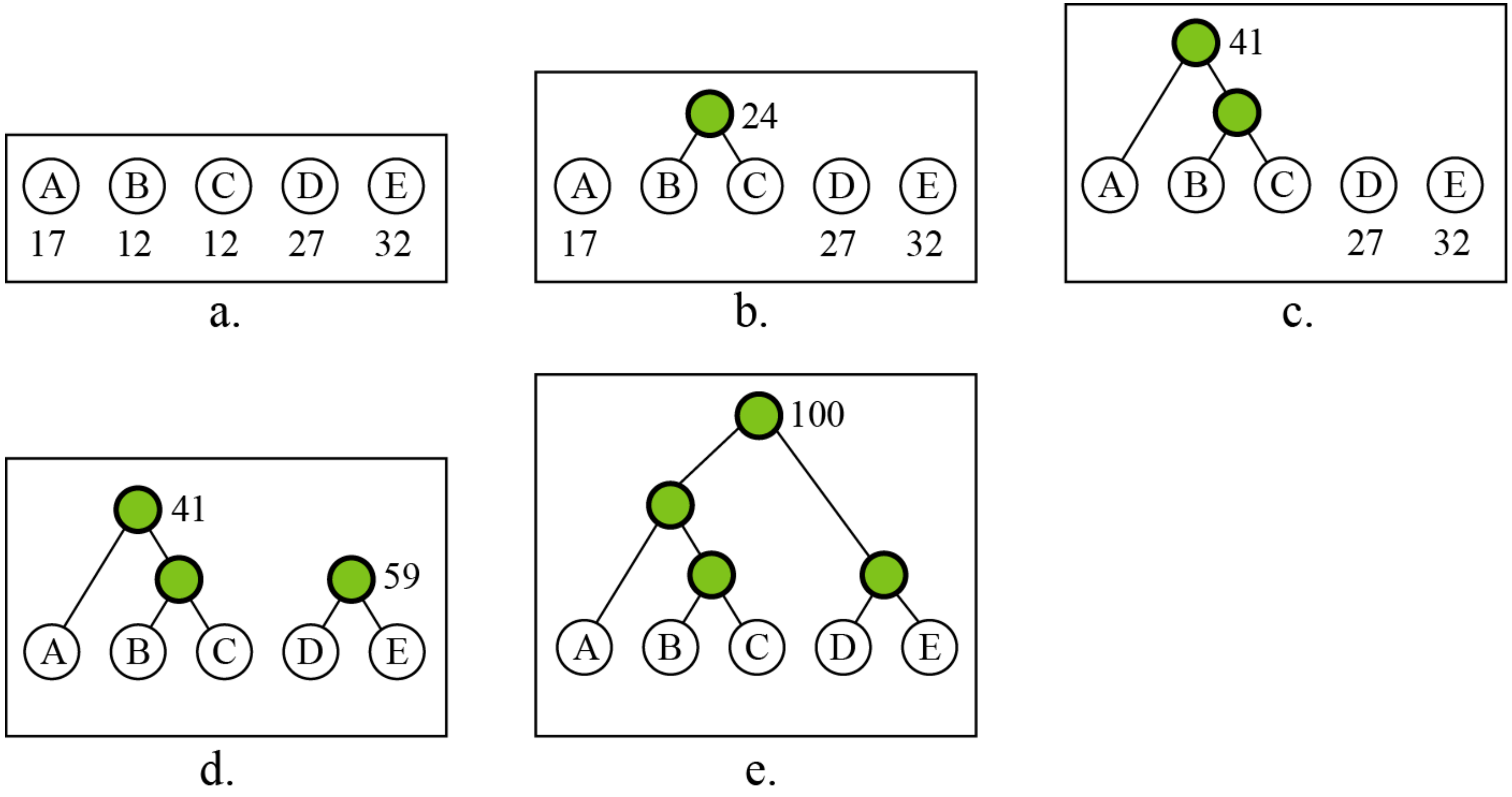
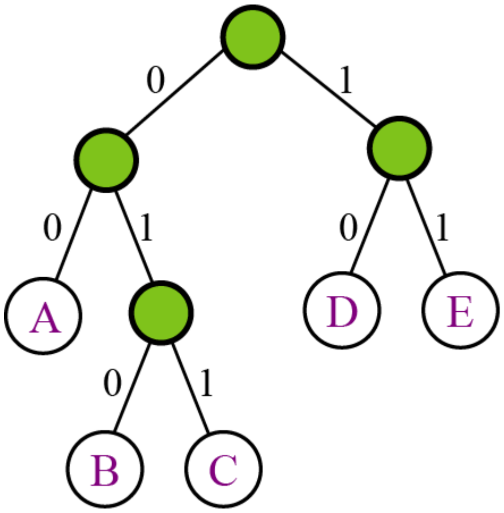


Figure 4 **Huffman coding**

A character’s code is found by starting at the root and following the branches that lead to that character. The code itself is the bit value of each branch on the path, taken in sequence.



