A book cover with text and purple text

Description automatically generated

**Khalid Sayood's textbook-style**

Introduction to Data Compression is the definitive guide to all kinds of compression schemes.

establish the mathematics involved in basic compression techniques, including lossless and lossy compression as well as the fundamentals of information theory that lay the groundwork for common forms of compression. (The book contains all the relevant formulas, although those who don't need such mathematical detail will still be able to understand the book.)

The Huffman method is simple, efficient, and produces the best codes for the individual data symbols. However, the only case where it produces ideal variable-size codes (codes whose average size equals the entropy) is when the symbols have probabilities of occurrence that are negative powers of 2 (i.e., numbers such as 1/2, 1/4, or 1/8). This is because the Huffman method assigns a code with an integral number of bits to each symbol in the alphabet. Information theory shows that a symbol with probability 0.4 should ideally be assigned a 1.32-bit code,

since −log2 0.4 ≈ 1.32.

The Huffman method, however, normally assigns such a symbol a code of 1 or 2 bits. Arithmetic coding overcomes the problem of assigning integer codes to the individual symbols by assigning one (normally long) code to the entire input file. The method starts with a certain interval, it reads the input file symbol by symbol and uses the probability of each symbol to narrow the interval. Specifying a narrower interval requires more bits, so the number constructed by the algorithm grows continuously. To achieve compression, the algorithm is designed such that a high-probability symbol narrows the interval less than a low-probability symbol, with the result that high-probability symbols contribute fewer bits to the output.

