**TEXT COMPRESSION ALGORITHMS**

Submitted By:

SAKSHI TIWARI, 20103216

YASH MANGLA, 20103231

KHUSHI RANA, 20103222

Submitted To:

DR. SHERRY GARG



ACKNOWLEDGEMENT

We are heartily thankful to our guide, DR. SHERRY GARG, whose encouragement, guidance and support enabled us to develop an understanding of the subject and made it possible for us to complete this project. We offer our regards to all the respected professors that have taught us since their contribution towards our knowledge has been immense enabling us to work on this project.

REGARDS

CONTENT

1.ACKNOWLEDGEMENT

2.ABSTRACT

3. INTRODUCTION TO HUFFMANN CODING

4. IMPLEMENTATION OF HUFFMAN CODING

5. INTRODUCTION TO SHANON FANO ALGORITHM.

6. IMPLEMENTATION OF SHANON FANO ALGORITHM

7. INTRODUCTION TO LZW ALGORITHM.

8. IMPLEMENTATION OF LZW ALGORITHM

ABSTRACT

Data Compression is an area that needs to be given almost attention is text quality assessment. Different methodologies have been defined for this purpose. Hence choosing the best machine learning algorithm is really important. In addition to different compression technologies and methodologies, selection of a good data compression tool is most important. There is a complete range of different data compression techniques available both online and offline working such that it becomes really difficult to choose which technique serves the best. Here comes the necessity of choosing the right method for text compression purposes and hence an algorithm that can reveal the best tool among the given ones. A data compression algorithm is to be developed which consumes less time while provides more compression ratio as compared to existing techniques. In this paper we represent a hybrid approach to compress the text data. This hybrid approach is combination of Dynamic Bit reduction method and Huffman coding,shanon fano and LZW algorithm.

INTRODUCTION TO HUFFMANN CODING

Let us suppose, we need to store a string of length 1000 that comprises characters a, e, n, and z. To storing it as 1-byte characters will require 1000 byte (or 8000 bits) of space. If the symbols in the string are encoded as (00=a, 01=e, 10=n, 11=z), then the 1000 symbols can be stored in 2000 bits saving 6000 bits of memory.

The number of occurrence of a symbol in a string is called its frequency. When there is considerable difference in the frequencies of different symbols in a string, variable length codes can be assigned to the symbols based on their relative frequencies. The most common characters can be represented using shorter codes than are used for less common source symbols. More is the variation in the relative frequencies of symbols, it is more advantageous to use variable length codes for reducing the size of coded string.

Since the codes are of variable length, it is necessary that no code is a prefix of another so that the codes can be properly decode. Such codes are called [prefix code](http://en.wikipedia.org/wiki/Prefix_code) (sometimes called "prefix-free codes", that is, the code representing some particular symbol is never a prefix of the code representing any other symbol). Huffman coding is so much widely used for creating prefix codes that the term "Huffman code" is sometimes used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Huffman was able to design the most efficient compression method of this type: no other mapping of individual source symbols to unique strings of bits(i.e. codes) will require lesser space for storing a piece of text when the actual symbol frequencies agree with those used to create the code.