A:	00	D:	10
B:	010	E:	11
C:	011		

Code

Figure 5 **Final tree and code**

Encoding

Let us see how to encode text using the code for our five characters. Figure 6 shows the original and the encoded text.

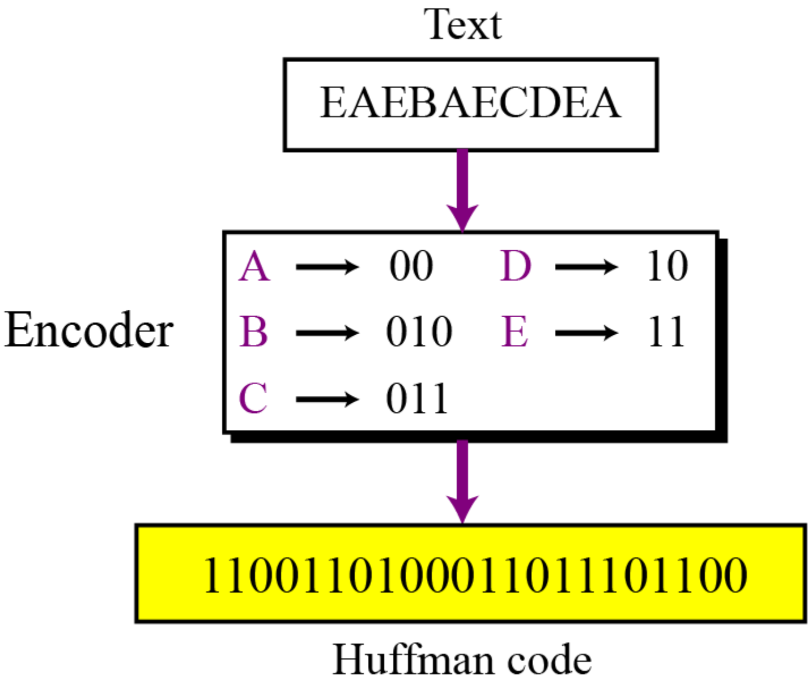


Figure 6 **Huffman encoding**

Decoding

The recipient has a very easy job in decoding the data it receives. Figure 7 shows how decoding takes place.