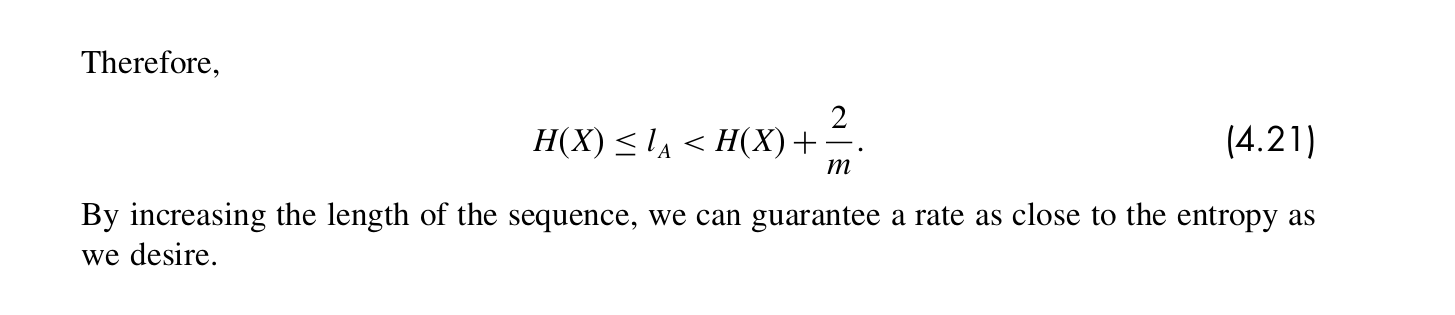
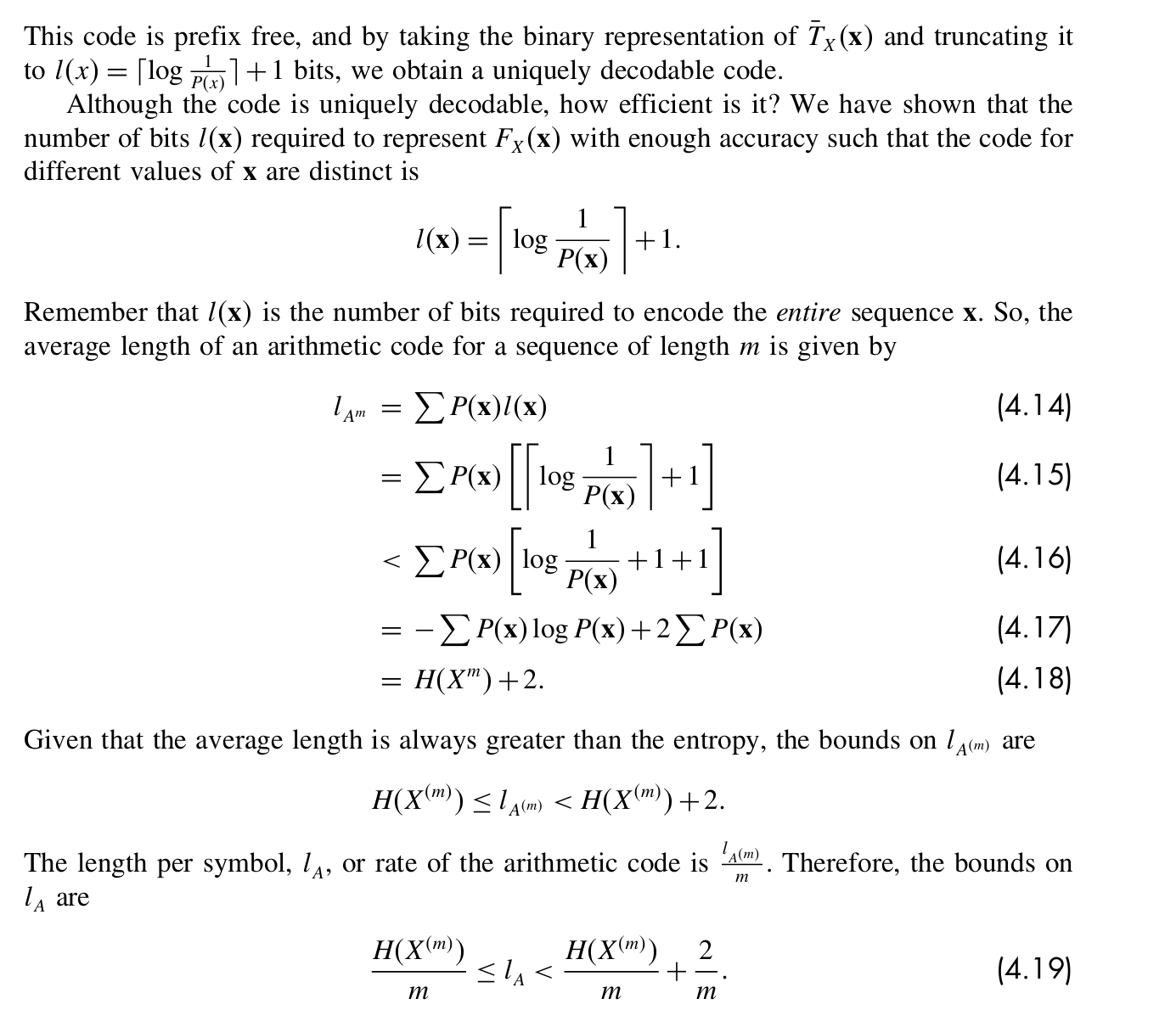
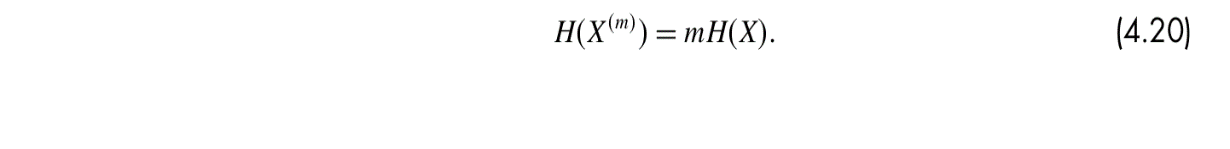