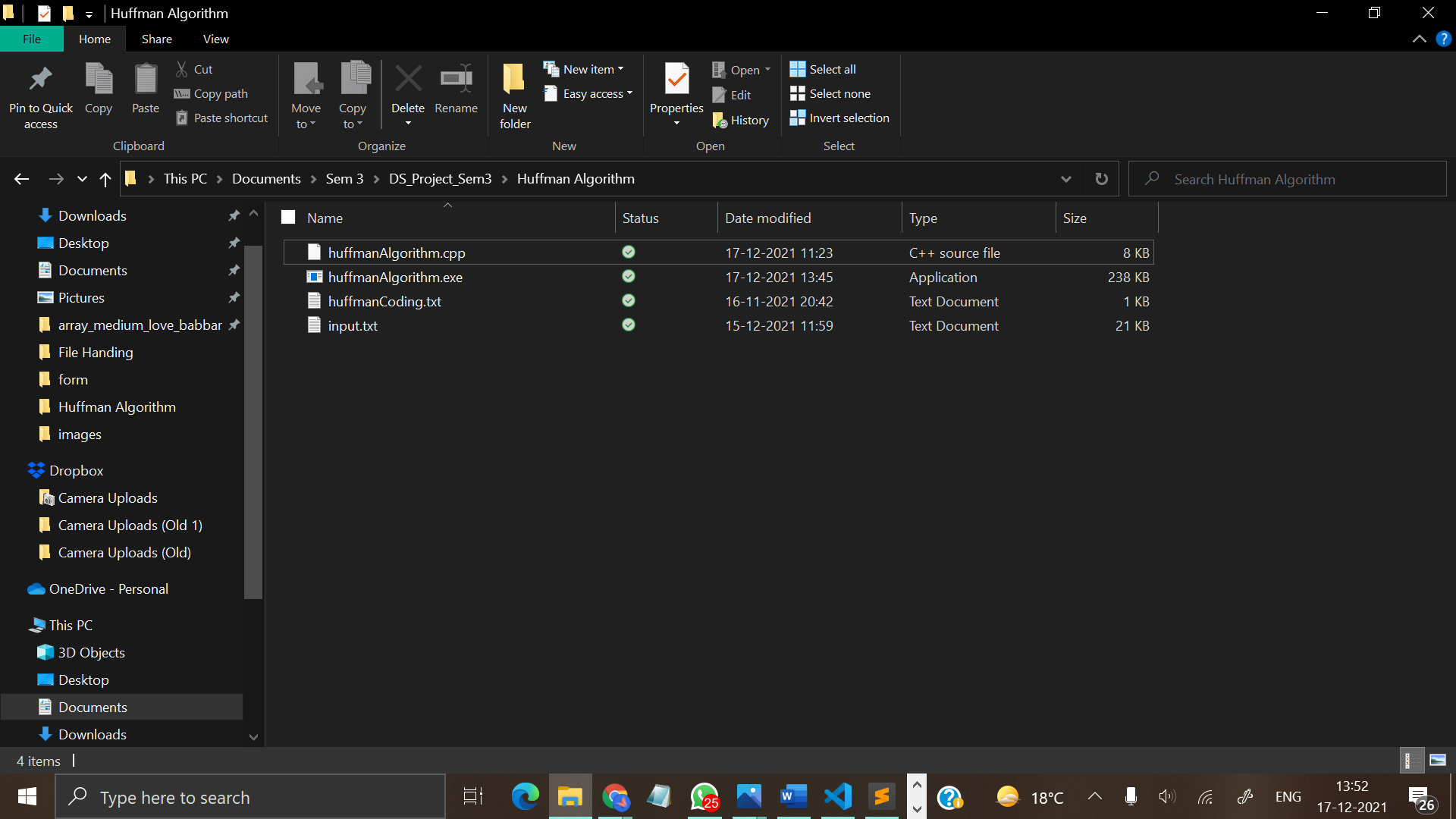
BASIC TECHNIQUE

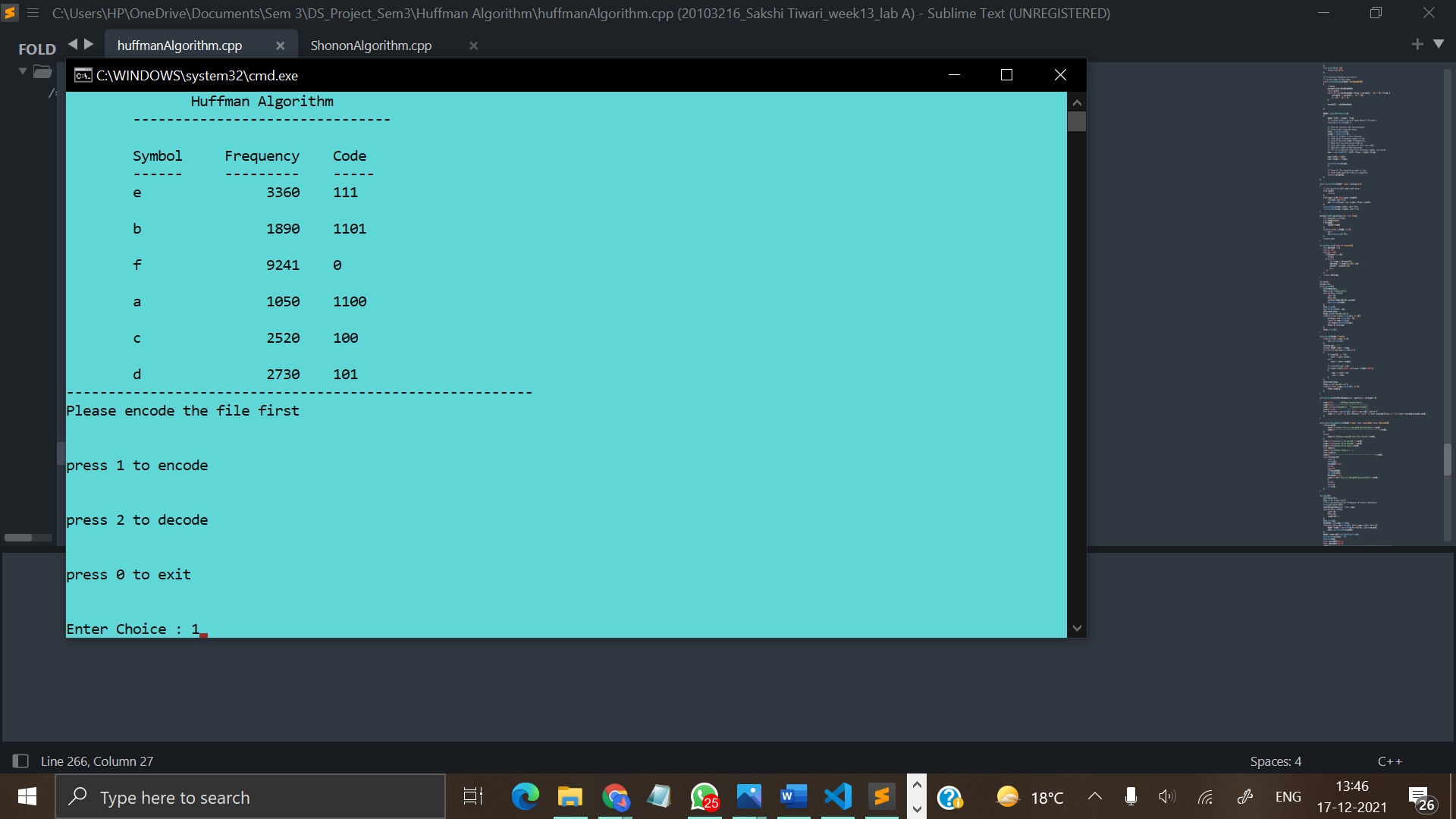
In Huffman Coding , the complete set of codes can be represented as a binary tree, known as a **Huffman tree**. This Huffman tree is also a **coding tree** i.e. a full binary tree in which each [leaf](http://www.itl.nist.gov/div897/sqg/dads/HTML/leaf.html) is an encoded symbol and the [path](http://www.itl.nist.gov/div897/sqg/dads/HTML/path.html) from the [root](http://www.itl.nist.gov/div897/sqg/dads/HTML/root.html) to a leaf is its codeword. By convention, bit '0' represents following the left child and bit '1' represents following the right child. One code bit represents each level. Thus more frequent characters are near the [root](http://www.itl.nist.gov/div897/sqg/dads/HTML/root.html) and are coded with few bits, and rare characters are far from the root and are coded with many bits.