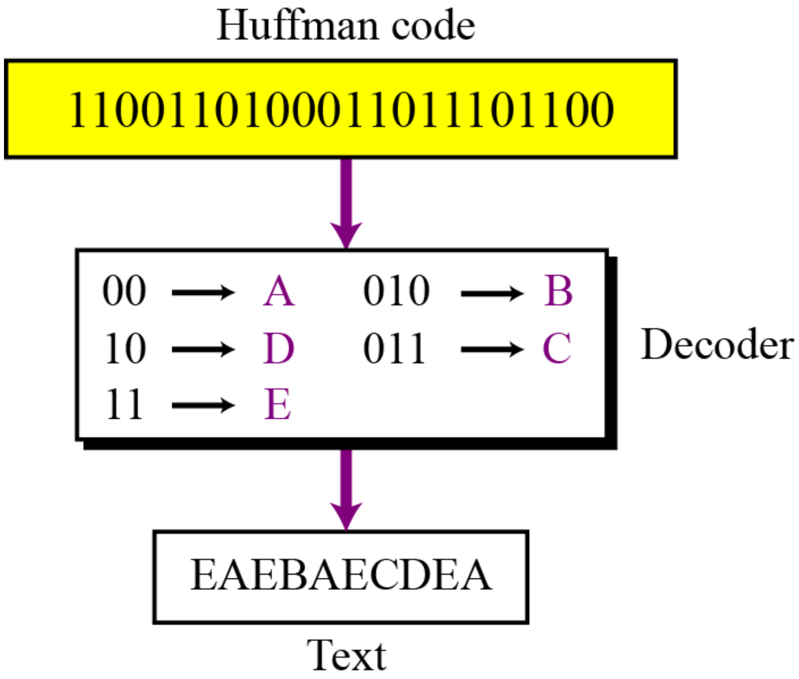


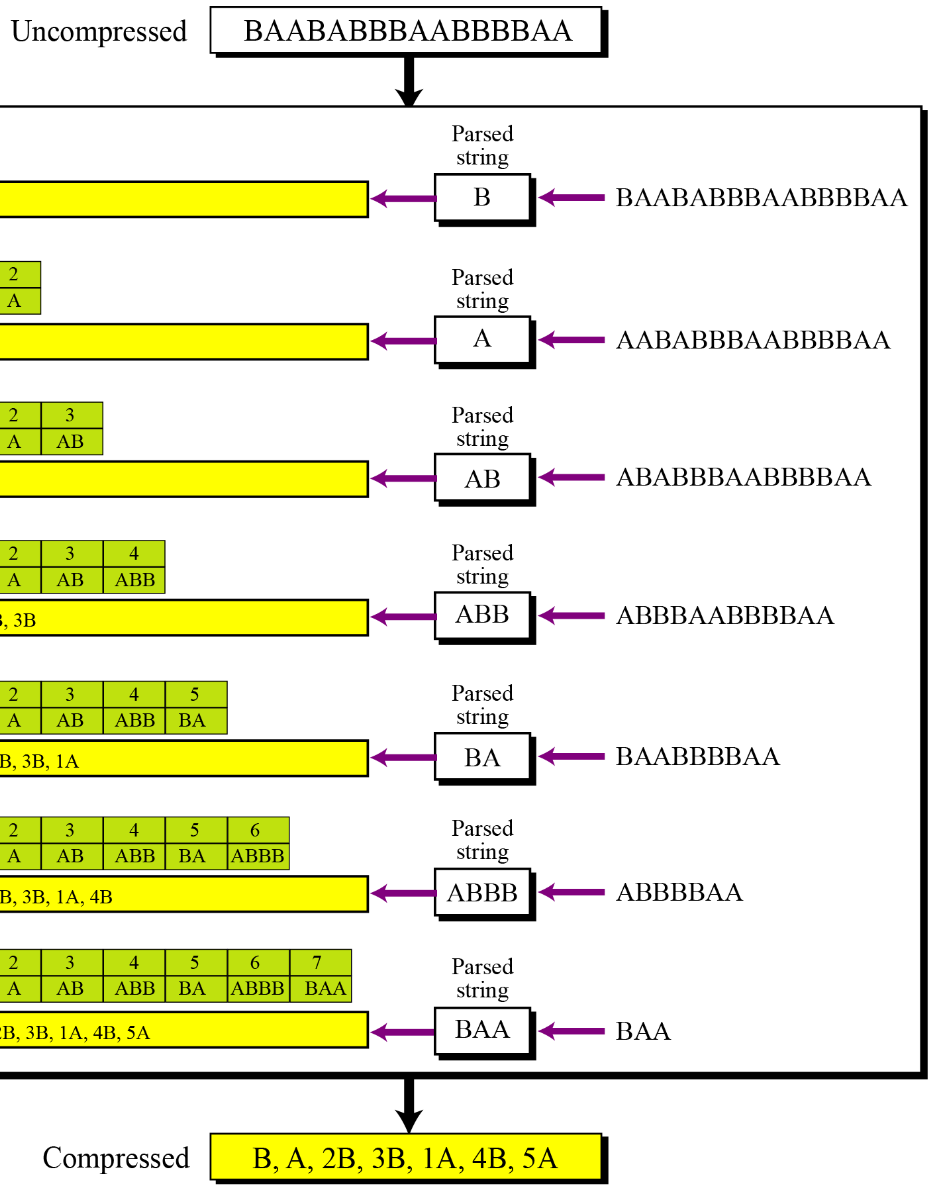
Figure 7 **Huffman decoding**

Lempel Ziv encoding

Lempel Ziv (LZ) encoding is an example of a category of algorithms called *dictionary-based* encoding. The idea is to create a dictionary (a table) of strings used during the communication session. If both the sender and the receiver have a copy of the dictionary, then previously-encountered strings can be substituted by their index in the dictionary to reduce the amount of information transmitted.

Compression

In this phase there are two concurrent events: building an indexed dictionary and compressing a string of symbols. The algorithm extracts the smallest substring that cannot be found in the dictionary from the remaining uncompressed string. It then stores a copy of this substring in the dictionary as a new entry and assigns it an index value. Compression occurs when the substring, except for the last character, is replaced with the index found in the dictionary. The process then inserts the index and the last character of the substring into the compressed string.