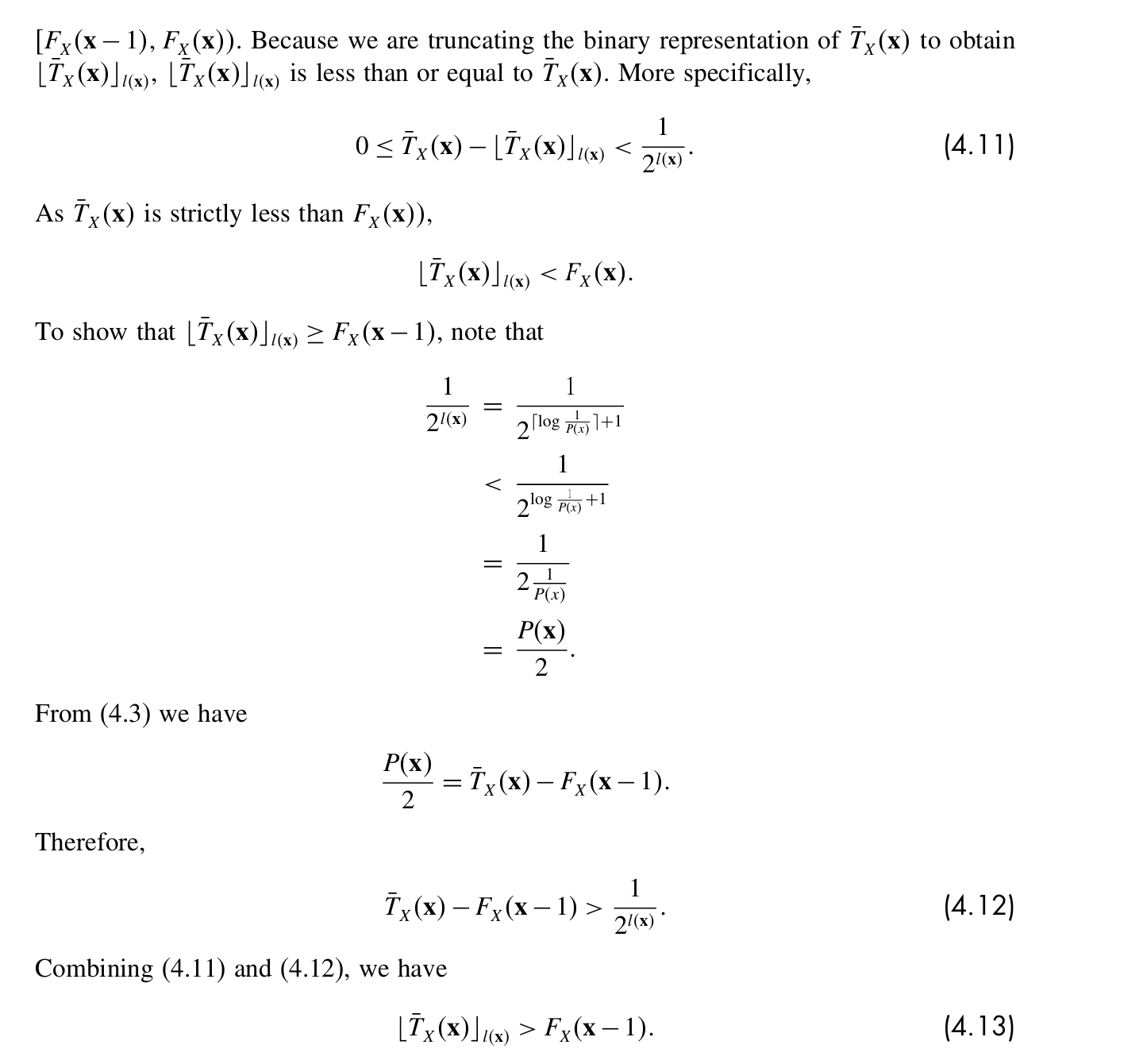
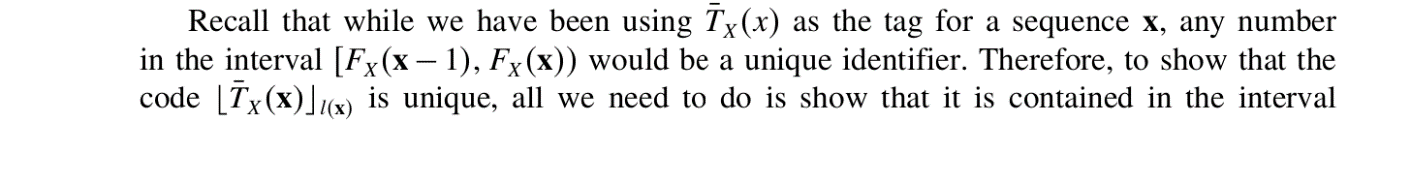
A diagram of a graph