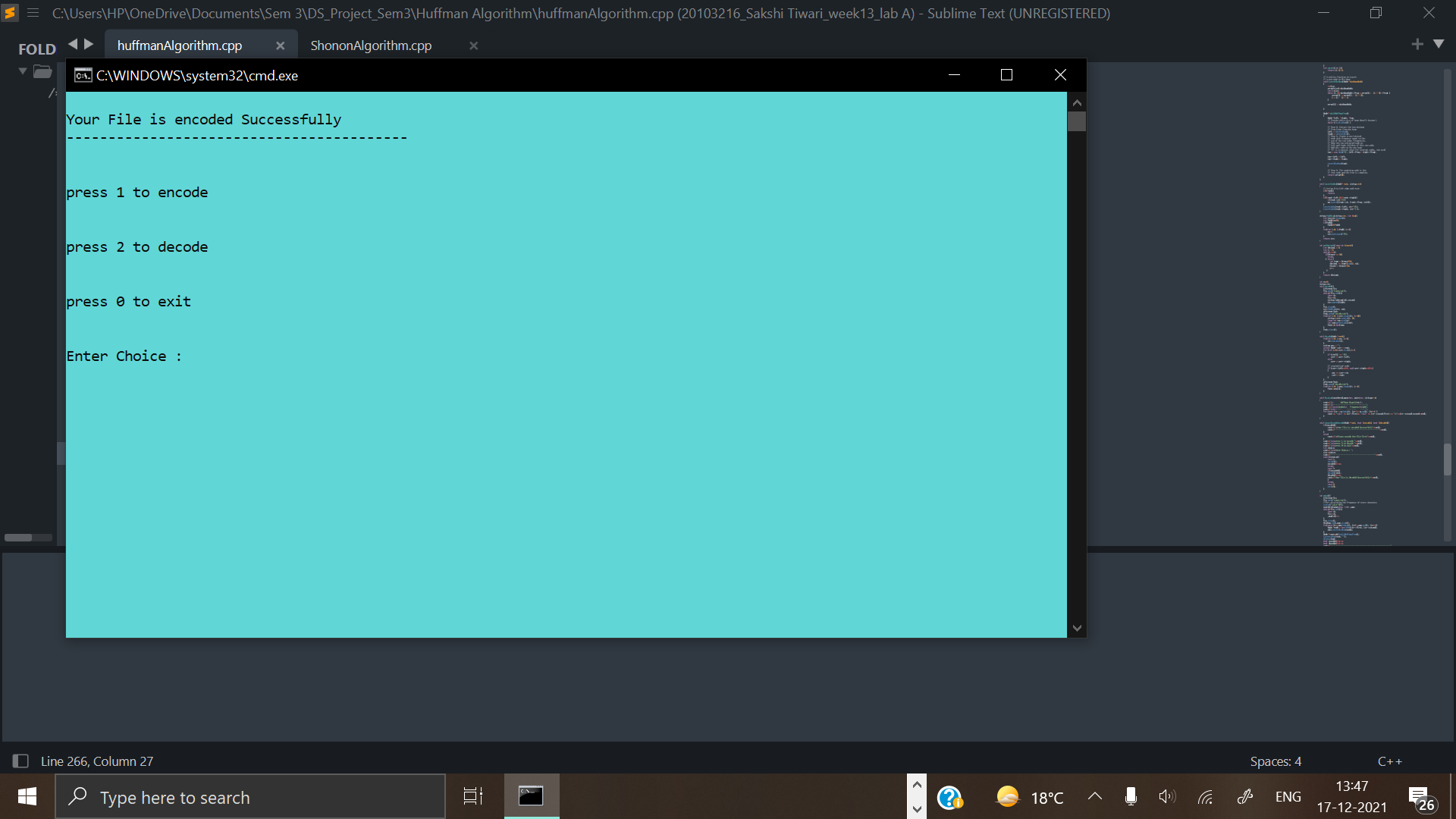
First of all, the source symbols along with their frequencies of occurrence are stored as leaf nodes in a regular array, the size of which depends on the number of symbols, n. A finished tree has up to n leaf nodes and n − 1 internal nodes.

