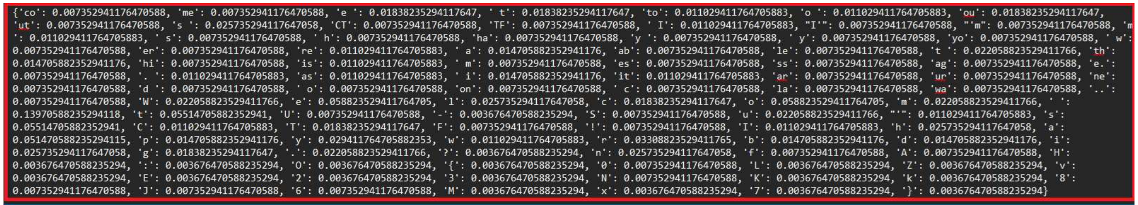
HUFFMAN

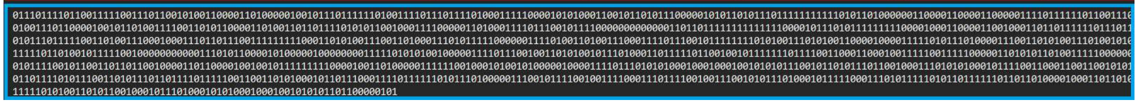
Description:

Trees are very pretty. I like trees!

P/S: the file was large. I’m just providing the image. Sorry for inconvenience.