Description automatically generated with medium confidence

We want to find a binary code that will represent the sequence x uniquely and efficiently. We have said that the tag forms a unique representation of the sequence. This means that the binary representation of the tag forms a unique binary code for the sequence. However, we have placed no restrictions on what values in the unit interval the tag can take. The binary 4.4 Generating a Binary Code 93 representation of some of these values would be infinitely long, in which case, although the code is unique, it may not be efficient. To make the code efficient, the binary representation has to be truncated. But if we truncate the representation, is the resulting code still unique? Finally, is the resulting code efficient? How far or how close is the average number of bits per symbol from the entropy?

Length of the tag

is a number in the interval [0,1). A binary code can be obtained by taking the binary representation of this number and truncating it to



This code implements an arithmetic encoder, along with a utility function to convert floating-point numbers to binary bit sequences. Let's break down the functionality and purpose of each part of the code:

1. tag\_to\_bits Function:

* This function takes a floating-point number (`tag`) and a precision (`precision`) as input.
* It converts the floating-point number into a binary bit sequence with the specified precision.
* Precision determines the number of bits used to represent the tag.
* The function returns a binary string representing the tag.

2. ArithmeticEncoder Class:

* This class implements an arithmetic encoder for lossless data compression.
* The constructor `\_\_init\_\_` initializes the encoder with the input sequence to be encoded.
* The `character\_probability` method calculates the probability of each character in the input sequence.
* The `arithmetic\_encoding` method performs the arithmetic encoding process:
* It calculates cumulative probabilities for each symbol in the sequence.
* It iterates through the sequence and narrows down the encoding interval based on the cumulative probabilities.
* During this process, it prints the upper and lower bounds of the interval.
* The `calculate\_metrics` method computes various metrics related to the encoding process:
* It calculates the encoded range, encoded tag, initial and final number of bits, compression ratio, entropy, average length, and efficiency.
* The encoded tag is the midpoint of the final encoding interval.
* The compression ratio represents the ratio of initial bits to final bits.
* The entropy measures the average information content of the source data.
* The average length represents the average number of bits per symbol needed to encode the sequence.
* The efficiency measures the compression efficiency as a percentage of the entropy.

The decimal module in Python is used for arbitrary-precision arithmetic. It provides support for decimal floating-point arithmetic, which is particularly useful when precision is critical, such as in financial applications or when dealing with extremely small or large numbers.

The Decimal module is utilized to ensure accurate precision during arithmetic operations, especially in scenarios where traditional floating-point arithmetic may lead to rounding errors. Here's how it is used:

1. Setting Precision:

The getcontext().prec method is called to set the precision to a high value (50 in this case). This sets the number of decimal places to be used in arithmetic operations involving Decimal objects.

1. Arithmetic Operations:

Throughout the code, Decimal objects are used instead of regular floating-point numbers for arithmetic calculations.

For example, in the tag\_to\_bits function, Decimal is used for calculating the midpoint of the interval to ensure an accurate representation of the tag.