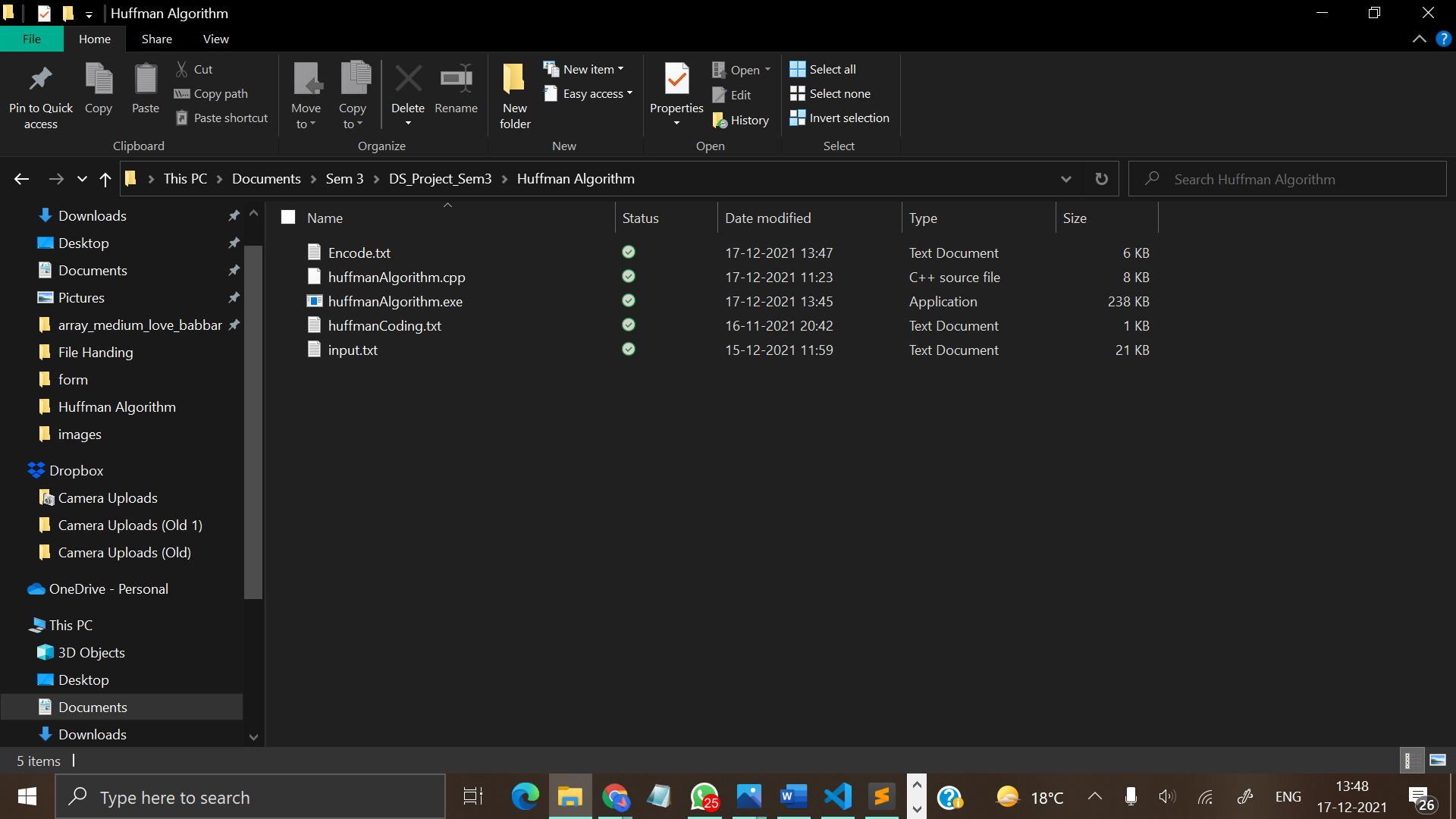
IMPLEMENTATION OF HUFFMAN CODING

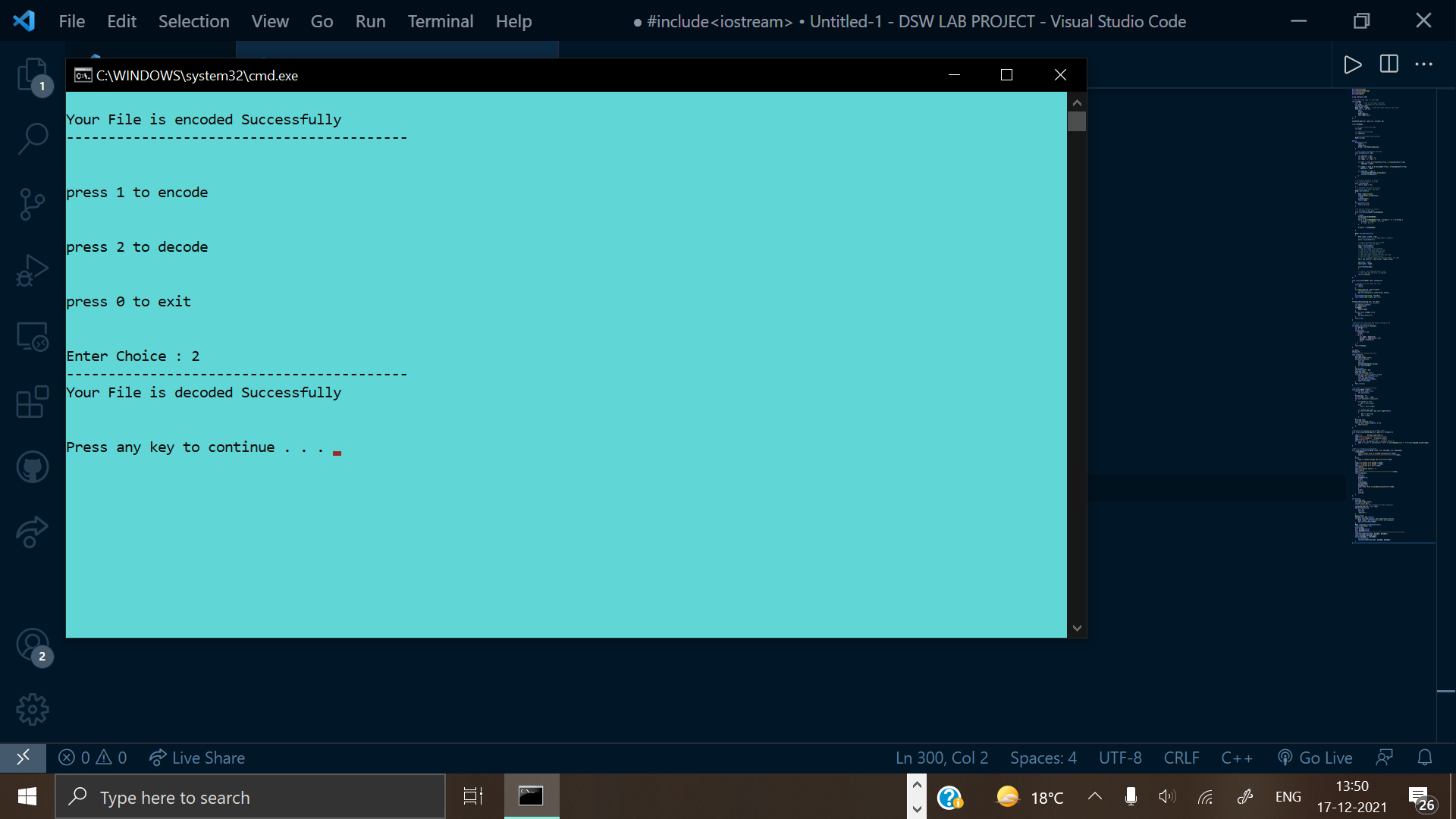
**Screenshots of output:**

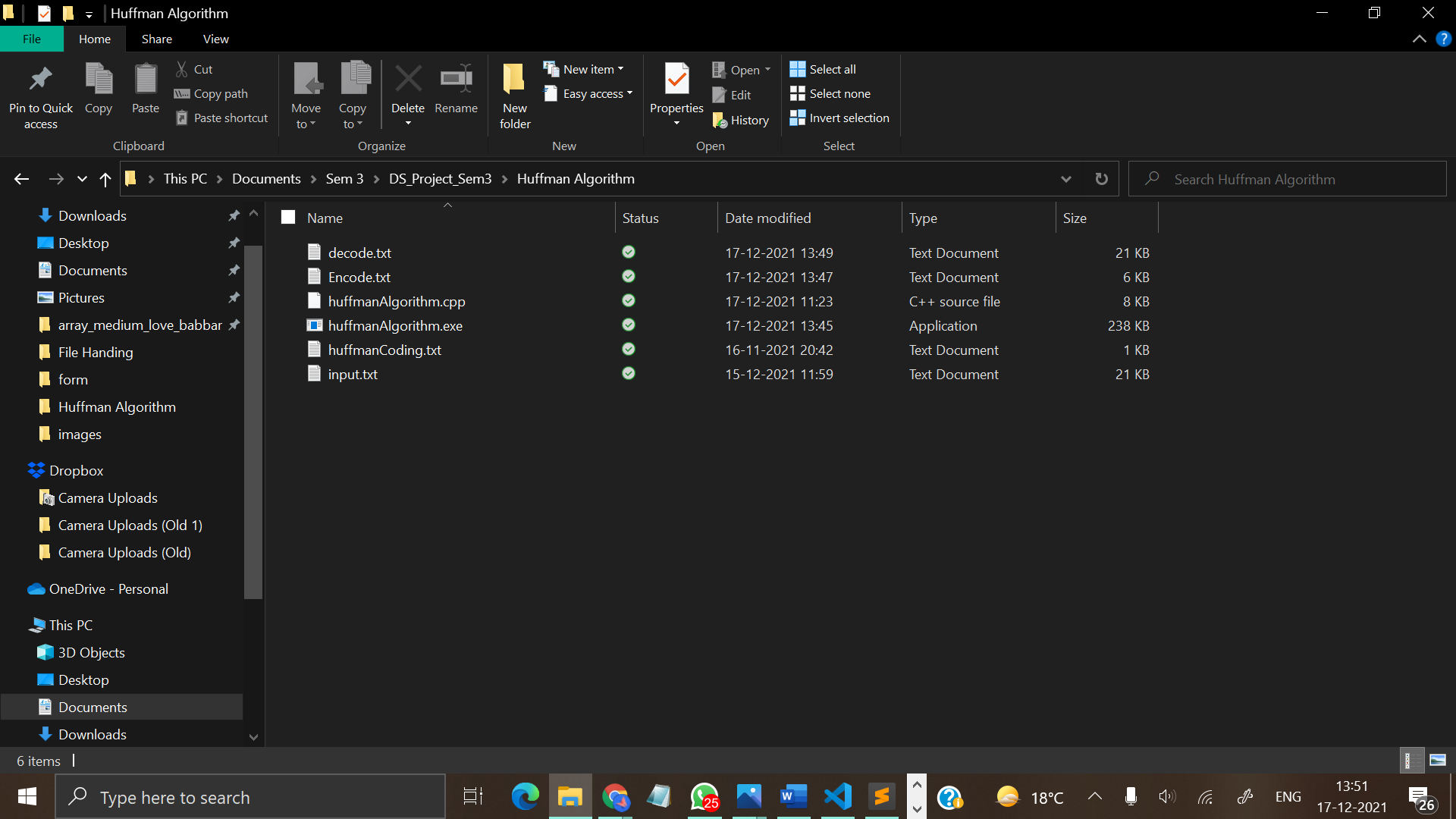












INTRODUCTION TO SHANON FANO

Shannon–Fano coding, named after Claude Elwood Shannon and Robert Fano, is a technique for constructing a prefix code based on a set of symbols and their probabilities. It is suboptimal in the sense that it does not achieve the lowest possible expected codeword length like Huffman coding; however unlike Huffman coding, it does guarantee that all codeword lengths are within one bit of their theoretical ideal I(x) = − log P(x). In Shannon–Fano coding, the symbols are arranged in order from most probable to least probable, and then divided into two sets whose total probabilities are as close as possible to being equal. All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes. When a set has been reduced to one symbol, of course, this means the symbol's code is complete and will not form the prefix of any other symbol's code. The algorithm works, and it produces fairly efficient variable-length encodings; when the two smaller sets produced by a partitioning are in fact of equal probability, the one bit of information used to distinguish them is used most efficiently.

BASIC TECHNIQUE

A Shannon–Fano tree is built according to a specification designed to define an effective code table.

The actual algorithm is simple:

1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol’s relative

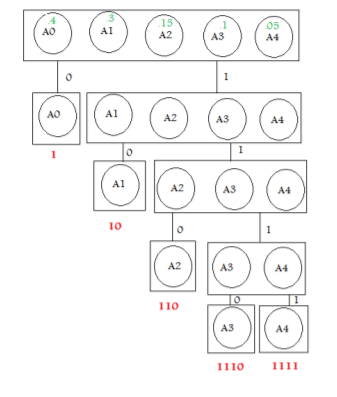
frequency of occurrence is known.

1. Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the left and the least common at the right.

3. Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible.