Uncompressed BAABABBBAABBBBAA

1
B

B

Parsed string B

BAABABBBAABBBBAA

1	2
B	A

B, A

Parsed string A

AABABBBAABBBBAA

1	2	3
B	A	AB

B, A, 2B

Parsed string AB

ABABBBAABBBBAA

1	2	3	4
B	A	AB	ABB

B, A, 2B, 3B

Parsed string ABB

ABBBAABBBBAA

1	2	3	4	5
B	A	AB	ABB	BA

B, A, 2B, 3B, 1A

Parsed string BA

BAABBBBAA

1	2	3	4	5	6
B	A	AB	ABB	BA	ABBB

B, A, 2B, 3B, 1A, 4B

Parsed string ABBB

ABBBBAA

1	2	3	4	5	6	7
B	A	AB	ABB	BA	ABBB	BAA

B, A, 2B, 3B, 1A, 4B, 5A

Parsed string BAA

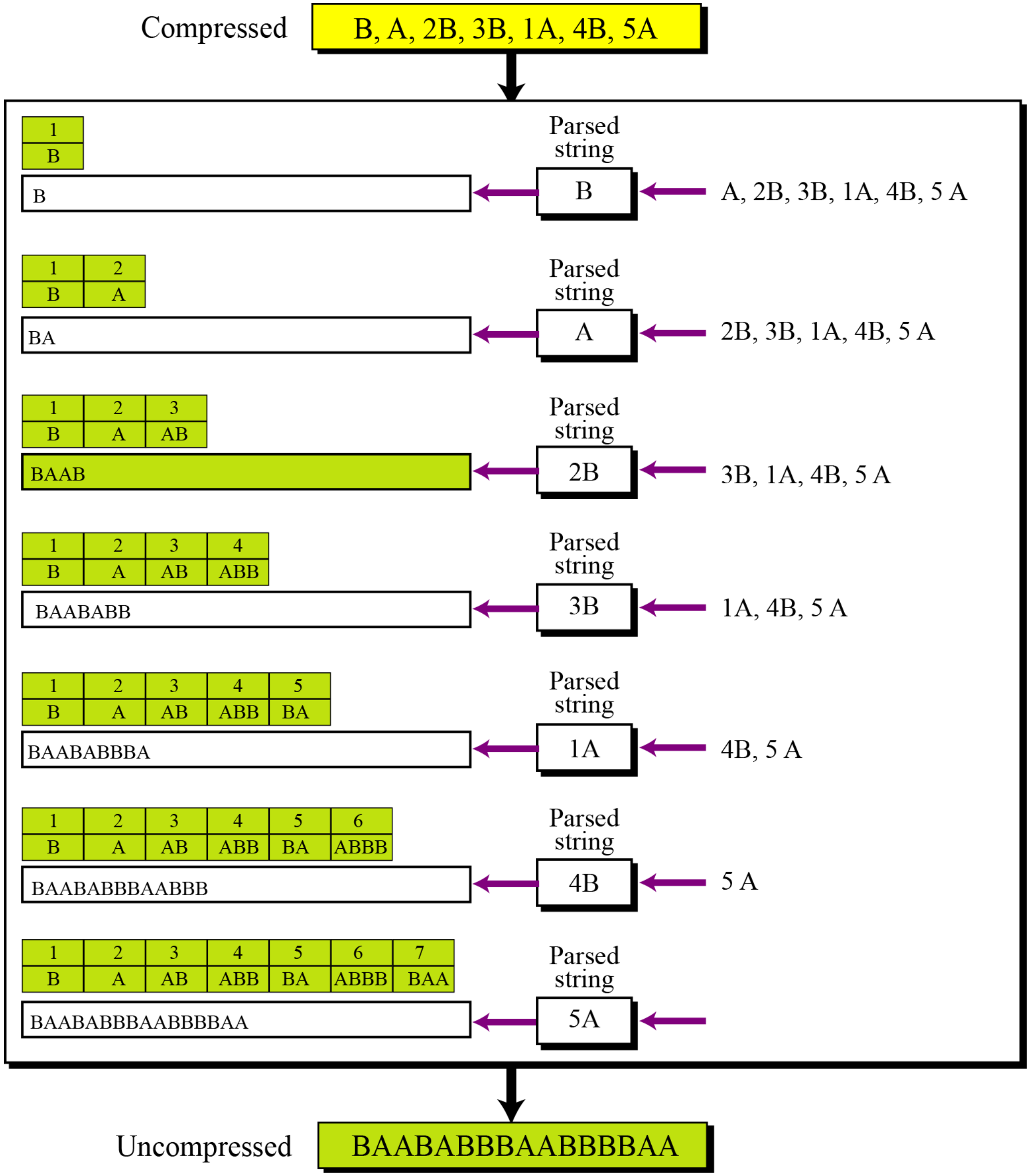
BAA

Compressed B, A, 2B, 3B, 1A, 4B, 5A

Figure 8 **An example of Lempel Ziv encoding**

Decompression

Decompression is the inverse of the compression process. The process extracts the substrings from the compressed string and tries to replace the indexes with the corresponding entry in the dictionary, which is empty at first and built up gradually. The idea is that when an index is received, there is already an entry in the dictionary corresponding to that index.