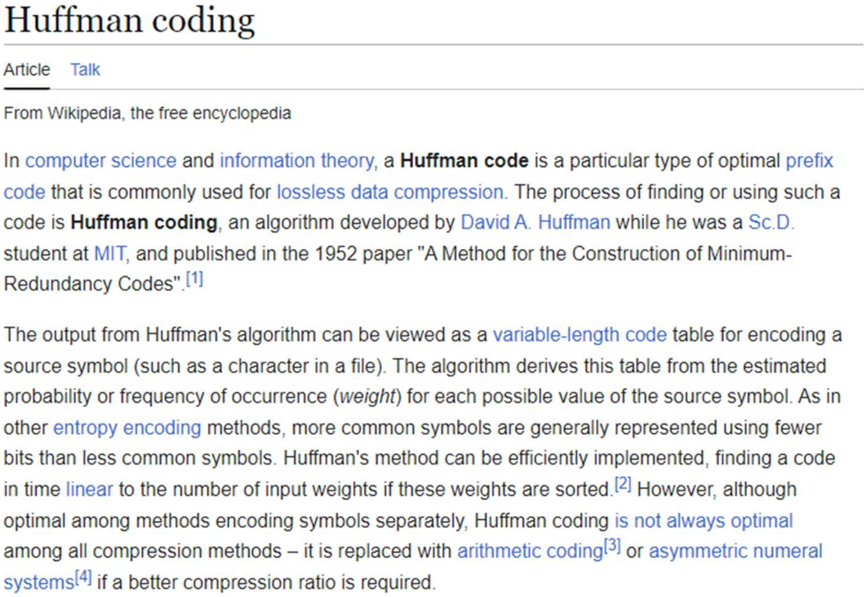


```
{'co': 0.007352941176470588, 'me': 0.007352941176470588, 'e ': 0.01838235294117647, ' t': 0.01838235294117647, 'to': 0.011029411764705883, 'o ': 0.011029411764705883, 'ou': 0.01838235294117647,
'ut': 0.007352941176470588, 's ': 0.025735294117647058, 'CT': 0.007352941176470588, 'TF': 0.007352941176470588, ' I': 0.011029411764705883, "I'": 0.007352941176470588, "'m": 0.007352941176470588, 'm
': 0.011029411764705883, ' s': 0.007352941176470588, ' h': 0.007352941176470588, 'ha': 0.007352941176470588, 'y ': 0.007352941176470588, ' y': 0.007352941176470588, 'yo': 0.007352941176470588, ' w':
0.007352941176470588, 'er': 0.007352941176470588, 're': 0.011029411764705883, ' a': 0.014705882352941176, 'ab': 0.007352941176470588, 'le': 0.007352941176470588, 't ': 0.022058823529411766, 'th':
0.014705882352941176, 'hi': 0.007352941176470588, 'is': 0.011029411764705883, ' m': 0.007352941176470588, 'es': 0.007352941176470588, 'ss': 0.007352941176470588, 'ag': 0.007352941176470588, 'e.':
0.007352941176470588, '. ': 0.011029411764705883, 'as': 0.011029411764705883, ' i': 0.014705882352941176, 'it': 0.011029411764705883, 'ar': 0.007352941176470588, 'ur': 0.007352941176470588, 'ne':
0.007352941176470588, 'd ': 0.007352941176470588, ' o': 0.007352941176470588, 'on': 0.007352941176470588, ' c': 0.007352941176470588, 'la': 0.007352941176470588, 'wa': 0.007352941176470588, '..':
0.007352941176470588, 'W': 0.022058823529411766, 'e': 0.058823529411764705, 'l': 0.025735294117647058, 'c': 0.01838235294117647, 'o': 0.058823529411764705, 'm': 0.022058823529411766, ' ':
0.13970588235294118, 't': 0.05514705882352941, 'U': 0.007352941176470588, '-': 0.003676470588235294, 'S': 0.007352941176470588, 'u': 0.022058823529411766, "'": 0.011029411764705883, 's':
0.05514705882352941, 'C': 0.011029411764705883, 'T': 0.01838235294117647, 'F': 0.007352941176470588, '!': 0.007352941176470588, 'I': 0.011029411764705883, 'h': 0.025735294117647058, 'a':
0.051470588235294115, 'p': 0.014705882352941176, 'y': 0.029411764705882353, 'w': 0.011029411764705883, 'r': 0.03308823529411765, 'b': 0.014705882352941176, 'd': 0.014705882352941176, 'i':
0.025735294117647058, 'g': 0.01838235294117647, '.': 0.022058823529411766, '?': 0.003676470588235294, 'n': 0.025735294117647058, 'f': 0.007352941176470588, 'A': 0.007352941176470588, 'H':
0.003676470588235294, ':': 0.003676470588235294, 'O': 0.003676470588235294, '{': 0.003676470588235294, '0': 0.007352941176470588, 'L': 0.003676470588235294, 'Z': 0.003676470588235294, 'v':
0.003676470588235294, 'E': 0.003676470588235294, '2': 0.003676470588235294, '3': 0.003676470588235294, 'N': 0.007352941176470588, 'K': 0.003676470588235294, 'k': 0.003676470588235294, '8':
0.007352941176470588, 'J': 0.007352941176470588, '6': 0.007352941176470588, 'M': 0.003676470588235294, 'x': 0.003676470588235294, '7': 0.003676470588235294, '}': 0.003676470588235294}
```



```
0111011110110011111001110110010100110000110100000010010111011111110100111101110111101000111110000101010001100101101011100000101011010111011111111111010110100000011000011000011000001111011111101100111 0
010011101100001001011010011110011010110000110100110110111101010110010001111000001101000111101110010111100000000000000011011011111111111111110000101110101111111111000010001100001100100011011011111101110 11
0101110111110011010011100010001110110111001111111110001101010011100110100011101011111000000111101001101001110001111011100101111111101010011101010011000010000111101011101000011100110101001110100101 0
1111101101001011111001000000000000111010110000101000001000000001111101010100100000111110111001001101010010111010001101111110110010010111111101111001100011000100111110011111100000011010101101001111100000 0
010111100101100110110110010000110110000100100101111111111000010011010000001111110010001010010100000100001111011101010100010001000100101010111001011010111011001000111010101000101111001100011001100101011
0110111101011100110101110110111101111100110011010100010110111000111101111110101110100000111001011110010011110001110111100100111001010111010001011111100011101011111010110111111101101101000010001101101 0
1111101010011010110010001011101000101010001000100101010101101100000101
```

I was unfamiliar with Huffman encoding, so I looked it up and learned that it is a lossless data compression technique based on tree structures.