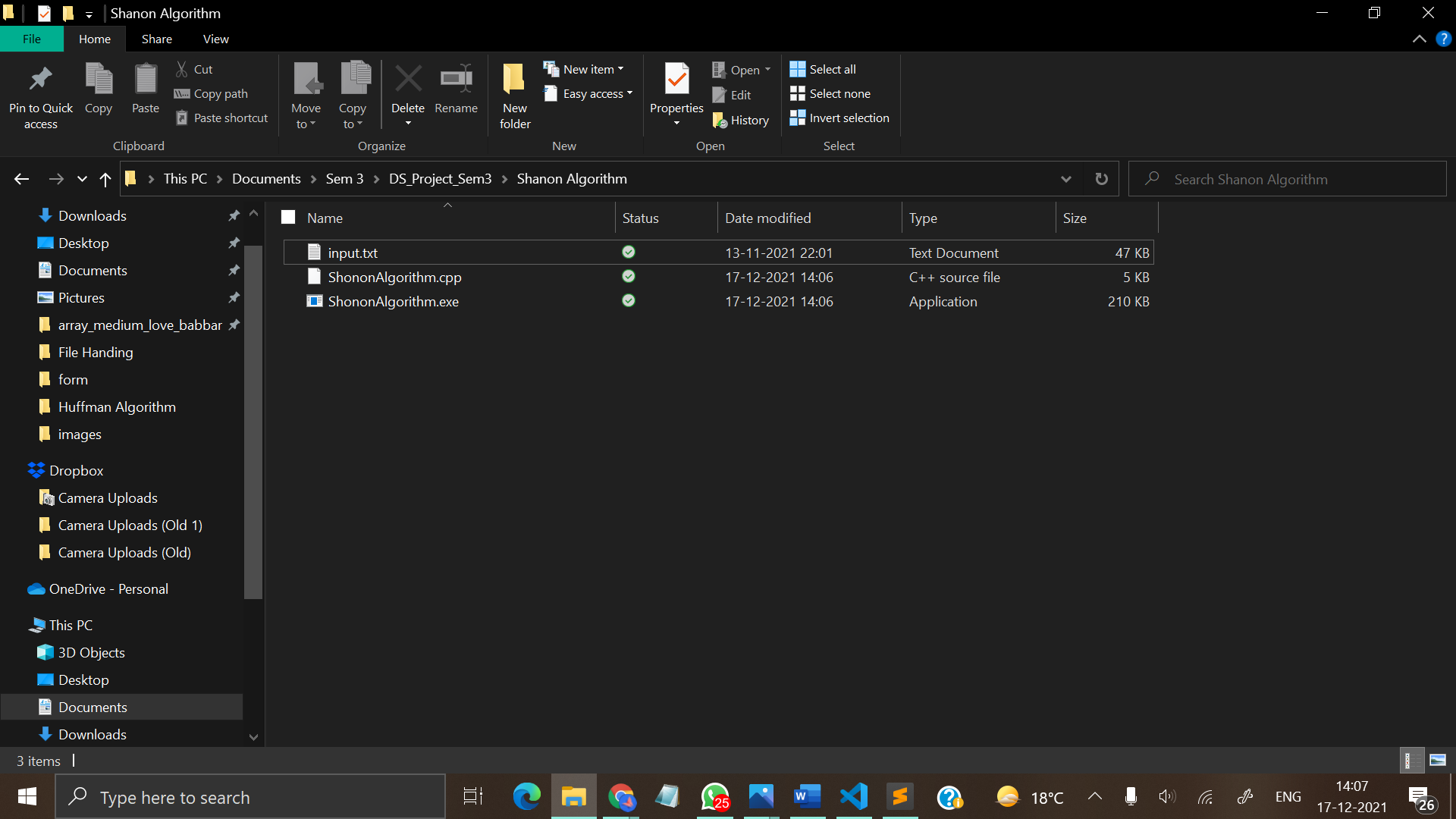
4. The left part of the list is assigned the binary digit 0, and the right part is assigned the digit 1. This means that the codes for the symbols in the first part will all start with 0, and the codes in the second part will all start with 1.

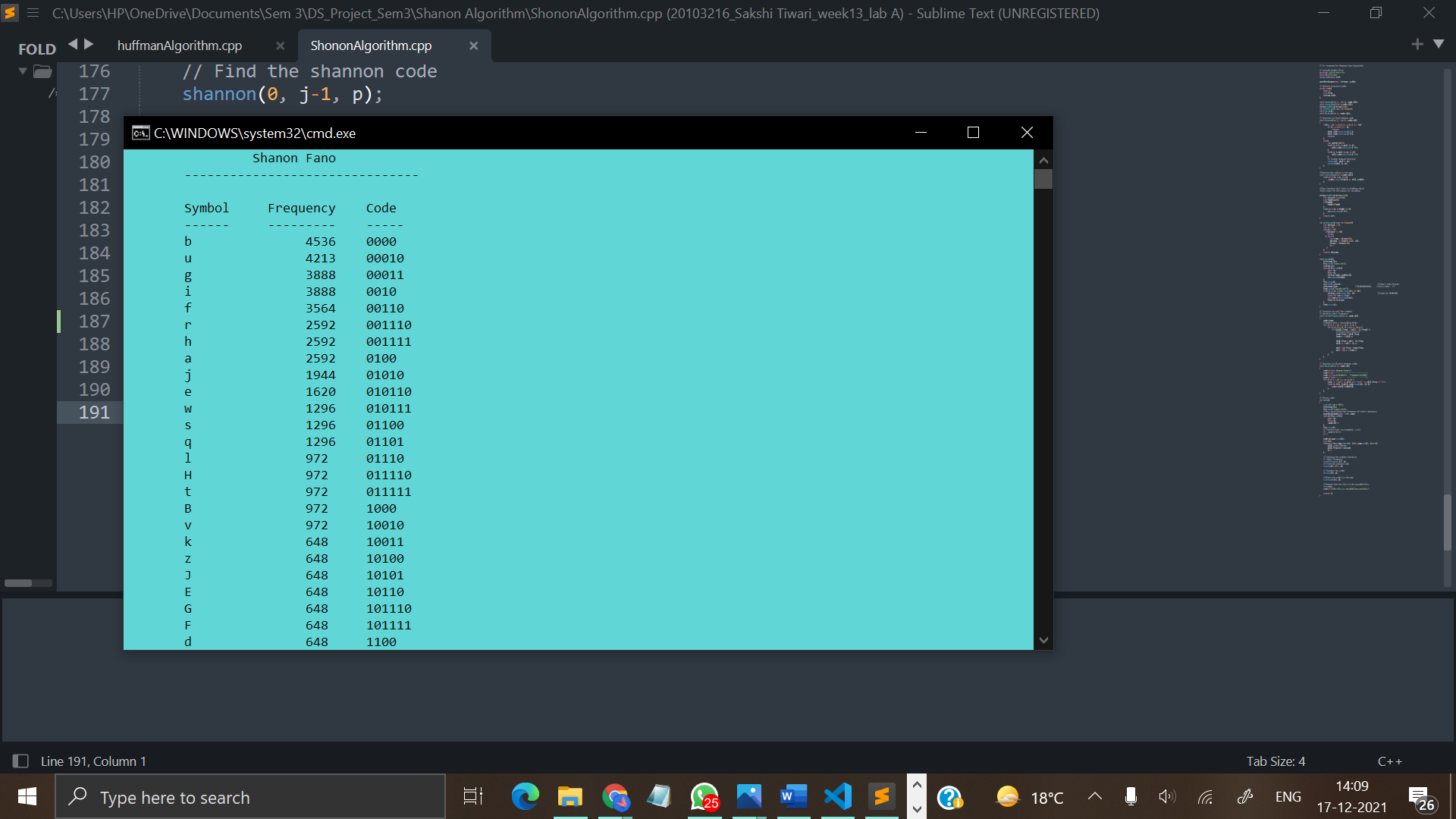
5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