Compressed B, A, 2B, 3B, 1A, 4B, 5A

1
B

B

Parsed string B ← A, 2B, 3B, 1A, 4B, 5 A

1	2
B	A

BA

Parsed string A ← 2B, 3B, 1A, 4B, 5 A

1	2	3
B	A	AB

BAAB

Parsed string 2B ← 3B, 1A, 4B, 5 A

1	2	3	4
B	A	AB	ABB

BAABABB

Parsed string 3B ← 1A, 4B, 5 A

1	2	3	4	5
B	A	AB	ABB	BA

BAABABBBA

Parsed string 1A ← 4B, 5 A

1	2	3	4	5	6
B	A	AB	ABB	BA	ABBB

BAABABBBAABBB

Parsed string 4B ← 5 A

1	2	3	4	5	6	7
B	A	AB	ABB	BA	ABBB	BAA

BAABABBBAABBBBAA

Parsed string 5A

Uncompressed BAABABBBAABBBBAA

Figure 9 **An example of Lempel Ziv decoding**

LOSSY COMPRESSION METHODS

Our eyes and ears cannot distinguish subtle changes. In such cases, we can use a lossy data compression method. These methods are cheaper—they take less time and space when it comes to sending millions of bits per second for images and video. Several methods have been developed using lossy compression techniques. **JPEG (Joint Photographic Experts Group)** encoding is used to compress pictures and graphics, **MPEG (Moving Picture Experts Group)** encoding is used to compress video, and **MP3 (MPEG audio layer 3)** for audio compression.

Image compression – JPEG encoding

An image can be represented by a two-dimensional array (table) of picture elements (pixels).

In JPEG, a grayscale picture is divided into blocks of 8 × 8 pixel blocks to decrease the number of calculations because, as we will see shortly, the number of mathematical operations for each picture is the square of the number of units.