Huffman coding

From Wikipedia, the free encyclopedia

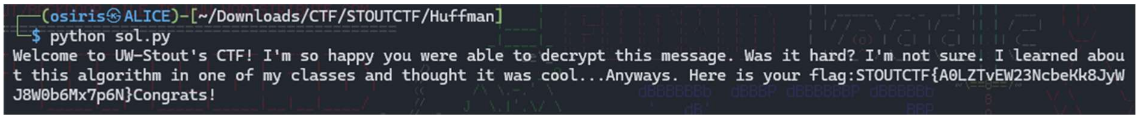
In computer science and information theory, a **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding or using such a code is **Huffman coding**, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".[1]

The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.[2] However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods – it is replaced with arithmetic coding[3] or asymmetric numeral systems[4] if a better compression ratio is required.

So I made a script to decrypt based on the given file and decode it Huffman with node ‘1’ or ‘0’. This script implements Huffman coding for text compression and decompression. It uses a Node class to represent each character and its frequency in a binary tree. A Huffman tree is built using a priority queue (using heapq), where nodes with lower frequencies have higher priority. The generate_codes function creates a mapping of characters to binary strings based on the tree structure, assigning shorter codes to more frequent characters. The decode_huffman function reconstructs the original text from a binary string using the character-to-code mapping. This process encodes and decodes data by minimizing the average code length based on character frequencies.

Code

```
import heapq
from collections import namedtuple

class Node:
    def __init__(self, weight, char, left=None, right=None):
        self.weight = weight
        self.char = char
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.weight < other.weight

def build_huffman_tree(frequency):
    priority_queue = []

    for char, freq in frequency.items():
        heapq.heappush(priority_queue, Node(freq, char))

    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = Node(left.weight + right.weight, None, left, right)
        heapq.heappush(priority_queue, merged)

    return priority_queue[0]

def generate_codes(node, prefix='', codebook={}):
    if node.char is not None:
        codebook[node.char] = prefix
    else:
        generate_codes(node.left, prefix + '0', codebook)
        generate_codes(node.right, prefix + '1', codebook)
    return codebook

def decode_huffman(encoded_string, codebook):
    reversed_codebook = {v: k for k, v in codebook.items()}
    decoded_string = ''
    current_code = ''

    for bit in encoded_string:
        current_code += bit
        if current_code in reversed_codebook:
            decoded_string += reversed_codebook[current_code]
            current_code = ''
    return decoded_string

frequency = {
    'co': 0.007352941176470588,
    'me': 0.007352941176470588,
    'e ': 0.01838235294117647,
    ' t': 0.01838235294117647,
    'to': 0.011029411764705883,
    'o ': 0.011029411764705883,
    #CONTINUE THE ARRAY GIVEN
}

huffman_tree = build_huffman_tree(frequency)
codebook = generate_codes(huffman_tree)

binary_string = (
    "0111011110110011111[redacted]"
)

decoded_text = decode_huffman(binary_string, codebook)
print(decoded_text)
```



```
┌──(osiris㉿ALICE)-[~/Downloads/CTF/STOUTCTF/Huffman]
└─$ python sol.py
Welcome to UW-Stout's CTF! I'm so happy you were able to decrypt this message. Was it hard? I'm not sure. I learned abou
t this algorithm in one of my classes and thought it was cool...Anyways. Here is your flag:STOUTCTF{A0LZTvEW23NcbeKk8JyW
J8W0b6Mx7p6N}Congrats!
```

Flag	STOUTCTF{A0LZTvEW23NcbeKk8JyWJ8W0b6Mx7p6N}