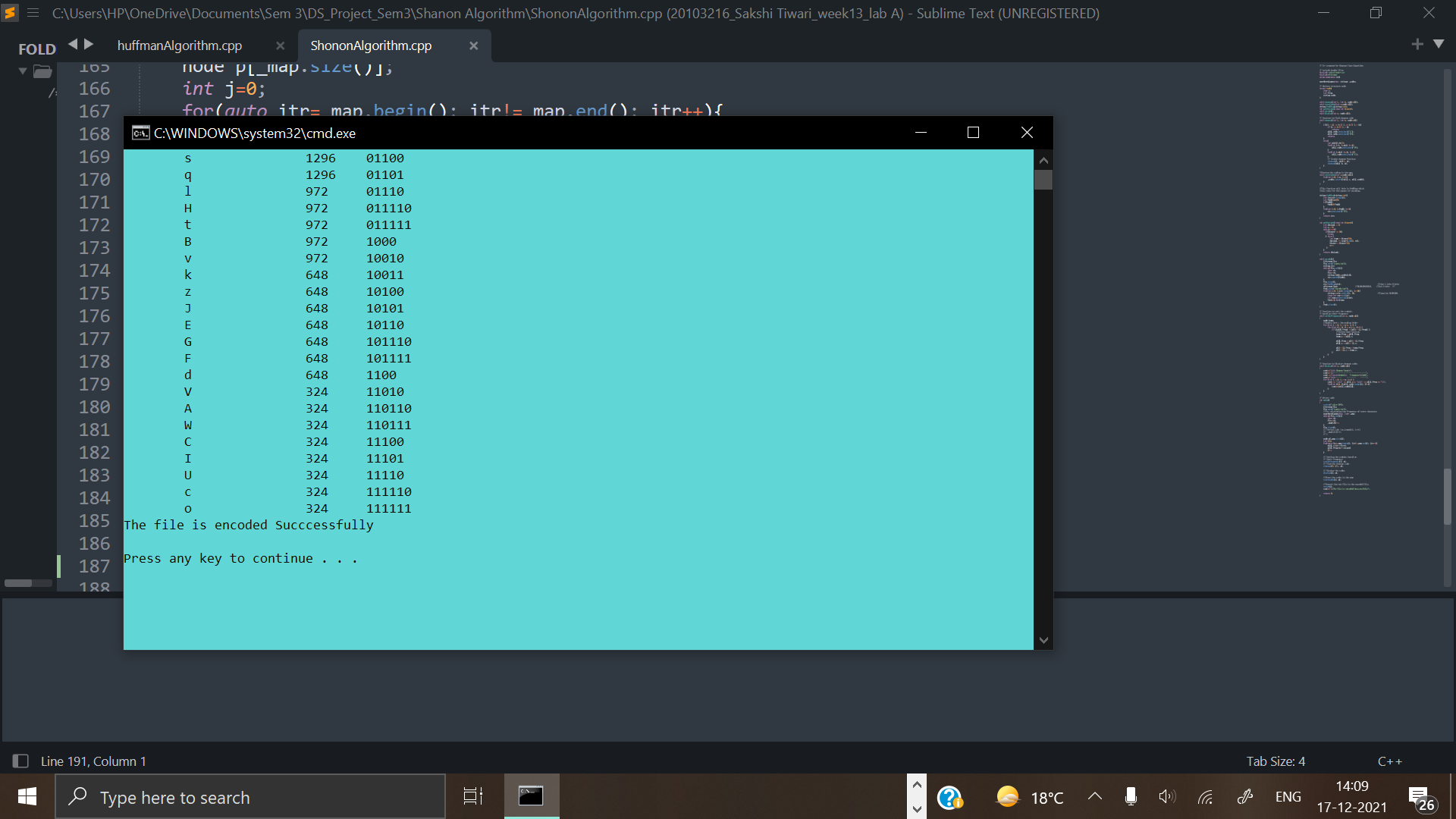


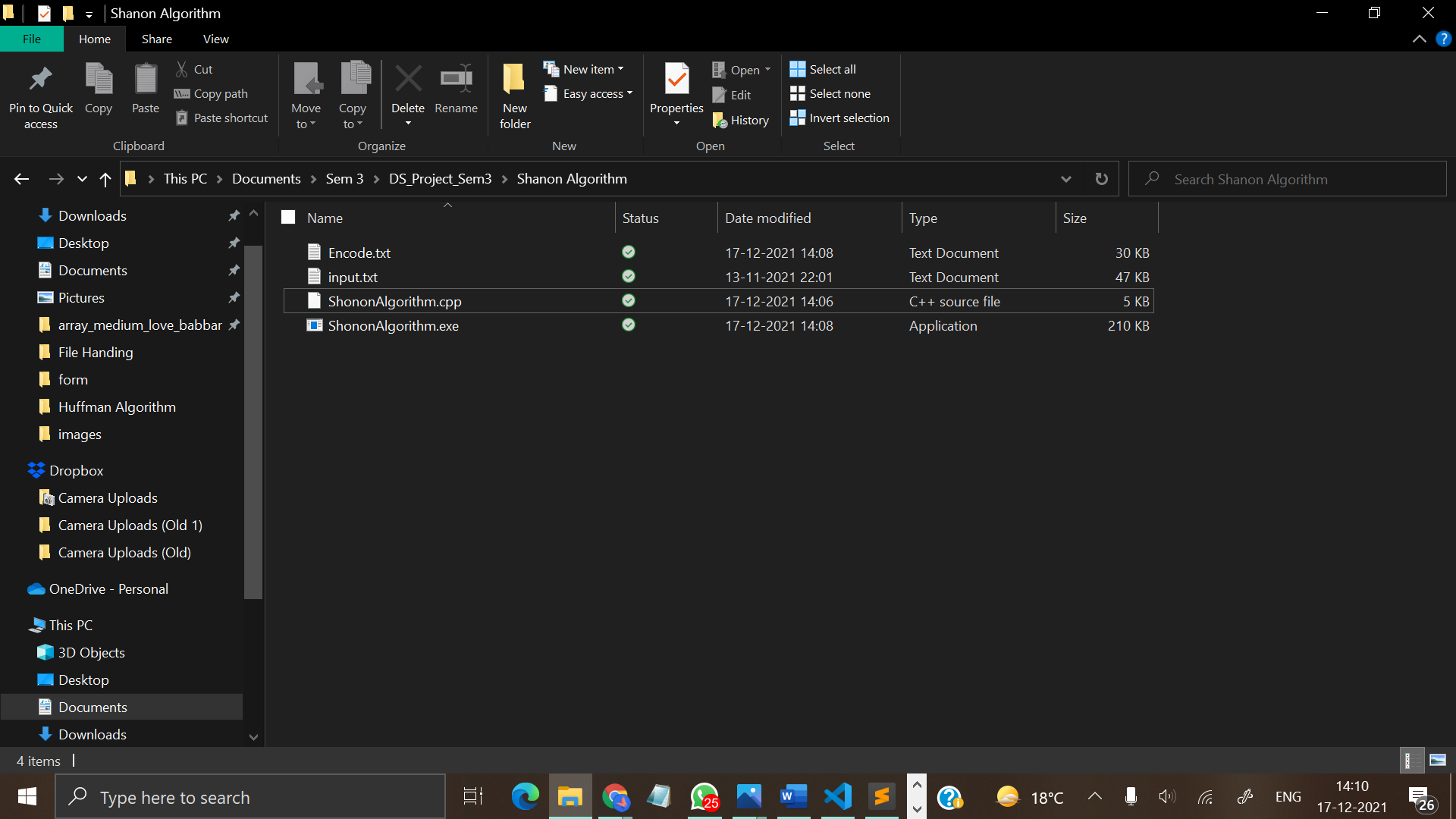
IMPLEMENTATION

**Screenshots of output:**









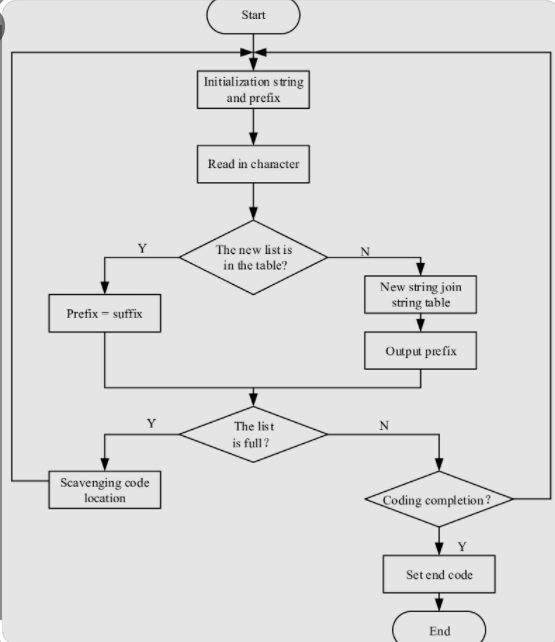
INTRODUCTION TO LZW

The technique encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095 are created in a dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence with no code yet in the dictionary. The code for the sequence (without that character) is added to the output, and a new code (for the sequence with that character) is added to the dictionary.

If you were to take a look at almost any data file on a computer, character by character, you would notice that there are many recurring patterns. LZW is a data compression method that takes advantage of this repetition. The original version of the method was created by Lempel and Ziv and was further refined by Welch in 1984, hence the LZW acronym. Like any adaptive/dynamic compression method, the idea is to (1) start with an initial model, (2) read data piece by piece, (3) and update the model and encode the data as you go along. LZW is a "dictionary"-based compression algorithm. This means that instead of tabulating character counts and building trees (as for Huffman encoding), LZW encodes data by referencing a dictionary. Thus, to encode a substring, only a single code number, corresponding to that substring's index in the dictionary, needs to be written to the output file. Although LZW is often explained in the context of compressing text files, it can be used on any type of file. However, it generally performs best on files with repeated substrings, such as text files.

BASIC TECHNIQUE

1. Create the Table to map the codes and the strings .
2. Create a list to store the codes for the string to be compressed and a variable to store the next code for each string pattern.
3. Create an empty variable to hold the current string.
4. For each character in the string (i), append the character to the current variable.
5. It checks if the current string is not in the dictionary.
6. Then maps it to the next\_code and adds it to the dictionary
7. Increases the next\_code by 1
8. Now, add the code for the current string ignoring the current character i.e. i to the answer list.
9. Now assign the current string to just the current character i.
10. And before we return the compressed codes, add the code to the current string to the dictionary.
11. Now, we return the compressed codes for the input string.



IMPLEMENTATION

**Screenshots:**