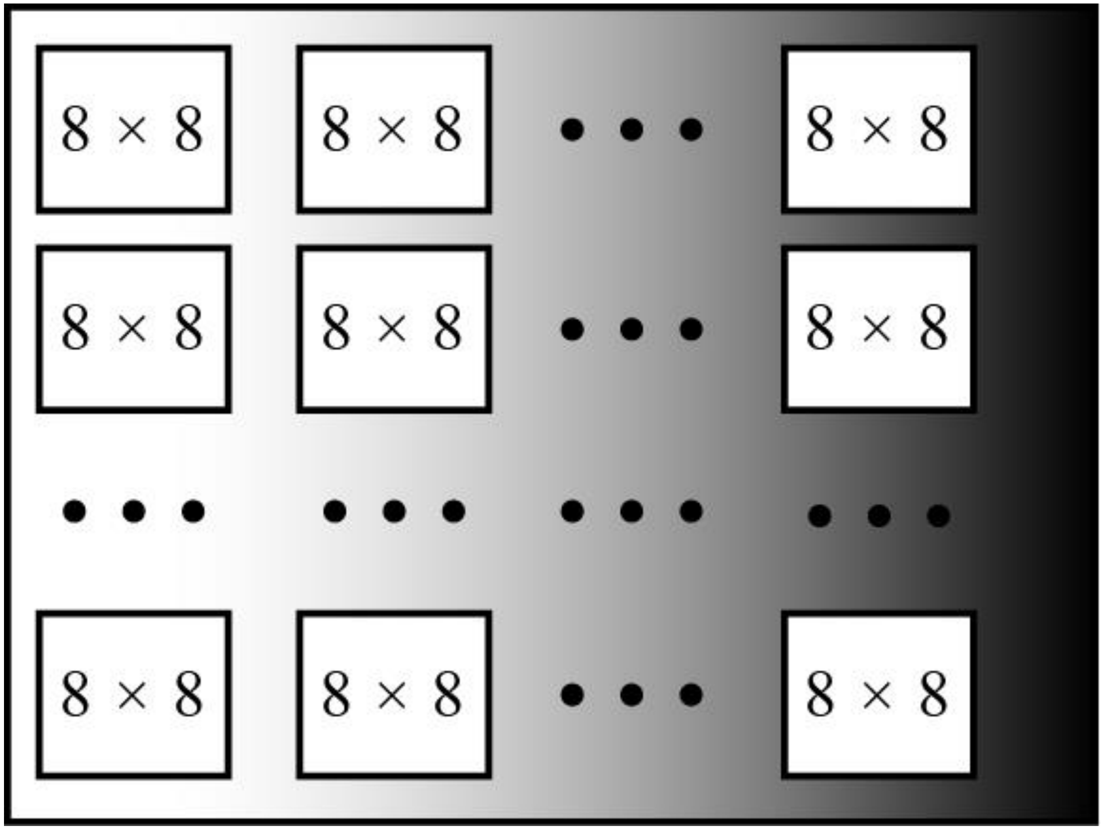


Figure 10 **JPEG grayscale example, 640 × 480 pixels**

The whole idea of JPEG is to change the picture into a linear (vector) set of numbers that reveals the redundancies. The redundancies (lack of changes) can then be removed using one of the lossless compression methods we studied previously. A simplified version of the process is shown in Figure 11.

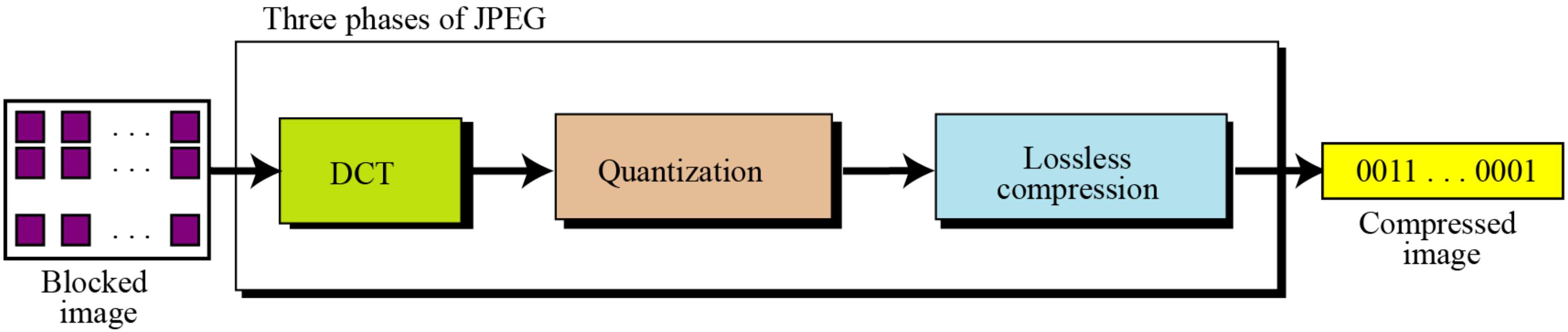
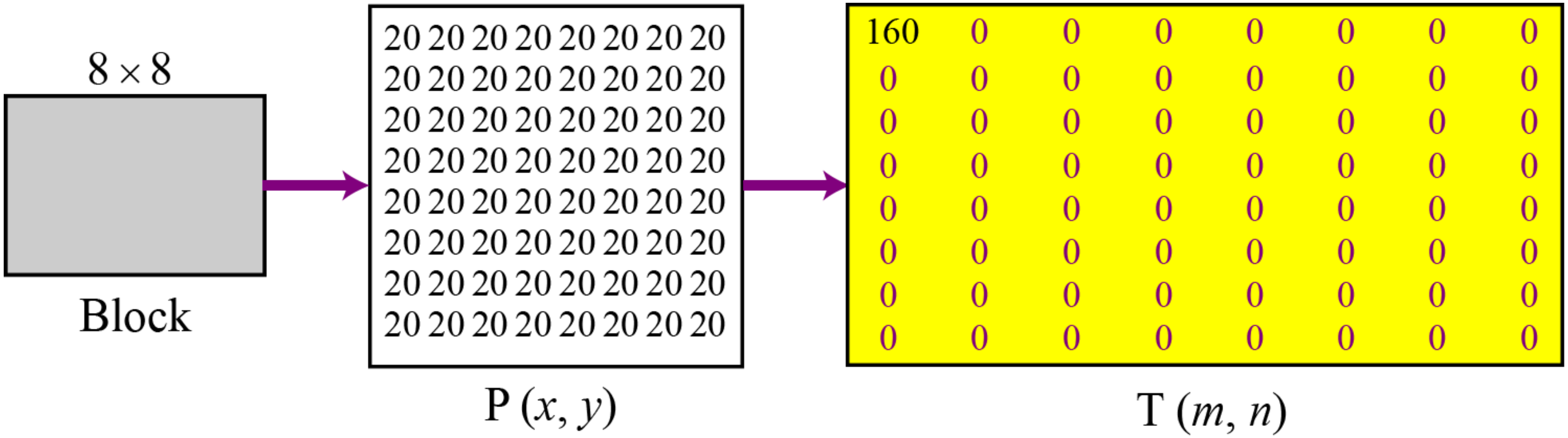


Figure 11 **The JPEG compression process**

Discrete cosine transform (DCT)

In this step, each block of 64 pixels goes through a transformation called the **discrete cosine transform (DCT)**. The transformation changes the 64 values so that the relative relationships between pixels are kept but the redundancies are revealed. The formula is given in Appendix G. P(*x*, *y*) defines one value in the block, while T(*m*, *n*) defines the value in the transformed block.

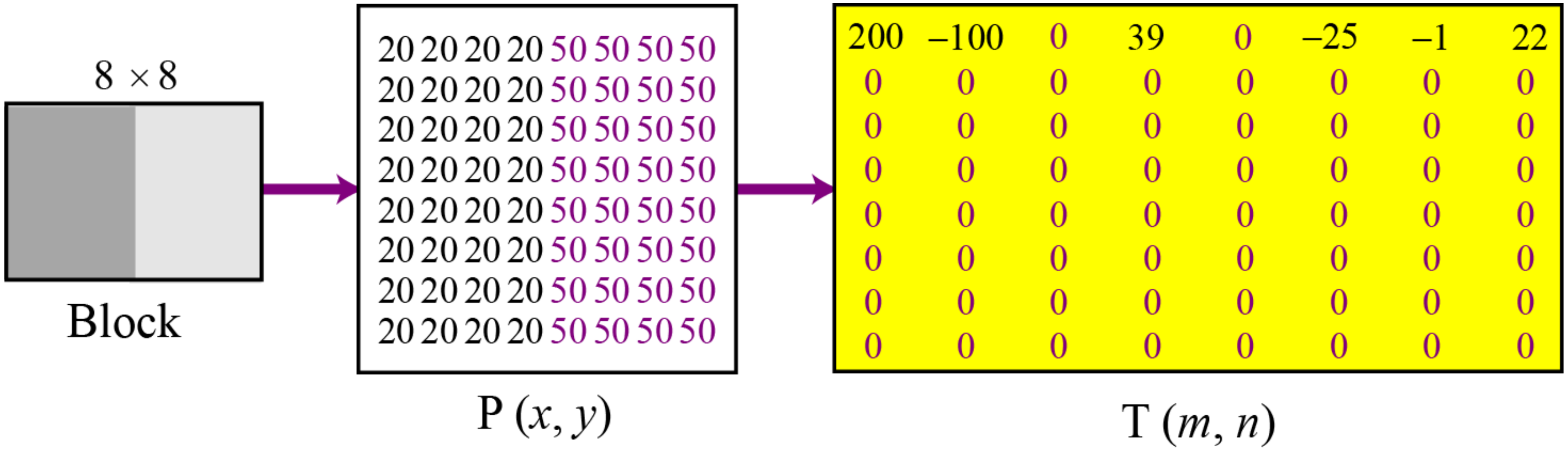
To understand the nature of this transformation, let us show the result of the transformations for three cases.



20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20
20	20	20	20	20	20	20	20

160	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
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 12 **Case 1: uniform grayscale**



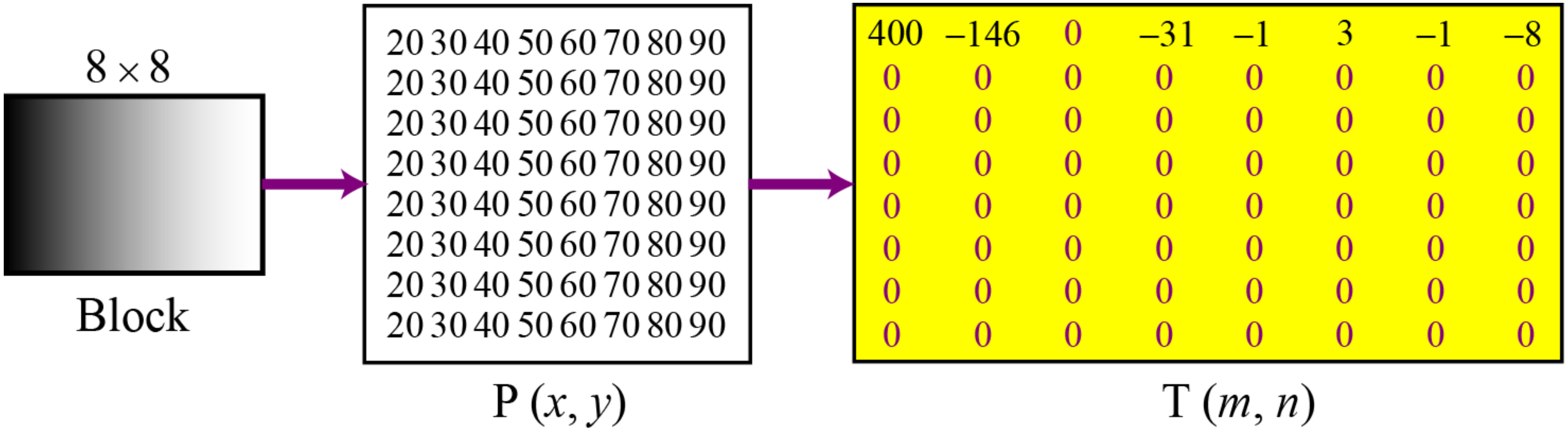
P (*x*, *y*)

20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50
20	20	20	20	50	50	50	50

T (*m*, *n*)

200	–100	0	39	0	–25	–1	22
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	0
0	0	0	0	0	0	0	0

Figure 13 **Case 2: two sections**



8 × 8

Block

20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90
20	30	40	50	60	70	80	90

P (x, y)

400	−146	0	−31	−1	3	−1	−8
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	0
0	0	0	0	0	0	0	0

T (m, n)

Figure 14 **Case 3: gradient grayscale**

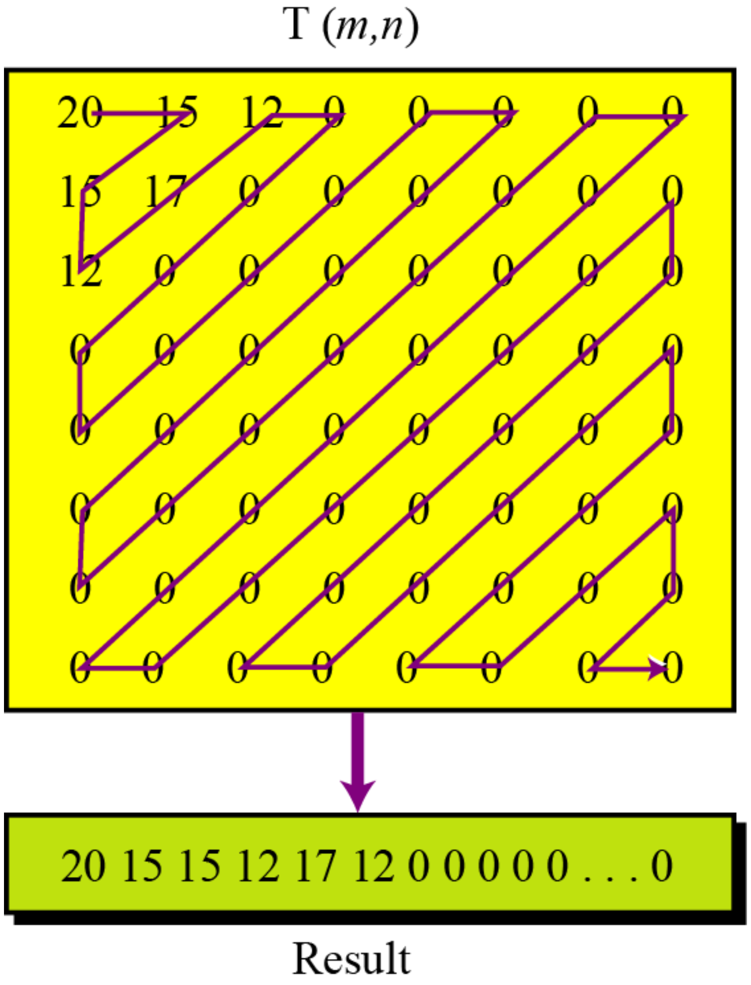
Quantization

After the T table is created, the values are quantized to reduce the number of bits needed for encoding. Quantization divides the number of bits by a constant and then drops the fraction. This reduces the required number of bits even more. In most implementations, a quantizing table (8 by 8) defines how to quantize each value. The divisor depends on the position of the value in the T table. This is done to optimize the number of bits and the number of 0s for each particular application.

Compression

After quantization the values are read from the table, and redundant 0s are removed. However, to cluster the 0s together, the process reads the table diagonally in a zigzag fashion rather than row by row or column by column. The reason is that if the picture does not have fine changes, the bottom right corner of the T table is all 0s.

JPEG usually uses run-length encoding at the compression phase to compress the bit pattern resulting from the zigzag linearization.



T (*m*,*n*)

20	15	12	0	0	0	0	0
15	17	0	0	0	0	0	0
12	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

20 15 15 12 17 12 0 0 0 0 0 . . . 0

Result

Figure 15 **Reading the table**